Studio Modeling Platform™

MQL Guide

3DEXPERIENCE R2015x

This page specifies the patents, trademarks, copyrights, and restricted rights for the 3DEXPERIENCE Platform R2015x:

Patents

The 3DEXPERIENCE Platform R2015x is protected by one or more U.S. Patents number 5,615,321; 5,774,111; 5,844,562; 5,844,566; 5,920,491; 6,044,210; 6,233,351; 6,292,190; 6,360,357; 6,396,522; 6,396,522; 6,396,522; 6,459,441; 6,459,441; 6,459,441; 6,499,040; 6,499,040; 6,499,040; 6,545,680; 6,573,896; 6,573,896; 6,573,896; 6,597,382; 6,597,382; 6,597,382; 6,654,011; 6,654,027; 6,697,770; 6,717,597; 6,745,100; 6,762,778; 6,762,778; 6,828,974; 6,828,974; 6,904,392; 6,918,095; 6,934,709; 6,993,461; 6,993,461; 6,993,461; 7,003,363; 7,016,821; 7,152,064; 7,250,947; 7,272,541; 7,289,117; 7,400,323; 7,428,728; 7,495,662; 7,499,845; 7,542,603; 7,555,498; 7,555,498; 7,587,303; 7,587,303; 7,595,799; 7,613,594; 7,620,638; 7,676,765; 7,676,765; 7,710,420; 7,814,429; 7,814,429; 7,814,429; 7,814,429; 7,873,237; 7,913,190; 7,952,575; 7,973,788; 8,010,501; 8,013,854; 8,095,229; 8,095,886; 8,222,581; 8,222,581; 8,248,407; 8,301,420; 8,368,568; 8,386,961; 8,421,798; 8,473,258; 8,473,259; 8,473,524; 8,554,521; 8,645,107; 8,670,957; 8,686,997; 8,694,284; 8,798,975; 8,798,975; 8,812,272; 8,825,450; 8,831,926; 8,832,551; 8,847,947; 8,854,367; 8,868,380; 8,878,841; 8,89,6598; 8,907,947; other patents pending.

Trademarks

3DEXPERIENCE, the Compass icon, the 3DS logo, CATIA, SOLIDWORKS , ENOVIA, DELMIA, SIMULIA, GEOVIA, EXALEAD, 3D VIA, BIOVIA, NETVIBES, 3DSWYM and 3DEXCITE, are commercial trademarks or registered trademarks of Dassault Systèmes or its subsidiaries in the United States and/or other countries. Use of any Dassault Systèmes or its subsidiaries trademarks is subject to their express written approval.

Other company, product, and service names may be trademarks or service marks of their respective owners.

Copyright

Certain portions of the 3DEXPERIENCE Platform R2015x contain elements subject to copyright owned by the following entities:

© Modelon AB
© Distrim
© NVIDIA ARC
(c) Copyright IBM Corporation 2007 All Rights Reserved" + cf http://www-03.ibm.com/software/sla/sladb.nsf/c7134e107cf0624e86256738007531d7/fd6322f964805a37002573a70058ab2e?OpenDocument
© DLR (Deutsches Zentrum f¸r Luft und Raumfahrt)
© DISTENE
© Kjell Gustafsson
© 2000 Geometric Limited
© Weber-Moewius, D-Siegen
© Oracle
© INRIA
© INRIA
© ITI (International Technegroup Corporation)
Raster Imaging Technology copyrighted by Snowbound Software 1996-2000
© Arsenal

© Allegorithmic
Copyright (c) 1997-2004 Lattice Technology, Inc. All Rights reserved
© NVIDIA
© Intel Corp
© Regents of the University of Minnesota
© Third Millenium Productions
© Scapos
© NuoDB
ìCAM-POST Æ Version 2001/14.0 © ICAM Technologies Corporation 1984-2001. All rights reservedî
© CENIT
© IMS
Contains copyrighted materials of MachineWorks Design Limited (c) 1995-2000
© NCCS
© Copyright 2003 - 2009 by Technia AB. All rights reserved. This product includes software developped by Technia AB (http://www.technia.com)
Unpublished - Copyright 1991-1997 Synopsys Inc. All rights reserved. The program and information contained herein are licensed only pursuant to a license agreement that contains use, reverse engineering, disclosure and other restrictions; accordingly, it is "Unpublished – rights reserved under the copyright laws of the United States" for purposes of the FARs. RESTRICTED RIGHTS LEGEND: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, this LICENSED PRODUCT, the rights of the Government regarding its use, reproduction and disclosure are as set forth in the Government contract.
XMLmind XSL-FO Converter Copyright © 2002-2009 Pixware SARL
Copyright Sun Microsystems, Inc.

This software is based in part on the work of the independent JPEG Group.

The **3D**EXPERIENCE Platform R2015x may include open source software components or free programs (together “OS Programs”). Each such program is distributed with **3D**EXPERIENCE Platform R2015x in binary form and, except as permitted by the applicable license, without modification. Each such program is available online for free downloading and, if required by the applicable OS Program license, the source code will be made available by Dassault Systèmes upon request.

IP Asset Name	IP Asset Version	Copyright notice
Under Academic Free License		
JAVA SWING DATE PICKER	v0.99-2006.09.01	
Under Apache 1.1		
Element Construction Set	1.4.2	Copyright (c) 1999-2003 The Apache Software Foundation. All rights reserved
Jakarta	2.07	Copyright 1999–2004, The Apache Software Foundation
JAKARTA Regular Expression	1.3	Copyright (c) 1999-2003 The Apache Software Foundation. All rights reserved.
JavaMail / Servlet-API	1.4.2 / 2.3	Copyright (c) 1999 The Apache Software Foundation. All rights reserved. VOIR AVEC R&D S IL S AGIT DU MEME COMPOSANT OU DEUX COMPOSANTS FONCTIONNANT ENSEMBLE

IP Asset Name	IP Asset Version	Copyright notice
Xalan	2.3.1	Copyright (c) 1999-2003 The Apache Software Foundation
XML4C	2.4	Copyright (C) 1998-2008, International Business Machines Corporation * and others
Code Generation Library (cglib)	2.2.2	This product includes software developed by Yale University
Under Apache 2.0		
ActiveMQ-activeIO-core	3.0.0	
Amazon Java SDK	1.3.26	
Ant	1.6.1	The Apache License Version 2.0 applies to all releases of Apache Ant starting with Ant 1.6.1
Apache Common Lang	2.0	Copyright 2001-2014 The Apache Software Foundation
Apache Commons	1.8	Copyright 2001-2012 The Apache Software Foundation
Apache Commons-cli	1.2	Copyright 2001-2009 The Apache Software Foundation
Apache Commons-codec	1.4	Copyright 2002-2013 The Apache Software Foundation
Apache Commons-Compress	1.8	Copyright 2002-2014 The Apache Software Foundation
Apache Commons-FileUpload	1.2.2	Copyright 2002-2010 The Apache Software Foundation
Apache Commons-httpclient	3.1	Copyright 1999-2011 The Apache Software Foundation
Apache Commons-io	2.1	Copyright 2002-2014 The Apache Software Foundation
Apache Commons-JEXL	1.1	Copyright 2006 The Apache Software Foundation
Apache Commons-lang		Copyright 2001-2014 The Apache Software Foundation
Apache Commons-logging	1.1.1	Copyright 2003-2013 The Apache Software Foundation
Apache HTTP Server	2.2.23	Copyright (c) 2011 The Apache Software Foundation.
Apache log4j	1.2.16	Copyright 2007 The Apache Software Foundation
Apache POI	2.5.1	Copyright 2002-2004 The Apache Software Foundation
Apache Storm	0.8	Copyright 2014 The Apache Software Foundation
Apache Tomcat	6	Copyright 1999-2014 The Apache Software Foundation
Apache.commons.fileupload	1.2.1	Copyright 2002-2008 The Apache Software Foundation
ApacheSSL	2.2.21	
Axis	1.4	Copyright 2001-2004 The Apache Software Foundation
Bean Validation API	1.0.0.GA	Copyright (c) Red Hat, Inc., Emmanuel Bernard
BoneCP	0.7.1.RELEASE	Copyright 2010 Wallace Wadge
CAS client (java)	2.1	Copyright 2010, JA-SIG, Inc
Commons Math Bundle	1.2	Minpack Copyright Notice (1999) University of Chicago. All rights reserved / This product includes software developed by the University of Chicago, as Operator of Argonne National Laboratory
Daisydiff	1.1	Copyright 2007 © Guy Van den Broeck <guy@guyvdb.eu>; Daniel Dickison

IP Asset Name	IP Asset Version	Copyright notice
Derby	10.8.2.2	(C) Copyright 1997,2004 International Business Machines Corporation. All rights reserved This product includes software developed by The Apache Software Foundation (http://www.apache.org/).
Ehcache	2.4.6	Copyright 2003-2010 Terracotta, Inc.
Formatting Objects Processor (FOP)		Copyright (c) 1998-1999, James Tauber. All rights reserved.
GChart	2.3	Copyright 2007,2008,2009 John C. Gunther
Google Web Toolkit (GWT)	1.5	Copyright 2007, Google Inc.
Guava	14.0	Copyright (c) 2011 Guava Authors. All rights reserved
GWT Drag and Drop	2.5.6	Copyright 2009 Fred Sauer
GWT Incubator	1.5	Copyright 2008, Google Inc.
GWTx	1.5-20081912	Copyright 2009 Google Inc.
Hibernate Validator Engine	4.3.1.Final	Copyright 2009, Red Hat, Inc. and/or its affiliates
HTTPClient	3.1	Copyright 1999-2007 The Apache Software Foundation
ibatis-core	3.0	
Ini4j 0.5.2	0.5.2	Copyright 2005,2009 Ivan SZKIBA
Inspektr	1.0.7.GA	Copyright 2010 Rutgers, the State University of New Jersey, Virginia Tech, and Scott Battaglia
iossim		Copyright (c) 2009-2013 by Appcelerator, Inc. All Rights Reserved. / A TRAN-CHER AVEC R&D
Jackson	1.9.12	Copyright (c) 2007- Tatu Saloranta, tatu.saloranta@iki.fi
jakarta.commons.lang	2.4	Copyright 2001-2008 The Apache Software Foundation
Jakarta.commons.logging	1.0.1	Copyright 2001-2004 The Apache Software Foundation.
jakarta.commons.net	1.4.0	Copyright 2001-2005 The Apache Software Foundation
Jasper	5	Copyright 1999-2010 The Apache Software Foundation
Jettison	1.3.2	Copyright 2006 Envoi Solutions LLC
JNRPE		Copyright (c) 2008 Massimiliano Ziccardi
Joda Time		Copyright 2001-2012 Stephen Colebourne
json simple	1.1	Copyright ©FangYidong<fangyidong@yahoo.com.cn>
JUG (Java UUID Generator)	1.1.2	Copyright (c) 2010 Tatu Saloranta
log4j		Copyright 2010 The Apache Software Foundation
opencsv	2.0	
Tomcat	6	Copyright 1999-2014 The Apache Software Foundation
TRUEZIP	6.7 Beta 2	Copyright (C) 2009 Schlichtherle IT Services
VIAMobileIntegration	2009	Copyright 2009 Facebook
WatiN	1.3	Copyright Jeroen van Menen 2011

IP Asset Name	IP Asset Version	Copyright notice
Xalan C++	1.10	Copyright (c) 1999-2012 The Apache Software Foundation
Xerces C++	3.01	Copyright © The Apache Software Foundation
Xerces-J	2.6.2	Copyright © The Apache Software Foundation
Under Apache 2.0	Or LGPL 2.1	
Javassist	3.15.0-GA	Copyright (c) 1999-2005 Shigeru Chiba. All Rights Reserved.
Under Apache 2.0	Or BSD 2	
CardMe	0.3.6.01	Copyright 2011 George El-Haddad. All rights reserved.
Under Apache 2.0	Or BSD 3	
CAS Client (PHP)	1.3.2	Copyright 2007-2011, JA-SIG, Inc.; Copyright © 2003-2007, The ESUP-Portail consortium; Copyright (c) 2009, Regents of the University of Nebraska All rights reserved.
Under Apple License		
KeychainItemWrapper		Copyright (C) 2010 Apple Inc. All Rights Reserved.
Under ASM license		
ASM Core	3.2	Copyright (c) 2000-2011 INRIA, France Telecom
Under BeOpen Python License Agreement		
Cookie	Python-2.7	Copyright 2000 by Timothy O'Malley
Under Boost license		
Wild Magic Library		Geometric Tools, LLC // Copyright (c) 1998-2014
Boost		Copyright Joe Coder 2004 - 2006.
Or BSD 2		
yasm	1.2.0	Copyright (C) 2003-2007 Peter Johnson
xmppframework	3.2	Copyright (c) 2007, Deusty Designs, LLC
FMI Interface MAProject		Copyright The Modelica Association
Or BSD 3		
_wincon.c	2001-05-08	Copyright (c) 1999-2001 by Secret Labs AB. # Copyright (c) 1999-2001 by Fredrik Lundh.
Adaptive Simulated Annealing (ASA)	v 23.7 2001/10/12 14:01:08	Copyright © 1987-2014 Lester Ingber. All Rights Reserved.
ANTLR	1.33MR33	Copyright © 2003-2006, Terence Parr ANTLR 3 / Public domain ANTLR 2
Atmosphere & Ocean	v2	Copyright (c) 2008 INRIA
jaxen	1.1.1	Copyright 2003-2006 The Werken Company. All Rights Reserved.
JGraphX	1.3.1.6	Copyright (c) 2001-2009, JGraph Ltd
Kiss FFT	1.3.0	Copyright (c) 2003-2010 Mark Borgerding
libogg	1.2.0	Copyright (c) 2002 Xiph.org Foundation

IP Asset Name	IP Asset Version	Copyright notice
libTheora	1.1.1	Copyright (c) 2002-2009 Xiph.org Foundation
libVorbis	Vorbis I Release: 1.3.1 (Feb, 3, 2010)	Copyright (c) 2002-2008 Xiph.org Foundation
Penner's easing functions		Copyright © 2001 Robert Penner
Skia Graphics Library		Copyright (c) 2011 Google Inc. All rights reserved
V8	3	Copyright 2006-2011, the Google V8 project authors
Visualization Toolkit (VTK)	5.10	Copyright (c) 1993-2008 Ken Martin, Will Schroeder, Bill Lorensen
Vorbis	1.3.3	Copyright (c) 2002-2008 Xiph.org Foundation
vpx	1.1.0	Copyright (c) 2010 The WebM project authors
yamdi	1.8	Copyright (c) 2007-2010, Ingo Oppermann
yuicompressor	2.4.7	Copyright (c) 2013, Yahoo! Inc.
Zend Framework	1.10.2	Copyright (c) 2005-2014, Zend Technologies USA, Inc. All rights reserved.
ZipJS		Copyright (c) 2013 Gildas Lormeau. All rights reserved.
ical4J		Copyright (c) 2012, Ben Fortuna * All rights reserved.
XStream	1.3.1	Copyright (c) 2003-2006, Joe Walnes Copyright (c) 2006-2009, 2011 XStream Committers
Data Driven Documents (D3)	3.0.0	Copyright (c) 2010-2014, Michael Bostock All rights reserved.
ESAPI	2.1	Copyright © 2009 The OWASP Foundation.
GCC-XML		Copyright 2002-2012 Kitware, Inc., Insight Consortium. All rights reserved.
hsqldb	2.2.9	Copyright (c) 2001-2011, The HSQL Development Group * All rights reserved.
Under BSD Style		
itcl	3.4	This software is copyrighted by Lucent Technologies, Inc., and other parties
dom4j	1.6.1	Copyright 2001-2005 MetaStuff Ltd.. All Rights Reserved
Exodus II	4.84	Copyright (c) 2005 Sandia Corporation
Kiss FFT	1.3.0	Copyright (c) 2003-2010 Mark Borgerding All rights reserved.
FreeType	2	Copyright 2000 The FreeType Development Team
JDOM	1.0	Copyright (C) 2000-2004 Jason Hunter & Brett McLaughlin. All rights reserved.
Natural Comparator	n.a.	Copyright (c) 2006, Stephen Kelvin Friedrich, All rights reserved. This a BSD license.
Under Castor License		
Castor	0.9.3.9	Copyright 2000 (C) Intalio Inc. All Rights Reserved.

IP Asset Name	IP Asset Version	Copyright notice
Under CDDL 1.0	Or GNU GPLV1.0	
JavaMail	1.4.2	Copyright 1997-2007 Sun Microsystems, Inc. All rights reserved.
Under CDDL 1.0	Or GNU GPL V2.0 classpath exception	
JBoss Transaction 1.1 API	1.0.0.Final	Copyright (c) 2011 Oracle and/or its affiliates. All rights reserved
Under CDDL 1.1		
Java Message Service	3.12.1.GA	Copyright (c) 2003-2010 Oracle and/or its affiliates. All rights reserved
Under Common Public License Version 0.5		
junit	3.8.1	
Under Custom Permissive License		
KeychainItemWrapper		Copyright (C) 2010 Apple Inc. All Rights Reserved
Under Custom Permissive License		
LibJPG		Thomas G. Lane, Guido Vollbeding.
Under Custom Permissive License		
LibTIFF		Copyright (c) 1988-1997 Sam Leffler Copyright (c) 1991-1997 Silicon Graphics, Inc.
Under Customized MIT License		
Tls	1.6	Copyright (C) 1997-2000 Matt Newman
Under Customized License		
Trf	2.1p2	This software is copyrighted by Andreas Kupries
Under Customized MIT License		
Tiff library	3.5.7	Copyright (c) 1988-1997 Sam Leffler Copyright (c) 1991-1997 Silicon Graphics, Inc.
Under Customized Boost license		
VRPN		Public domain until version 7.27 and then customized Boost License with credit to "The CISMM project at the University of North Carolina at Chapel Hill, supported by NIH/NCRR and NIH/NIBIB award #2P41EB002025"
Under Customized License		
libPNG		Copyright (c) 2004, 2006-2014 Glenn Randers-Pehrson depending on the asset version - to be confirmed with R&D)
Under Eclipse Public License 1.0		
AspectJ	1.6.11	Copyright (c) 1998-2001 Xerox Corporation, 2002 Palo Alto Research Center, Incorporated, 2003-2008 Contributors. All rights reserved.
Eclipse Platform	3.3.1	A VOIR AVEC LA R&D
Graphviz	none	* Copyright (c) 2011 AT&T Intellectual Property. All rights reserved

IP Asset Name	IP Asset Version	Copyright notice
Under CDDL		
jersey	1.17	Copyright (c) 2010-2011 Oracle and/or its affiliates. All rights reserved
Info-ZIP license		
Unzip (from InfoZip)	6.0	Copyright (c) 1990-2009 Info-ZIP. All rights reserved.
Zip	3.0	Copyright (c) 1990-2009 Info-ZIP. All rights reserved.
Under Jasig License for USE		
cas-client-core	3.1.6	Copyright 2007 The JA-SIG Collaborative. All rights reserved
Under LGPL		
unix ODBC	2.2.14	A revoir avec Rodolphe
GWT Beans Binding	0.2.3	* Copyright (C) 2006-2007 Sun Microsystems, Inc. All rights reserved.
GWT Mosaic	0.1.9.1	Copyright (C) 2009 Georgios J. Georgopoulos, All rights reserved.
Hibernate Commons Annotations	4.0.1.Final	Copyright (c) 2008, Red Hat Middleware LLC
Hibernate	4.1.6.Final	Copyright (c) 2009 by Red Hat Inc and/or its affiliates
WxPython	2.8	Copyright (c) 1992-2013 Julian Smart, Vadim Zeitlin, Stefan Csomor, Robert Roebling, and other members of the wxWidgets team
Under LGPL 2.1		
Hibernate	3.17.1-GA	Copyright (c) 2007, Red Hat Middleware, LLC. All rights reserved.
HTMLPurifier	4.4	Copyright 2006-2008 Edward Z. Yang
JACOB	1.14.3	Copyright (c) 1999-2004 Sourceforge JACOB Project. All rights reserved. Originator: Dan Adler (http://danadler.com)
JBOSS	6.1.0	Copyright 2011 Red Hat Inc. and/or its affiliates and other contributors as indicated by the @author tags. All rights reserved
JBoss Logging 3	3.1.0.GA	Copyright 2011 Red Hat, Inc., and individual contributors as indicated by the @author tags
JCIFS Libraries	1.2.25b	
jregistrykey	1.0	Copyright © 2001, BEQ Technologies Inc.
jfreechart		
Tiny MCE	3.4.6	Copyright 2009, Moxiecode Systems AB
Under LGPL 2.1	Or under GNU GPL V2	
FFMpeg	1.1	Copyright © 2000 Fabrice Bellard et al.
Under LGPL 2.1	Or under Eclipse public license - v 1.0	
c3p0	0.9.1.2	Copyright: (C) 2001-2007 Machinery For Change, Inc.
Under GNU LGPL 3.0		
libmcrypt		Copyright (C) 1998,1999,2000,2002 Nikos Mavroyanopoulos
Under MIT License		

IP Asset Name	IP Asset Version	Copyright notice
_random	Python-2.7	Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.
asyncore-asynchat	Python-2.7	Copyright 1996 by Sam Rushing
Bouncy Castle Libraries		Copyright (c) 2000 - 2013 The Legion of the Bouncy Castle Inc.
Com4J		Copyright (c) 2003, Kohsuke KawaguchiAll rights reserved.
Code mirror		Copyright (C) 2012 by Marijn Haverbeke <marijnh@gmail.com> and others
Credis		Copyright (c) 2009 Justin Poliey <jdp34@njit.edu> Copyright (c) 2011 Colin Mollenhour <colin@mollenhour.com>
ctypes	1.1.0	Copyright (c) 2000 - 2006 Thomas Heller
curl.js	0.7.3	Copyright (c) 2010-2013 Brian Cavalier and John Hann
Dynamic Java	1.1.5	DynamicJava - Copyright 1999 Dyade
Easysax		Copyright (c) 2012 Vopilovskii Constantine <flash.vkv@gmail.com>
Expat	Expat XML Parser-2.0.0	
Express	3.1.0	Copyright (c) 2009-2014 TJ Holowaychuk <tj@vision-media.ca>
f2c	20100827	Copyright 1990 - 1997 by AT&T, Lucent Technologies and Bellcore.
FTGL		Copyright (c) 2001-2004 Henry Maddocks <ftgl@opengl.geek.nz> Copyright (c) 2008 Sam Hocevar <sam@zoy.org> Copyright (c) 2008 Sean Morrison <learner@brlcad.org>
Hammerjs	1.0.5	Copyright (C) 2013 by Jorik Tangelder (Eight Media)
Java Cup	11	Copyright 1996-1999 by Scott Hudson, Frank Flannery, C. Scott Ananian
Java Service Wrapper	3.0.2	Copyright (c) 1999, 2004 Tanuki Software
jQuery		Copyright 2014 jQuery Foundation and other contributors
jQuery Simple Context Menu	1.0	Copyright (c) 2011, Joe Walnes
libffi	Python-2.7	Copyright (c) 1996-2012 Anthony Green, Red Hat, Inc and others.
libxml	2.4.5	Copyright (C) 1998-2003 Daniel Veillard
LittleCMS		Copyright (c) 1998-2012 Marti Maria Saguer
Markdown-js	1.0	Copyright (c) 2009-2010 Dominic Baggott // Copyright (c) 2009-2010 Ash Berlin
MD5	(none)	Copyright (C) 1999 Aladdin Enterprises. All rights reserved.
OpenCL API	1.1	Copyright (c) 2008-2010 The Khronos Group Inc.
Three.js	R58	Copyright (c) 2010-2012 three.js authors
uu	Python-2.7	Copyright 1994 by Lance Ellinghouse, Modified by Jack Jansen, CWI, July 1995
when.js	1.7.1	Copyright (c) 2011 Brian Cavalier
WINP	1.14	Copyright (c) 2004-2009, Sun Microsystems, Inc., Kohsuke Kawaguchi
Winston	0.7.2	Copyright (c) 2010 Charlie Robbins

XML Xpat Parser	1.95.4	Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper Copyright (c) 2001, 2002 Expat maintainers.
jsPlumb	V6R2014x	Copyright (c) 2010 - 2013 Simon Porritt (http://jsplumb.org)
xmlrpclib	1.0.1	Copyright (c) 1999-2002 by Secret Labs AB Copyright (c) 1999-2002 by Fredrik Lundh
Under the Modelica License		
Modelica Standard Library		Copyright The Modelica Association
Under Mozilla Public License Version 1.1a		
Mozilla Rhino	1.7R2	
Extended Message boxes		Copyright (c) 2004 Michael P. Mehl. All rights reserved (Version 1.1a)
Under Ms-LPL License		
ATLSOAPInterfaces		Copyright © Microsoft Corporation
Under Python 2.2 License		
Trace	(no version)	copyright 2001, Autonomous Zones Industries, Inc., Copyright 2000, Mojam Media, Inc, Copyright 1999, Bioreason, Inc., Copyright 1995-1997, Automatrix, Inc., Copyright 1991-1995, Stichting Mathematisch Centrum
Jython	2.5.2	Copyright (c) 2007 Python Software Foundation; All Rights Reserved
Python	Python-2.5	Copyright © 2001-2014 Python Software Foundation; All Rights Reserved
Under Zlib License		
NanoXml	2.2.5	Copyright (C) 2000-2002 Marc De Scheemaecker, All Rights Reserved.
Natural Order Sort	2004-10-10 mbp	This software is copyright by Martin Pool, and made available under the same licence as zlib
ZLib		Copyright (C) 1995-2013 Jean-loup Gailly and Mark Adler
Zlib	Zlib-1.2.4	Zlib software copyright © 1995-2012 Jean-loup Gailly and Mark Adler

This clause applies to all acquisitions of Dassault Systèmes Offerings by or for the United States federal government, or by any prime contractor or subcontractor (at any tier) under any contract, grant, cooperative agreement or other activity with the federal government. The software, documentation and any other technical data provided hereunder is commercial in nature and developed solely at private expense. The Software is delivered as "Commercial Computer Software" as defined in DFARS 252.227-7014 (June 1995) or as a "Commercial Item" as defined in FAR 2.101(a) and as such is provided with only such rights as are provided in Dassault Systèmes standard commercial end user license agreement. Technical data is provided with limited rights only as provided in DFAR 252.227-7015 (Nov. 1995) or FAR 52.227-14 (June 1987), whichever is applicable. The terms and conditions of the Dassault Systèmes standard commercial end user license agreement shall pertain to the United States government's use and disclosure of this software, and shall supersede any conflicting contractual terms and conditions. If the DS standard commercial license fails to meet the United States government's needs or is inconsistent in any respect with United States Federal law, the United States government agrees to return this software, unused, to DS. The following additional statement applies only to acquisitions governed by DFARS Subpart 227.4 (October 1988): "Restricted Rights - use, duplication and disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(l)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252-227-7013 (Oct. 1988)

Table of Contents

Preface

The *Studio Modeling Platform MQL Guide* is intended for users who...

- use a command driven interface to all of the functions that can be performed in Live Collaboration.
- use MQL to perform most of the operations that can be done using Matrix Navigator or Business Modeler.
- use MQL to perform other functions of Live Collaboration that you can not do with the GUI applications, including system administration functions.

About This Guide

Purpose

This guide provides the MQL programmers with conceptual and reference information concerning the MQL language.

MQL is the Matrix Query Language. Similar to SQL, MQL consists of a set of commands that help the administrator set up and test a Live Collaboration database quickly and efficiently.

MQL is primarily a tool for building the Live Collaboration database. Also, you can use MQL to add information to the existing database, and extract information from the database.

This guide provides MQL programming concepts and reference information to persons responsible for building and maintaining the Live Collaboration database.

Intended Audience

You can use MQL to perform most of the operations that can be done using Matrix Navigator or Business Modeler. MQL also provides additional commands for functions that can't be done with these GUI applications. MQL commands can be run on a command line or via a script. For these purposes, you can review the introduction and concepts chapters of this guide, or the following related documentation:

- *Matrix Navigator Guide*

  Review this guide to become familiar with the Matrix application.
- *Business Modeler Guide*

  Review this guide to become familiar with the Business Modeler application, and the kinds of tasks the Business Administrator performs.

Skills You Need

Programmers who will use MQL in Java or.Net programs to be used across the enterprise should have a strong foundation in the following skills:

- Knowledge of database management and Web servers
- Object-oriented programming techniques
- Programming in a multi-threaded environment
- Java, JavaServer Pages, and JavaScript or C# and ActiveServer Pages programming experience.

Introducing Live Collaboration™

An Information Management System

Live Collaboration is a standards-based, open and scalable system able to support the largest, most complex, product lifecycle management deployments. It provides the flexibility to easily configure business processes, user interfaces, and infrastructure options to ensure that they meet your organization's needs. The system enables you to continually drive business process improvements to operate more efficiently using pre-built metrics reports, while virtual workplace capabilities enable ad-hoc collaboration for cross-functional and geographically dispersed teams to securely share product content.

The Live Collaboration system operates in virtually any configuration to support your unique operating, organizational, and performance needs—on a single computer, in a networked system environment, over the internet or an enterprise intranet.

This chapter introduces you to the concepts and features of Live Collaboration.

Live Collaboration Database Components

Live Collaboration (CPF) provides the backbone for product lifecycle management activies within a workgroup, enterprise, or extended enterprise, depending on the organization's needs. Live Collaboration is the underlying support for all 3DEXPERIENCE products and other Dassault Systèmes products for Product Lifecycle Management, which include products from CATIA, 3DLive, SIMULIA, and DELMIA product lines. It is comprised of the following end-user software components:

- Matrix Navigator, which is the tool for searching the data objects based on the data model. This application, or its Web-based version (MXW), is required to administer triggers and other configuration objects for pre-V6 legacy clients. Both the Rich client and Web version Navigator products are included with CPF. The rich client is installed as part of the Studio Modeling Platform Rich Clients distribution for ease of installation. The CPF license entitles your use of this component of Studio Modeling Platform only.
- *Live Collaboration Server*, which is a Java/RMI-based business object server. The MQL command line executable is included with the server for the purpose of system installation and data/file storage setup functions only. If any other schema configurations are needed, Studio Modeling Platform (DTE) must be licensed. Note that if distributed file stores are created with Sites and Locations, one or more File Collaboration Server licenses may also be needed.
- *Live Collaboration Business Process Services*, include the capabilities from the Application Exchange Framework, Common Components, Team Central, and Business Process Metrics applications.

The following administrative add-ons are also available:

- *Studio Modeling Platform (DTE)* provides the development tools for a company to define and test configurations that are needed in their production system. While the 3DEXPERIENCE platform offers PLM products that cover many product development business processes, the need for companies to tailor or extend products to meet their specific needs is inevitable, since a company's competitive advantage often requires a unique product development process compared to how other companies execute. Therefore, in order to deploy the 3DEXPERIENCE system, companies may need to set up an environment that allows them to develop changes to the standard products and test them in conditions that duplicate the actual or projected network and server performance. The Studio Modeling Platform is required in order to tailor and configure the production system.

Users of Studio Modeling Platform must be valid licensees of the products that are configured and tested. However, a user's production license can also be used with the system installed for Studio Modeling Platform. The Studio Modeling Platform includes the following primary tools:

- MQL - the command line interface tool for executing commands and scripts.
- System Administration - the tool for managing and configuring data and file storage vaults.
- Business Modeler - the tool for managing the data model and its associated business processes including types, attributes, relationships, and the web user interface design.
- Matrix Navigator - the tool for searching the data objects based on the data model. This application is required to administer triggers and other configuration objects.While the Matrix Navigator is installed as part of Studio Modeling Platform for ease of installation, this application license is included in both CPF and DTE. For legacy customers that are still based on thick client deployments vs. Web, the CPF license entitles you to use Matrix Navigator.

- *Studio Federation Toolkit* (ADT), which provides documentation and examples for writing custom programs that use the Adaplet libraries available in Live Collaboration. The Studio Federation Toolkit must be licensed for each Studio Modeling Platform environment that requires development of custom connectors with the Adaplet APIs.
- *File Collaboration Server* (FCS), which enables administrators to distribute file data across the enterprise for optimal upload and download performance. A system license is required for each physical site that requires remote file storage. There is no limit to the number of users that can use the File Collaboration Server at a given licensed site, however, all of these users must be licensed to use Live Collaboration, which is the prerequisite.

Overview of Business Process Services

Business Process Services (BPS) is the foundation for all other apps. It includes the schema for all products, as well as the programs and JavaServer Pages needed to construct the user interface shared by all the applications. It can also be used as the basis for creating your own applications.

BPS (and the other components and prerequisites) must be installed before you can install any other apps. Refer to the *Business Process Services Installation Guide* for details.

Business Process Services Components

Application Components

Each Dassault Systemes product contains the items listed in this table.

Application Components	
Item	**For information, refer to:**
Web pages used by the application's users	The user guide that accompanies the application.
Programs specific to the application	To configure programs and for descriptions of utility trigger programs, see *Business Process Services Administration Guide : Triggers*. For application-specific trigger programs, see the Administration Guide that accompanies the application. For information on how to call the included JavaBeans in your custom applications, see the Javadocs located at: ENOVIA_INSTALL\Apps\APP_NAME\VERSION\Doc\javadoc. Refer to *Directories for shared and common components* for other details.
Other administrative objects specific to the application, such as formats	The Administration Guide for the application.
Business objects that accomplish system-related tasks, such as objects for automatically-naming objects and for executing trigger programs	For general information on how the objects function and how to configure them, see the *Business Process Services Administration Guide* : *Configuring Business Process Services Functions*. For a list of the objects included in the application, see the Administration Guide that accompanies the application.

Directories for shared and common components

Some applications, such as Materials Compliance Central, install other components that may be shared between applications. When this is the case there are 2 directories installed under ENOVIA_INSTALL\Apps, one which includes "base" in the name, such as MaterialsComplianceBase.

Use of General Client Applications

Some of the instructions in this and other Administration Guides require the use of a general Matrix client navigator. It is important to restrict the use of these general navigator applications to only a few specially-trained business administrators.

The general client navigator is the desktop version of Matrix Navigator (also known as the thick client), including MQL, Matrix, and Business Modeler.

and to only the purposes described in the *Business Process Services - AEF User Guide* and app Administration Guides. Apps run JavaBean code that requires data to have specific characteristics and conditions. For example, objects may have to have certain relationships defined, have specific values entered for attributes, be in specific lifecycle states, or be in particular vaults. When a person works within the app user interface, these data conditions are met. However, the general Matrix navigators are not necessarily aware of these conditions and therefore a person working within the general navigators can easily compromise data integrity.

Another reason to restrict access to the general clients is that certain actions have different results depending on where the action is taken. A command on a JSP page may include options (such as additional MQL clauses) to ensure that the operation is completed as the application expects, but a user in a general client has no guidance on what options should be chosen. For example, when a file is checked into Live Collaboration using a general client, the store set in the policy is used; when using an app to check in a file, the person or company default store is used regardless of the store set by the policy.

The general navigators must or can be used in situations such as:

- App features require data that cannot be created within the user interface.

  For example, some user profile information and template information must be created in a general navigator.
- Automated business rules and processes need to be configured, such as triggers and autonamers.
- Data needs to be investigated for troubleshooting, testing, or data conversion.

The general navigators should only be used in these situations, using the instructions provided in documentation, and only by specially-trained business administrators. Standard users of apps should never be allowed to work with their data in a general navigator and external customers should never be given access to a general navigator.

Related Documentation

This section lists the documentation available for BPS and 3DEXPERIENCE apps.

BPS Documentation

BPS installs with this documentation:

- *Business Process Services Administration Guide*

  This guide is available in HTML format. It is for people in the host company who need to configure and customize apps. It describes the schema that underlies the applications and how to configure it.
- *Business Process Services - Application Exchange Framework User Guide* and online help

  This guide is available in HTML format and as a context-sensitive online help system. It describes how to use features installed with the Application Exchange Framework portion of BPS, such as history pages and pages accessed from the global toolbar. It also explains how to navigate through the user interface, such as how to use table pages and the Context Navigator. You can access this help system by clicking the help button on any framework-specific application page or clicking AEF Help at the top of any application help page.
- *Business Process Services - Common User Guide* and online help

  This guide is available in HTML format and as a context-sensitive online help system. It describes how to use features installed with the common components portion of BPS, such as the common document model. You can access this help system by clicking the help button on any common-specific application page or clicking Common Components Help at the top of any application help page.
- *Business Process Services - Team User Guide* and online help

  This guide is available in HTML format and as a context-sensitive online help system. It describes how to use features installed with the Team portion of BPS, such as routes. You can access this help system by clicking the help button on any team-specific application page.
- *Business Process Services - Metrics User Guide* and online help

  This guide is available in HTML format and as a context-sensitive online help system. It describes how to use features installed with the business metrics portion of BPS. You can access this help system by clicking the help button on any metrics-specific application page.
- JavaDoc for BPS

  For descriptions of methods in framework packages and classes, see `ENOVIA_INSTALL\Apps\Framework\VERSION\Doc\javadoc`.

App Product Documentation

All apps install with this documentation:

- User Guide and online help for each app

You can access online help for an app by clicking the Help (?) tool on the toolbar of every page. These guides are for the people who will log in and use any part of the application, including Administrators who use the profile management portions of an app to manage person and company profiles.

- Administrator Guide for each app

  Each app has a separate guide for company administrators who work with 3DEXPERIENCE apps. These are the same people who will use BPS and the Studio Modeling Platform to configure an app. The Administration Guide for each app contains information that is unique to the app and therefore not appropriate for the *Business Process Services Administration Guide*. The Administration Guides are provided in PDF format.

Related Documentation Not Installed with BPS or Applications

A Program Directory is available for each version of BPS and the 3DEXPERIENCE platform apps. Each version comes with media that includes the program directory for that release. The program directory is a website that organizes all the release information for all Dassault Systèmes products for a given release. It contains information about prerequisites, installation, licensing, product enhancements, general issues, open issues, documentation addenda, and closed issues.

For instructions on installing BPS and applications, see the *Business Process Services Installation Guide*.

How to Use this Guide

This guide explains the concepts and reference material you need to setup the Live Collaboration database and also to write applications for it. This guide is organized in conceptual chapters and provides complete syntax and reference information in the latter sections:

- Chapter 1, *The MQL Language* - Introduces the MQL Language, syntax conventions, scripts, input modes, commenting, explains how to build an initial database and edit an existing one.
- Chapter 2, *Building the System Basics* - Describes how to use vaults, stores, sites, locations. It provides a progression of setting up your platform infrastructure: adding a vault, adding a store, etc.
- Chapter 4, *Working with Metadata* - Provides information about how to define meta data. It describes attributes, dimensions, types, interfaces, formats, relationships, and policies.
- Chapter 5, *Manipulating Data* - Explains how to use business objects to manipulate data.
- Chapter 6, *Working with Workspace Objects* - Explains how to use workspace objects to manipulate data. It provides information on queries, sets, filters, and cues.
- Chapter 7, *Extending an Application* - Explains how to use other features in the system to extend an application such as programs, and applications.
- Chapter 8, *Modeling Web Elements*- Provides information about commands, menus, tables, portals, etc.
- Chapter 9, *Maintenance* - Provides information about how to maintain the system. There are sections on monitoring clients, system-wide settings, database constraints, diagnostics, etc.
- Chapter 10, *Working with Import and Export* - Discusses concepts about exporting and importing administrative and business objects.
- Chapter 10, *Working with Workflows* - Explains how to use and maintain workflows.
- *Reference Commands*

Published examples in this document, including but not limited to scripts, programs, and related items, are intended to provide some assistance to customers by example. They are for demonstration purposes only. It does not imply an obligation for Dassault Systemes to provide examples for every published platform, or for every potential permutation of platforms/products/versions/etc.

What To Do Next

Start by understanding the basic concepts explained in the chapters of this guide and then use the reference as a tool when programming in MQL. The reference section will provide complete syntax and description of all the parameters, clauses, and grammar of MQL commands.

You can also use MQL within Java or .Net programs when using the MQL Command package of the Studio Customization Toolkit. Before you begin programming for Live Collaboration, you should review the following information:

- *Live Collaboration Installation Guide*

  The Live Collaboration Server must be installed before installing Studio Customization Toolkit. Information on installing the server is included in the this guide.

Always refer to the current Program Directory for any changes that have been made since the publication of this manual.

1 The MQL Language

This chapter introduces the MQL command language in the following sections:

- *About MQL*
- *Accessing MQL*
- *Using Commands and Scripts*
- *Building an Initial Database*
- *Modifying an Existing Database*
- *Working with Implicit and Explicit Transactions*
- *Working With Threads*
- *Using Tcl*

About MQL

The Matrix Query Language (MQL) is similar to SQL. MQL consists of a set of commands that help the administrator set up, build, and test an Live Collaboration database quickly and efficiently.

You can also use MQL to add information to the existing database, and extract information.

MQL acts an interpreter for Live Collaboration and can be used in one of three modes:

1. Interactive Mode - which means executing each command from the command line.This mode is typically used when you have only a few modifications to make or tests to perform.
2. Script Mode - which means using scripts to run commands. This lets you combine commands and also have a repeatable history of commands. You should use MQL scripts as long as you are in the building process. Later, you may decide to add information and files into Live Collaboration. When adding information you may require additional MQL commands. Rather than entering them interactively, you can create a new script to handle the new modifications.
3. Tool Command Language (Tcl) Mode - which means you can use the tcl/tk scripting language. With Tcl embedded in MQL, common programming features such as variables, flow control, condition testing, and procedures are available.

After the database is built, you will most likely maintain it through the Business Modeler interface which helps you see what you are changing.

Using Interactive Mode

When you use MQL in interactive mode, you type in one MQL command at a time. As each command is entered, it is processed. This is similar to entering system commands at your terminal.

MQL is not intended to be an end-user presentation/viewing tool, and as a consequence there are cases where some non-ASCII character sets (eg.some Japanese characters) will have characters that do not display properly, such as when a history record is printed to the console in interactive mode. This is due to low-level handling of the byte code when attempting to display. However, the data is intact when retrieved in any programmatic way, such as:

- Retrieving data into a tcl variable: set var sOut [mql print bus T N R select history]
- Retrieving data from a java program via ADK calls.
- Writing data into a file: print bus T N R select history output d:/temp/TNR_History.txt;

Interactive mode is useful if you have only a few commands to enter or want more freedom and flexibility when entering commands. However, interactive mode is very inefficient if you are building large databases or want to input large amounts of information. For this reason, you also have the ability to use MQL in a script mode.

Command Line Editing

Command line editing is available in interactive MQL. MQL maintains a history list of commands and allows you to edit these commands using the arrow keys and some control characters.

To edit, use the arrow keys and move your cursor to the point where a change is required. Insert and delete characters and/or words, as needed.

Using Control Characters

All editing commands are control characters which are entered by holding the Ctrl or Esc key while typing another key, as listed in the following table. All editing commands operate from any place on the line, not just at the beginning of the line.

Command Line Editing Control Characters

Control Character	Description
Ctrl-A	Moves the cursor to the beginning of the line.
Ctrl-B	Moves the cursor to the left (back) one column.
Esc-B	Moves the cursor back one word.
Ctrl-D	Deletes the character to the right of the cursor.
Ctrl-E	Moves the cursor to the end of the line.
Ctrl-F	Moves the cursor right (forward) one column.
Esc-F	Moves the cursor forward one word.
Ctrl-H	Deletes the character to the left of the cursor.
Ctrl-I	Jumps to the next tab stop.
Ctrl-J	Returns the current line.
Ctrl-K	Kills from the cursor to the end of the line (see Ctrl-Y).
Ctrl-L	Redisplays the current line.
Ctrl-M	Returns the current line.
Ctrl-N	Fetches the next line from the history list.
Ctrl-O	Toggles the overwrite/insert mode, initially in insert mode.
Ctrl-P	Fetches the previous line from the history list.
Ctrl-R	Begins a reverse incremental search through the history list. Each printing character typed adds to the search substring (which is empty initially). MQL finds and displays the first matching location. Typing Ctrl-R again marks the current starting location and begins a new search for the current substring.
Type Ctrl-H	Or press the Del key to delete the last character from the search string. MQL restarts the search from the last starting location. Repeated Ctrl-H or Del characters, therefore, unwind the search to the match nearest to the point where you last typed Ctrl-R or Ctrl-S (described below).Type Esc or any other editing character to accept the current match and terminate the search.
Type Ctrl-H	Or press the Del key until the search string is empty to reset the start of the search to the beginning of the history list. Type Esc or any other editing character to accept the current match and terminate the search.
Ctrl-S	Begins a forward incremental search through the history list. The behavior is like Ctrl-R but in the opposite direction through the history list.
Ctrl-T	Transposes the current and previous character.
Ctrl-U	Kills the entire line (see Ctrl-Y).

Control Character	Description
Ctrl-Y	Yanks the previously killed text back at the current location.
Backspace	Deletes the character left of the cursor.
Del	Deletes the character right of the cursor.
Return	Returns the current line.
Tab	Jumps to the next tab stop.

The MQL Prompts

Interactive MQL has two prompts. The primary prompt is, by default:

```
MQL<%d>
```

where %d is replaced with the command number. The secondary prompt is, by default:

```
>
```

The secondary prompt is used when a new line has been entered without terminating the command.You can change the primary and secondary prompts with the MQL command:

```
prompt [[PROMPT_1] [PROMPT_2]]
```

Without arguments, the prompts reset to the defaults.

Using Script Mode

When working in the script (or batch) mode, you use a text editor to build a set of MQL commands contained in an external file (a script), that can be sent to the command interface. The interface then reads the script, line by line, and processes the commands just as it would in the interactive mode.

Working in script mode has many advantages, particularly when you are first building a database, such as:

- It gives you a written record of all your definitions and assignments. This enables you to review what you have done and make changes easily while you are testing the database. When you are first building the database, you may experiment with different definitions to see which one works best for your application. A written record saves time and aggravation since you do not have to print definitions when there is a question.
- You can use text editor features to duplicate and modify similar definitions and assignments. Rather than entering every command, you can copy and modify only the values that must change. This enables you to build large and complicated databases quickly.
- Testing and debugging the database is simplified. MQL databases can be wiped clean so that you can resubmit and reprocess scripts. This means you can maintain large sections of the MQL commands while easily changing other sections. Rather than entering commands to modify the database, you can simply edit the script. If you are dissatisfied with any portion of the database, you can change that portion of the script without eliminating the work you did on other portions. Since the script contains the entire database definition and its assignments, there is no question as to what was or was not entered.

Tcl Overview

Tcl (tool command language) is a universal scripting language that offers a component approach to application development. Its interpreter is a library of C procedures that have been incorporated into MQL.

With Tcl embedded in MQL, common programming features such as variables, flow control, condition testing, and procedures are available for use. The Tk toolkit is also included. Tcl and Tk are widely available and documented. The information in this section is simply an overview of functionality. For more detailed procedures, consult the references listed in the section.

Tcl enhances MQL by adding the following functionality:

- Simple text language with enhanced script capabilities such as condition clauses
- Library package for embedding other application programs
- Simple syntax (similar to sh, C, and Lisp):
  Here are some examples of using tcl

Command	Output
set a 47	47

Substitutions:

Command	Output
set b $a	47
set b [expr $a+10]	57

Quoting:

Command	Output
set b "a is $a"	a is 47
set b {[expr $a+10]}	[expr $a+10]

- Variables, associative arrays, and lists
- C-like expressions
- Conditions, looping

```
if "$x<<3" {
   puts "x is too small"
}
```

- Procedures
- Access to files, subprocesses
- Exception trapping

MQL/Tcl offers the following benefits to Live Collaboration users:

- Rapid development—Compared with toolkits where you program in C, there is less to learn and less code to write. In addition, Tcl is an interpreted language with which you can generate and execute new scripts on the fly without recompiling or restarting the application. This enables you to test and fix problems rapidly.
- Platform independence—Tcl was designed to work in a heterogeneous environment. This means that a Tcl script developed under Windows can be executed on a UNIX platform. After you complete design and testing, a program object can be created using the code providing users with immediate access to the new application. This also solves the problem of a need to distribute new applications.
- Integration support—Because a Tcl application can include many different library packages, each of which provides a set of Tcl commands, an MQL/Tcl script can be used as a communication mechanism to allow different applications to work with Live Collaboration.
- User convenience— MQL commands can be executed while in Tcl mode by preceding the correct MQL syntax with mql.

Tcl is described in detail at the end of this chapter.

Accessing MQL

There are several ways to access the MQL from the operating system.

For example, if you are working on a PC, select the icon.

Or

If you are working on a UNIX platform, enter the MQL command using the following syntax:

```
mql [options] [-] [file...]
```

Brackets [] *indicate optional information. Do not include the brackets when you enter the command.* **Braces** {} *may also appear in a command syntax indicating that the items in the braces may appear one or more times.*

When the system processes the mql command, the command interface starts up in one of two modes: interactive or script. The mode it uses is determined by the presence of the hyphen and file name(s):

Interactive mode is used when the command includes the hyphen and/or does not include a file name(s). For example, to run MQL interactively, enter either:

```
mql -
```

Or:

```
mql
```

In both of these commands, no files are specified so the interpreter starts up in an interactive mode to await input from the assigned input device (usually your terminal).

MQL Syntax	Specifies that...
mql	The MQL interpreter should be invoked.
options	One or more MQL command options
- (the hyphen)	Entries come from standard input which is interactive mode.
file	A script will be used. File is the name of the script(s) to be processed. Live Collaboration processes the script of MQL commands and then returns control to the operating system. For example, `mql TestDefinitions.mql TestObjects.mql` This command invokes the MQL interpreter, processes the MQL commands within the TestDefinitions.mql script, and then processes the commands within the TestObjects.mql script.

MQL Command Options

In addition to the hyphen and file name qualifiers, several MQL command options are available:

mql Command Options	Specifies that MQL…
-b FILENAME	Use the bootfile FILENAME instead of matrix-r.
-c "command;command... "	Use command;command… as the input script. Processes the MQL commands enclosed within the double quotes.
-d	Suppress the MQL window but do not suppress title information.
-install -bootfile BOOTFILE -user DBUSER -password DBPASSOWORD -host CONNECTSTRING -driver DRIVER	Creates a bootfile with the parameters passed. For DB2, the -host parameter is required.
-k	Does not abort on error. Continues on to the next MQL command, if an error is detected in an MQL command. The -k option is ignored for interactive mode.
-q	Set Quote on.
-t	Suppress the printing of the opening title and the MQL window.
-v	Work in verbose mode to print all generated messages.

Each MQL command and command option is discussed in the reference section.

- FILENAME is the name of the file to be created.
- DBUSER is the name used to connect to the database. This is the database user that is created to hold the database. It is established by the database administrator.
- DBPASSWORD is the security password associated with the Username. This is established by the database administrator. The user password is encrypted as well as encoded.
- CONNECTSTRING For Oracle, the connect string is the Oracle “connect identifier”, which can be a net service name, database service name, alias or net service alias. For DB2, the connect string is the database name or alias. For SQL Server, it is the name of the ODBC datasource.
- DRIVER is one of the following depending on which database you use:
  - Oracle/OCI80
  - DB2/CLI
  - MSSQL/ODBC

Using Commands and Scripts

All commands perform an action within Live Collaboration. For example, actions might define new structures within the database, manipulate the existing structures, or insert or modify data associated with the database structures.

An MQL script is an external file that contains MQL commands (see the example below). You can create this file using any text editor. After you insert all the desired MQL commands, you can batch process them by specifying the name of the script file in the mql command.

```
############################################################
# Script file: SampleScript.mql
# These MQL commands define a vault and create an object.
############################################################
verbose on;
#
# Set context to Matrix System Administrator's and define a new vault.
#
set context dba;
add vault "Electrical Parts";
output "Electrical Parts vault has been added"
#
# Set context to that of person Steve in vault "Electrical
Subassemblies"
#
set context vault "Electrical Subassemblies" user Steve;
#
# Add a business object of type Drawing with name C234 A3
# and revision 0 and check in the file drawing.cad
#
output "now adding a business object for Steve";
add businessobject Drawing "C234 A3" 0
   policy Drawing
   description "Drawing of motor for Vehicle V7";
checkin businessobject Drawing "C234 A3" 0 drawing.cad;
#
quit;
```

Entering (Writing) MQL Commands

An MQL command consists of a keyword and, optionally, clauses and values. MQL commands follow their own set of rules and conventions, as in any other programming or system language. In the following sections, you will learn about the conventions used in this book for entering, as well as reviewing, displayed MQL commands.

You must follow a fixed set of syntax rules so that Live Collaboration can interpret the command correctly. If you do not, the system may simply wait until the required information is provided or display an error message.

When in interactive mode, each command is prompted by:

```
mql<#>
```

The syntax rules are summarized in the following table. If you have a question about the interpretation of a command, see the reference section of this guide.

Writing and Entering MQL Commands	
Commands	Consist of words each separated by one or more spaces, tabs, or a single carriage return.
	Begin with a keyword. Most keywords can be abbreviated to three characters (or the least number that will make them unique).
	All NAMES, VALUES, etc. must be enclosed within single or double quotes when they have embedded spaces, tabs, newlines, commas, semi-colons, or pound signs.
	End with either a semicolon (`;`) or double carriage return.
	For example, this is the simplest MQL command. It contains one keyword and ends with a semicolon: `quit;`
Clauses	May or may not be necessary.
	Begin with a keyword.
	Are separated with a space, tab, or a carriage return.

	The following example is a command with two clauses separated by both carriage returns and indented spaces.
	`add attribute "Ship's Size"`
	`  description "Ship size in metric tons"`
	`  type integer;`

Values	Are separated within clauses by a comma (`,`). Values may be single or, if the keyword accepts a list of values, they can be specified either separately or in a list. For example:
	`attribute Size, Length`
	Or:
	`attribute Size`
	`attribute Length`

Comments	Begin with a pound sign (#) and end with a carriage return. For example: `# list users defined within the database`
	Comments are ignored by Live Collaboration and are used for your information only.

MQL Command Conventions

When commands are displayed on your screen, the commands are displayed in diagrams which obey a set of syntax rules. These rules are summarized in the following table:

Reviewing Displayed MQL Commands	
keywords	All keywords are shown in lowercase. These words are required by MQL to correctly process the command or clause. Though they are displayed in all lowercase in the syntax diagram, you can enter them as a mixture of uppercase and lowercase characters.
	All keywords can be abbreviated providing they remain distinctive. For example, you can use `bus` for `businessobject`.

Values	User-defined values are shown in uppercase. The words in the syntax diagram identify the type of value expected. When entering a value, it must obey these rules:
	1. You can enter values in uppercase or lowercase. However, values are case-sensitive. Once you enter the case, later references must match the case you used.
	`For example, if you defined the value as Size, the following values would` *not* `match: SIZE, size, sIZE.`
	2. You must enclose a value in double (`" "`) or single (`' '`) quotes if it includes spaces, tabs, carriage returns (new lines), commas, semicolons (`;`), or pound signs (`#`).
	A space is included in the following examples, so the values are enclosed in quotes:
	`"Project Size"`
	`'Contract Terms'`
	3. If you need to use an apostrophe within a value, enclose the value in quotes. For example:
	`"Project's Size"`
	4. If you need to use quotes within a value, enclose the value in single quotes. For example:
	`'Contract "A" Terms'`

`[option]`	Optional items are contained within square brackets `[ ]`. You are not required to enter the clause or value if you do not need it.
	Do not include the brackets when you are entering an option.
`{option}`	All items contained within braces `{ }` can appear zero or more times.
	Do not include the braces when you are entering an option.

`|option1|`
`|option2|`
`|option3|`
	All items shown stacked between vertical lines are options of which you must choose one.

Important MQL Commands

When you first use MQL, there are two important MQL commands you should know: Quit and Help.

Quit - exits the current MQL session and returns control to the operating system. To terminate your MQL session, simply enter:

```
quit;
```

When writing the code for program objects, you may want the program to return an integer value to the system that ran the program. For this reason, the quit command can be followed by an integer, as follows::

```
quit [INTEGER];
```

- The INTEGER value is passed and interpreted as the return code.

Help -is available for various MQL command categories. If you do not know which category you want or you want to get a listing of the entire contents of the help file, enter:

```
help all;
```

Eventually, you will probably want help only on a selected MQL category:

application	association	attribute
businessobject	channel	command
config	connection	context
cue	dimension	download
error	export	filter
form	format	group
import	index	inquiry
license	location	menu
monitor	person	policy
portal	program	property
query	relationship	role
rule	set	site
store	table	thread
transaction	type	upload
user	validate	vault

To get help on any of these categories, enter::

```
help MQL_CATEGORY;
```

- help is the keyword.
- MQL_CATEGORY is one of the categories listed above.

If you have a question about the interpretation of a command, see the reference page on that command.

When you enter this command, Live Collaboration displays a list of all the commands associated with the category along with the valid syntax. For example, if you enter the following command:

```
help context;
```

It will show the following syntax diagram:

```
set context [ITEM {ITEM}];
 where ITEM is:
    | person PERSON_NAME |
    | password [PASSWORD_VALUE] |
    | vault [VAULT_NAME] |
print context;
```

This information indicates that there are two commands associated with the context category: the set context and print context commands.

The first command has three clauses associated with it: the person, password, and vault clauses. The square brackets ([]) on either side of PASSWORD_VALUE mean that it is optional. If a clause is used, you must obey the syntax of the clause. In the person clause, the syntax indicates that a value for PERSON_NAME is required. Words shown in all uppercase indicate user-defined values. (Refer to Entering (Writing) MQL Commands for a complete description of syntax rules.)

Building an MQL Script

Scripts are useful when you are performing many changes to the Live Collaboration database. This is certainly true in the initial building process and it can be true when you are adding a number of old files or blocks of users into an existing database. MQL scripts provide a written record that you can review. Using a text editor makes it easy to duplicate similar blocks of definitions or modifications.

Running Scripts and Using Basic MQL Commands

When building a script, there are several MQL commands that help run other scripts and let you monitor the processing MQL commands.

MQL Commands to Use When Building a Script	
`output VALUE;`	The Output command enables you to print a message to your output device. This is useful within scripts to provide update messages and values while the script is processed. For example, if you are processing large blocks of commands, you may want to precede or end each block with an Output command. This enables you to monitor the progress of the script.
`password PASSWORD;`	The Password command enables you to change the current context password. The Password command enables you to change your own password (which can also be done with the Context command).
`run FILE_NAME [continue];`	The Run command enables you to run a script file. This is useful if you are working in interactive mode and want to run a script. The `continue` keyword allows the script to run without stopping when an error occurs. This is essentially the same as running MQL with the -k option, but it is available at run time, making it usable by programs.
`shell COMMAND;`	The Shell command enables you to execute a command in the operating system. For example, you can perform an action such as automatically sending an announcement to a manager after a set of changes are made. The Output command (see above) sends a message to your output device; but, it cannot send a message to a different device. You can do so with the Shell command.
`verbose [on|off];`	The Verbose command enables you to increase or decrease the amount of message detail generated by the MQL interpreter as the script is processed. This is similar to the -v Option. See *mql Command* in the programming reference section of this guide. When set to `ON`, more detail is provided. When set to `OFF`, only errors and important system messages are displayed.
`version;`	The Version command enables you to see which version of MQL you are using. For example, this is useful if you want a record of the version used to process a script.

Using Comments

Comments visually divide scripts into manageable sections and remind you of why structures and objects were defined and modified. Comments, which are preceded in the script by a pound sign (#), are ignored by MQL:

```
# script comments
```

Each comment line must begin with a pound sign (#).

Parameterized MQL Commands

Using parameterized MQL prevents MQL injection flaws in apps. An MQL injection atack consists of insertion or "injection" of an MQL command via the input data from the client to the application. A successful MQL injection exploit can read sensitive data from the database, modify database data (e.g., through Insert/Update/Delete commands), execute administration operations on the system (e.g., Clear Vault command), and in some cases, issue commands to the operating system.

The following Java example is unsafe because it allows an attacker to inject code into the command that the system would execute:

```
String command = "temp query bus Part * * where owner == " +
    request.getParameter("customerName");
mql.executeCommand(context, command);
```

If `"customerName"` is equal to `"; clear all"`, then the `"temp query"` will fail because the command is incomplete, but the `"clear all"` command will execute as a valid command.

MQL injection flaws are introduced when software developers create dynamic commands that include user-supplied input. Avoiding MQL injection flaws is straightforward. Developers should either stop writing dynamic commands, or prevent user-supplied input that contains malicious MQL from affecting the logic of the executed command.

In support of this, the Open Web Application Security Project (OWASP) recommends that software developers apply the following defenses to prevent MQL injection attacks:

- Use prepared statements (parameterized commands).
- Use stored procedures.
- Escape all user-supplied input.
- Enforce "least privilege" (e.g., by minimizing the privilege of user accounts). See *Least Privilege*, below.
- Perform "white list input validation" (e.g., by providing a list of the available inputs, such as attribute ranges). See *White List Input Validation*, below.

The Server already provided capabilities to implement stored procedures (Java Program Objects or JPOs), enforce "least privilege" (P&O security) and to perform "white list input validation" (attribute ranges). Additionally, web application development teams use APIs that allow them to encode and escape data.

Release V6R2013 adds the ability to use parameterized MQL commands. Parameterized MQL commands allow the developer first to define the MQL command and then to pass in each parameter to the command at a later time. This coding style makes it possible to distinguish between code and data, regardless of what user data is supplied.

Escaping of user-supplied input is unnecessary in MQL. In other words, for code that read:

```
"temp query bus '"+type+"' Part * select id"
```

where `"type"` was bracketed by two single quotes, now rather than writing:

```
"temp query bus '$1' Part * select id"
```

you can simply write:

```
"temp query bus $1 Part * select id"
```

Parameterized commands ensure that an attacker is not able to change the intent of a query, even if the MQL commands are inserted by the attacker. In the following safe example, if an attacker were

to enter a userID of `tom' or '1'='1`, the parameterized query would not be vulnerable and would instead loook for a username that literally matches the entire string `tom' or '1'='1`.

```
String customerName = request.getParameter("customerName");
String whereClause = "'owner == " + customerName + "'";
String command = "temp query bus $1 $2 $3 where $4";
mql.executeCommand(context, command, "Part", "*", "*",
  whereClause);
```

For details on how to implement parameterized MQL commands, see the Javadoc online help for the matrix::db::MQLCommand::executeCommand() method.

Building an Initial Database

Building a database usually involves writing two scripts. One script will contain all the definitions used by the database. The second script creates business objects, associates files with the objects, and defines relationships between objects.

By separating these two steps, it is easier to test the database. You cannot create business objects or manipulate them until the definitions are in place. By first creating and testing all the definitions, you can see if the database is ready to add objects. You may want to alter the definitions based on test objects and then use the altered definitions for processing the bulk of the object-related data.

Clearing the Database

When you are creating an initial database and you want to start over, the Clear All command enables you to work with a clean slate. The Clear All command clears all existing definitions, vaults, and objects from the database. It wipes out the entire content of the database and leaves only the shell. While this is useful in building new databases, it should NOT be used on an existing database that is in use.

Once this command is processed, the database is destroyed and can only be restored from backups.

Only the person named “creator” can use the Clear All command.

```
clear all;
```

Do not attempt to add the persons creator, guest, or Test Everything using MQL. Adding or modifying these objects could cause triggers, programs or other application functions not to work.

The clear all command should NEVER be used while users are on-line.

Clearing Vaults

To test and build new vaults, use the Clear Vault command. This command ensures that the vault does not contain any business objects. It can be run by a person with System Administrator privileges.

```
clear vault VAULT_NAME;
```

- VAULT_NAME is the names of the vault(s) to be cleared.

When the Clear Vault command is processed, all business objects are cleared within the named vaults. The definitions associated with the vault and the database remain intact. Only the business objects within the named vaults are affected.

The clear vault should NEVER be used while users are on-line.

Creating Definitions in a Specific Order

When creating the script of definitions for the initial database, the order in which you make the definitions is significant. Some commands are dependent on previous definitions. For example, you cannot assign a person to a group if that group does not already exist. For this reason, it is recommended that you write your definition commands in the following order

:

Definition Order		
1	Roles Groups Persons Associations *Or:* Persons Groups Roles Associations	You can define roles, groups, and persons either way depending on your application. Since associations are combinations of groups and roles, they must be created after groups and roles are defined.
2	Attributes Types Relationships Formats Stores Policies	Order is important for this set of definitions. For example, types use attributes, so attributes must be defined before types.
3	Vaults	Vaults can be defined at any time although they are most commonly defined last.

Processing the Initial Script

After your script is written, you can process it using the mql command, described in *Accessing MQL*. As it is processed, watch for error messages and make note of any errors that need correction.

Once the script that creates the definitions is successfully processed, you can use either the interactive MQL interpreter or the Business Administrator account to check the definitions. From Matrix Navigator or MQL, you can create objects, associate files, and otherwise try out the definitions you have created. If you are satisfied with the definitions, you are ready to write your second MQL script.

Writing the Second MQL Script

The second MQL script in this example creates business objects, manipulates them, and defines relationships. For more information on the types of information commonly found in this second script file, see *Manipulating Data* in Chapter 5.

Modifying an Existing Database

After you define the initial database, you may want to modify it to add additional users and business objects. If the number of users or the amount of information is large, you should consider writing a MQL script. This enables you to use a text editor for editing convenience and provides a written record of all of the MQL commands that were processed.

Often when you are adding new business objects, you will define new vaults to contain them. This modular approach makes it easier to build and test. While some new definitions may be required, it is possible that you will use many of the existing definitions. This usually means that the bulk of the modification script will involve defining objects, manipulating them, and assigning relationships. These actions are done within the context of the vault in which the objects are placed.

To test and build new vaults, use the Clear Vault command. This command ensures that the vault does not contain any business objects. See *Clearing the Database* for details.

When writing your script for modifying an existing database, remember to include comments, occasional output commands, and a Quit command at the end (unless you'll want to use the interactive MQL interpreter after script completion).

Working with Implicit and Explicit Transactions

A *transaction* involves accessing the database or producing a change in the database. All MQL commands involve transactions.

Implicit Transactions

When you enter an MQL command, the transaction is started and, if the command is valid, the transaction is *committed* (completed). For example, assume you enter the following MQL command:

```
add person Debbie;
```

As soon as you enter the command, the system starts the transaction to define a person named Debbie. If that person is already defined, the command is invalid and the transaction aborts—the command is not processed and the database remains unchanged. If the person is not already defined, the app is committed to (the completion of) the transaction to add a new person. Once a transaction is committed, the command is fully processed and it cannot be undone. In this case, Debbie would be added to the database.

Ordinarily, starting, committing, and aborting transactions is handled *implicitly* with every MQL command. Each command has two implied boundaries: the starting keyword at the beginning of the command and the semicolon or double carriage return at the end. When you enter an MQL command, a transaction is implicitly started at the keyword and is either committed or aborted depending on whether the content of the command is valid.

This implicit transaction control can also be performed explicitly.

Explicit Transaction Control

Several MQL commands enable you to explicitly start, abort, and commit transactions:

`abort transaction [NAME];`
`commit transaction;`
`print transaction;`
`start transaction [read];`
`set transaction wait|nowait|savepoint [NAME];`

These commands enable you to extend the transaction boundaries to include more than one MQL command.

- If you are starting a transaction, use the `read` option if you are only reading.
- Without any argument, the `start transaction` command allows reading and modifying the database.

Extending Transaction Boundaries to Include Several Commands

Including several MQL commands within a single set of transaction boundaries enables you to tie the success of one command to the success of some or all of the other MQL commands.

It will appear that all valid MQL commands are performed; but, they are not permanent until they are committed. You should not quit until you terminate the transaction or you will lose all your changes. Also, the longer you wait before committing changes, the more likely you are to encounter

an error message, particularly if you are entering the commands interactively when typing errors are common.

Let's look at an example in which you want to create a set of business objects and then associate a collection of files with those objects. You might successfully create the objects and then discover that you cannot place the files in them. With normal script or interactive MQL processing, the objects are created even though the checkin fails. By altering the transaction boundaries, you can tie the successful processing of the files to the creation of the business objects to contain them. For example:

```
start transaction update;
add businessobject Drawing "C234 A3" 0
   policy Drawing
   description "Drawing of motor for Vehicle V7";
checkin businessobject Drawing "C234 A3" 0 V7-234.MTR;
add businessobject Drawing "C234a A3" 0
   policy Drawing
   description "Drawing of alt. motor for Vehicle V7";
checkin businessobject Drawing "C234a A3" 0 V7-234a.MTR;
commit transaction;
```

This transaction is started and the current state of the database is saved. The `add businessobject` commands create business objects, and the `checkin businessobject` commands add the files.

When the `commit transaction` command is processed, MQL examines all the commands that it processed since the `start transaction` command. If no error messages are detected, all changes made by the commands are permanently made to the database.

If ANY errors are detected, NONE of the changes are committed. Instead, the system returns the database to the recorded state it was in at the start of the transaction. This essentially gives you an *all or nothing* situation in the processing of MQL commands. However, savepoints can be set in the transaction to be used in conjunction with the `abort transaction` command, which can save some of the work already done.

The `set transaction savepoint [NAME]` command allows you to set a point within a transaction that will be used for any required rollbacks. If you specify the `NAME` in an `abort transaction` command, the transaction is rolled back to the state where the savepoint was created. If no name is specified, the entire transaction is rolled back. Of course the transaction must still be committed before the commands between the `start transaction` and the savepoint are processed. The `set transaction savepoint NAME` command may be issued multiple times without error. The effect is that the savepoint is changed from the original savepoint state in the transaction to the current state of the transaction. The `abort transaction NAME` command may also be issued multiple times.

If `set transaction savepoint NAME` is issued and the transaction has already been marked as aborting, the command will fail and return an error. When using this command via Studio Customization Toolkit or Tcl, be sure to check for the returned error "Warning: #1500023: Transaction aborted". If you assume `set transaction savepoint NAME` works, a subsequent `abort transaction NAME` may roll the transaction back to an older state than expected.

Nested savepoints cannot be used in MQL transactions on DB2. An attempt to nest savepoints will result in a DB2 error stating that nested savepoints are unsupported.

The `wait` and `nowait` arguments are used to tell the system if you want to queue up for locked objects or not. When `nowait` is used, an error is generated if a requested object is in use. The default is to wait; the `wait` keyword is a toggle.

The Oracle SQL `nowait` function is used by Oracle only.

One advantage to using the transaction commands to process command blocks involves the processing time. Commands processed within extended transaction boundaries are processed more quickly than the normal command transactions.

If you choose to process blocks of MQL commands within explicit transaction boundaries, you need to carefully consider the size and scope of the commands within that transaction. When you access anything with an MQL command, that resource is locked until the transaction is completed. Therefore, the more commands you include within the transaction boundaries, the more resources that will be locked. Since a locked resource cannot be accessed by any other user, this can create problems if the resource is locked at a time when heavy use is normal. The use of large transactions may also require you to adjust the size of your Oracle rollback segments. See your Oracle documentation for more details.

Be careful that the blocks are not too large.

You should immediately attend to an explicit transaction that either has errored or is awaiting a `commit` or `abort` command. Many database resources other than objects are also locked with the pending transactions. Users will begin to experience lock timeouts as they attempt typical database operations.

Access Changes Within Transactions

A variety of information about access rights are cached during a transaction within the 3DExperience platform. This information includes not only the security model, but also other parts of the administration model, such as which attributes belong to which types. These caches are not reset until the transaction is committed, which guarantees the integrity of all updates within a transaction.

Changes to the metamodel - that is, any part of the administration model that influences the behavior of the system - must be done within their own transactions. The metamodel includes all administrative types, but can also include data on business objects or connections that control behavior, such as when an attribute value is referenced in an access rule. This type of data is cached as part of the metamodel, and therefore if you make changes to them within a longer transaction, you may get unexpected results.

Specifically, if the current context gains access to an object within a transaction due to an access expression that evaluates to TRUE based on a current attribute value, then that access is granted for the entire transaction, even if the attribute value is changed. The same could be said about changing the user's group or role assignments. With regard to group and role assignments, if you have access at the beginning of a transaction, you will have it at the end. That is, once a transaction is begun, the system does not check back into the database to detect changes to person assignments that may affect the current user's access within the person's cache.

Consider the example below:

```
set context user creator;
add group GRP1;
add person PERS1 type application,full;
add type ACCESS attribute int-u;
add policy NONE type ACCESS state one public none owner all user GRP1
read,show,modify;
add bus ACCESS a1 0 vault unit1 policy NONE int-u 1;
# First, get the object in PERS1's cache by failing to modify attr:
```

```
start trans;
set context user PERS1;
mod bus ACCESS a1 0 int-u 2;
# Now switch back to creator and modify the group so that PERS1 can
modify
# >>>>> OR make this change from a separate MQL/Business session.
set context user creator;
mod person PERS1 assign group GRP1;
# CASE 1: Try to modify - this is not allowed
set context user PERS1;
mod bus ACCESS a1 0 int-u 2;
commit trans;
# Now switch back to creator and modify object to kick it out of cache
set context user creator;
mod bus ACCESS a1 0 int-u 3;
# Modify the group so that PERS1 can modify
mod person PERS1 assign group GRP1;
# Start a transaction and get the object in PERS1's cache with
successful mod
start trans;
set context user PERS1;
mod bus ACCESS a1 0 int-u 4;

# Now switch back to creator and modify the group so that PERS1 can NOT
modify
# >>>>> OR make this change from a separate MQL/Business session.
set context user creator;
mod person PERS1 remove assign group GRP1;

# CASE: Try to modify - it is allowed.
set context user PERS1;
mod bus ACCESS a1 0 int-u 5;
commit trans;
```

Working With Threads

MQL has a notion of a “thread,” where you can run multiple MQL sessions inside a single process. Each thread is like a separate MQL session, except that all threads run within the same MQL process. This concept was introduced into MQL to support the MSM idea of multiple “deferred commit” windows.

The syntax for the thread command is discussed in *thread Command* in the programming reference section of this guide.

Using Tcl

To enter Tcl mode, enter the following in MQL:

```
tcl;
```

In Tcl mode, the default Tcl prompt is % and MQL will accept only Tcl commands. Native MQL commands are accessed from Tcl by prefixing the command with the keyword mql. For example:

```
% mql print context;
```

Otherwise, Tcl command syntax must be followed. It differs from MQL in several ways:

Tcl Command Syntax Differences	
Comments	If the first nonblank character is #, all the characters up to the next new line are treated as a comment and discarded. The following is not a comment in Tcl: `set b 101; #this is not a comment`
Command Separators	New lines are treated as command separators. They must be preceded by a backslash (\) to not be treated as such. A semi-colon (;) also is a command separator.
Commas	For MQL to properly parse commands with commas, the commas must be separated with blanks. For example: `access read , write , lock`
Environment Variables	Environment variables are accessible through the array, `env`. For example, to access the environment variable `MATRIXHOME`, use: `env(MATRIXHOME)`
History	Tcl has its own history mechanism.

For a description of common Tcl syntax errors, see *Tcl Syntax Troubleshooting*.

Command line editing is not currently available.

The Tcl command, exit, returns you to native MQL. Like the MQL Quit command, Exit can be followed by an integer value which is passed and interpreted as the return code of the program.

Displaying Special Characters

When displaying special symbols in a Tk window, it may be necessary to specify the font to be used. Tcl/Tk 8.0 tries to map any specified fonts to existing fonts based on family, size, weight, etc. Depending on where you are trying to use a special symbol, you may be able to use the -font arg.

MQL and Tcl

When Tcl passes an MQL command to MQL, it first breaks the command into separate arguments. It breaks the command at every space, so you have to be careful to use double quotes ("") or braces ({}) to enclose any clause which is to be passed along to the system as a single argument.

The parsing of Tcl commands containing backslashes within an MQL program object is different than when those same commands are run from MQL command prompt.

For example, create a text file with the command lines:

```
tcl;
set a "one\\two\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\} $a {/} b
puts "now b contains $b"
puts "now b contains $b"
```

Run this file from MQL (or alternatively type each line in MQL command window). The final line of output will be:

```
now b contains one/two/three
```

This is consistent with how the same lines of code behave in native Tcl.

However if you use this same code as a MQL program object, to get the same output you need to use triple rather than double backslashes so your code becomes:

```
tcl;
set a "one\\\two\\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\\} $a {/} b
```

Tcl Syntax Troubleshooting

The Tcl executable has strict rules regarding syntax, which can cause problems when the rules are not followed. Below are some tips for avoiding common syntax-related errors.

1. Extra Spaces

A common error is to include extra spaces in a line of Tcl. You can easily miss excess white space, but the Tcl compiler examines white space closely. This can happen when you join two lines together during editing and something is deleted. An extra space before the carriage return may end up on the line of code. Tcl will catch this error every time, but it won't report anything back to you. A quick way to catch this error is to write the program to a text file and to search for all occurrences of a blank space. The syntax for this is:

```
grep -n "{ $" *.<text_file_name_suffix>
```

The second common error occurs with extra spaces in an attribute name in an MQL select command. For example:

```
set sClass [ mql print businessobject $sOid select
attribute\[Classification \] dump $cDelimiter ]
```

In using the Classification attribute, you may unintentionally leave an extra space before the "\" character. The correct form of this query looks like this:

```
set sClass [ mql print businessobject $sOid select
attribute\[Classification\] dump $cDelimiter ]
```

To check for this type of problem, first write all the programs to text files, then search the files for spaces prior to a backslash. The syntax for searching this is:

```
grep -n "attribute\\\\\[" *.tcl | grep " \\\\\]"
```

2. Matching Braces

Another common error results when the Tcl compiler determines that the number of begin/end braces does not match. The typical cause for this error is commented-out code containing braces, which the Tcl compiler includes in its brace count, leading to unexpected results. For example, the

code sequence below looks like it should run correctly. However, the Tcl compiler will determine that the brace count does not match, causing problems.

```
# foreach sUser $lUserList {
foreach sUser $lEmployeeList {
   puts $sUser
}
```

The correct solution is to either comment-out the entire block of code, or to delete it, so the compiler will generate a correct brace match, for example:

```
# foreach sUser $lUser {
#     puts $sUser
# }

foreach sUser $lEmployeeList {
   puts $sUser
}
```

or

```
foreach sUser $lEmployeeList {
   puts $sUser
}
```

Again, write the program to a text file, search for the braces, and ensure that all braces match. The syntax for retrieving the lines containing braces and storing them in a file is:

```
grep "\#" *.tcl | grep "\{" | cut -d: -f1,1 > filelist.log
```

The file filelist.log will then contain only the lines with braces for analysis.

3. Tcl Versions

To determine what version of Tcl you are using, switch to tcl mode then use the info tclversion or info patchlevel commands:

```
MQL<1>tcl;
% info tclversion
8.3.3
```

Or

```
% info patchlevel
8.3.3
```

The Tk library must be loaded in order to retrieve the Tk version. On Windows, a command similar to the following should be used:

```
% load tk83
% info loaded
{tk83 Tk}
```

For More Information

For more information about Tcl and Tk, refer also to the following books:

Ousterhout, John K., Tcl and the Tk Toolkit. Addison-Wesley, April 1994, ISBN 0-201-63337-X.

Welch, Brent, Practical Programming in Tcl and Tk. Prentice Hall, May 1995, ISBN 0-13-182007-9.

World Wide Web documents for reference:

USENET newsgroup: comp.lang.tcl

Parameterized MQL Commands

Using parameterized MQL prevents MQL injection flaws in apps. An MQL injection atack consists of insertion or "injection" of an MQL command via the input data from the client to the application. A successful MQL injection exploit can read sensitive data from the database, modify database data (e.g., through Insert/Update/Delete commands), execute administration operations on the system (e.g., Clear Vault command), and in some cases, issue commands to the operating system.

The following Java example is unsafe because it allows an attacker to inject code into the command that the system would execute:

```
String command = "temp query bus Part * * where owner == " +
    request.getParameter("customerName");
mql.executeCommand(context, command);
```

If `"customerName"` is equal to `"; clear all"`, then the `"temp query"` will fail because the command is incomplete, but the `"clear all"` command will execute as a valid command.

MQL injection flaws are introduced when software developers create dynamic commands that include user-supplied input. Avoiding MQL injection flaws is straightforward. Developers should either stop writing dynamic commands, or prevent user-supplied input that contains malicious MQL from affecting the logic of the executed command.

In support of this, the Open Web Application Security Project (OWASP) recommends that software developers apply the following defenses to prevent MQL injection attacks:

- Use prepared statements (parameterized commands).
- Use stored procedures.
- Escape all user-supplied input.
- Enforce "least privilege" (e.g., by minimizing the privilege of user accounts). See *Least Privilege*, below.
- Perform "white list input validation" (e.g., by providing a list of the available inputs, such as attribute ranges). See *White List Input Validation*, below.

The Server already provided capabilities to implement stored procedures (Java Program Objects or JPOs), enforce "least privilege" (P&O security) and to perform "white list input validation" (attribute ranges). Additionally, web application development teams use APIs that allow them to encode and escape data. Release V6R2013 adds the ability to use parameterized MQL commands. Parameterized MQL commands allow the developer first to define the MQL command and then to pass in each parameter to the command at a later time. This coding style makes it possible to distinguish between code and data, regardless of what user data is supplied.

Parameterized commands ensure that an attacker is not able to change the intent of a query, even if the MQL commands are inserted by the attacker. In the following safe example, if an attacker were to enter a userID of `tom' or '1'='1`, the parameterized query would not be vulnerable and would instead loook for a username that literally matches the entire string `tom' or '1'='1`.

```
String customerName = request.getParameter("customerName");
String whereClause = "'owner == " + customerName + "'";
String command = "temp query bus $1 $2 $3 where $4";
mql.executeCommand(context, command, "Part", "*", "*",
  whereClause);
```

For details on how to implement parameterized MQL commands, see the Javadoc online help for the matrix::db::MQLCommand::executeCommand() method.

Least Privilege

To minimize the potential damage of a successful MQL injection attack, you can minimize the privileges assigned to every account in the system. Do not assign "business" or "system" access rights to standard application accounts. In many cases, this is easy and everything just works when done this way, but it is very dangerous. It is preferable to start from the ground up to determine what access rights the application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access and only to the data to which they need to access.

The privileges of the operating system account for the Java server for MCS and FCS should be minimized. Do not run the Java server application as `root`! Create an OS account to something more appropriate and with restricted privileges.

The system provides extensive capabilities to manage access to data. This access control can be applied at the group, role, or individual user level. It can also be applied against a class of data types, or as granularly as individual attributes.

White List Input Validation

White list input validation is appropriate for all input fields in which the user enters data. It involves defining exactly what data is authorized; by definition, everything else is not authorized. If the data is well-structured (such as dates, social security numbers, zip codes, email addresses, and the like), then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input. If the input field comes from a fixed set of options, such as a drop-down list or radio buttons, then the input can only match exactly one of the values offered in the first place.

The most difficult fields to validate are free-text fields, such as blog entries. However, even those types of fields can be validated to some degree, for example by excluding all non-printable characters and defining a maximum size for the input field.

The system provides capablities to create attribute ranges and to perform input validation in web user interfaces.

2 Building the System Basics

This chapter explains the infrastruture schema. It will take you through the progression of setting up the infrastructure.

In this section:

Vaults

A *vault* is a storage location residing in the underlying database that is defined by a System Administrator. Vaults allow the designer of the database to construct one or more logical storage locations within the database. The Business Administrator determines what the vault is for, while the System Administrator defines where the vault is located on the network. Vaults should use actual host and path names, not mounted directories. Paths must be exported on the host to all users who require access to the vaults.

You must be a System Administrator to access vaults.

In addition to the business object vaults created by the System Administrator, a vault called the Administration vault that is created automatically when the 3DEXPERIENCE platform is installed on your system. The Administration vault is used for administrative purposes only and serves as the master definition vault. The Administration vault is used for definitions only. You cannot use it for storing business objects.

Vaults contain metadata (information about objects), while stores contain the application files associated with business objects.

All vaults contain a complete set of definitions. These definitions identify the characteristics of items such as persons, roles, types, formats, etc. When you make changes to a definition (such as add, modify, or delete), all definition copies must be updated to reflect the change. This update of the vaults occurs simultaneously if all the copies are available. If any of the copies are not available (a vault is not available), you cannot alter the definitions. This prevents partial alteration of the definitions.

For example, assume you want to add a new format definition. After you enter the Add Format command, the system attempts to add the definition. If the definition is valid (no errors), all copies of the definitions are changed to include this new format. But assume that a vault resides on a host that is currently offline. In this case, no changes to the definitions are made. If changes were allowed, the one vault would not be updated to contain the change. Therefore, you should ensure that all defined vaults are available before modifying the definitions.

Types of Vaults

There are three types of vaults: local, foreign and external.

- Most vaults are local.
- Foreign vaults are used with Adaplets, which allow data from virtually any source to be modeled as objects.
- External (Web Service Adaplet) vaults are used with External stores that contain data maintained by external servers.

When defining a vault in MQL, you don’t need to specify which type it is. The system knows which type of vault you are defining by the parameters you specify for the vault.

- For local vaults, you define Oracle tablespaces using the Tablespace and Indexspace clauses.
- For foreign vaults, you specify tablespaces, and Interface and Map fields (using the Interface and Map clauses).
- External vaults need only a parameters file.

The Vault command is described in the reference section of this guide.

Defining a Vault

Use the Add Vault command to define a vault:

```
add vault NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the vault you are creating.

`ADD_ITEM` is an Add Vault clause that provides more information about the vault you are creating.

The complete syntax and clauses for working with vaults are discussed in *vault Command* in the programming reference section of this guide.

Stores

A *store* is a storage location for checked-in files. All files checked in and used by the system are contained in a file store. These files can contain any information and be associated with any variety of business objects. A file store simply defines a place where you can find the file. You can define file stores for CAD drawings, documentation files, problem reports, and so on.

A store is used to divide the database for improved performance. An object's policy determines the store that is used for its files by default. A file store provides:

- Information on the physical storage location of a file checked in. Stores, like vaults, should be strategically placed within the network topology.
- Access to that information. Remember the store contains more data than typically found within the vault.
- Optional integration with the version control system, DesignSync, that allows the system to handle object metadata while a version control system handles file and folder versioning. Files are accessed via HTTP or HTTPS.

You must be a System Administrator to access stores.

Multiple file stores are possible at different locations. For example, two or more file stores could contain CAD drawings. These stores might be located on the same host or on different hosts. An object's policy determines the store that is used for its files.

A lock feature enables you to lock a store from the user for all write activities. Business objects with a policy using a locked store cannot have files checked in (written), but files can be checked out (read). This is useful when a store becomes obsolete. Stores are unlocked by default.

For information on replacing a store with a new store, see *Implications of Changing Stores*.

The sysem has few restrictions on the the characters that can be used in the names of checked-in files. See *Administrative Object Names* for details.

Types of File Stores

There are 3 types of file stores:

- Captured
- DesignSync
- External

Each type specifies the amount of control and knowledge the system has over the files. As such, each type has different parameters associated with it.

Stores should use actual host names and paths, not mounted directories.

Captured Stores

A *captured file store* contains *captured files*. A captured store offers flexibility in regard to system control while still taking advantage of the file and access control. Captured files are maintained by the system and are subject to the access rules defined in the policy that governs the file associated

with the business object. The primary means of accessing the file is from the 3DEXPERIENCE platform although it is possible to access it from the file system.

Files should be accessed through the 3DEXPERIENCE Platform

Files in captured stores should not be manipulated or altered outside of the 3DEXPERIENCE platform (for example, through the operating system). If a file that is being checked out is a different size than when it was checked in, the following warning message displays:

“File size has changed from XXXX to YYYY since it was checked in. File may be damaged.”

Filename hashing

Captured stores can use file name *hashing*, which is the ability to scramble the file name. When file name hashing is on (the default), captured stores generate hashed names for checked in files based on a random number generator. If a name collision occurs, it will retry with a new hashname up to 100 tries, then return an error. Since the files for captured stores are physically stored on disk, the names are hashed to be recognized by the 3DEXPERIENCE platform only.

When file name hashing is off, the file names appear in the protected captured directory with original file names. Unhashed file names *collide* in a store more often than when the 3DEXPERIENCE platform generates unique hashed file names for each checked-in file. Since two physical files of the same name cannot reside in a single directory, the 3DEXPERIENCE platform scrambles the name of one copy whenever a collision would occur.

DesignSync Stores

A *DesignSync file store* represents a DesignSync server and is used to associate *DesignSync* files and folders with business objects. Business objects that use DesignSync stores will use vcfile, vcfolder, and vcmodule operations to map DesignSync files, folders, or modules to them. Refer to the *Semiconductor Accelerators Administration Guide* for information about vc commands.

When creating a DesignSync store, system administrators provide information on how to access a DesignSync server. Once created, the store communicates with the specified DesignSync server. Any file, folder, or module that is put in the store actually gets checked into a DesignSync server.

DesignSync stores use HTTP or HTTPS to connect to DesignSync. Matrix Navigator cannot be used for checkin.

Access controls

The table below indicates the access controls required on the DesignSync server, in order to perform various tasks.

User	DesignSync Accesses Required
System Administrator (user creating the DesignSync store)	BrowseServer
User defined in DesignSync Store	SwitchUser
3DEXPERIENCE platform users that perform DesignSync file/folder/module operations	BrowseServer Checkin Checkout with Lock = “yes” Checkout with Lock = “no” Unlock
3DEXPERIENCE platform users that navigate DesignSync files/folders/modules	BrowseServer
Of these required accesses, SwitchUser is the only one that is not allowed in DesignSync by default.	

DesignSync file/folder/module operations include those available from the Semiconductor Accelerator: checkin, checkout (checkout with lock = yes), download (checkout with lock = no), unlock, display file properties, display file versions, and display folder contents. You can remove DesignSync accesses except BrowseServer for users as required. For example, for users that only need the “display” operations, only BrowseServer is required.

The access controls in DesignSync should be setup before creating the DesignSync store.

External Stores

An external file store contains files managed by an external file server or vault. External Stores are used with External (Web Service Adaplet) vaults. You specify the External Vault as a parameter when an External Store is created.

Defining a Store

There are several parameters that can be associated with a store. Each parameter enables you to provide information about the new store. Some parameters are common to all stores, others depend on the type of store you are creating. While only the Name and Type clauses are required, the other parameters can further define the store, as well as provide useful information about the store.

Before defining a DesignSync store, configure the DesignSync server to allow BrowseServer access to the system administrator user that will create the store.

A file store is defined with the Add Store command:

```
add store NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the store you are defining.

`ADD_ITEM` is an Add Store clause that provides more information about the store you are creating.

The complete syntax and clauses for working with stores are discussed in *store Command* in the programming reference section of this guide.

Locations and Sites

Modeling of replicated captured stores relies on the creation of *locations* and *sites*.

- *Locations* provide alternate host and path information for captured stores. They can be added to existing captured store definitions only.
- A *site*, which is a set of locations, can be added to a person or group definition to specify location preferences.

Add the `MX_SITE_PREFERENCE` variable to the Live Collaboration Server's initialization file (enovia.ini). This adjustment overrides the setting in the person or group definition for the site preference, and should be set to the site that is local to the server. This ensures optimum performance of file checkin and checkout for Web clients.

The reason for introducing sites and locations is to enhance checkin, checkout, and open for view or edit performance for clients that require WAN access to a centralized storage location. The central store (“default”) is mirrored to one or more remote machines (“locations”). In this way, a client at the remote site has LAN access to the data.

When a user performs a file checkin, it is written to that user’s preferred location. If none is specified, the system writes the file to the store’s default path. In the schema, the business object is now marked as having a file checked in at this location. When another user requests the file for checkout, the system attempts the checkout from the following locations:

1. Locations that are part of the checkout person’s preferred site. If none of these locations contains the newest copy, then;
2. The store’s ‘default’ location. If this does not contain the newest copy, then;
3. Any location that contains a valid copy of the file. This means that a “sync on demand” is performed—the requested file is copied to the user’s preferred location, and then the local file checkout is performed.

The files can be published to all locations associated with the store by running the MQL `sync` command against either the business object or the store. Following a synchronization, the files for a business object will be available in all locations. Refer to *Synchronizing Captured Stores* for more information.

Implications of Changing Stores

There may be times when you need to create a new store to replace an existing store. For example, the existing store may be running low on space. When you replace a store, you’ll need to change the policies that reference the old store and have them reference the new store. That way, all new files that are checked into objects governed by the policies will be placed in the new store. Refer to *Monitoring Disk Space* for more information.

Remember that the old store will still be used because files checked in before the policy change will still reside in the old store. Also, if these objects are revised or cloned, the new revision/clone references the original file and its storage location. When the time comes for the reference to become an actual file (as when the file list changes between the 2 objects) the file copy is made in the same store the original file is located in.

Defining a Location

Locations contain alternate host and path information for a captured store. The host and path in the store definition is considered to be the *default* location for the store, while any associated location objects identify *alternate* file servers.

Think of a location as another store—it is defined as disk "location." The same rules apply as in specifying a store.

A location is defined with the Add Location command:

```
add location NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the location you are defining.

`ADD_ITEM` is an Add Location clause that provides more information about the location you are creating.

The complete syntax and clauses for working with locations are discussed in *location Command* in the programming reference section of this guide.

Defining Sites

Sites are nothing more than a set of locations. A site can be associated with a person or group object. When associated with a person, the site defines the list of locations preferred by a particular person. When associated with a group, the site defines the list of locations preferred by all members of the group.

In addition, the enovia.ini file for the Studio Modeling Platform or Live Collaboration Server may contain the following setting:

```
MX_SITE_PREFERENCE = SITENAME
```

where SITENAME is the name of the site object.

This setting overrides the setting in the person or group definition for the site preference. It is particularly designed for use in the enovia.ini file for the Live Collaboration Server, where all Web clients should use the site preference of the Server to ensure optimum performance. Refer to the *Installation Guide* for more information.

The complete syntax and clauses for working with sites are discussed in *site Command* in the programming reference section of this guide.

Configuring FCS Routing

FCS routing is a way to manage FCS access for both local and remote users. Remote users include users using the central FCS site, an FCS deployed at their site, or a central FCS site through a private network. To configure FCS routing, you must add an FCS route to the store or location in your environment.

A route includes an external FCS URL and an internal FCS URL. An internal URL is used for direct FCS access and an external URL is used for reverse-proxy FCS access. A reverse-proxy is a gateway for incoming network HTTP traffic. The FCSURLSolver, an MCS component, determines the appropriate URL to use for each FCS request depending on the user's site and the store or location's site.

- If user's site and the store or location's site are the same, the internal URL is used.
- If user's site and the store or location's site are different and there is no site specific URL defined for that store or location, the external URL is used.

A site specific FCS URL can be defined for a store or location which gives all user's associated with that site access to the store or location through that URL. For details, see *Adding a Site Specific FCS URL*.

To add an FCS route to a store or location:

1. In MQL, execute the following commands for each store or location in your environment. The syntax for locations is shown:

   ```
   modify location NAME fcs
   "${CLASS:com.matrixone.fcs.backend.FCSURLSolver}";
   add property externalURL on location NAME value URL;
   add property internalURL on location NAME value URL;
   ```

   Where NAME is the name of the location to which you are adding the route and URL is the internal or external URL.

2. Next, you can verify that the store or location has been configured correctly by printing it.

Adding a Site Specific FCS URL

When a site specific FCS URL is defined for a store or location, all user's associated with that site will be able to access the store or location through that URL. A site specific FCS URL overrides the internal and external URLs.

To add a site specific URL to a store or location:

1. In MQL, add the following property to the store or location to define a site specific route. The syntax for stores is shown:

   ```
   add property siteSITE on store NAME value URL;
   ```

   Where SITE is the name of the site, NAME is the name of the store to which you are adding the property and URL is the FCS URL.

   For example:

   ```
   add property siteFrance on store Documents value http:/host3:port3/
   enovia;
   ```

Synchronizing Captured Stores

Synchronization is the means that the software uses to ensure that users access the latest versions of files when checking out or viewing files in a replicated store environment. Synchronization occurs in one of two ways:

- The software automatically synchronizes files for a user's preferred location during checkout and open for viewing operations.
- System Administrators can use MQL commands to synchronize files for a business object or for a store to ensure that all locations contain the most recent copy of the files.

The lxfile table row is locked during both kinds of file synchronization (with the "select for update" SQL command) to avoid concurrency issues with data integrity. However, when very large files are checked out simultaneously from different sites, a performance hit may occur. For example, when two users connected to different sites/locations do a concurrent file checkout and the file resides only at the store. If the remote user in this case obtains the lock first, the file synchronization is performed and the local user (or any other user for that matter) is locked out until the sync is complete. For very large files, the delay will be noticeable if the subsequent checkout is done before the first has completed.

The software uses HTTP/S to transfer files between FCS locations and stores. This includes all communication between the MCS and FCS.

If an FCS will be performing large synchronization operations of 1500 files or more, its application server must be configured for these large operations. For details about configuring the application server, see the Installation Guide : Installing the Server.

Automatic Synchronization

Synchronization occurs automatically when a person checks out or opens a file for viewing and the file at the user's preferred location is not the latest version. In such a case, the software copies the latest version of the file to the user's preferred location and then checks out or opens the local file.

Manual Synchronization

In order to publish newly checked-in files to other locations associated with the store, System Administrators must use the `sync` command against either the business object or the store. In such a case, the software performs a *sync on demand*, using one of the following:

- To synchronize using the business object, use the following command:

```
sync businessobject TYPE NAME REV [to [store] [location
LOCATION_TARGET{,LOCATION_TARGET}]] [from [store][location
LOCATION_SOURCE{,LOCATION_SOURCE]] [update] [overwrite];
```

  This command copies the business object's files to the specified locations.

- To synchronize only a business object file's metadata, use the following command:

```
sync businessobject TYPE NAME REV metadata format FORMAT_NAME
file FILENAME destination STORE_NAME/LOCATION_NAME capturedfile
FILE_STORE_NAME;
```

The `metadata` keyword in the command indicates that only the meta data for the given business object will be synchronized. When this option is used, it will be up to the implementation to physically move the file from one location to another. This is useful if you want to use 3rd party tools for transferring files between locations instead of the standard file transfer mechanism offered by the 3DEXPERIENCE platform.

This command can only be used with an individual file. The `FILENAME` specified cannot be a directory. If a synchronized record with the `FILENAME` and `FORMAT_NAME` specified in the command is not found in the lxfile table row, an error will result. This command does not delete any files from stores or locations.

- To synchronize using the store name, use the following command:

```
sync store STORE_NAME [continue] [commit N] [to [store]
[location LOCATION_TARGET{,LOCATION_TARGET}]] [from
[store][location LOCATION_SOURCE{,LOCATION_SOURCE]] [update]
[overwrite];
```

All files within the named store are copied to the specified locations.

When comparing files between a remote and host store, the software looks for the hashed name that is known to the database, for the business object associated with the store or location, as well as the file size and permissions.

- To synchronize on more than one businessobject, use the Sync businessobjectlist command.

Do not use the `sync bus` command within a transaction that also performs a file checkin or filenames will be hashed.

The sync command clauses and related sync commands are described in the sections that follow.

Capturedfile FILE_STORE_NAME (sync metadata only)

Use the `capturedfile` clause to specify the file name in the store. If a row in the database is found that points to the location specified in this command, the row will be replaced with the `FILE_STORE_NAME` specified in the command.

Commit N clause (sync store only)

Include the `commit N` clause when syncing large stores. The number “N” that follows specifies that the command should commit the database transaction after this many objects have been synced. The default is 10. For example:

```
sync store “Engineering-Dallas” continue commit 20
```

Continue clause (sync store and sync buslist only)

Include the keyword continue if you don’t want the command to stop if an error occurs. In case of an error, the current businessobject is skipped and the command continues on the next businessobject. If the log file is enabled, failures are listed in the file. Refer to the *Administration Guide* : *About Tracing* in the online documentation.

For example:

```
sync store “Engineering-Dallas” continue
```

If an error occurs when using the `continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next

business object. For this reason, when using the `continue` clause you should also include the `commit` clause, described below.

Destination STORE_NAME/LOCATION_NAME (sync metadata only)

Use the `destination` clause to specify the store or location where the physical file was updated with the external tool.

From clause

Use the `from` clause to specify locations from which files will be replicated to other specified locations. For example:

```
sync store "Engineering-Dallas" continue commit 20 from
location London,Paris,Milan to location Moscow,Boston;
```

```
sync bus Assembly R123 A from location London,Paris,Milan
store;
```

If you do not include the `to` keyword, all locations will be updated, potentially including those listed in the `from` clause. You can optionally include the `store` keyword in either the from or to clause to include the store's host and path.

If an on-demand sync is initiated for multiple locations, the sync will be rolled back if the sync fails at any of the locations, as described above in *To clause*.

Overwrite clause

The `overwrite` clause, used with the `to|from location` and/or `store` clauses, forces an overwrite of files on those servers. All files are copied to the specified locations/store without doing a comparison. For example:

```
sync store "Engineering-Dallas" continue commit 20 from
location London,Paris,Milan to location Moscow,Boston
overwrite;
```

```
sync bus Assembly R123 A from location London,Paris,Milan store
overwrite;
```

Sync businessobjectlist

Use the `sync businessobjectlist` clause to specify a list of business objects to sync:

```
sync businessobjectlist SEARCHCRITERIA [SYNC_ITEM {SYNC_ITEM}];
```

The keyword `businessobjectlist` can be shortened to `buslist`.

For example:

```
sync buslist set MyAssemblies continue commit 20 to location
London,Paris,Milan store;
```

SEARCHCRITERIA can include:

```
| set NAME                                                    |
| query NAME                                                  |
| temp set BO_NAME{,BO_NAME}                                  |
| temp query [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM}]              |
| expand [EXPAND_ITEM {EXPAND_ITEM}]                          |
```

You can also use these specifications in boolean combinations.

SYNC_ITEM can be any of the following:

```
| overwrite                                                   |
| update                                                      |
| commit N                                                    |
| continue                                                    |
| from [store | location NAME{,NAME}                          |
| to [store | location NAME{,NAME}                            |
| vcconnection                                                |
| format NAME                                                 |
```

Format clause

Each file managed by FCS has a format attribute. The policy governing the businessobject owner of the file determines the formats that are allowed for the file. It is possible to synchronize a list or set of businessobjects based on their format attribute. The ability to synchronize only the files of a specified format reduces the amount of data that needs to be synchronized, thus improving performance. You must provide the name of the format that you want to synchronize after the "format" keyword.

Example:

```
sync businessobjectlist SET1 from STORE to location locationS;
```

Here, SET1 is a set of businessobjects that is the same as a businessobjectlist, STORE is a store, and locationS is a location for store STORE.

A limitation of this feature is that only one format can be passed as a filter argument at a time.

To clause

If you want to sync only a subset of all of a store's locations, include the `to location` and/or `store` clause. For example:

```
sync bus Assembly R123 A to location London,Paris,Milan store;
```

```
sync store "Engineering-Dallas" continue commit 20 to location
London,Paris,Milan store;
```

When specifying only some locations, a comma (but not a space) is used as a separator. Including the keyword `store` indicates that the store itself (default host and path) should be updated, too.

If an on-demand sync is initiated for multiple locations, the sync will be rolled back if the sync fails at any of the locations. For example, assume an on-demand sync for five locations. The first two locations sync, but the third fails because the machine is down. The fourth and fifth stores will probably not get synced, depending on where the file is physically located and what order the stores are created in.

- If the file lives in a downed store, none will get synced.
- If the file lives in a store that is online, then the sync will fail when it reaches the downed store. Any stores that would have been synced after that store will not be synced.

While the stores that were synced will retain the new file, the software will rollback the sync command and think that only the original location of the file will still have that file. The software will not know of any other locations that now have the file.

Update clause

Use the `update` clause to copy files only to locations that contain a previous version of the file. For example:

```
sync store "Engineering-Dallas" continue commit 20 update
```

```
sync bus Assembly R123 A update;
```

Using format.file and format2.file

When working with distributed file stores you must be able to identify which of several locations holds up-to-date versions of a particular file checked into a given business object. The `file.format` select clauses shown below are available to more efficiently manage on demand synchronization. Using the `print bus TNR select` command, you can find all locations that hold up-to-date versions of a particular file in order to sync files using a specific location.

For example, suppose you have three locations over a WAN and a sync needs to happen to one location because a user is requesting the file there. You can force sync from the "nearest" synced location, that is, the location from which the traffic is fastest.

- `format.file.location`

  Finds locations which are in the current user's preferred sites *and* are synchronized, and returns the first of them. If no locations satisfy both conditions, returns the first synchronized one.
- `format.file.synchronized`

  Returns all locations holding synchronized versions of the checked in file(s), regardless of user preference.
- `format.file.obsolete`

  Returns all locations holding a copy of the file that has been marked as obsolete (needing synchronization), regardless of user preference.
- `format.file.modified`

  Returns all files of the specified format that were modified and checked in at a particular date.

For both `format.file.originated` and `format.file.modified` subselects:

- If the `filename` is omitted, all files of the specified `format` are returned.
- If both the `format` and `filename` are omitted, all files in all formats are returned.
- If the `format` is omitted but the `filename` is specified all files matching filename are returned regardless of format.

When using `format.file[FILENAME].*`, you can use either of the following formats for the filename:

- the complete `host:/directory/filename.ext` from which it was last checked in.
- just the filename (`FILENAME.EXT`), which must be unique within this object/format. This makes the extraction of data specific to a single file much easier.

Using format2.file

The format2[].file.* selectable is a selectable on business object files used to precisely identify locations where files are synchronized, obsolete or none. The format2.file.synchronized selectable is used to identify the locations where the files are synchronized and the format2.file.status selectable is used to display the status of the file as either: synchronized, obsolete or none.

Additionally, the subfield needsynchronization can be used to identify files needing synchronization from one store/location to another store/location. This selectable returns either True or False. Items in `[FROM_STORE/LOCATION_NAME TO_STORE/LOCATION_NAME]` must be valid for the file store and separated by a space inside the square brackets.

```
print businessobject BO_NAME select
format2.file.needsynchronization [FROM_STORE/LOCATION_NAME
TO_STORE/LOCATION_NAME];
```

For example:

```
Print bus TNR select format2.file.needsynchronization[loc1 loc2]
Format2[f].file[test1.txt].needsynchronization[loc1 loc2]= FALSE
Format2[f1].file[test3.txt]. needsynchronization[loc1 loc2] = TRUE
```

The needsynchronization selectable can be used with the format.file.store selectable:.

```
temp query bus TYPE NAME REV
format.file.store=='Mystore'&&format2.file.needsynchronizat
ion[Mystore]==TRUE
```

Although the needsynchronization selectable is easily used in where clauses, the most efficient way to return a list of business objects that need to be synchronized is through the sync store validate command. You can use the validate keyword with the sync store command or sync businessobject to return a list of business objects that need synchronization from the source store/location to the destination store/location specified in the command. Validate may be followed by the optional keyword size and will allow for the alternative dump character. The locations belonging to the store should not have the same name.

```
sync store STORE_NAME [from [store][location
LOCATION_SOURCE{,LOCATION_SOURCE]] [to [store] [location
LOCATION_TARGET{,LOCATION_TARGET}]] [validate] [size] [dump
[SEPARATOR_STRING]];
```

Or

```
sync businessobject TYPE NAME REV [from [store][location
LOCATION_SOURCE{,LOCATION_SOURCE]][to [store] [location
LOCATION_TARGET{,LOCATION_TARGET}]] validate [dump
[SEPARATOR_STRING]];
```

The list of business objects will be printed in the following format:

```
OID1|format|file1|size
OID2|format|file1|size
OID2|format|file2|size
Total bytes: Number of bytes
```

Tidying all locations

In replicated environments, when a user deletes a file checked into an object (via checkin overwrite or file delete), by default all other locations within that store maintain their copies of the now obsolete file. You can run the `tidy store` command to remove obsolete copies.

To remove obsolete files at each location

- Use the following command:

```
tidy store NAME;
```

You can change the system behavior going forward such that all future file deletions occur at all locations by using the `set system tidy on` command. Since this command changes future behavior and does not cleanup existing stores, you should then sync all stores, so all files are updated, and then tidy so obsolete files are removed at all locations. Once done, the system will remain updated and tidy, until and unless system tidy is turned off.

Running with system tidy turned on may impact the performance of the delete file operation, depending on the number of locations and network performance at the time of the file deletion.

FCS Compressed Synchronization

Compressed synchronization can improve the FCS server synchronization process in a WAN environment.

Compressed synchronization reduces the synchronization-process-elapse time as soon as wide area network latency exceeds 1 ms. Compressed synchronization is available for all synchronization scenarios, including when using the SyncServer tool, the sync bus command line, or on-demand synchronization.

A synchronization process occurring between two locations, or a store and a location, is considered to be WAN-wide (in other words, occurring in a WAN-environment) if the two locations, or the store and the location, belong to distinct sites.

For example, assume there is a store with two locations, A and B. Location A belongs to site A and location B belongs to site B. Any synchronization that occurs between location A and location B is considered to be WAN-wide. A synchronization that occurs between the store and location A, or the store and location B, is also considered to be WAN-wide because the store belongs to the central site by default.

If compressed synchronization is activated, any WAN-wide FCS synchronization is compressed. This means that the files exchanged during the synchronization process are compressed. The FCS server that sends the file compresses the data (if it is not already compressed), and the FCS server that receives the file decompresses it. The compression function does not compress files that are already compressed, as this would cause a performance loss. FCS determines whether a file is already compressed or not by examining its file extension.

MQL provides commands to activate and configure compressed synchronization. For example, you can configure the list of file extensions that specify whether files are compressed by using MQL commands.

Enabling or Disabling FCS Compressed Synchronization

You can use the set system fcssettings zipsync command to enable or disable FCS compressed synchronization.

To enable FCS compressed synchronization

Use the command:

```
set system fcssettings zipsync on;
```

To disable FCS compressed synchronization

Use the command:

```
set system fcssettings zipsync off;
```

The table below summarizes MQL compressed synchronization commands for the inter-site case:

MQL Commands for the Inter-Site Case

	sync store	sync buslist	sync bus	sync bus with zip
zipsync on	cond zip	cond zip	cond zip	force zip
zipsync off	no zip	no zip	no zip	force zip

- `cond zip` means that the synchronization stream depends on the synchronized file extension.
- `force zip` means that the synchronization stream is always compressed.
- `no zip` means that the synchronization stream is not compressed.

The table below summarizes MQL compressed synchronization commands for the intra-site case:

MQL Commands for the Intra-Site Case

	sync store	sync buslist	sync bus	sync bus with zip
zipsync on	no zip	no zip	no zip	force zip
zipsync off	no zip	no zip	no zip	force zip

Printing the Compressed Synchronization Feature Activation State

You can print the FCS compressed synchronization feature activation state using the following commands.

To print the FCS compressed synchronization feature activation state

Use the command:

```
print system fcssettings;
```

The result is:

```
zipsync on|off
```

A result of `on` means the compressed synchronization feature is enabled, `off` means that it is disabled.

For an abbreviated form of the output, you can use the command:

```
print system fcssettings zipsync;
```

For this command the result is simply the following, with the same meanings as above:

```
on|off
```

Configuring FCS Compressed Synchronization

You can configure FCS compressed synchronization settings using the following commands.

To configure file extensions for compressed files

Use the command:

```
set system fcsextensions [FILE_EXTENSION{,FILE_EXTENSION}];
```

File extensions are expressed using alphanumeric ASCII characters, avoiding control characters and punctuation characters. Each extension comprises one to five characters, where each character must match the pattern:

```
[a-z]|[A-Z]|[0-9]
```

The keyword `fcsextensions` is a system property whose value is a comma-separated list of file extensions that identify files that are considered to be compressed. For example:

```
lzw,giff,ar,bz2,jpg
```

The `fcsextensions` list can contain a maximum of 16 extensions, which are stored in lower case.

FCS considers any file with one of these extensions to be compressed. For example, if FCS detects a file with an extension of .jpg (for example, a file named picture.jpg) during the synchronization process, it considers this file to be already compressed and will not try to compress it again as this would cause a performance loss. Compressed file format detection through the file extension is not case sensitive.

The `fcsextensions` property complements an FCS-internal list of compressed-file extensions, which includes the most commonly encountered and used compressed file format extensions:

```
gz,tgz,zip,rar,z,bz,bz2,tbz,png,gif,jpg,tif,mp4,mpg,avi,wmv,aac,mp3,
mpa,wma
```

This list is "hard-coded" into the FCS server and cannot be altered, but you can extend it by using the set system fcsextensions command, as described above.

Printing FCS Compressed File Extensions

You can print all file extensions that are configured in FCS to indicate compressed files.

To print FCS compressed file extensions

Use the command:

```
print system fcsextensions;
```

The result is the list of file extensions indicating compressed files, for example:

```
jpg,giff,tiff
```

Limitations with FCS compressed synchronization

Be aware of the following limitations with regard to the compressed synchronization function:

- File extensions can include only alphanumeric ASCII characters, and must not include control characters or punctuation characters.
- The CATIA V6 and VPLM Solutions use the file extensions '1' and '2' extensively. Therefore, it is strongly recommended not to add these extensions to the `fcsextensions` list.
- If a file has several extensions (for example, `file.tar.gz`), only the trailing one is effective for FCS compressed file detection. In the example cited, FCS would consider this file to have an extension of `gz` and therefore to be compressed.

- Regarding the set system fcsextensions command:
  - The maximum number of extensions that can be added with this command is 16. This limit has been set with the intention of maximizing synchronization process performance.
  - Each extension comprises a string of one to five characters, where each character must match the pattern `[a-z]|[A-Z]|[0-9]`.
  - This command automatically converts all characters in the string to lower case.

FCS Bucket Replication

FCS bucket replication improves FCS synchronization process performance by dividing (by 10 or more) the number of FCS requests required to perform replication on relatively small files (<5MB). The new algorithm for the Sync Store commands, updated in V6R2013, improves performance by reducing the number of database requests, and reduces memory consumption by using low-level database streaming services.

The MQL command to activate FCS bucket replication is:

```
set system fcssetting bucketsync on;
```

The MQL command to deactivate FCS bucket replication is:

```
set system fcssetting bucketsync off;
```

The MQL command to change the FCS bucket replication parameter `syncmaxfilenum` is:

```
set system fcssetting syncmaxfilenum NUM;
```

The MQL command to change the FCS bucket replication parameter `synmaxsize` is:

```
set system fcssetting syncmaxsize SIZE;
```

The MQL command to start synchronization between location of a store or the store itself is:

```
sync store;
```

The commit N argument of the Sync Store command is obsolete and will be ignored, because you can now control the scope of the synchronization through the system-wide settings fcssettings.syncmaxfilenum and fcssettings.syncmaxsize.

For more information on FCS bucket replication, see the *File Collaboration Server Administration Guide : FCS Bucket Replication* in the online documentation.

3 Controlling Access

This chapter discusses all the different types of users and also the access model.

In this section:

User Access

The software lets you control the information users see and the tasks they can perform. Like most Business Administrator functions, you control user access by defining administrative objects in MQL.

This chapter describes all the administrative objects that allow you to control user access. These administrative objects include: persons, groups, roles, associations, policies, and rules. It also defines the various kinds of user access and explains which ones take precedence over others. Use this chapter to help you plan the administrative objects you should create and the accesses you should assign.

When a user attempts to perform a task on a business object — for example, view a file checked into the object or change the value of an attribute for the object — they can only perform the task if the user has been assigned *access*. To see a list and descriptions of the accesses you can assign and deny, see *Accesses*.

If you are working with an application that has a user interface external to the 3DEXPERIENCE Platform, you can also control access by controlling who sees various components of the user interface. For example, if you are building a custom applet, to determine who sees specific pages and links on the pages. If you are working with an application that is built with configurable component objects, you can use access features of those components to control access. Although configurable component objects are administrative objects that can be used to control access, they are not described here because they are specific to a particular type of user interface. For information, see the *Studio Modeling Configuration Guide*.

Persons

You must define a person object for every person who uses this software. There are many components to a person definition, such as the user's full name and e-mail address, and several of these components effect user access.

Use the Configure My ENOVIA Console, instead of MQL, to create or modify persons. For more information, see the One-Click Deployment Experience Administrator's Guide : About the Configure My ENOVIA Console.

A *person* is someone who uses *any* application, not only Matrix Navigator, Business Modeler, and MQL, but also all 3DEXPERIENCE apps, as well as custom apps. The system uses the persons you define to control access, notification, ownership, licensing, and history.

Two persons are defined when you first install the sofware:

- **creator** This user has full privileges for all 3DEXPERIENCE platform applications.
- **guest** This definition uses 3DEXPERIENCE platform infrequently.

You, as the Business Administrator, should first add yourself as a person (Business Administrator). You should then add a person defined as a System Administrator. (The Business and System Administrators may or may not be the same person.)

After you define a person, you can assign the person to user categories (groups, roles, and associations). Use the user categories in policies and rules to control user access.

Person Access

Part of creating a person definition involves specifying the accesses the user should have for business objects. If the user will ever need to perform a task for any business object, you must assign the access in the user's person definition. Other administrative objects, such as policies and rules, allow you to restrict the access that users and user categories (groups, roles, and associations) have for specific business objects and for specific attributes, programs, relationships, and forms.

If you want to prevent a person from ever having access, deny that access in the person definition. For example, assume you have a user who continually overrides the signature requirements for a business object. You could prevent that user from overriding signatures, even if a policy grants the access to the user, by denying the access in the person definition.

User Type

Another element of a person definition that can effect user access is the user type. If a user's type is System Administrator, the software performs no access checking. The system allows such a user to perform any task on any object, even if a policy or rule limits the user's access.

You should assign the System Administrator user type only to people who need full access to all objects, such as a person who will be importing and exporting data or maintaining vaults and stores. For such a user, you should create a second person definition that is not assigned the System Administrator user type. When performing routine work with business objects, the user should set the session context using this second person definition.

Also, if the user's type is Trusted, the system allows the user to perform any task that requires read access only (viewing basics, attributes, states, or history for the object).

Improving Workflow

To improve workflow, you may also want to define a person that doesn't represent a real person. For example, you could define a person to represent the company. When objects reach the end of their lifecycle and are no longer actively worked on, you can have the system reassign the objects to this person. Having the company person own inactive objects allows standard users to perform owner-based queries that return only objects that currently require the users' attention. (The Corporate person in 3DEXPERIENCE Platform software serves this purpose.)

You can also define persons who are not users of the system. This is useful for sending notifications to people outside the system or for maintaining history records associated with people who no longer work in the organization.

User Categories

Three administrative objects allow you to identify a set of users (persons) who require the same accesses: groups, roles, and associations. The shared accesses can be to certain types of business objects (as defined in policies) or other administrative objects, such as specific attributes, forms, relationships, or programs (as defined in rules). When you create a group or role, you assign specific users to the category. When you create an association, you assign groups and roles to the association. A user can belong to any number of groups, roles, and associations.

If many users need access to a type of business object, you should consider creating a user category to represent the set of users. Creating a user category saves you the trouble of listing every user in the policy or rule definition. Instead, you just list the user category. For example, suppose you create a user category, such as a group, and assign 25 users to the group. Then you assign the group to a state in the policy and grant the group full access. All 25 users within that group will have full

access to objects governed by the policy. It is easier to build and maintain user categories than to specify individual users in all policy and rule definitions.

To decide which kind of user category you should create, consider what the users have in common and why they need some of the same accesses.

- *Groups*—A collection of people who work on a common project, have a common history, or share a set of functional skills.

  In a group, people of many different talents and abilities may act in different jobs/roles. For example, an engineering group might include managerial and clerical personnel who are key to its operation. A documentation group might include a graphic artist and printer/typesetter in addition to writers. While the groups are centered on the functions of engineering or documentation, they include other people who are important for group performance.
- *Roles*—A collection of people who have a common job type: Engineer, Supervisor, Purchasing Agent, Forms Adjuster, and so on. Additionally, a role can be defined as either a project or organization. The ability to create projects and organizations is important for applications that use access models based on these user categories. Projects and organizations are still of the type role user category and have the same parameters as roles. Projects and organizations can be used in user access definitions for policy states and rules. A person can be assigned directly to a role, project, or organization.
- *Association*—A collection of groups, roles, or other associations. The members of the user categories have some of the same access requirements based on a combination of the roles users play in the groups in which they belong. For example, perhaps a notification that an object has reached a certain state should be sent to all Managers in an Engineering department. Or maybe, only Technical Writers who are not in Marketing are allowed to approve a certain signature.

Groups, roles, and associations can be used in many ways to identify a set of users.

- As recipients of IconMail sent manually or sent automatically via notification or routing of an object as defined in its policy.
- As the designated Approver, Rejecter, or Ignorer of signatures in the lifecycle of an object.
- In the user access definitions in the states of a policy.
- As object owners, through reassign.

There are just a few differences between the three categories:

- Users can grant their accesses to objects to persons and groups, but not to roles and associations.
- Persons can be assigned directly to groups and roles. Groups and roles make up associations. So a person belongs to an association by virtue of belonging to a group or role that is assigned to the association.
- Groups and roles can be hierarchical. A group or role can have multiple parents and multiple child groups/roles. A role defined as a project or organization can only have ancestors of the same role definition: a role defined as project or a role defined as an organization respectively. Associations are not hierarchical.

The key to determining who should be in which group depends on:

- Whether the user requires access to the business objects.
- How much access is required for the user to do their job.
- When access is needed.

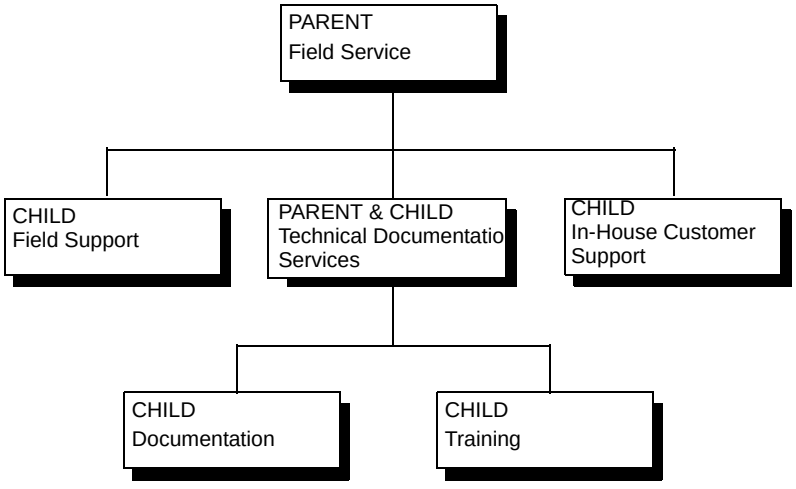
For example, suppose a company is manufacturing a new product, called product ABC. Depending on where the project is in its lifecycle, different people will work on the project, require access for different tasks, or use information related to it:

Lifecycle	Users Needing Access
Planning	*ABC Product Manager association*—An association made up of persons who belong to the Product Manager role and the Product ABC group. The Product Managers create the schedule and project plan and write specifications. *Product ABC group*—All persons involved with the product. This group will view the schedules and plans prepared by the Product Managers.
Review	*Quality Assurance role*—Persons who test products for the company. These people view the specifications and write test plans. *Product ABC group*—Review the specifications prepared by the Product Managers. *ABC Product Manager association*—Review and update items as needed.
Release	*Implementation group*—Engineers, clerical help, a financial manager, and a supervisor help implement the product after it is sold. This group includes some individuals from the user categories assigned to the first two states. For these individuals, you can change the amount of access they have to reflect their changing tasks. *ABC Marketing association*—Persons who belong to both the Marketing Manager role and to the Product ABC group. Advertise the product and manage order processing.

Group Hierarchy

Groups are frequently hierarchical. In hierarchical groups, access privileges that are available to a higher group (*parent group*) are available to all of the groups below it (subgroups or *child groups*).

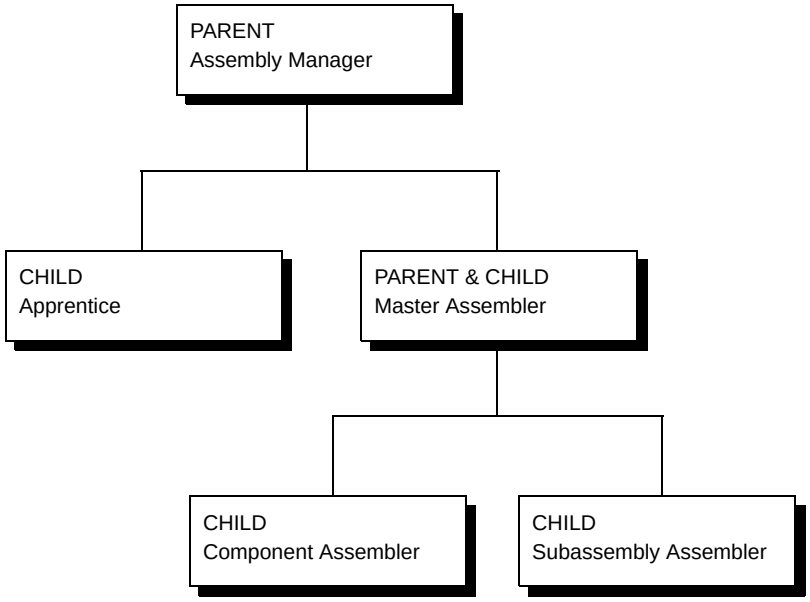
For example, in the figure below, the Field Service group is the parent of the Field Support, Technical Documentation Services, and In-House Customer Support groups. The Technical Documentation Services group is also a parent of the Documentation and Training groups.



You can establish group hierarchy using the procedures in this chapter.

Role Hierarchy

Roles are often similarly hierarchical. For example, in the figure below, the Assembly Manager role is the parent of the Apprentices and Master Assembler roles. The Master Assembler role is also a parent of the Component Assembler and Subassembly Assembler roles.



A child group or role can have more than one parent. Child groups/roles with more than one parent inherit privileges from each of the parents. You can establish role hierarchy using the procedures in this chapter.

Prior to the V6R2013 release, the VPM roles VPLMViewer, VPLMCreator, and VPLMProjectLeader were not hierarchically structured, even though the associated access rights were cumulative. In other words, the VPLMCreator role had the same access rights as the VPLMViewer role, plus some more, and the VPLMProjectLeader role had the same access rights as the VPLMCreator role (and thus VPLMViewer), plus some more.

This resulted in a significant redundancy in policy and rule access definitions, as can be seen in the following example:

```
policy "VPLM_SMB"
   state "IN_WORK"
      user "VPLMViewer"
         read,show,checkout
         filter "(organization.ancestor match
                     context.user.assignment[$CHECKEDUSER].org) &&
                  (project ==
                     context.user.assignment[$CHECKEDUSER].project)"
      user "VPLMCreator"
         read,show,create,delete,modify,promote,demote,grant,revoke,
         changename,changeowner,changetype,checkin,checkout,lock,
         unlock,fromconnect,fromdisconnect,toconnect,todisconnect
         filter "(organization.ancestor match
                     context.user.assignment[$CHECKEDUSER].org) &&
                  (project ==
                     context.user.assignment[$CHECKEDUSER].project)"
```

```
        user "VPLMProjectLeader"
           read,show,create,delete,modify,promote,demote,grant,revoke,
           changename,changeowner,changetype,checkin,checkout,lock,
           unlock,fromconnect,fromdisconnect,toconnect,todisconnect
           filter "(organization.ancestor match
                      context.user.assignment[$CHECKEDUSER].org) &&
                   (project ==
                      context.user.assignment[$CHECKEDUSER].project)"
```

By reorganizing these roles hierarchically, it is possible to leverage the native inheritance role mechanism of the 3DEXPERIENCE Platform so as to specify only those accesses that are specific to each role:

```
policy "VPLM_SMB"
   state "IN_WORK"
      user "VPLMViewer"
         read,show,checkout
         filter "(organization.ancestor match
                      context.user.assignment[$CHECKEDUSER].org) &&
                   (project ==
                      context.user.assignment[$CHECKEDUSER].project)"
      user "VPLMCreator"
         delete,modify,promote,demote,grant,revoke,changename,
         changeowner, changetype,checkin,checkout,lock,unlock,
         fromconnect,fromdisconnect,toconnect,todisconnect
         filter "(organization.ancestor match
                      context.user.assignment[$CHECKEDUSER].org) &&
                   (project ==
                      context.user.assignment[$CHECKEDUSER].project)"
```

Here, the `read`, `show`, and `checkout` accesses are no longer specified in the access definition of the VPLMCreator role since they are inherited from VPLMViewer. The access definition of the VPLMProjectLeader role was completely removed, as it was exactly the same as VPLMCreator.

Policies

A policy controls many aspects of the objects it governs, including who may access the objects and what tasks they can perform for each state defined in the policy. Policies grant access to objects.

There are three general categories. For each category, you may assign full, limited, or no access.

- *Public*—Everyone in the database. When the public has access to perform a task in a particular state, any user can perform the task (except if denied in person definition). When defining public access, it is important to define access limits. Should the public be able to check in files to the object, override restrictions for promotions, and delete objects? These are some of the access questions that you should answer when defining the public access.
- *Owner*—The person, group, role, or association that currently owns the object. When a user initially creates an object, the user (person) who creates it is the *owner*. This user remains the owner unless ownership is transferred to someone else. In an object's lifecycle, the owner usually (though not always) maintains control or involvement. In some cases, the original owner might not be involved after the initial state. Typically, the owner has full access to objects.
- *User*—A person, group, role, or association that has specific access requirements for a particular state. When a group, role, or association is assigned access, all the persons who belong to the group, role, or association will have access. Additionally, all persons assigned to

groups and roles that are children of the assigned group or role will have access. (Child groups inherit all accesses from the parent group and child roles inherit all accesses from the parent role.)

For example, you may not want the public to make flight reservations. Therefore, the public is not given access to create reservation objects. Instead, you establish that a Travel Agency group can originate flight reservations. Any member of that group can create a reservation object. Note that once an agent creates a reservation object, that agent is the owner and has all access privileges associated with object ownership.

Assigning user access to groups, roles, and associations is an effective means of providing access privileges to a user. Under most circumstances, a person will have both a group and a role assignment and may also have multiple group and role assignments. In many cases, it is easier to specify the roles, groups, or associations that should have access in a policy rather than list individual users. This way, if personnel changes during a stage of the project, you do not need to edit every policy to change user names.

If a user is assigned access (public, owner, or user) in the current state of an object, the system allows the user to perform the task. For example, suppose a user belongs to a group and a role. If the policy allows the role to perform the task but does not allow the group to perform the task, then the user can perform the task.

For more information about policies, see policy Command.

Rules

Once the 3DEXPERIENCE Platform determines that a user's person definition and the policy allows the user to perform a task for an object's current state, the software then checks to see if any rules prevent the user from performing the task. Rules restrict access to attributes and relationships. Without rules, attributes and relationships are visible to everyone.

As described above, a policy controls the tasks users can perform for each state of an object. In contrast, a rule controls the tasks users can perform regardless of the object. The tasks that rules apply to are limited to only those involving attributes, relationships, forms, and programs. For example, a policy might allow all users in the Engineering group to modify the properties of Design Specification objects when the objects are in the Planning state. But you could create a rule to prevent those users from changing a particular attribute of Design Specifications, such as the Due Date. In such a case, Engineering group users would be unable to modify the Due Date attribute, no matter what type of object the attribute is attached to.

When you create a rule, you define access using the three general categories used for assigning access in policies: public, owner, and user (specific person, group, role, or association). For a description of these categories, see the *Policies* section above. Note that owner access does not apply to rules that govern relationships because relationships don't have owners.

Access that is Granted

There is one way users can gain access privileges that aren't controlled by the Business Administrator using administrative objects. In Matrix Navigator and MQL, users can grant any or all the access privileges they have for a business object to another user or group. However, you can only grant accesses on an object to another user or group if you have the "grant" access privilege for the object. As a Business Administrator, you control who has grant access by assigning or denying the grant access in the person definition and in policy definitions.

You can only grant accesses that have been assigned to you in your person definition or in a policy for an object's current state. For example, if a your person definition denies the override access privilege, you cannot grant that privilege to another user. However, you can be granted privileges

that are denied in your person definition. (This is the only way you can perform a task that is denied in your person definition.) You could not then grant the privilege to another user.

The MQL command allows users to:

- grant an object to multiple users
- have more than one grantor for an object
- grant to any user (person/group/role/association)
- use a key to revoke access without specific grantor/grantee information

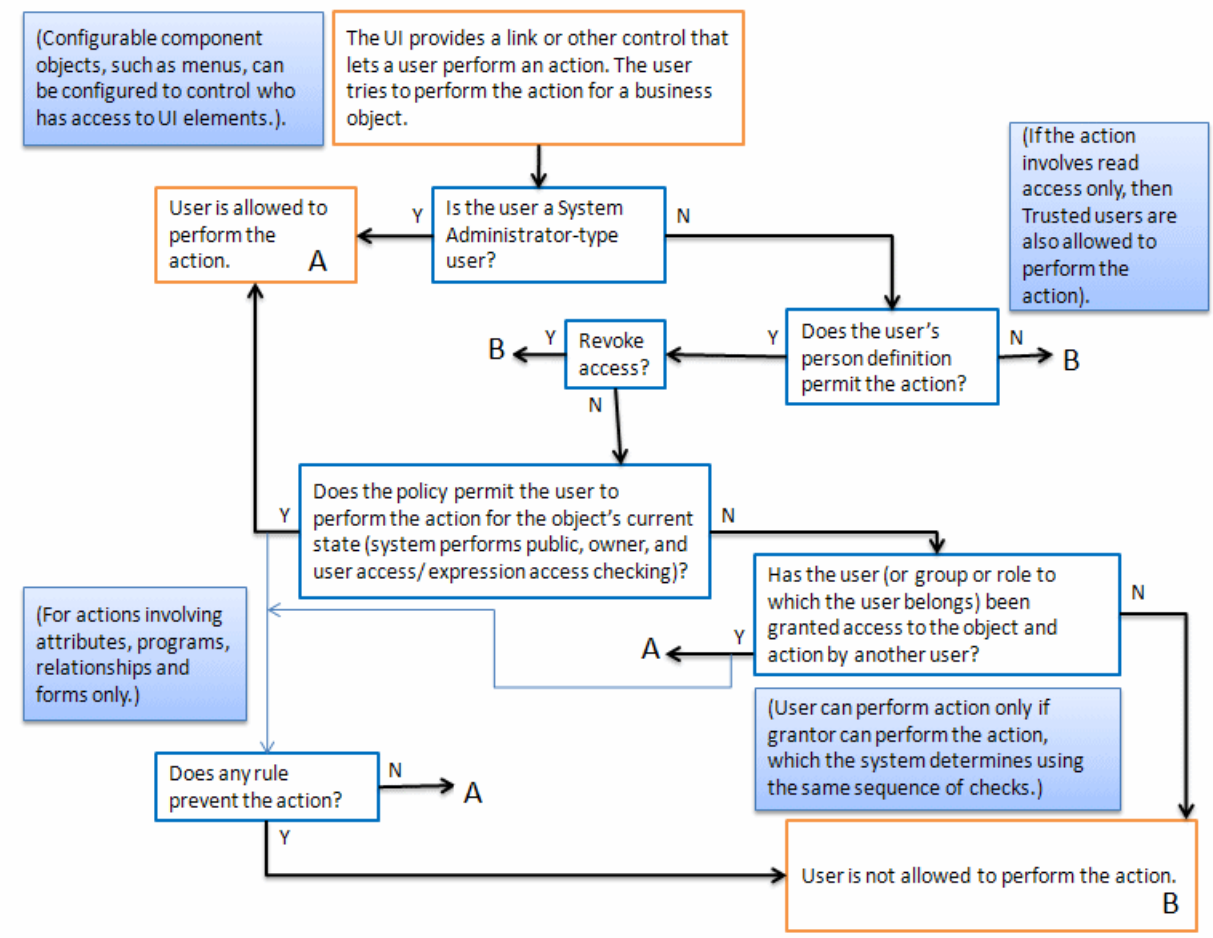
Users cannot give the grant access itself. The intent is to provide just 1 level of delegation. Although including grant in the list of accesses will not fail, the grantee of the grant access will not be able to grant, unless they already have grant access.

Apps can use the MQL commands. However, the Matrix Navigator user interface does not support these features (only 1 grantor and grantee are allowed).

For information on how to grant access to objects, see *Granting Business Object Access*.

Which Access Takes Precedence Over the Other?

Suppose a person definition does not include delete access but a policy assigns delete access to the user. Will the user be able to delete an object governed by the policy? The answer is no, the user won't be able to perform the action because the person definition takes precedence over the policy. This section describes which kinds of access take precedence over others.



The highest level of access control occurs through the user interface: users who have delete access for an object can only delete the object if the user interface provides some mechanism, such as a Delete link, button, or menu option, that lets users delete the object. If you are working with an application that is built using configurable components (for example, command, menu, table, form objects), you can use access features for these components to control who can see these user interface elements. The access features include restricting user interface components to specific roles or access privileges. Some components also let you control access using a select expression or JPO. For information on the access controls available for configurable components, see the *Administration Guide : Which Access Takes Precedence?*

When you attempt to perform an action for a business object, the system checks to see what type of user is defined in the person definition. If you are a System Administrator-type user, the system allows the action and performs no further checking. If the user is Trusted and the action involves read access only, the action is permitted.

If you are not System Administrator or Trusted (or if you are Trusted and the action involves more than read access), Live Collaboration checks your person definition. A person definition takes precedence over accesses granted in the policy definition—if an access is denied in the person definition, you will not have that access, even if a policy assigns the access. However, you can be "granted" access to a business object even if the action is denied in the person definition. (See *Access that is Granted*.)

If the person definition allows access, Live Collaboration next examines the current state of the policy to see if you are allowed access. The policy allows the user access if the access for the action is assigned to the:

- public *or*
- owner and the user is the owner *or*
- user or to a user category (group, role, association) to which the user belongs

If the user is denied access in the policy or person definition, the system checks to see if the user has been granted the access for the business object by another user. If so, the system makes sure the grantor has the access by going through the same access checking as described above. If the grantor has access, then the user is allowed to perform the action. If the user has not been granted access or the grantor doesn't have the access, the action is denied.

If the policy and person definition allow the access or if the user has been granted access, the user is allowed to perform the action with one important exception. If the action involves an attribute, relationship, form, or program, the system first checks to see if any rules deny access. If the system finds a rule that governs the attribute, relationship, form, or program for which the action applies, it goes through the same kind of checking as it did for the policy. If the access is assigned for the user in any access (public, owner, or user), the action is allowed. If access is not specifically assigned in a rule that governs the object, access is denied.

WHERE access is found is not as important as *IF* it is found. For example, you may receive access by virtue of belonging to a group that is assigned to a parent group that is assigned access in a policy. Your group may be denied access but the user's role is allowed access. In addition, you might have read access to a business object, allowing attributes or a form to be displayed, but then have modify access to an attribute denied.

Accesses

An access is the permission to perform a particular task or action. You assign accesses in person definitions, policies, and rules. In policies and rules, you can assign or deny accesses to the public (all users), owner, or users (persons, groups, roles, and associations). Users also can grant their accesses for an object to other users. This table lists and describes all the accesses available and shows which administrative object uses each access.

Access Privilege	Allows a user to:	Can be assigned in:
`Approve`	Approve an object for promotion to the next state.	Person definition Policy definition
`Reject`	Prevent an object from being promoted until it meets the approval of the user.	Person definition Policy definition
`Ignore`	Override the approval or rejection of an object and to sign in place of others.	Person definition Policy definition
`Read`	View the properties of an object, including basics attributes, states, and history. For more information, refer to *Read Access*. To delete files checked into a business object, a user must have read and checkin access.	Person definition Policy definition Rule for attributes

Access Privilege	Allows a user to:	Can be assigned in:
`Modify`	Edit the attributes of an object or relationship.	Person definition Policy definition Rule for attributes Rule for relationships
`Delete`	Delete an object from the database. Does not apply to files. To delete files checked into a business object, a user must have both read and checkin access.	Person definition Policy definition
`CheckOut`	Copy files contained within a business object to the local workstation. Also allows the user to open a file for viewing. To allow a user to use the open for edit command for files checked into an object, the user must have checkin, checkout, and lock access for the object.	Person definition Policy definition
`CheckIn`	Copy files from the local workstation to a business object. To allow a user to use the open for edit command for files checked into an object, the user must have checkin, checkout, and lock access for the object. To delete files checked into a business object, a user must have both read and checkin access.	Person definition Policy definition
`Schedule`	Set and modify schedule dates for the states of a business object.	Person definition Policy definition
`Lock`	Restrict other users from checking files into a business object and from opening files for editing. To allow a user to use the open for edit command for files checked into an object, the user must have checkin, checkout, and lock access for the object. If an object is governed by a policy with enforce locking turned on, users can only lock the object when checking out a file. Users cannot manually lock the object. Enforce locking prevents one user from overwriting changes to a file made by another user. See also the discussion of enforced locking in *Enforce Clause*.	Person definition Policy definition
`Execute`	Execute a program. This access applies only when assigning rules for programs. It does not apply in person or policy definitions. A program rule establishes who has the right to use the programs to which it is assigned by setting owner, public, and user accesses in the rule. Applies only to programs, including wizards, executed explicitly by the user; that is business object methods and those executed via a custom toolbar.	Rule for programs
`UnLock`	Release a lock placed on a business object *by another user*. Users may release locks they themselves have placed on objects without this access. Reserve unlock access only for those users who may need to override someone else's lock, such as a manager or supervisor. See *Unlock Access*. Unlocking an object locked by another user should especially be avoided for objects governed by policies with enforce locking turned on. For information on enforce locking and how unlocking objects manually can cause confusion, see *Enforce Clause*.	Person definition Policy definition

Access Privilege	Allows a user to:	Can be assigned in:
Freeze	Freeze, or lock, a relationship so that business objects may not be disconnected until the relationship is thawed. Also, the type or attributes of a frozen relationship may not be modified.	Person definition Policy definition Rule for relationships
Thaw	Thaw, or unlock, a relationship so that it may be modified or deleted.	Person definition Policy definition Rule for relationships
Note: When a user attempts to perform a task that requires freeze or thaw access, the system checks the access privileges for the objects on both sides of the relationship (defined in the relevant policies), as well as accesses defined for the relationship type (defined in relevant access rules).		
Create	Create original and clone business objects. Create access applies only for the first state of an object. If a policy gives the owner or the public create access in the first state of an object, anyone will be able to create that type of object (when objects are created, the owner is the one performing the function). To allow only a certain group, role, person, or association to be able to create a specific type of object, deny create access for the owner and public in the object's first state. Then add a user access for the person, role, group, or association that includes create access.	Person definition Policy definition
Revise	Create a minor revision of the selected business object. This is a synonym for MinorRevise access, which also makes the revision backwards compatible.	Person definition Policy definition
Promote	Change the state of an object to be that of the next state.	Person definition Policy definition
Demote	Change the state of an object to that of a prior state.	Person definition Policy definition
Grant	Grant the access privileges the user has for a business object to another user. *Users cannot grant the grant access itself. The intent is to provide just 1 level of delegation. Although including grant in the list of accesses will not fail, the grantee of the grant access will not be able to grant, unless s/he already has grant access.*	Person definition Policy definition Rule definition
Revoke	Revoke the access privileges that have been granted for a business object. *Use with caution! Anyone with this privilege can revoke grants for another user.*	Person definition Policy definition Rule definition
Enable	Unlock the state so that a business object can be promoted or demoted.	Person definition Policy definition
Disable	Lock a state so that a business object cannot be promoted or demoted.	Person definition Policy definition
Override	Disable requirement checking allowing for promotion of an object even when the defined conditions for changing the state have not been met.	Person definition Policy definition

Access Privilege	Allows a user to:	Can be assigned in:
ChangeName	Change the name of a business object.	Person definition Policy definition
ChangeType	Change the type of a business object or relationship. To change a relationship type, the user needs changetype access for the object on both ends of the relationship and on the relationship.	Person definition Policy definition Rule for relationship
ChangeOwner	Change the owner of a business object.	Person definition Policy definition
ChangePolicy	Change the policy of a business object.	Person definition Policy definition
ChangeVault	Change the vault of a business object.	Person definition Policy definition
FromConnect	Link business objects together on the “from” side of a relationship.	Person definition Policy definition Rule for relationships
ToConnect	Link business objects together on the “to” side of a relationship.	Person definition Policy definition Rule for relationships
FromDisconnect	Dissolve a relationship on “from” business objects.	Person definition Policy definition Rule for relationships
ToDisconnect	Dissolve the “to” side relationship between business objects.	Person definition Policy definition Rule for relationships
Note: When a user attempts to perform a task that requires connect or disconnect access, the system checks the access privileges for the objects on both sides of the relationship (defined in the relevant policies), as well as accesses defined for the relationship type (defined in relevant access rules).		
ViewForm	View a form. The ViewForm and ModifyForm accesses apply only when assigning rules for forms. It does not apply to person or policy definitions. A form rule establishes who can view and modify the form to which it is assigned by setting owner, public, and user accesses in the rule.	Rule for forms
ModifyForm	Edit attribute and other field values in a form.	Rule for forms
Show	Control whether a user knows that a business object exists. The access privilege is designed to prevent a user from ever seeing the type, name, or revision of an object. See *Show Access*.	Policy definition Rule definition

Read Access

The **read access** privilege allows a user to view the properties of an object, including basic information, attributes, states, and history. For example, if a user does not have read access for an object, the user will not be able to see the Object Inspector, Attributes, Basics, States, or History dialog boxes when the object is selected. Furthermore, the user won't be able to expand on the object using the Navigator browser or view the Image assigned to the object. When a user performs queries based on criteria other than type, name, and revision, Live Collaboration only finds those objects to which the user has read access. This applies also to visual cue, tip, and filter queries: Live Collaboration will not apply the visual, filter, or tip to objects found by the query if the user does not have read access to them.

All users have access to tables (both indented and flat) and reports. If a user does not have read access to an object, Live Collaboration displays only the type, name, and revision. Table cells that provide additional information on objects to which there is no read access display "`#DENIED!`". Reports work in a similar manner, providing only the information to which the user has access. MQL commands using `SELECT` or `EXPAND` also return `#DENIED!` if the user does not have read access. When read access is denied on an attribute, the `#DENIED!` message is displayed there as well.

If read access to a business object is denied, the system does not display the Attributes or the States browser. This prohibits functions such as modify, promote, demote, and so on. In MQL, however, if users have these privileges, the transactions are allowed. For example, if a user has no read access but does have modify access, the user may modify the object's attributes in MQL. This is because the user does not have to print the business object (which would not be permitted) before modifying it.

However, if a user has modify or promote access for an object, it makes little sense to deny read access. Similarly, allowing a user to check out files without assigning read access only partially prohibits the user from seeing information for the object. Carefully think through how you assign accesses, whether you are assigning accesses for users (person, group, role, and association), policies, or access rules. Read access must be used logically in conjunction with the other accesses.

Unlock Access

Take extra care when granting unlock access. Unlock access allows a user to unlock objects that have been locked by *other* users. Reserve unlock access only for those users who may need to override someone else's lock, such as a manager or supervisor.

Unlocking an object locked by another user should especially be avoided for objects governed by policies with enforce locking turned on. For information on enforce locking and how unlocking objects manually can cause confusion, see *Enforce Clause*.

Show Access

When a user does not have show access, 3DEXPERIENCE Platform behaves as if the object does not exist. So, for instance, the `print businessobject` command errors out with the same error as if the object does not exist.

Without show access for an object, the user cannot see the object or any information about that object in any browser or in MQL. Specifically, the user will not see the object displayed under these circumstances:

- Performing a "find" in the desktop or Web versions.
- Clicking on the plus sign in a Navigator dialog in the desktop or Web clients.

- Evaluating a query in MQL.
- Running the `expand businessobject` command in MQL.
- Running the `print set` command in MQL.
- Running the `expand set` command in MQL.
- Running the `print connection` command in MQL.
- Reading an email message sent by Live Collaboration that includes a business object that was routed or sent.
- Opening the revisions dialog.
- Opening the History of an object that references a “no show” object.

Usually it makes no sense to remove show access from a Person definition access mask.

In the History of an object, other objects are sometimes referred to. The performance impact of determining whether the current user has access to see the Type, Name and Revision of such objects is significant and unavoidable. To workaround this issue, you can delete individual history records using the Delete History clause of the `modify businessobject` or `modify connection` command. This can be used in action triggers to remove such records.

Additionally, a user who does not have show access for an object will not see references to the object when these operations occur:

- Running the `evaluate expression` command in MQL.
- This command evaluates an expression against either a single, named business object or against a collection of them specified via some combination of sets, queries, and/or expansions. In the former case, when a user does not have show access, the command behaves as if the object does not exist. In the latter case, it simply leaves the object out of the collection.
- Running any of the `businessobject` commands in MQL.
- If a user has no show access to an object, then the object is not presented as the next or previous revision of an object to which the user does have show access. Such objects are simply left out of (skipped from) the revision sequence. So if there are three objects in a revision sequence with revisions 1, 2, and 3 in that order and the user has show access only to 1 and 3, the next revision of 1 will be represented as 3.
- If the user tries to create an object that exists, but they cannot see, they will receive an error message indicating “Access denied.”

With show access, you will have access to the type, name, and revision fields of the object. This means that if you only have show access and print an object in MQL, you will be able to select the type, name, and revision but will receive #DENIED for any other select. This also means that if you include selectables in the Where clause other than type, name, and revision, the object will not be found even if the Where clause evaluates to true, because you do not have access to the fields in the Where clause (for example, vault, policy, state, or attributes).

Show access should not be used as an access strategy for objects that you wish to search for through the Live Collaboration Server. If you have only show access on an object, all searches via the Web UI will never show the object in the results list. This applies to all searches done through the Web interface, not only Advanced Search. The search component automatically inserts elements into the Where clause of a query that are not available to a user with only show access. If you want to use show access for searching in the Web UI, you will need to create a custom tool.

Checking Show Access

The MQL command `set checkshowaccess [ON|OFF]` sets/unsets the global flag to indicate whether the new access checking required by show access is to be executed or not. As this is a one-time-only command, it is provided only via MQL. If set to OFF, the software will not perform the checking, and the feature will not be enabled. If set to ON, the software will always perform the checks.

By default, the database is upgraded with a global setting that indicates that show access should NOT be checked, and therefore the feature will not actually be enabled until this flag is changed. Since show access will not be checked, all users, states and rules will have show access. This will guarantee upward compatibility.

Until the `checkshowaccess` flag is explicitly set to ON, all the new access checks required for this feature will be skipped. Even after the `checkshowaccess` flag is set to ON, the only effect will be the negligible time spent in the new access checks since all users will have Show access to all objects by default.

Show access status is included in export and imports. (However, the access of the global flag is not included).

Show Access in Sorted Sets

The checks for show access do not occur for all set commands for the following reasons:

- A performance penalty is paid for evaluating show access on each object in a set.
- The only information that is hidden by show access is an object's type, name, and revision, but that very information had to be known to the set owner at the time the set was created.
- The primary use case for Sorted Sets is to support efficient pagination in html applications via the following technique:

  a) First by performing a query/expand into a temporary set

  b) Then optionally sorting that set

  c) And then printing the set with start/end indices to page through it.

  Since in this case show access is applied during the query/expand evaluation, re-applying it for the print set command is unnecessary.

Connection Accesses

Objects can be connected in desktop using drag and drop, and the relationship to use may be chosen from the connect bar. You can use the same technique to connect a new object and remove an old object in one step. When replacing an object using drop connect, if you drop onto a child object to replace it, 3DEXPERIENCE platform keeps the same relationship that was already there rather than using the relationship shown in the connect bar. You are actually *modifying* one end of the connection, not deleting the relationship and creating a new one. This means that to/fromconnect and to/fromdisconnect accesses are checked only on the end of the connection that is changing.

You could create a relationship rule to control this behavior by limiting *modify* access to those who should be allowed to do so. Alternatively, a trigger program could control the ability to perform the replacement.

Working With Expression Access Filters

User access lists defined on a policy or rule can accept a filter expression in order to grant or deny access to a specific user. For example, the access portion of the policy or rule might be:

```
user Writer read,modify,fromconnect,toconnect filter ACCESS_FILTER;
```

Where ACCESS_FILTER is any valid expression.

If the filter expression evaluates to “true,” the specified access will be granted; otherwise the access is denied.

Policies and rules should use organization, project, maturity, and reserve logic instead of access filters whenever possible. These security checks ensure the completeness of search results. You can use the options in Configure My ENOVIA to define People and Organizations. For more information, see the One-Click Deployment Experience - Adminstrator’s Guide: About the Configure My ENOVIA Console.

In order to evaluate a filter, at least Read access (and Show access, if enabled) is required so these access checks are disabled when analyzing filters. However, if the filter includes the access selectable, such as one that checks access on a connected object, these access checks are turned back on for evaluation.

A filter in a policy is always executed in the context of the login user.

Use localfilter instead of filter in policies and access rules to return only results to which the current user definitely has access. When you use localfilter, the expression is not evaluated by full-text search.

The following describes operands of expression access filters.

- Anything that you can select on a business object can be used as an operand of an expression access filter. For example:

  ```
  ("attribute[Priority Code]" == "High") && (description ~~ "*test**")
  ```

- Anything that you can select from the context user object can be included as an operand of an expression access filter. For example:

  ```
  context.user.isassigned[Group_Name] == true
  ```

- Any property defined on the “context user” can be used as an operand of an expression. For example:

  ```
  context.user.property[Export Allowed].value == true
  ```

- Any expression that checks the access inherited from a connected object can be an operand of an expression access filter. For example:

  ```
  to.from.current.access[modify] ~~ true
  ```

  This expression evaluates to true for a business object at the “to” end of a connection only if the business object at the “from” end has the named access.

  The ACCESS macro can be used generically to check the access required for any requested operation. For example:

  ```
  to.from.current.access[$ACCESS] ~~ true
  ```

  With this expression access filter in place, attempting a “mod bus T N R” on the child object results in “modify” replacing $ACCESS during macro processing. The expression will then evaluate to true only if the object at the from end of the connection has “modify” access.

If an object is connected to more than one parent object, the expression will evaluate to true if any one of the parents has the required access. To specify that a child should inherit the access from only one of the parents, the expression must identify the relevant connection between the parent and the child. For example:

```
to[relationship1].from.current.access[$ACCESS] ~~ true
```

The $ACCESS macro within brackets is recognized for the “access” selectable of the state of a business object, as in one of the following:

```
current.access[$ACCESS]
state[STATE1].access[$ACCESS]
```

The macro is not expanded if it appears anywhere else in the expression. This includes the case where a program is called as part of the access filter. There will be no macro substitution for $ACCESS in the following access filter:

```
program[checkAccess $ACCESS]
```

However, if the checkAccess program itself creates a command of the form `print bus $OBJECTID select current.access[$ACCESS]` it will be evaluated.

In the following show, read, modify access filter, the ACCESS macro appears outside brackets:

```
filter '("$ACCESS" == "show") || ("$ACCESS" == "read") ||
("$ACCESS" == "modify" && (from.owner == context.user))'
```

This expanded use of $ACCESS can be used to create rules that define the access a user has on a relationship.

- The $ACCESS macro can be used in an access rule filter and is populated with the current access being checked whenever the filter is evaluated. For example, the following rule allows a different condition for modify vs. revise access:

  ```
  ($ACCESS == modify && <modify-condition>) || ($ACCESS == revise $$
  <revise-condition>)
  ```

  Output from the revise portion of this filter continues to be "revise" rather than "minorrevise" in order to maintain backward compatibility.

- You can also define an expression access filter in a business object attribute value and have Live Collaboration evaluate the expression stored in the attribute when checking access. For example, the expression filter might be:

  ```
  evaluate attribute[expattr]
  ```

  Objects governed by a policy that includes this filter, must be of a type that includes the expattr attribute. Live Collaboration first extracts the expression stored as the value of the attribute “expattr” and then evaluates that expression against the business object. For example, if the value of the attribute “expattr” is:

  ```
  context.user.isassigned[MX-GROUP-1] == true &&
  context.user.isassigned[Role_1] == true
  ```

  then access is given only to users with these assignments.

- You can define an expression access filter in a business object description and have Live Collaboration evaluate the expression stored in the description. For example the expression filter would be:

  ```
  evaluate description
  ```

  For objects governed by a policy that includes this filter, Live Collaboration first extracts the expression stored as the description and then evaluates that expression against the business object. For example, if the value of the description is:

  ```
  context.user.isassigned[MX-GROUP-1] == true &&
  context.user.isassigned[Role_1] == true
  ```

  then access is given only to users with these assignments.

- You can define a dynamic expression and store it in a description of a connected control object. For example, if we have the following structure:

  Assembly MTC1234 0 is connected to Control Rule1 0 by relationship Control

  You might have an expression filter defined as:

  ```
  evaluate to[Control].businessObject.description
  ```

  For objects that are governed by the policy that includes this filter, Live Collaboration first extracts the expression stored as the description of the object connected by the Control relationship and then evaluates that expression against the business object. For example, if the value of that description is:

  ```
  context.user.isassigned[MX-GROUP-1] == true &&
  context.user.isassigned[Role_1] == true
  ```

  then access is given only to users with these assignments. This allows you to store the expression filter rule in some central object to which all objects to be controlled are connected.
- If a revoke filter expression evaluates to “true,” the specified access will not be granted. If it evaluates to “false,” the specified access will not be revoked but it will also not be granted. Before access is granted, the system checks to see if access has been explicitly granted.
- Filter expressions of the form `current.access[ACCESS_TYPE] == TRUE` accept `minorrevise` as an ACCESS_TYPE.

Filter expressions of the form `current.access ~~ *ACCESS_TYPE*` are risky since `revise` is now a substring of both `minorrevise` and `majorrevise`.

Expression Access Filter Example

When developing filters, you can use the `eval expr` command on the filter to qualify that the expression is valid before including it in the policy or rule.

For example:

```
MQL<15> eval expr '(attribute[Actual Weight] > 100) AND
("relationship[Designed Part Quantity].to.type" == "Body Shell")' on
bus Comment 12345 1;
FALSE
```

From the above we can say that the expression

```
'(attribute[Actual Weight] > 100) AND ( "relationship[Designed Part
Quantity].to.type" == "Body Shell")'
```

is a valid expression and thus can be used for an expression access filter.

Context in which the filter is executed, for example:

```
MQL<1>add person pgrantor;
MQL<2>add person pgrantee;
MQL<3>add type test_typ;
MQL<4> add policy ptest type test_typ state exists public show,read
user pgrantor grant,revoke,changevault filter
'state[toggle].access[execute]' state toggle public show user pgrantor
execute;

# Add the bus and the grant
MQL<5> add bus test_typ bus1 0 policy ptest vault xxxx;
MQL<6> set context user pgrantor;
MQL<7> mod bus test_typ bus1 0 grant pgrantee access changevault;
MQL<8> set context user pgrantor;
MQL<9> print bus test_typ bus1 0 select current.access[changevault];
```

```
>> TRUE
MQL<10> set context user pgrantee;
MQL<11> print bus t2 ptest 0 select current.access[changevault];
>> FALSE
```

This is because the pgrantor has execute access in state toggle. Even though pgrantee is granted the changevault access, the login user (`pgrantee`) has no execute access in state '`toggle`' resulting in filter ('`state[toggle].access[execute]`') condition to fail & thus it shows FALSE for:
'`print bus test_typ bus1 0 select current.access[changevault]`'

```
MQL<12> set context user creator;
MQL<13> mod policy ptest state toggle remove user pgrantor execute;
MQL<14> mod policy ptest state toggle user pgrantee execute;
MQL<15> set context user pgrantor;
MQL<16> print bus test_typ bus1 0 select current.access[changevault];
>> FALSE
MQL<17> set context user pgrantee;
MQL<18> print bus test_typ bus1 0 select current.access[changevault];
>> TRUE
```

It should be TRUE since even though the policy state '`exist`' has NO access or rule for `pgrantee`, the business object already has a changevault access granted to `pgrantee` by `pgrantor`. `Pgrantor` is then validated against the access rule from policy state '`exist`' and it is found to be valid. Even the filter condition is satisfied since the login user (`pgrantee`) has execute access in state '`toggle`'.

Summary of User Access Goals

The left column of this table lists goals you may have for controlling user access to information and tasks. The right column summarizes what you need to do to accomplish the goal.

To accomplish this user access goal:	Do this:
Restrict access to a user interface component—such as a menu item, link, table column, or form page row—when the UI is constructed using configurable components	Configure the appropriate dynamic UI object to restrict access based on roles, access privileges, select expressions, or the result of a JPO. For details, see the *Adminstration Guide : Which Access Takes Precedence?*
Disable read access checking for a single user	In the user's person definition, make the person a Trusted user.
Grant one user's accesses for an object to another user	Have the user grant access to the other user.
Prevent a user from performing a particular task for all objects	Deny access in the user's person definition. For example, suppose you have a user who continually overrides the signature requirements for a business object. You could remove the override access from this person by including `!override` in the user's person definition. Then, even if the person is allowed this access through a policy, the access is denied.
Control access in different states of an object.	Create groups, roles, and associations that represent a set of users with shared access requirements. Define a policy that allows the public only minimum access and then assign owner and user access as required.

To accomplish this user access goal:	Do this:
Allow only a certain group, role, person, or association to be able to create a specific type of object	In the policy that governs the object type, edit the access for the first state as follows: - deny create access for the owner and public - add a user access for the user, role, group, or association and assign create access.
Allow only certain users to execute a program	Define a rule for the program. Assign execute access to the user category that needs to run the program.
Allow all users to view a form but only some to modify it	Define a rule for the form. Assign the viewform access to the public and assign the modifyform access to the user category that needs to edit the form.
Hide an attribute's value from certain users	Define a rule for the attribute. Deny read access to the public and grant it to the user category that should see the value.
Hide an attribute from all users	Make the attribute hidden in the attribute definition. When an administrative object is hidden, users don't see any evidence of it in Live Collaboration.
Allow certain users to create connections of a certain type	Define a rule for the relationship type. Assign create access to the user category that needs to create the relationship, ensuring that they also have toconnect and fromconnect in the policies governing the types at each end.
Allow certain users to remove connections of a certain type	Define an access rule for the relationship type. Assign delete access to the user category that needs to remove the relationship, ensuring that they also have todisconnect and fromdisconnect in the policies governing the types at each end.
Allow certain users to freeze and thaw relationships of a specific type, change the relationship type, and modify attributes	Define an access rule for the relationship type. Assign freeze, thaw, changetype, and modify access to the user category that needs to perform these tasks.
Allow access based on specific attribute values of an object's instance.	Define an access filter in the policy such as filter attribute[TargetCost] > attribute[ActualCost].

Working with Users and Access

Business Modeler contains four kinds of administrative objects that represent individual users and sets of users: persons, groups, roles, and associations. The primary function of these objects is to allow you to control the information users can see and the tasks they can perform.

System-Wide Password Settings

Before defining users, you should consider what your company's password policies are and set system-wide password settings to enforce them. One setting allows you to deny access in the current session to a user who makes repeated failed login attempts. Other settings allow you to control the composition of passwords. For example, you can require that users change their passwords every 90 days, that passwords be at least six characters, and that reusing the old password be prohibited.

The system-wide password settings apply to:

- Every person defined in the database, except users whose person definition includes either the No Password or Disable Password clause.
- Every attempt at setting a context, whether the attempt be in Matrix Navigator, the Business Modeler, the System application, or MQL.
- Only passwords that are created or changed after the setting is defined (except for the expiration setting which affects all passwords). For example, suppose you set the minimum password size to 4 characters. From that point on, any password entered in a user's person definition and all new passwords defined by the user in 3DEXPERIENCE platform must be at least 4 characters. Any existing password that contains less than 4 characters is unaffected. (Tip: You can make passwords for existing users conform to new system-wide password settings by making users change their passwords. Do this for all users by using the `expires` clause, or per user by using the `passwordexpired` clause.)

To setup your company's password policies, use the clauses and arguments discussed in *password Command* in the programming reference section of this guide.

Encrypting Passwords

For LDAP environments, the following MQL command encrypts a password using the same algorithm used for encrypting the bootstrap file password. After executing the command, MQL outputs the encrypted text string. Copy and paste it to the file or location where you want to save it.

```
encrypt password PASSWORD_STRING
```

For example, to encrypt the password "secret", enter:

```
encrypt password secret
```

For details on LDAP authentication, see the *Installation Guide*.

Using Special Characters in Passwords

The default cipher (crypt) for encrypting passwords handles only 7-bit ASCII characters. For passwords that use extended character sets (e.g., for letters with diaeresis symbols such as ä, ë, ö, and ü characters in the German language), you must change the system password setting to use an encryption algorithm other than UTF-8.

This can be accomplished with the following command:

```
set password cipher <cipher-type>
```

where <cipher-type> is one of `md5`, `sha`, `smd5`, or `ssha`.

Defining a Person

There are many parameters that you can enter for a person. While only a name is required, you can use the other parameters to further define the person's relationship to existing user categories, assign accesses, establish security for logging in, as well as provide useful information about the person.

Users are defined with the Add Person command:

```
add person NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the person you are creating.

`ADD_ITEM` is an Add Person clause that provides more information about the person you are creating.

The complete syntax and clauses for working with persons are discussed in *person Command* in the programming reference section of this guide.

Defining a Group or Role

The clauses for defining groups and roles are almost identical. While only a name is required, the other parameters can further define the relationships to existing users, as well as provide useful information about the group or role.

Groups and Roles are created and defined with one of following MQL commands:

```
add group NAME [ADD_ITEM {ADD_ITEM}];
add role NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the group or role you are creating.

`ADD_ITEM` is an Add Group or Role clause that provides more information about the group or role you are creating.

The complete syntax and clauses for working with groups and roles are discussed in *group Command* and *role Command* in the programming reference section of this guide.

Defining an Association

Associations are created and defined with the MQL Add Association command:

```
add association NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the association you are creating.

`ADD_ITEM` is an Add Association clause that provides more information about the association you are creating.

The complete syntax and clauses for working with associations are discussed in *association Command* in the programming reference section of this guide.

Working with Associations

If users from a combination of different groups and roles will need access to a business object or set of objects, you should create an association.

Associations allow defining signature definitions such as “a Manager in the Products group AND a member of the Engineering group OR a member of the Design group OR a Vice-President.”

Signature Definitions

Suppose you are trying to define policy signature requirements that would govern “Software Development” business objects. Suppose the Approve signature definition for promotion of business objects from the state “Design” to the state “Implement” requires that the person must be a Project Leader and member of the Kernel Engineering group.

Without an association definition, you would have to hard code names of individuals who are Project Leaders and members of the Kernel Engineering group. However, if the number of individuals with such group/role assignments is large, you would have to manually enter all such names, which could be error prone. Also this signature definition would have to be maintained as group/role assignments change.

With associations, you could define the following

```
Project Leader and Kernel Engineering
```

All individuals who are assigned a role of Project Leader and belong to the Kernel Engineering group.

Another example of an association definition:

```
Designer or Project Leader and Kernel Engineering or Management
```

With this association definition, the following individuals would have signature authority:

- All individuals who are assigned a role of Designer.
- All individuals who are assigned a role of Project Leader who also belong to the Kernel Engineering group.
- All individuals who belong to a group called Management.

Notify

The Notification facility sends messages to a group of people when a business object enters a new state.

Suppose you want to remind all Quality Assurance people associated with the testing of Assembly 101 that the customer requested an extra test.

You could accomplish this by notifying the entire Quality Assurance Group, but the message would unnecessarily go out to people who did not work on Assembly 101.

You could also accomplish this by notifying the entire Assembly 101 Group, but the message would unnecessarily go out to people who do marketing, documentation, etc. for Assembly 101.

The association feature allows you to more effectively control the recipients of the notification message. You can send the message to only the desired set of people by using the following association definition: `Quality Assurance and Assembly 101.` The only people who receive the message are those in both the Quality Assurance and Assembly 101 Groups.

Integrating with LDAP and Third-Party Authentication Tools

The 3DEXPERIENCE Platform integrates with Lightweight Directory Access Protocol (LDAP) services. You can use an LDAP service, such as openLDAP or Netscape Directory Server, as a repository to store information about users. The integration uses a toolkit from openldap.org (http://www.openldap.org/) for the underlying access protocols and is compliant with LDAPv3. The integration lets you authenticate users based on the users defined in the LDAP database. The integration lets you specify the user information to retrieve from the LDAP service, including address, comment, email, fax, fullname, groups, password, phone, and roles.

The 3DEXPERIENCE Platform also lets you authenticate users (persons, groups, or roles) with an external authentication tool instead of authenticating through Live Collaboration. The 3DEXPERIENCE Platform provides Single Sign-on when external authentication is used. This means when a user attempts to access Live Collaboration (by logging into a Business Process app or the Web version of Matrix Navigator) after having been authenticated externally, Live Collaboration allows the user access and does not present a separate login dialog.

The following limitations related to LDAP integration and/or external authentication apply:

- MatrixServletCORBA does not support external authentication.
- External authentication using LDAP integration is not supported for loosely-coupled databases.
- LDAP integration is not supported on SGI IRIX or Compaq True 64 operating systems.
- The integration works with LDAP version 2 servers but version 3 features, such as TLS/SSL, will not work when running on a version 2 server. It is recommended that you use version 3 servers.
- LDAP user names and passwords can contain special characters, but do not use the following: " , ' * (that is, double quote, comma, single quote, asterisk), since they are used within Live Collaboration as delimiters. The local operating system of the LDAP directory may have further character restrictions.

For details on how to set up integration with an LDAP service and/or an external authentication tool, see the Installation Guide.

Role-Based Visuals

Visuals (Filters, Tips, Cues, Toolsets, Tables, Views) are generally defined by users for their own personal use. They serve to set up a user's workspace in a way that is comfortable and convenient. Visuals can be used for many purposes, including organizing, prioritizing tasks, providing reminders, or streamlining access to information. Each user can define Visuals in a way that is most helpful to that person.

Visuals can also be defined and shared among users who are assigned to a role. For example, every person belonging to the role Accountant can have access to the same Visuals. This not only makes the work environment easier for a person just joining the department, but also facilitates communication among persons within a group.

For example, suppose there is a role called Manager. At a weekly manager's phone conference, each person sitting in front of a computer can, with the click of a mouse, switch Visuals so that the whole group is looking at the same thing. One could then suggest a filter to view a particular subset of objects, or look at a table, or refer to "all objects highlighted in red...."

The person setting up shared Visuals must have Business Administrator privileges.

Sharing Visuals

There are 3 methods to share visuals so that members of a group, role or association can use the visuals:

- A Business Administrator can copy the visuals from one user (person, group, role or association) to another using MQL commands.
- A Business Administrator can use the `set workspace` command to change the Workspace currently active so that the Workspace of some other User (person, group, role or association) is active, and then create visuals within that new context. See *Setting the Workspace* for details.
- Workspace objects (including visuals) can be made “visible” to other users via the MQL Visible clause.

Copying Visuals

After Visuals are defined within a session context, Business Administrators can copy the definition. If you don’t have Business Administrator privileges, then you need to be defined in the group, role, or association in order to copy from it to yourself.

The Copy command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy VISUAL SRC_NAME DST_NAME [fromuser USER] [touser USER] [overwrite] [MOD_ITEM
{MOD_ITEM}];
```

`VISUAL` can be any one of the following: `filter, tip, cue, toolset, table, view.`

`SRC_NAME` is the name of the visual definition (source) to copied.

`DST_NAME` is the name of the new definition (destination).

`USER` can be the name of a person, group, role, or association.

`Overwrite` replaces an existing visual (or member, in the case of views) in the destination user.

The order of the FromUser, ToUser and Overwrite clauses is irrelevant, but MOD_ITEMS, if included, must come last.

`MOD_ITEM`s are modifications that you can make to the new definition. `MOD_ITEM`s vary depending on which visual you are copying. A complete list can be found in the sections that follow.

For example, you can copy a cue definition using the following command:

```
copy cue RedHiLite RedHiLite fromuser purcell touser Engineer
```

This command copies a Cue named RedHiLite from the person purcell to the role Engineer.

If an error occurs during the copying of a Visual, the database will be left intact.

Setting the Workspace

The `set workspace` command allows you to change the Workspace currently active in MQL so that the Workspace of some other User (person, group, role or association) is visible.

While `set workspace` still works as described below, in version 10.6 and higher, it is no longer necessary because the business administrator can refer to any user's workspace objects by appending 'user X' to the name, so adds/modifies/deletes can be done directly. For example:

```
set context user creator;
add cue c1 user P1 type t2 name * revision * color blue;
mod cue c1 user P1 color red;
print cue c1 user P1;
del cue c1 user P1;
```

Users can change to the Workspace of groups, roles, and associations to which they belong. Business Administrators can change to the Workspace of any group, role, association, or person.

The syntax for this command is:

```
set workspace user USERNAME
```

This command affects the behavior of all commands to which Workspace objects are applicable, including:

- all commands specific to Workspace objects (for example, add/modify/delete filter)
- `expand bus` (affects what filters are used to control output)

`USERNAME` is the name of a person, group, role or association.

When users (other than Business Administrators) set their Workspace to that of a group, role, or association, they cannot use commands that modify the Workspace, that is, the `add`, `modify`, and `delete` commands for Workspace objects. This restriction enforces the rule that only Business Administrators are permitted to change the Workspace of a group, role or association.

Note that `set context user USERNAME` will change the context to that of `USERNAME` regardless of an earlier invocation of `set workspace`.

Multiple/Mono Security Context Enforcement

Multiple Access Definitions Per User

It is possible to specify multiple access definitions for the same user in policies and rules. This makes it possible to split access definitions using complex filter expressions into several smaller and simpler access definitions.

For example, the following access definition:

```
policy "VPLM_SMB"
   state "SHARED"
      user "VPLMProjectLeader"
         read,show,checkout,promote,demote,changeowner,unlock,
          toconnect,todisconnect, revise
         filter "(organization.ancestor match
                   context.user.assignment[$CHECKEDUSER].org) &&
                   (($ACCESS matchlist 'read,show,checkout' ',') ||
                  (project ==
                   context.user.assignment[$CHECKEDUSER].project))"
```

filters data depending on both their organization and project ownership in a similar way to the example shown in *Organization and Project-Based Access*, with a small but significant diffference. It checks whether the organization of the object (`organization`) or any of its parent (`organization.ancestor`) matches the organization component of the security context granting the access definition (`context.user.assignment[$CHECKEDUSER].org`; for example, `Company Name` if the user is assigned context `VPLMDesign.Company Name.Engineering`).

It also checks whether the project of the object (`project`) matches the project component of the security context granting the access definition (`context.user.assignment[$CHECKEDUSER].project`; for example, `Engineering` if the user is assigned context `VPLMDesign.Company Name.Engineering`), but only for accesses other than `read`, `show`, and `checkout` (`$ACCESS matchlist 'read,show,checkout'`, `$ACCESS` being a symbolic variable corresponding to the access being checked).

Taking advantage of the kernel matching options described in *Organization and Project-Based Access*, that access definition could be implemented as follows:

```
policy "VPLM_SMB"
   state "SHARED"
      user "VPLMProjectLeader"
         read,show,checkout,promote,demote,changeowner,unlock,
         toconnect,todisconnect,revise
         ancestor org
         filter "($ACCESS matchlist 'read,show,checkout' ',') ||
                   (project ==
                    context.user.assignment[$CHECKEDUSER].project)"
```

Because the filtering on the project of the object depends on the access being checked, it is not possible to model it using the matching option `single project`, as was done above, and consequently the access definition still contains the filter expression. However, if this access definition can be split into two as follows:

```
policy "VPLM_SMB"
   state "SHARED"
      user "VPLMProjectLeader"
         key "read"
         read,show,checkout
         descendant org
      user "VPLMProjectLeader"
         key "others"
         descendant org
         single project
         promote,demote,changeowner,unlock,toconnect,todisconnect,
         revise
```

Then there are now two separate access definitions for the same user `VPLMProjectLeader`. The first access definition contains only `read`, `show`, and `checkout` accesses, and only specifies a matching option on the organization. The second access definition contains the other accesses, and specifies both organization and project matching options.

In this way, it is possible to specify access definitions with separate sets of accesses and thus to eliminate the remaining filter expression:

```
$ACCESS matchlist 'read,show,checkout' ',') || (project ==
     context.user.assignment[$CHECKEDUSER].project)
```

The `key` option is used to name each access definition to indicate that there are actually two distinct entries. Otherwise, the kernel would think that there was only one per user and overwrite it with subsequent access definitions.

Mono-Security Context Enforcement

The `login` keyword may be specified in policy and rule access definitions in place of the `user` keyword to indicate that the following definition must apply only to the security context the user selected when connecting to the 3DEXPERIENCE Platform (the login or current security context).

This makes it possible to specify mono-context access definitions that are compatible with the new organization and project matching options and multiple access definition per user as described in *Multiple Access Definitions Per User*.

For example, the following access definition:

```
policy "VPLM_SMB"
   state "WAITAPP"
      user "VPLMCreator"
         read,show,changeowner,checkout,toconnect,todisconnect
         filter "(organization.ancestor match
                   context.role[$CHECKEDUSER].org) && (project ==
                   context.role[$CHECKEDUSER].project)"
```

can be implemented as follows:

```
policy "VPLM_SMB"
   state "WAITAPP"
      login "VPLMCreator"
         read,show,changeowner,checkout,toconnect,todisconnect
         descendant org
         single project
```

Owner and Public Access Definition Enhancement

Owner and public policy and rule access definitions have been enhanced with the same capabilities as regular user-based access definitions, including:

- Filter expressions
- Organization and project matching
- Mulitple access definitions per user
- Revoke access

Working with Context

Setting *context* identifies the user and the areas of access the current user maintains. For example, setting context to a person who is defined as a Business Administrator allows access to Business Administrator functions such as adding a Type. In addition, a default vault is associated with context so that newly created objects are assigned to that user's typical vault (unless specified otherwise).

Personal settings that a user creates such as saved sets, queries, views and visuals (filters, cues, tips, toolsets and tables) are accessed only when context is set to that person.

Context is defined by persons' names and vault they are working on. As described in *Vaults* in Chapter 2, a vault defines a grouping of objects. For example, a vault may contain all the information and files associated with a particular project, product line, geographic area, or period of time.
By default in MQL, context is set to "guest," assuming the user "guest" has not been deleted, made inactive, or assigned a password. If MQL does not (cannot) set a context as guest, no default context is set.

Setting Context

Context identifies a person and indicates the type of person (such as a System Administrator), and optionally, provides security with a password. Any user can set context, but restrictions apply based on the type of person and password. For example, only a System Administrator can perform System Administrator functions.

Setting the context of a user implies that you are the user. Once the context is set, any commands you enter are subject to the same policies that govern the defined person. This is useful when you need to perform a large number of actions for a defined user.

For example, assume you want to include a person's files in the database. When you include them, you want the person to maintain ownership. Also, you do not want to create objects the person cannot access or perform actions prohibited to the person. You need to *act as* the person when those files are processed. In other words, you want to identify yourself as the person in question so that the actions you take appear to have been done by the actual owner of the files.

Context is controlled with the Set Context command which identifies a user to by specifying the person name and vault:

```
set context [ITEM {ITEM}];
```

`ITEM` is a Set Context clause. Each clause and the arguments they use are discussed in *context Command* in the programming reference section of this guide.

Setting Context With Passwords

When a person is added to the database, the Business Administrator can include a Password clause as part of the person's definition. This clause assigns a password to the person. Once assigned, the password is required to access this person's context (unless the password is removed).

The password should be kept secret from all unauthorized users. If the defined person never shares its password with any other user, the effect is the same as using the Disable Password clause in the

person's definition (refer to *Disable Password Clause*). Use the following Set Context command if a person is defined as having a password:

```
set context person PERSON_NAME password VALUE [vault VAULT_NAME];
```

PERSON_NAME is the name of a user defined in the database.

VALUE is the password value assigned to the named person in the person definition that was created by the Business Administrator.

VAULT_NAME is a valid vault defined in the database.

In this command, you must enter both a person name and the password associated with the person. If either value is incorrect, an error message is displayed. However, if you are the Business Administrator, you can bypass a defined password. If you are assigned a user type of Business Administrator, you can change your context to that of another person by entering the following command:

```
set context person PERSON_NAME [vault VAULT_NAME];
```

For example, assume a person is defined as follows:

```
add person mcgovern
    fullname "Jenna C. McGovern"
    password PostModern
    assign role Engineer
    assign group "Building Construction"
    vault "High Rise Apartments";
```

If you are defined as a Full User and want to set your context to mcgovern, you would enter:

```
set context user mcgovern password PostModern;
```

In this case, the Password clause must be included in the Set Context command. If you are defined as a Business Administrator, you can set your context to mcgovern by entering:

```
set context person mcgovern;
```

No password is required even though a password was assigned. For more information on the different user types, see *Type Clause*.

Changing Your Password

You can change your password as you set context using the keyword newpassword within the Set Context clause. Use this keyword with the user or person keywords and the password keyword. Enter the new password after the keyword newpassword. If the change is successful, context will be set as well. For example, the following MQL command will set context to the user mcgovern and change the user's password from "Jurassic" to "PostModern".

```
set context user mcgovern password Jurassic newpassword
PostModern;
```

No Password Clause

When a person is defined with a *No Password Clause*, anyone can set context to that person name. Since no password is required, the Set Context command is:

```
set context person PERSON_NAME [vault VAULT_NAME];
```

For example, assume you want to access the business objects created by a person named MacLeod. To do this, you enter:

```
set context person macleod;
```

After this command is executed, you have the same privileges and business objects as MacLeod.

Setting Context With Disabled Passwords

When a person is defined with a *Disable Password Clause*, the security for logging into the operating system is used as the security for setting context. When a user whose password is disabled attempts to set context, the system compares the user name used to log into the operating system with the list of persons defined in the 3DEXPERIENCE. If there is a match, the user can set context without a password. (The context dialog puts the system user name in as default, so the user can just hit enter.) If they do not match, the system denies access.

When Disable Password is chosen for an existing person, Live Collaboration modifies the password so that others cannot access the account. This means that the user with a disabled password can only log in from a machine where the O/S ID matches the 3DEXPERIENCE ID. This is similar to the way automatic SSO-based user creation is handled. To re-enable a password for such a person, create a new password for the person as you normally would.

Temporarily Setting Context

The context settings that the user provides at login time define what types of accesses that user has to Live Collaboration. Programs, triggers and wizards may be available to users who do not have appropriate access privileges to run them, so context must be changed within the program and then changed back to the original user's context when the program completes.

For example, a trigger program may need to switch context to perform some action which is not allowed under the current user context. It must then return to the original context to prohibit invalid access or ownership for subsequent actions.

Two MQL commands are available that are useful in scripts that require a temporary change to the session context.

Initialization Context Variable

There is an initialization file variable that controls whether context is restored or not at the termination of a program. If the `MX_RESTORE_CONTEXT` variable is set to `true`, the original user's context is restored after the program/wizard/trigger terminates. If set to `false`, any context changes made by a program/wizard/trigger will remain changed after the program terminates. This ensures that the appropriate action is taken if there is a failure within the program before the `pop context` command executes.

Organization and Project-Based Access

Syntax options are provided in the Policy and Rule access definitions to support filtering by organization and project. This option makes it possible to specify whether organization and/or project should be matched against the security context(s) of the user, and whether ascendants and descendants should also be checked. For example, the following access definition filters data depending on both the organization and project ownership:

```
policy "VPLM_SMB"
   state "WAITAPP"
      user "VPLMCreator"
         read,show,changeowner,checkout,toconnect,todisconnect
         filter "(organization.ancestor match
                     context.user.assignment[$CHECKEDUSER].org) &&
                  (project ==
                     context.user.assignment[$CHECKEDUSER].project)"
```

This checks whether the organization of the object (`organization`) or that of its parent (`organization.ancestor`) matches the organization component of the security component granting the access definition (`context.user.assignment[$CHECKEDUSER].org`; for example, `Company Name` if the user is assigned context `VPLMDesign.Company Name.Engineering`).

It also checks whether the project of the object (`project`) matches the project component of the security context granting the access definition (`context.user.assignment[$CHECKEDUSER].project`; for example, `Engineering` if the user is assigned context `VPLMDesign.Company Name.Engineering`).

Organization and project-based filtering is the main recurring pattern in most access definitions. Taking advantage of the kernel-matching options, the access definition is implemented as follows:

```
policy "VPLM_SMB"
   state "WAITAPP"
      user "VPLMCreator"
         read,show,changeowner,checkout,toconnect,todisconnect
         descendant org
         single project
```

The filter expression:

```
organization.ancestor match context.user.assignment[$CHECKERUSER].org
```

has been replaced with the option `descendant org`, which specifies that the organization of the object must match that of the security context granting the access definition (`org`) or any of its descendants (`descendant`).

Similarly, the filter expression:

```
project == context.user.assignment[$CHECKEDUSER].project
```

has been replaced with the option `single project`, which specifies that the project of the object must exactly match that of the security context granting the access definition (`single project`).

Ultimately, all of the contents of the filter expression of the original access definition:

```
organization.ancestor match context.user.assignment[$CHECKEDUSER].org)
&& (project == context.user.assignment[$CHECKEDUSER].project
```

are defined using the new options, so the new access definition has no filter at all.

The complete set of organization and project matching options is listed in the following table:

Possible Options for STATE_ITEM/ STATE_MOD_ITEM	Description
`any org`	The value of `organization` is not checked.
`single org`	The object organization must exactly match the organization of the security context granting the access. The value of `organization` must match the login role or assigned security context.
`ancestor org`	The object organization must be an ancestor (parent organization) of the organization of the security context granting access. The ancestor of `organization` must match the login role or assigned security context.
`descendant org`	The object organization must be a descendant (child organization) of the organization of the security context granting the access.
`any project`	The value of `project` is not checked.
`single project`	The object project must exactly match the project of the security context granting the access. The value of `project` must match login role or assigned security context
`ancestor project`	The object project must be an ancestor (parent object) of the project of the security context granting the access. The ancestor of `project` must match login role or assigned security context.
`descendant project`	The object project must be a descendant (child project) of the project of the security context granting the access.

MQL also allows you to specify organization and project-based access in commands with a State clause, such as Modify Businessobject and Add Policy.

Possible Values for STATE_ITEM/ STATE_MOD_ITEM	Description
`login`	May be used in place of `user` to represent an `ACCESS_ITEM` associated with the user's login role, as opposed to their assignment list.
`key`	May be used to distinguish between multiple ACCESS_ITEMs with the same USER_NAME (owner, or public).

Possible Filter for STATE_ITEM/ STATE_MOD_ITEM	Description
`[filter EXPRESSION]`	Allowed for owner or public.

Enhanced Selectables on Access Rules

A first-class object has been added to the select interface to represent a user (owner, or public). You can navigate to this object using the following fields:

```
policy[POLICY_NAME].state[STATE_NAME].owner[KEY]
policy[POLICY_NAME].state[STATE_NAME].public[KEY]
policy[POLICY_NAME].state[STATE_NAME].user[USER_NAME|KEY]
```

Since `KEY` can include wildcards, a `KEY` of '*' selects all access items for owner, public, or a named user. Once the targeted user has been selected, the following fields are available:

Field	Description
`organization`	Returns `any`, `single`, `ancestor`, or `related`, depending on which option was selected.
`project`	Returns `any`, `single`, `ancestor`, or `related`, depending on which option was selected.
`access`	Returns the user's access mask.
`filter`	Returns the user's filter expression.
`revoke`	Returns TRUE if the access item denies access, FALSE if it grants access.
`key`	Returns the user-assigned identifier string for this access item.
`login`	Once the targeted user has been selected, the login field is also available. Returns TRUE if the access item applies to the login role, FALSE if the access item applies to the user assignment.

For details, see *3DEXPERIENCE Open Configuration : Appendix: Selectables* in the online documentation.

4 Working with Metadata

This chapter explains the metadata available in the 3DEXPERIENCE Platform.

In this section:

Attributes

An *attribute* is any characteristic that can be assigned to an object or relationship. Objects can have attributes such as size, shape, weight, color, materials, age, texture, and so on.

You must be a Business Administrator to add or modify attributes. (Refer also to the Business Modeler Guide.)

Defining an Attribute

Before types and relationships can be created, the attributes they contain must be defined as follows:

```
add attribute NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the attribute.

`ADD_ITEM` is an Add Attribute clause that provides additional information about the attribute you are creating.

The complete syntax and clauses for working with attributes are discussed in *attribute Command* in the programming reference section of this guide.

Assigning Attributes to Objects

In the 3DEXPERIENCE Platform, objects are referred to by type, name, and revision. Object types are defined by the attributes they contain. When an object is created, you specify the object type and then Live Collaboration prompts for attribute values for that object instance.

For example, assume the object is clothing. It might have attributes such as article type (for example, pants, coat, dress, shirt, and so on), size, cost, color, fabric type, and washing instructions. Now assume you are creating a new article of clothing. When you create this object, Live Collaboration prompts you to provide values for each of the assigned attributes. You might assign values such as jacket, size 10, $50, blue, wool, and dry clean.

The specific value for each attribute can be different for each object instance. However, all objects of the same type have the same attributes.

Assigning Attributes to Relationships

Like business objects, a relationship may or may not have attributes. Since a relationship defines a connection between two objects, it has attributes when there are characteristics that describe that connection.

For example, an assembly could be connected to its components with a relationship called, “component of,” which could have an attribute called, “quantity.” When the component and the assembly are connected, the user would be prompted for the quantity or number of times the component is used in the assembly.

The same attributes could apply to either the objects or the relationship. When an object requires additional values for the same attribute in different circumstances, it is easier to assign the attributes to the relationship. Also, determine whether the information has more meaning to users when it is associated with the objects or the relationship.

Assigning Attribute Types

The Type clause is always required. It identifies the type of values the attribute will have. An attribute can assume one of five different types of values: string, Boolean, Real, Integer, Date. When determining the attribute type, you can narrow the choices by deciding if the value is a number or a character string. Once a type is assigned to an attribute, it cannot be changed. The user can associate only values of that type with the attribute.

Assigning Attribute Ranges

You can define a range of values the attribute can assume. This range provides a level of error detection and gives the user a way of searching a list of attribute values. If you define an attribute as having a specific range, any value the user tries to assign to that attribute is checked to determine if it is within that range. Only values within all defined ranges are allowed. There are several ways to define a range:

- Using a relational operator, such as is less than (“<”) or is equal to (“=”)
- Using a range with a pattern, or special character string, such as starts with A (“A*”)
- Using multiple ranges to provide a list of allowed values
- Using a program to define a range depending on conditions

See also *Multi-Value and Range-Value Attributes*.

Assigning a Default Value

When a business object is created and the user does not fill in the attribute field, the default value is assigned. When assigning a default value, the value you give must agree with the attribute type. If the attribute should contain an integer value, you should not assign a string value as the default.

For example, assume you want to define an attribute called PAPER_LENGTH. Since this attribute will specify the size of a sheet of paper, you defined the type as an integer. For the default value, you might specify 11 inches or 14 inches, depending on whether standard or legal size paper is more commonly used.

Applying a Dimension to an Existing Attribute

You can add a dimension to attributes of type integer or real, whether or not the attributes already belong to instantiated business objects. The existing value stored for that attribute will be treated as the normalized value.

However, if the existing value is a unit other than what the dimension uses as its default, you need to convert those values. For example, an existing attribute Length has stored values where the business process required the length in inches. You want to add a Length dimension to the Length attribute, and that dimension uses centimeters as the default. If you just add the dimension to the attribute, the existing values will be considered as centimeters without any conversions.

As another example, the Length attribute can be used with multiple types. Some types may use the attribute to store lengths in millimeters, while others store lengths in meters. Because there is only one attribute Length that represents two kinds of data, the dimension cannot be applied without normalizing the business types on the same unit.

Use the convert attribute command to convert the attribute to 1 unit of measure.

```
mod attribute ATTRIBUTE_NAME dimension DIMENSION_NAME;
```

Replace `ATTRIBUTE_NAME` with the attribute name and `DIMENSION_NAME` with the dimension name.

Specify the Existing Units of the Attribute

```
convert attribute ATTRIBUTE_NAME to unit UNIT_NAME on temp query bus
"TYPE" * *;
```

This command applies the unit `UNIT_NAME` to the attribute, which causes the conversion from the existing value with the applied units to normalized values.

The system displays this message:

```
Convert commands will perform mass edits of attribute values.
You should consult with MatrixOne support to ensure you follow
appropriate procedures for using this command.
Proceed with convert (Y/N)?
```

If the system message includes the following warning,

```
WARNING: The search criteria given in the convert command includes objects
that currently have unit data. This convert command will overwrite that
data.
```

the data for this attribute has already been converted and you should not continue (type N).

Changing the Default (Normalized) Units of a Dimension

The Weight attribute included in Business Process Services, is defined to have a dimension containing units of measure with the default units set to grams. If your company uses the Weight attribute representing a different unit, you have these options:

- Convert existing values to use the new dimension with its default as described in *Applying a Dimension to an Existing Attribute*.
- Change the Dimension's default units, offsets, and multipliers as described in this section

After applying a dimension, you can only change the default units of measure before any user enters any data for the attribute. After values are stored in the as entered field, you cannot change the default units of measure.

To change a dimension's default units

1. Using Business Modeler, locate the dimension and open it for editing.
2. Click the **Units** tab.
3. For the unit you want to use as the default:
   - **a)** Highlight the Unit.
   - **b)** Change the multiplier to 1.
   - **c)** Change the offset to 0.
   - **d)** Check the **Default** check box.
   - **e)** Click **Edit** in the Units section.
4. For all other units:
   - **a)** Highlight the Unit.
   - **b)** Change the multiplier to the appropriate value.
   - **c)** Change the offset to the appropriate value.

d) Click **Edit** in the Units section.

5. Click **Edit** at the bottom of the dialog box.

Multiple Local Attributes

It is possible to manage local attributes with the same name defined on multiple interfaces on the same object/relationship instance. The objects/relationships can also have a global or local attribute with the same attribute name.

In V6R012 and earlier, it was not possible to add an interface to a business object if the business object hierarchy had a local attribute with the same name as one of the interface attributes. For example:

If Type `T1` owned a local attribute named `a1`, and Interface `I1` also owned a local attribute named `a1`, attempting to add interface `I1` to a business object of type `T1` generated the following error:

```
Error : #1500779 : Conflicting local attribute 'a1' on interface 'I1'
```

In V6R2012x and later, this limitation no longer applies. The above scenario is allowed, as shown in the following MQL session. Note the naming convention for the local attributes—`I1.a1` denotes the local attribute `a1` belonging to interface `I1`.

```
MQL<35>add type T1;
MQL<36>add attribute a1 type string default a1-def-value;
MQL<37>mod type T1 add attribute a1;
MQL<38>add interface I1 type T1;
MQL<39>add attribute a1 type string owner interface I1 default
I1-a1-def-value;
MQL<40>add bus T1 test 0 vault unit1 policy simple-u;
MQL<41>mod bus T1 test 0 add interface I1;
MQL<42>print bus T1 test 0 select attribute;
business object T1 test 0
    attribute = a1
    attribute = I1.a1
MQL<43>print bus T1 test 0 select attribute.value;
business object T1 test 0
    attribute[a1].value = a1-def-value
    attribute[a1].value = I1-a1-def-value
MQL<44>print bus T1 test 0 select attribute[a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
    attribute[a1] = a1-def-value
MQL<45>print bus T1 test 0 select attribute[*a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
    attribute[a1] = a1-def-value
MQL<46>print bus T1 test 0 select attribute[I1.a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
```

In V6R2012 and earlier, it was also not possible to add two interfaces on a business object if both interfaces owned a local attribute with the same name. For example:

If Type `T1` owned some attribute, but no `a1`, Interface `I1` owned a local attribute named `a1`, and Interface `I2` owned a local attribute named `a1`,

there was no way to add interfaces `I1` and `I2` to a business object of type `T1`, as attempting to add interface `I2` generated the following error:

```
Error : #1500779 : Conflicting local attribute 'a1' on interface 'I2'
```

In V6R2012x and later, this limitation no longer applies. The above scenario is allowed, as shown in the following MQL trace:

```
MQL<28>add type T1;
MQL<29>add interface I1 type T1;
MQL<30>add interface I2 type T1;
MQL<31>add attribute a1 type string owner interface I1 default
I1-a1-def-value;
MQL<32>add attribute a1 type string owner interface I2 default
I2-a1-def-value;
MQL<33>add bus T1 test 0 policy simple-u vault unit1;
MQL<34>mod bus T1 test 0 add interface I1;
MQL<35>mod bus T1 test 0 add interface I2;
MQL<36>print bus T1 test 0 select attribute;
business object T1 test 0
    attribute = I1.a1
    attribute = I2.a1
MQL<37>print bus T1 test 0 select attribute.value;
business object T1 test 0
    attribute[a1].value = I1-a1-def-value
    attribute[a1].value = I2-a1-def-value
MQL<38>print bus T1 test 0 select attribute[a1];
business object T1 test 0
    attribute[a1] = I2-a1-def-value
    attribute[a1] = I1-a1-def-value
MQL<39>print bus T1 test 0 select attribute[*a1];
business object T1 test 0
    attribute[a1] = I2-a1-def-value
    attribute[a1] = I1-a1-def-value
MQL<40>print bus T1 test 0 select attribute[I1.a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
MQL<41>print bus T1 test 0 select attribute[I2.a1];
business object T1 test 0
    attribute[a1] = I2-a1-def-value
```

Attribute Names

A fully qualified name includes both the object name and the attribute name, using the format *ADMIN.ATTRIBUTE*. An unqualified name includes the attribute name only. The only time the fully qualified name of an attribute is used is when `select attribute` is specified. Otherwise, the display name is always the unqualified attribute name. This enables multiple attributes to have the same name, because if the names are displayed "fully qualified," (i.e., `T1.a1`), they do not appear as having the same name.

No matter how the selectable is written (`attribute[X]` where `x` = `I1.a1`, `I2.a1`, `a1`, etc.), the returned attribute is always the simple/common attribute name. The correct value is always returned if you select using the fully qualified local attribute name (*ADMIN.ATTRIBUTE*, e.g., `T1.a1`), for example:

```
MQL<219>print bus tp1 test2 1 select attribute[i1.*];
business object tp1 test2 1
    attribute[i1.a1] = 3DPLM
MQL<220>print bus tp1 test2 1 select attribute[].value;
business object tp1 test2 1
    attribute[tp1.a1].value = 3DPLM
    attribute[a1].value = geo
    attribute[i1.a1].value = 3DPLM
MQL<221>print bus tp1 test2 1 select attribute[a1];
business object tp1 test2 1
    attribute[tp1.a1] = 3DPLM
    attribute[i1.a1] = 3DPLM
    attribute[a1] = geo
```

One limitation of this feature is that there is no way to select only local attributes. For example, specifying `attribute[*.a1]` returns an error:

```
MQL<55>print bus T1 test 0 select attribute[*.a1];
Error: #1900068: print business object failed
Error: #1500063: Unknown field name 'attribute[*.a1]'
```

However, you can use '*' to match all global and local attributes matching a pattern, for example:

```
MQL<4>add type k1;
MQL<5>add interface k1 type k1;
MQL<6>add attribute ak1 type string owner interface k1;
MQL<7>add interface k2 type k1;
MQL<8>add attribute ak1 type string owner interface k2;
MQL<9>add attribute ak1 type string;
MQL<10>mod type k1 add attribute ak1;
MQL<11>add bus k1 test 0 vault unit1 policy simple-u;
MQL<12>mod bus k1 test 0 add interface k1 add interface k1 add
interface k2;
MQL<13>print bus k1 test 0 select attribute [*k1];
business object k1 test 0
   attribute[k2.ak1] =
   attribute[ak1] =
   attribute[k1.ak1] =
```

Using Patterns for Attribute Names in Temp Query Bus Where clause

When a pattern is provided for an attribute name in a `temp query bus' where` clause, the pattern is resolved against all attributes, not just attributes available for specific objects. This can result in unsupported criteria when using local attributes. For example, in a database with attributes of different derivations of names beginning with 't', the following occurs:

```
MQL<43>add attr t3 type integer;
MQL<44>add attr t4 type string;
MQL<45>list attr t*;
t3
t4
MQL<46>add type typ1 attr t4;
MQL<47>add policy pol1 type typ1 state s1;
MQL<48>add bus typ1 aaa 1 policy pol1 vault v1 t4 abc;
```

```
MQL<49>temp query bus typ1 * * where 'attribute[t*] ~~ "*b*"';
Warning: #1500802: Different attribute types are found in expanded
attribute list
```

Multi-Value and Range-Value Attributes

Attributes can have multiple values and can also specify a range of values.

Multi-value attributes also support the existing concept of ranges.

Multi-Value Attributes

Traditionally, an attribute can specify only one value. However, in some cases it may be necessary for an attribute to have multiple values, which could be stored as comma-separated values. This would require additional logic in the form of a workaround in order to parse the comma-separated values and display them properly. However, this could in turn lead to issues when searching or indexing the data, as the kernel might not be aware of the workaround convention.

To deal with these issues, MQL provides the following support for multi-value attributes:

- Multi-value attributes can be applied to any attribute type, including date, integer, real, boolean, and string.
- Multi-value attributes can define an order. An order number is used to display the attritbue values in a specific order.
- Dimension can be applied to the numeric fields.
- Each value can have its own unit of measure when Unit Of Measure (UOM) is applicable.
- Searching is performed against each value separately to determine a match.
- A multi-value attribute can have more than one instance of the same value.

You define an attribute to be multi-value by specifying its type as `multivalue` at the time of creation. You can also provide an order number when setting a multi-value attribute so that its value is positioned correctly.

To define an attribute as multi-value, use the command:

```
add attribute NAME multivalue;
```

To define an attribute as single-value, use one of the commands:

```
add attribute NAME notmultivalue;
add attribute NAME !multivalue;
add attribute NAME;
```

By default, attributes are single-value. It is not allowed to specify default values for multi-value attributes. Import and export can handle multi-value attribute definitions and instances.

The lxString table indexes have been changed as follows:

- The index on lxType, lxVal, lxOid is now only LxType, lxVal. In addition, the index order is lxType, lxVal on all databases.
- The unique index on lxOid, lxType includes the non-key column lxVal on DB2 and SQL Server.

See also *attribute Command*.

Range-Value Attributes

While multi-value attributes allow you to enter distinct values, range-value attributes allow you to enter maximum and minimum values (for example, to model a range of 100%-20%). This capability is enabled for numeric fields as well as dates (for example, to model a start and end date).

MQL provides the following support for range-value attributes:

- Range-value attributes can be applied to numeric and date attributes.
- Dimension can also be applied to the numeric fields.
- The range can have only one UOM value.
- Searching is performed against both values.
- When setting values for a range-value attribute, you can specify either endpoint as inclusive (by default), exclusive (using the keywords minexclude and maxeclude), or open-ended (if no value is specified).

You define an attribute as having a range by specifying its type as `rangevalue` at the time of creation. Only numeric and date attribute types can be defined as range-value attributes. If you attempt to define a string or boolean attribute type as `rangevalue`, an error will be thrown.

To define an attribute as specifying a range of values, use the command:

```
add attribute NAME rangevalue;
```

To define an attribute as not specifying a range, use one of the commands:

```
add attribute NAME notrangevalue;
add attribute NAME !rangevalue;
add attribute NAME;
```

By default, attributes do not specify a range of values. It is not allowed to specify default values for range-value attributes. Import and export can handle range-value attribute definitions and instances.

See also *attribute Command*.

Upgrading the Database to Add Support for Multi-Value Attributes

An upgrade of the database to V6R2013x or later is required in order to enable support for multi-value and range-value attributes. If the database is not upgraded and you attempt to add a multi-value or range-value attribute, an error will be thrown. For more information, see the *Installation Guide: Upgrading the Database After Installing a New Version of Software*.

The upgrade adds a new table in the database for integer, real, date, boolean, and string attributes. This new table supports both multi-value and range-value attributes. In addition, the existing unit table is modified to add a new order column, which allows each value of a multi-value attribute to have its own UOM value.

Modifying Multi-Value Attributes

To modify an existing attribute to become multi-value, use the command:

```
modify attribute NAME multivalue;
```

Only existing `singlevalue` attributes can be modified to become `multivalue`. If the attribute is already a `rangevalue`, an error will be thrown.

If there are instances of a `singlevalue` attribute in the database, the modification moves all instances from the existing attribute tables to the new multi-/range-value attribute tables in the database.

If an attribute is already `multivalue` and has multiple values (instances), changing it to `singlevalue` is not allowed and will cause an error to be thrown. For example, you can use either of the following commands to change a `multivalue` attribute to `singlevalue`:

```
modify attribute NAME notmultivalue;
modify attribute NAME !multivalue;
```

The above is OK if the attribute is already either `multivalue` or `rangevalue` and instances exist.

```
modify attribute NAME rangevalue;
```

The above is OK if the attribute is not already `multivalue`. In other words, if it is a `singlevalue` attribute with no instances, then this is permitted.

```
modify attribute NAME notrangevalue;
modify attribute NAME !rangevalue;
```

The above is OK if the existing `rangevalue` attribute does not have any instances in the database. If the attribute is already specified as `multivalue`, an error will be thrown. The following table shows a summary of what is permitted and what is not:

Modification	Allowed?
`singlevalue` (without any instances) -> `rangevalue`	Yes
`singlevalue` (with instances) -> `rangevalue`	No
`singlevalue` (without any instances) -> `multivalue`	Yes
`singlevalue` (with instances) -> `multivalue`	Yes
`rangevalue` (without any instances) -> `singlevalue`	Yes
`rangevalue` (with instances) -> `singlevalue`	No
`rangevalue` (without any instances) -> `multivalue`	No
`rangevalue` (with instances) -> `multivalue`	No
`multivalue` (without any instances) -> `singlevalue`	Yes
`multivalue` (with instances) -> `singlevalue`	Yes (as long as there is only one value for each instance)
`multivalue` (without any instances) -> `rangevalue`	No
`multivalue` (with instances) -> `rangevalue`	No

Adding/Modifying Business Objects/Connections for Multi-Value and

Range-Value Attributes

Once a multi-value or range-value attribute has been added to a type or relationship, you can add business objects and connections with attribute values. You can provide a comma-separated list of values for a multi-value attribute using one of the following methods:

```
add bus T N R ATTR_NAME 1,2,3;
mod bus T N R ATTR_NAME 3,4,5;
mod bus T N R ATTR_NAME test1,test2,test3;
```

If the values themselves have spaces or commas, then they must be passed in single or double quotes using one of the following methods:

```
mod bus T N R ATTR_NAME 'foo bar1','foo bar2','foo bar3';
mod bus T N R ATTR_NAME "foo bar1","foo bar2","foo bar3";
```

Attempting to provide a comma-separated list of values for an attribute that is not multi-value will cause an error to be thrown.

For range-value attributes, the keywords `minexclude` and `maxexclude` indicate that the range endpoints do not include the minimum and maximum values, respectively. This is similar to interval notation in algebra.

Range-value attributes can have the following possible values:

Values	Description
minval::maxval For example: 5::10	• Both minval and maxval are included for any query operation. In the example, it considers the value of the range from 5 to 10 (i.e., including 5 and 10). • The selectables `includeminval` and `includemaxval` both return TRUE.
minval::maxval minexclude For example: 5::10 minexclude	• Only maxval is included for any query operation. In the example, it considers the value of the range from 6 to 10 (excluding 5). • The selectable `includeminval` returns FALSE.
minval::maxval minexclude maxexclude For example: 5::10 minexclude maxexclude	• Both minval and maxval are excluded for any query operation. In the example, it considers the value of the range from 6 to 9 (excluding 5 and 10). • The selectables `includeminval` and `includemaxval` both return FALSE.
::maxval [|maxexclude] For example: ::10	• The value of the range is considered from the negative system limit for integer or float values (e.g., from -2,147,483,648 to +10). • The selectable `openminval` returns TRUE.
minval:: [|minexclude] For example: 5::	• The value of the range is considered from minval up to the positive system limit for integer or float values. • The selectable `openmaxval` returns TRUE.

If a business object has to set a range-value attribute to have open-ended minimum and maximum values, then one of the values does not need to be provided.

The following table shows some examples of adding/modifying range-value attributes for a business object.

`add bus T N R intattr 1::3 minexclude maxexclude;`	The attribute `intattr` is stored in the database as two separate table rows with values 1 and 3, and orders 1 and 2, respectively.
`add bus T N R intattr ::3;`	The minval is open in this case.
`add bus T N R intattr 3::;`	The maxval is open in this case.
`mod bus T N R intattr add 4::5;`	Multiple values are not allowed for range-value attributes. Since the attribute already has a range of 1::3, attempting to add a second range will throw an error.
`mod bus T N R intattr 4::5;`	This range will replace the existing range value of 1::3 with the range value of 4::5.

The following is an example of using a combination of single-value, multi-value, and range-value attributes together with other business objects in a single MQL command:

```
mod bus T N R singleattr 2 multiattr test1,test2,test3 rangeattr 1::5
  policy p multiattr add test4;
```

The above command has the following effect:

1. The single-value attribute named `singleattr` is assigned the value 2.
2. The multi-value attribute named `multiattr` is assigned the values `test1,test2,test3`.
3. The range-value attribute named `rangeattr` is assigned the range 1::5.
4. The policy `p` is assigned to the business object.

A fourth value of `test4` is added to the multi-value attribute named `multiattr`.

Removing Values from Multi-Value Attributes

You can remove a value from a multi-value attribute, for example, as follows:

```
mod bus T N R remove ATTR_NAME 3;
```

The above command removes the value 3 from the multi-value attribute on the business object. If there are multiple rows with the same value, then all of those rows are removed.

A multi-value attribute can have more than one instance of the same value. The command `add bus T N R intattr 1,2,3,1;` will create four separate rows in the database table.

The following table shows some examples of adding/modifying/removing multi-value attributes for a business object. The same applies to attributes for relationships. In these examples, `intattr` is an integer, multi-valued attribute. The order values in the database are 1-based.

`add bus T N R intattr 1,2,3;`	Attribute values 1,2,3 are added in that order.
`mod bus T N R intattr add 4 order 2;`	Attribute value 4 is inserted in the third position, so that the values are 1,2,4,3.
`mod bus T N R intattr add 8;`	Attribute value 8 is added in the last position, so that the values become 1,2,4,3,8.
`mod bus T N R intattr remove 4;`	The value 4 is removed from the attribute. If there are several instance of the same value, all are removed.
`mod bus T N R intattr remove order 2;`	The value in the third position is removed.
`mod bus T N R intattr replace 5 order 2;`	The current value in the third position is replaced with the value 5.
`mod bus T N R intattr 7;`	The attribute value 7 replaces all other values, and the attribute value for the object becomes only 7.

Querying a Business Object for Multi-Value or Range-Value Attributes

Printing a business object returns multiple values for a multi-value attribute, in ascending order. For example, if there exists a type T1 with an integer multi-value attribute M1 and there is one object of this type, printing this object produces the following results:

```
add bus T1 test 0 policy P1 M1 1,2,3;
print bus T N R select attribute[M1].value;
  attribute[M1].value = 1
  attribute[M1].value = 2
  attribute[M1].value = 3
```

There is no selectable that can retrieve all of the values of a multi-value attribute as a single string.

For range-value attributes, the existing `value` selectable returns two values, as shown below:

```
add bus T1 test 0 policy P1 M1 1::3;
print bus T N R select attribute[M1].value;
  attribute[M1].value = 1
  attribute[M1].value = 3
```

The `minval` and `maxval` selectables for business objects print the minimum and maximum values, respectively, of range-value attributes. For example:

`print bus T N R select attribute[NAME].minval;`	Returns the minimum value of a range-value attribute. Returns nothing for multi-value and single-value attributes.
`print bus T N R select attribute[NAME].maxval;`	Returns the maximum value of a range-value attribute. Returns nothing for multi-value and single-value attributes.
`print bus T N R select attribute[NAME].size;`	Returns 2, since there are two instances of a range-value attribute on the business object.

Using the `minval` and `maxval` selectables in query Where clauses may have performance issues. It is recommended not to use either of these selectables as part of Where clauses.

Given a business object with a multi-value attribute M1 created with the command:

```
add bus T1 test 0 policy P1 M1 "1","2","3";
```

The following table lists the outputs produced by various queries of this business object:

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value >= 1;`	`T1 test 0`
`temp quer bus T1 * * where attribute[M1].value == 1;`	`T1 test 0`
`temp quer bus T1 * * where "attribute[M1].value == 1 || attribute[M1].value == 2";`	`T1 test 0`
`temp quer bus T1 * * where attribute[M1].value > 3;`	Nothing

Given a business object with a range-value attribute M1 created with the command:

```
add bus T1 test 0 policy P1 M1 1:3;
```

The results from greater-than '>' and less-than '<' operators for range-value attributes are interpreted such that the entire range for the attribute should be '>' or '<' the literal.

The following table lists the outputs produced by various queries of this business object:

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value >= 1;`	`T1 test 0` Both the minval and the maxval are >= 1.
`temp quer bus T1 * * where attribute[M1].value == 1;`	`T1 test 0` Minval is == 1.
`temp quer bus T1 * * where "attribute[M1].value == 1 || attribute[M1].value == 2";`	`T1 test 0`

Query	Returns:
`temp quer but T1 * * where attribute[M1].value > 0;`	`T1 test 0` Both minval and maxval are > 0.
`temp quer bus T1 * * where attribute[M1].value > 3;`	Nothing Both minval and maxval are <= 3.
`temp quer bus T1 * * where attribute[M1].value < 2;`	Nothing Maxval > 2.

Given a business object with a multi-value attribute M1 created with the commands:

```
add bus T1 test 0 policy P1 M1 1,2,3;
add bus T1 test2 0 policy P1 M1 4,5,6;
add bus T1 test3 0 policy P1 M1 7,2,3;
```

The following table lists the outputs produced by various queries of this business object:

Query	Returns:

`temp quer bus T1 * * where attribute[M1].value >= 1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`

`temp quer bus T1 * * where attribute[M1].value == 1;`	`T1 test 0`

`temp quer bus T1 * * where attribute[M1].value != 1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`

`temp quer bus T1 * * where attribute[M1].value > 3;`	`T1 test2 0` `T1 test3 0`

Given a business object with a multi-value attribute M1 created with the commands:

```
add bus T1 test 0 policy P1 M1str val1,val2,val3;
add bus T1 test2 0 policy P1 M1 val4,val5,val6;
add bus T1 test3 0 policy P1 M1str Val1,val2,val3;
```

The following table lists the outputs produced by various queries of this business object:

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value == val1;`	`T1 test 0`

`temp quer bus T1 * * where attribute[M1].value != val1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`
	All three objects have values other than "val1".

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value match val1;`	`T1 test 0` This is a case-sensitive match.

`temp quer bus T1 * * where attribute[M1].value nmatch val1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`
	This is a case-sensitive not-match.

`temp query bus T1 * * where attribute[M1].value smatch val1;`	`T1 test 0`
	`T1 test3 0`
	This is a case-insensitive match. "Val1" is equal to "val1" when the comparison is case-insensitive.

`temp query bus T1 * * where attribute[M1].value nsmatch 1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`
	All three objects have values other than "val1".

In the case of multi-value attributes, using ".value" in the Where clause (e.g., `where attribute[M1].value == 'val1'`) is less efficient than without (e.g., `where attribute[M1] == 'val1'`). This is because the Where clause condition is resolved by means of a precise select statement, and when ".value" is not used, evaluation is done in memory, which is more intensive. It is, therefore, advisable to construct Where clauses without ".value".

Expressions with Multi-Value Selectables

Starting in V6R2013x, expressions with multi-value selectables in the Where clause maintain the data type of the underlying data. Prior to this release, multi-valued selectables in expressions were compared as strings. For example, given the following scenario:

```
businessobject SynthPart Alternate Alternating Assembly 0
attribute[SynthReal1] = -304.66
to[SynthExpand].from.attribute[SynthReal1] = -317.72
to[SynthExpand].from.attribute[SynthReal1] = 631.53
businessobject SynthPart Electronic Shielding Terminal 0
attribute[SynthReal1] = 341.34
to[SynthExpand].from.attribute[SynthReal1] = 125.79
to[SynthExpand].from.attribute[SynthReal1] = 711.95
```

The function that retrieves the data, `ConstPatternOperand::getData(int\real)`, had been incorrectly implemented. It simply called `atoi/atof` on the complete pattern string. In the above example, the pattern `-317.72\n631.53` would have been converted to a real as -317.72, and 631.53 would not even have been considered. Since -304.66 is greater than -317.72, the following query would not have returned this business object prior to V6R2013x:

```
MQL<n>temp query bus SynthPart * * orderby name where
'(attribute[SynthReal1] < to[SynthExpand].from.attribute[SynthReal1])'
select attribute[SynthReal1]
to[SynthExpand].from.attribute[SynthReal1];
```

Similarly, `125.79\n711.95` would have been converted to 125.79, which is less than 341.34, and would not have returned this business object. In V6R2013x and later, data-typed comparisons are correctly performed.

Also, multi-valued selectables in expresssions were previously treated like an OR (i.e., the expression evaluated to true if any of the multiple values made it true).

In cases where there are multiple expressions involving the same attribute (e.g., where `[(attribute[M1] == val1) && (attribute[M1] == valN) .. ]`, only the first expression `(attribute[M1] == val1)` is SQL'ized, and the rest are resolved in memory.

In Where Used queries, it is prohibited to use multi-value or range-value attributes. The Orderby clause is also not allowed if multi-value or range-value attributes are used in a query.

In the case of range-value attributes, some operators (mainly the ones dealing with strings and patterns such as match, nmatch, smatch, etc.) do not produce accurate results. They treat the value `[|(minval::maxval)|]` as a string and make a comparison against it.

Dimensions and Multi-Value/Range-Value Attributes

Dimensions can be added for multi-value attributes, as is currently possible in the system (see *Dimensions*). Only one dimension is allowed per attribute type, whether multi- or single-value. However, for multi-value attributes, every single value can have its own UOM when a dimension is applied to the attribute type (i.e., each attribute value can have its own UOM value). There is a one-to-one correspondence between the entries in the attribute table and in the unit table. For example:

```
add dimension d;
mod dimension d add unit km label kilometers unitdescription
  kilometer_desc multiplier 1000 offset 0 setting METRIC true;
add attribute M1 type integer dimension d multival true;
add attribute R1 type integer dimension d rangeval true;
add type T1 attribute M1,R1;
add bus T1 N1 0 M1 "100km","200km","300km" R1 "10 km::100 km";
```

Since the attribute M1 has three values, the above `add bus` statement should see three separate entries for the UOM values in the unit table and three corresponding values in the attribute table in the database. The entries in the two tables are tied together with the order number.

The following attribute selectables return multiple values, all of which are output based on the order number:

```
attribute.inputvalue;
attribute.inputunit;
attribute.dbvalue;
attribute.dbunit;
attribute.unitvalue;
attribute.generic;
```

For range-value attributes, there is just one UOM value for the entire range in the unit table.

History and Multi-Value/Range-Value Attributes

The history entry for a multi-value attribute is delimited by a comma and truncated to 255 characters maximum length. If a single attribute setting sets the attribute to multiple values, which concatenates the values delimited by ',' and generates a string that is longer than 255 characters, only the first 255 characters of the attribute value are stored in the history tables in the database.

The history entry for the attribute modifications also includes the existing value of the attribute, and the same 255 character limitation applies to the existing attribute values.

For a multi-value attribute, an example of a history entry would be:

```
history = modify - user: creator time: Fri Jul 20, 2012 11:00:54 AM
  EDT state: s1 attr: 1,2,3,[..up to 255 chars] was 20,30,40[..up to
  255 chars]
```

For a range-value attribute, the range is specified. An example of a history entry would be:

```
history = modify - user: creator time: Fri Jul 20, 2012 11:00:54 AM
  EDT state: s1 attr: 20::30 was 10::20
```

Triggers and Multi-Value/Range-Value Attributes

The new ATTRTYPEKIND macro specifies whether an attribute is single-value, multi-value, or range-value, respectively, by having one of three values:

- Single
- Multi
- Range

The existing NEWATTRVALUE macro can be used to enter *new* multi-value and range-value attribute values delimited by "^G" (the beep character). The trigger would then have to parse the string to get the multiple values for such attributes.

The existing ATTRVALUE macro can be used to enter *current* multi-value and range-value arribute values, again delimited by "^G" (the beep character). The trigger would then have to parse the string to get the multiple values for such attributes.

Unique Keys/Indexes and Multi-Value/Range-Value Attributes

Unique keys and indexes are not supported for multi-value or range-value attributes. This means that:

- If a multi-value or range-value attribute is used as a field while defining an index or unique key, an error will be thrown.
- If there is an existing unique key or index that has a single-value attribute and you try to modify that attribute to become a multi-value or range-value attribute, an error will be thrown.

Adaplets and Multi-Value/Range-Value Attributes

Adaplets do not support multi-value or range-value attributes. This means that:

- If you modify an existing attribute to become a multi-value or range-value attribute, the first time that the attribute vault is loaded after the change, an error will be thrown.
- If any new attributes are added that are multi-value or range-value attributes and they are used in an adaplet mapping file, again an error will be thrown.

Dimensions

The dimension administrative object provides the ability to associate units of measure with an attribute, and then convert displayed values among any of the units defined for that dimension. For example, a dimension of Length could have units of centimeter, millimeter, meter, inch and foot defined. Dimensions are used only with attributes; see *Dimension Clause* for instructions.

The definition of the units for a dimension includes determining which unit will be the default (the normalized unit for the dimension), and the conversion formulas from that default to the other units. The conversion formulas are based on a multiplier and offset entered when the unit is defined. The normalized unit has a multiplier of 1 and an offset of 0.

To convert to the normalized value stored in the database to a different unit, the system uses this formula:

```
normalized value = unit value * multiplier + offset
```

To display a value in units other than the normalized units, the system uses this formula:

```
unit value = (normalized value - offset) / multiplier
```

Only the normalized value is stored in the database. When an application requires the value for an attribute to be displayed, the system converts the normalized value to the units required. The value can be entered in any supported unit of the dimension, but it will be converted and stored in the default units.

Real attribute normalized values are stored with the same precision as real attribute values with no dimension applied. See *Attributes* for more information. To avoid round-off errors with integer attributes, the default units should be the smallest unit (for example, millimeters rather than centimeters or meters).

The conversion process affects the precision. In general, up to 12 digits of precision can be assumed. For each order of magnitude that the offset and the converted value differ, another digit of precision is lost.

Dimensions help qualify attributes that quantify an object. For example, for a type with an attribute Weight, the user needs to know if the value should be in pounds or kilograms, or another dimension of weight. When the attribute definition includes a dimension, the user is provided with that information in the user interface. In addition, the user has the ability to choose the units of the dimension to enter values.

When applying dimensions to attributes that already belong to business object instantiations, refer to *Applying a Dimension to an Existing Attribute* for information on converting the existing values to the required normalized value.

Defining a Dimension

Before attributes can be defined with a dimension, the dimension must be created. You can define a dimension if you are a business administrator with Attribute access, using the add dimension command:

```
add dimension NAME [ADD_ITEM {ADD_ITEM} ];
```

`NAME` is the name you assign to the dimension.

`ADD_ITEM` is an Add Dimension clause that provides additional information about the dimension you are creating.

The complete syntax and clauses for working with dimensions are discussed in *dimension Command* in the programming reference section of this guide.

Choosing the Default Units

Before you define a dimension, you need to decide which units of that dimension will be the default. That unit will have a multiplier of 1 and an offset of 0. You must calculate the multiplier and offset values for all other units of the dimension based on the default.

For example, the following table shows the definition for a Temperature dimension normalized on Fahrenheit:

Unit	Label	Multiplier	Offset
Fahrenheit	degrees Fahrenheit	1	0
Celsius	degrees Celsius	1.8	32

If you wanted to normalize the dimension on Celsius, you would enter these values when defining the units:

Unit	Label	Multiplier	Offset
Fahrenheit	degrees Fahrenheit	.555555555555555	17.7777777777777777
Celsius	degrees Celsius	1	0

For dimension definitions that are to be applied to an integer, all multiplier and offset values in the dimension should be whole numbers, and the "smallest" unit should be the default.

Interfaces

You may want to organize data under more than one classification type. For instance, a Part may have a classification based on its function, which is most typical, but it may also require classification on other issues such as production process, manufacturing location, etc. For each classification type, there is typically a collection of attributes that can be defined for each instance of the classification type and used for searching.

An *Interface* is a group of attributes that can be added to business objects as well as connections to provide additional classification capabilities. When an Interface is created, it is linked to (previously-defined) attributes that logically go together. Interfaces are defined by the Business Administrator and are added to business object or relationship instances.

You must be a Business Administrator to add or modify interfaces.

An Interface can be *derived* from other Interfaces, similar to how Types can be derived. Derived Interfaces include the attributes of their parents, as well as any other attributes associated with it directly. The types or relationships associated with the parents are also associated with the child.

The primary reason to add an interface to a business object or a connection is to add the attributes to the instance that were not defined on the type. Moreover, when you add an interface to a business object or a connection, it gives you the ability to classify it by virtue of the interface hierarchy.

Attribute values that come from interfaces cannot be used in a create access rule, because the interfaces are not applied until AFTER the create access checks.

Defining an interface

An object interface is created with the Add Interface command:

```
add interface NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the interface.

`ADD_ITEM` is an Add Interface clause which provides additional information about the interface you are creating.

The complete syntax and clauses for working with interfaces are discussed in *interface Command* in the programming reference section of this guide.

Types

A *type* identifies a kind of business object and the collection of attributes that characterize it. When a type is created, it is linked to the (previously defined) attributes that characterize it. It may also have methods and/or triggers associated (refer to *Programs* and the *Configuration Guide : Triggers*). Types are defined by the Business Administrator and are used by users to create business object instances.

A type can be *derived* from another type. This signifies that the derived type is of the same kind as its parent. For example, a Book is a kind of Publication which in turn is a kind of Document. In this case, there may be several other types of Publications such as Newspaper, Periodical, and Magazine.

This arrangement of derived types is called a *type hierarchy.* Derived types share characteristics with their parent and siblings. This is called *attribute inheritance*. When creating a derived type, other attributes, methods, and triggers can be associated with it, in addition to the inherited ones. For example, all Periodicals may have the attribute of Page Count. This attribute is shared by all Publications and perhaps by all Documents. In addition, Periodicals, Newspapers, and Magazines might have the attribute Publication Frequency.

You must be a Business Administrator to add or modify types. (Refer also to your Business Modeler Guide.)

Type Characteristics

Implicit and Explicit

Types use explicit and implicit characteristics:

- *Explicit characteristics* are attributes that you define and are known to Live Collaboration.
- *Implicit characteristics* are implied by the name only and are known only to the individual user.

For example, you may create a type called “Tax Form” which contains administrator-defined explicit attributes such as form number, form type, and tax year. Or, Tax Form may contain no explicit attributes at all.

When a type exists without administrator-defined attributes, it still has implicit characteristics associated with it. You would know a tax form when you saw it and would not confuse it with a type named “Health Form.” But the characteristics you use to make the judgment are implicit—known only by you and not Live Collaboration.

Inherited Properties

Types can inherit properties from other types:

- *Abstract types* act as categories for other types.

  Abstract types are not used to create any actual instances of the type. They are useful only in defining characteristics that are inherited by other object types. For example, you could create an abstract type called Income Tax Form. Two other abstract types, State Tax Form and Federal Tax Form, inherit from Income Tax Form.

- *Non-abstract types* are used to create instances of business objects.

  With non-abstract types, you can create instances of the type. For example, assume that Federal Individual Tax Form is a non-abstract type. You can create business objects that contain the actual income tax forms for various individuals. One object might be for a person named Joe Smith and another one for Mary Jones. Both objects have the same type and characteristics although the contents are different based on the individuals.

Defining a Type

An object type is created with the Add Type command:

```
add type NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the type.

`ADD_ITEM` is an Add Type clause which provides additional information about the type you are creating.

The complete syntax and clauses for working with types are discussed in *type Command* in the programming reference section of this guide.

Formats

A *format* definition is used to capture information about different application file formats. A format stores the name of the application, the product version, and the suffix (extension) used to identify files. It may also contain the commands necessary to automatically launch the application and load the relevant files from Collaboration and Approvals. Formats are the definitions used to link Collaboration and Approvals to the other applications in the users' environment.

Applications typically change their internal file format occasionally. Eventually older file formats are no longer readable by the current version of the software. It is wise to create new format definitions (with appropriate names) as the applications change so that you can later find the files that are in the old format and bring them up to date.

The system does not do any checking of the type of file that is checked into a format. For example, a word document with a .doc extension can be checked into a format defined for HTML files. This means that formats can be used to define directories for the checked in files of a business object.

A business object can have many file formats and they are linked to the appropriate type definition by the policy definition (see *Policies*).

You must be a Business Administrator to add or modify formats.

Defining a Format

Format definitions are created using the MQL Add Format command. This command has the syntax:

```
add format NAME [ADD_ITEM {ADD_ITEM}]
```

`NAME` is the name you assign to the format.

`ADD_ITEM` is an Add Format clause which provides more information about the format you are creating.

The complete syntax and clauses for working with formats are discussed in *format Command* in the programming reference section of this guide.

Policies

A *policy* controls a business object. It specifies the rules that govern access, approvals, lifecycle, revisioning, and more. If there is any question as to what you can do with a business object, it is most likely answered by looking at the object's policy.

You must be a Business Administrator to add or modify policies. (Refer also to your Business Modeler Guide.)

A policy is composed of two major sections: the first describes the general behavior of the governed objects and the second describes the lifecycle of the objects.

General Behavior

The first section controls the creation of the object and provides general information about the policy. This information includes:

- The types of objects the policy will govern.
- The types of formats that are allowed.
- The default format automatically assigned.
- Where and how checked in files are managed.
- How revisions will be labeled.

Lifecycle

The second section provides information about the lifecycle of the objects governed by the policy. A lifecycle consists of a series of connected states, each of which represents a stage in the life of the governed objects. Depending on the type of object involved, the lifecycle might contain only one state or many states. The purpose of the lifecycle is to define:

- The current state of the object.
- Who will have access to the object.
- The type of access allowed.
- Whether or not the object can be revised.y
- Whether or not files within the object can be revised.
- The conditions required for changing state.
- Whether or not the state has been published. See “Published States” on page 139.

A policy can exist without any states defined. This is enabled mainly to support legacy data. There are certain disadvantages to not defining any states:

- It is not possible to have access control without any state definition.
- Some operations, like disabling checkout history, cannot be performed since no state exists to control this.

If a policy is created without state, a warning is displayed stating "Policy has no STATE defined."

Determining Policy States

When creating a policy, defining the policy states is most often the most difficult part. How many states does the policy need? Who should have access to the object at each state and what access should each person have at each state? Which access takes precedence over the other? Should you allow revisions at this state? Should you allow files to be edited? What signatures are required to move the object from one state to another? Can someone override another's signature? As described below, all of these questions should be answered in order to write the state definition section of a policy.

How Many States are Required?

A policy can have only one state or many. For example, you might have a policy that governs photographic images. These images may be of several types and formats, but they do not change their state. In general, they do not undergo dramatic changes or have stages where some people should access them and some should not. In this situation, you might have only one state where access is defined.

Let's examine a situation where you might have several states. Assume you have a policy to govern objects during construction of a house. These objects could have several states such as:

State	Description
Initial Preparation	The building site is evaluated and prepared by the site excavator and builder. After the site is reviewed and all preparations are completed, the excavator and builder sign off on it and the site enters the second state, Framing.
Framing	Carpenters complete the framing of the house and it is evaluated by the builder, architect, and customer. In this state, you may want to prohibit object editing so that only viewing is allowed. If the framing is complete to the satisfaction of the builder, architect, and customer, it is promoted to the third state, Wiring.
Wiring	The electrician wires the house. However, the electrician may sign off on the job as completed only to have the builder reject it. When approval is rejected, promotion to the next state is prevented from taking place.

As the house progresses through the building states, different persons would be involved in deciding whether or not the object is ready for the next state.

When determining how many states an object should have, you must know:

- What are the states in an object's life.
- Who requires access to the object.
- What type of access they need.

Once a policy is defined, you can alter it even after business object instances are created that are governed by it.

Who Will Have Object Access?

There are three general categories used to define who will have access to the object in each state:

- Public—refers to everyone in the database. When the public has access in a state, any defined user can work with the business object when it is in that state.

- Owner—refers to the specific person who is the current owner of the object instance. When an object is initially created in the database, the person who created it is identified as the *owner* of the object. This person remains the owner unless ownership is transferred to someone else.
- User—refers to a specific person, group, role, or association who will have access to the object. You can include or exclude selected groupings or individuals when defining who will have access.

For additional information on access privileges, including which access takes precedence over the other, see *User Access* in Chapter 3.

Is the Object Revisionable?

In each state definition are the terms *Versionable* and *Revisionable*. The term Revisionable indicates whether a new Revision of the object can be made. Versionable is not used at this time, and setting it has no affect on policy behavior.

You can decide when in the object's lifecycle revisions are allowed by setting the switch ON or OFF in each state definition. This setting is independent of who (which person, role or group) has access to perform the operations.

Published States

Every state has a published flag. In policies where this flag is turned on, a best-so-far (BSF) flag is automatically propagated to business objects that enter such states. By default, the published flag is turned off. A migration tool is available to migrate policies, both to incorporate the new published flag and to update all business objects governed by policies that have the flag turned on in order to set the published flag correctly. See the *Database Migration Guide*, available in the Program Directory, for details on the migration tool.

The definition of a policy accepts that each of its states can be marked as *published*. You can set this flag on the state administrative object by invoking the following MQL command:

```
mod policy NAME state STATE published TRUE|FALSE;
```

To determine whether the published flag has been set on an object, you can use:

```
print policy NAME select state.published;
```

To retrieve the boolean value of the published flag as it is propagated to business objects, use:

```
print bus T N R select.current.published;
```

Policies allow definition of two revision sequences, major and minor. You can define a revision sequence for a policy as follows:

```
add policy NAME majorsequence A,B,C,... minorsequence 1,2,3
  delimiter '-';
```

If both major and minor sequences are defined, then a delimiter is required. The delimiter is used to concatenate major and minor revision strings for storage in the database, so it must be an ASCII non-alphanumeric character that could never be part of a major/minor revision string calculated from the two sequences.

The Modify Policy command allows you to edit either sequence, but not to change the delimiter or add/remove either sequence, as this would make major/minor revision strings impossible to parse.

You can use the Transition Revisions command to add a second sequence to a policy that only one (see the *Database Migration Guide* for details).

The Print Policy command also allows the corresponding selectables, such as:

```
print policy NAME select majorsequence minorsequence delimiter;
```

Policy and state access definitions have the following access flags:

- *minorrevise* is synonymous with the earlier *revise* flag, and is used to control the Revise Minor command for adding a new minor revision.
- *majorrevise* is used to control the Revise Major command for adding a new major revision.

The following events support type definitions:

- *minorrevision* is synonymous with the earlier *revision* keyword, used for defining triggers on a type.
- *majorrevision* has been added as a new event for defining triggers on a type.

Minorrevision and majorrevision are values that are derived from the revision field using the policy delimiter. Object uniqueness that incorporates revision information must include both minor and major revision fields. The revision keyword should be used to define such a uniquekey. Therefore, minorrevision and majorrevision are not supported as fields for unique keys.

These capabilities in policy definition have corresponding pieces of data and selectables for business objects.

You can mark any state in a policy to indicate whether it is allowed to create a major/minor revision of an object in that state as follows:

```
mod policy NAME state STATE minorrevision TRUE|FALSE
  majorrevision TRUE|FALSE;
```

The terms "minorrevise" and "revise" are synonymous when defining access rules in policy and in the `print bus select current.access[minorrevise]` command. However, as output from the Print Policy command (e.g., `print policy select state` and `print bus select current.access`), the earlier "revise" keyword continues to be used for forward compatibility. Likewise, the keyword "revisioned" continues to be used to mark minorrevise events in history.

Input Keywords

In the following cases, "revise" and "revision" have the same meaning as long as the object's policy supports ONLY minor revisioning. If, however, the governing policy supports both major and minor revisions, there is a difference:

- `print bus T N R select revision` prints the full revision string of the object. If the policy supports both minor and major revisions, this will be major-minor.
- `print bus T N R select revisions` prints the full revision string for all objects in the object's minor revision sequence.
- `copy/mod bus T N R name NEWNAME revision NEWREV: NEWREV` uses the full revision string (major-minor) as specified in the governing policy.

In an access rule filter:

- A filter-including expression of the form `current.access[ACCESS_TYPE] == TRUE` accepts either minorrevise or revise as the ACCESS_TYPE.
- A filter-including expression of the form `current.access ~~ *ACCESS_TYPE*` is risky since revise is now be a substring of both minorrevise and majorrevise (but see Output Keywords below).

Output Keywords

In various places, the kernel outputs the keywords revise/revision or populates macros with such a string. Changing such output could break any application code that depends on the current 'revise/revision' keyword. Although it would be cleaner and more consistent to migrate these outputs to minorrevise/minorrevision, the current output has been retained to satisfy forward compatibility.

- `print bus T N R select current.access` maintains 'revise' as the keyword describing minor revise access (e.g., `current.access == read,modify,revise,show`).
- The $ACCESS macro can be used in an access rule filter and is populated with the current access being checked whenever the filter is evaluated (e.g., the following rule allows a different condition for modify vs. revise access):

  ```
  ($ACCESS == modify && <modify-condition>) || ($ACCESS == revise &&
  <revise-condition>)
  ```

This continues to be populated as 'revise'.

- The EVENT macro continues to be populated as 'Revision' to pass to revision triggers.
- History records still say `history = revisioned - user: creator ….`

How Do You Change From One State to the Next?

Most often a change in state is controlled by one or more persons, perhaps in a particular role or group. For example, during the construction of a house, the customer and the builder might control the change in state. If you break the building stage down into smaller states, you might have the object's transition controlled by the site excavator, foundation expert, electrician, or plumber. As the house progresses through the building states, different persons would be involved in deciding whether the object is ready for the next state. You certainly would not want the carpenters to begin working before the foundation is done.

Signatures are a way to control the change of an object's state. Signatures can be associated with a role, group, person, or association. Most often, they are role-related. When a signature is required, a person must approve the object in order for the object to move on to the next state. If that person does not approve it, the object remains in the current state until the person does approve or until someone with higher authority provides approval.

More than one signature can be associated with the transition of an object. Lifecycles can be set up such that the signature that is approved determines which state is the next in the object's life.

A signature can be approved or rejected. For example, an electrician could say a job is done only to have the builder reject it. When approval is rejected, promotion to the next state is prevented from taking place.

Filters can be defined on a signature requirement to determine if it is fulfilled. If the filter evaluates to true, then the signature requirement is fulfilled. This is useful for adding required signatures that are conditional, dependent on some characteristic of a business object.

In the sections that follow, you will learn more about the actual procedures to define a policy and the object states as well as the procedures that manipulate and display policy definitions.

Defining an Object Policy

Policies are defined using the Add Policy command:

```
add policy NAME [ITEM {ITEM}];
```

`NAME` is the name you assign to the policy.

`ITEM` is an Add Policy clause which defines information such as the types of objects governed by the policy, the types of formats permitted by the policy, the labeling sequence for revisions, the storage location for files governed by the policy, and the states and conditions that make up an object's lifecycle.

The complete syntax and clauses for working with policies are discussed in *policy Command* in the programming reference section of this guide.

Relationships

A *relationship* definition is used along with the policy to implement business practices. Therefore, they are relatively complex definitions, usually requiring planning.

For example, in manufacturing, a component may be contained in several different assemblies or subassemblies in varying quantities. If the component is later redesigned, the older design may then become obsolete. Component objects could be connected to the various assembly and subassembly objects that contain it. Each time objects are connected with this relationship, the user could be prompted for the quantity value for the relationship instance. If the component is later redesigned, the older design may become obsolete. When a revision of the component object is then created, the relationship would disconnect from the original and connect to the newer revision. If the component is cloned because a similar component is available, the cloned component may or may not be part of the assembly the original component connects to. The connection to the original should remain but there should be no connection to the cloned component.

For the process to work in this fashion, the relationship definition would include the attribute "quantity." The cardinality would be "many to many" since components could be connected to several assemblies and assemblies can contain many components. The revision rule would be "float" so new revisions would use the connections of the original. The clone rule would be "none" so the original connection remains but no connection is created for the clone.

A relationship can be *derived* from another relationship. This signifies that the derived relationship is of the same kind as its parent. This arrangement of derived relationships is called a *relationship hierarchy.* Derived relationships share characteristics with their parent and siblings. This is called *relationship inheritance*. When creating a derived relationship, other attributes, methods, and triggers can be associated with it, in addition to the inherited ones.

You must be a Business Administrator to define relationships. (Refer also to your Business Modeler Guide.) Relationships are typically initially created through MQL when all other primary administrative objects are defined. However, if a new relationship must be added, it can be created with Business Administrator.

Collecting Data for Defining Relationships

In an MQL schema definition script, relationship definitions should be placed after attributes and types. Before writing the MQL command for adding a relationship definition, the Business Administrator must determine:

- Of the types of business objects that have been defined, which types will be allowed to connect directly to which other types?
- What is the nature and, therefore, the name of each relationship?
- Relationships have two ends. The *from* end points to the *to* end. Which way should the arrow (in the Indented and Star Browsers) point for each relationship?
- What is the meaning of the relationship from the point of view of the business object on the *from* side?
- What is the meaning of the relationship from the point of view of the business object on the *to* side?
- What is the cardinality for the relationship at the *from* end? Should a business object be allowed to be on the *from* end of only one or many of this type of relationship?

- What is the cardinality for the relationship at the *to* end? Should a business object be allowed to be on the *to* end of only one or many of this type of relationship?
- When a business object at the *from* end of the relationship is revised or cloned, a new business object (similar to the original) is created. What should happen to this relationship when this occurs? The choices for revisions and clones are: `none`, `replicate`, and `float`.

  Should the relationship stay on the original and not automatically be connected to the new revision or clone? If so, pick `none`.

  Should the relationship stay on the original and automatically connect to the new revision or clone? If so, pick `replicate`.

  Should the relationship disconnect from the original and automatically connect to the new revision or clone? If so, pick `float`.
- When a business object at the *to* end of the relationship is revised or cloned, a new business object (similar to the original) is created. What should happen to this relationship when this occurs? The choices are the same as for the *from* end.
- What attributes, if any, belong on the relationship? Quantity, Units, and Effectivity are examples of attributes which logically belong on a relationship between an assembly and a component rather than on the assembly or component business object. Each instance, or use of the relationship, will have its own values for these attributes which apply to the relationship between the unique business objects it connects.

Use a table like the one below to collect the information needed for relationship definitions.

Relationship Name	_____	_____	_____	_____	_____	_____
From Type	_____	_____	_____	_____	_____	_____
From Meaning	_____	_____	_____	_____	_____	_____
From Cardinality	_____	_____	_____	_____	_____	_____
From Rev Behavior	_____	_____	_____	_____	_____	_____
From Clone Behavior	_____	_____	_____	_____	_____	_____
To Type	_____	_____	_____	_____	_____	_____
To Meaning	_____	_____	_____	_____	_____	_____
To Cardinality	_____	_____	_____	_____	_____	_____
To Rev Behavior	_____	_____	_____	_____	_____	_____
To Clone Behavior	_____	_____	_____	_____	_____	_____
Attributes	_____	_____	_____	_____	_____	_____
Dynamic Relationship?	_____	_____	_____	_____	_____	_____

Dynamic Relationships

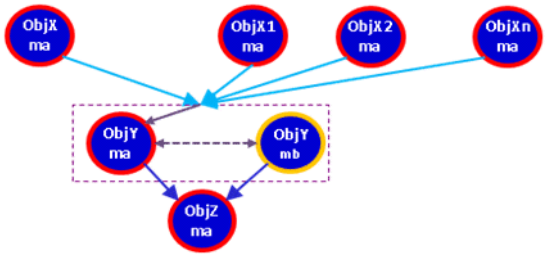
To support major/minor revisioning of business objects, "dynamic" relationships have a pointer to a MajorId representing a minor revision family rather than an individual business object. The TO end of a dynamic relationship resolves to a specific business object by identifying the best-so-far (BSF) object within the minor revision family.

When used with the Add Relationship command, the `dynamic` keyword implements support for minor-revision insensitivity. This keyword is mutually exclusive with any of the Replicate/Float/None options on the TO end, since it defines a built-in dynamic behavior of its *to* pointer.

Together with *best-so-far*, the concept of *published* objects makes it possible unambiguously to identify a single object within a minor revision family in order to resolve dynamic relationships. An object is marked "published" when it is at an appropriately mature state in its lifecycle. An unambiguous best-so-far object is then either:

- The last object in the family that is marked "published," or
- The last object in the family if none are published.

The *to* end of a dynamic relationship resolves to a specific business object by identifying the best-so-far (BSF) object within the minor revision family).



The dynamic relationship feature adds an additional requirement for database definition with Oracle installations: the Oracle user (i.e., database) must have CREATE VIEW privileges. Run the following command to add this privilege:

```
SQL> grant connect, resource, create view to USER:
Grant succeeded.
SQL> commit;
Commit complete.
```

where USER is the V6 Oracle user.

For more information, see Reference Commands: relationship Command > Add Relationship > Dynamic Clause.

MQL Export/Import

Policies are exported/imported with added fields.

Business object export/import is not supported on objects governed by policies with `majorsequence` set.

Dynamic relationships are not exported with business objects.

Defining a Relationship

A relationship between two business objects is defined with the Add Relationship command:

```
add relationship NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the relationship.

`ADD_ITEM` is an Add Relationship clause which provides more information about the relationship you are creating.

The complete syntax and clauses for working with relationships are discussed in *relationship Command* in the programming reference section of this guide.

Rules

Use rules to limit user access to attributes, forms, programs, and relationships. Unlike policies, rules control access to these administrative objects regardless of the object type or state. For example, a policy might allow all users in the Engineering group to modify the properties of Design Specification objects when the objects are in the Planning state. But you could create a rule to prevent those users from changing a particular attribute of Design Specifications, such as the Due Date. In such a case, Engineering group users would be unable to modify the Due Date attribute, no matter what type of object the attribute is attached to. For a explanation of how rules work with other objects that control access, see *Which Access Takes Precedence Over the Other?* in Chapter 3.

When you create a rule, you define access using the three general categories used for assigning access in policies: public, owner, and user (specific person, group, role, or association). For a description of these categories, see *Policies*.

Creating a Rule

Rules are defined using the Add Rule command:

```
add rule NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the rule.

`ADD_ITEM` is an Add Rule clause which defines information about the rule including the description and access masks.

The complete syntax and clauses for working with rules are discussed in *rule Command* in the programming reference section of this guide.

Ownership

Business objects and relationships can have multiple owners. In addition, several objects may need to have a common baseline access level. This section describes how ownership of business objects and relationships functions, including multiple ownerships and ownership inheritance.

Multiple Ownerships

There are many cases where a business object or relationship may require multiple ownerships, such as a part that has multiple RDO, RMO, and RSO security attributes, each of which represent an organization or project owner. It is possible to specify multiple ownerships for an object. A complete ownership definition includes the following:

- Organization—a "role" object that represents an organization
- Project—a "role" object that represents a project
- Comment—a short string used primarily for annotation
- Access—a comma-separate list of security tokens

Each of these fields can be specified when providing an ownership. The first three fields (org, project, and comment) together provide a unique identifier for the ownership entry.

You can add ownerships to or subtract them from an existing business object or relationship. Ownership information is used to determine object access rights dynamically. Commands that are used to get or set the single organization/project ownership on business objects continue to function, but generate an error if more than one ownership is present.

See Adding or Removing Business Object Ownerships and Adding or Removing Relationship Ownerships for details.

Ownership Inheritance

Rather than maintaining security on a per-object basis, it is possible to govern a list of objects where the entire collection of objects has a common baseline access level by aggregating ownerships based on another object. This allows an object to inherit the ownership list from another object as its baseline access in addition to having its own direct ownership. Thus, it is no longer necessary to manage ownership by adding or subtracting ownerships individually on each object in a workspace since ownership for an object comprises an embedded reference to a parent object.

Existing applications that store parent information as a relationship (e.g., workspace) must add an ownership definition in addition to the relationship. Applications that currently lack a parent or folder notation can create the parent ownership entry without creating a new relationship entity by specifying a parent object. The access rule engine dynamically extends an entity’s ownership list to include the (non-inherited) ownerships of the parent.

To do this, you specify an object ID and another optional comment relating to the parent object. Ownerships, constructed in this form implement ownership inheritance from the parent. The access rule engine dynamically extends an entity's ownership list to include the ownerships of the parent. These inherited ownerships include all ownerships added to the entity using the multiple-ownership feature. The ownerships do not include the parent’s primary ownership identified in the parent’s organization and project basic properties. You can use ownership inheritance on multiple levels. Object A can inherit from object B, and object B can inherit from object C. This implies that Object A also inherits from object C. See *Adding Ownership Inheritance* for details.

5 Manipulating Data

This chapter explains how to create and use business objects to manipulate data.

In this section:

Creating and Modifying Business Objects

Business objects form the body of Collaboration and Approvals. They contain much of the information an organization needs to control. Each object is derived from its previously-defined type and governed by its policy. Therefore, before users can create business objects, the Business Administrator must create definitions for the types (see *Types* in Chapter 4) and policies (see *Policies* in Chapter 4) that will be used. In addition, the users (persons, groups, and roles) must be defined before they can have access to the application (see *Working with Users and Access* in Chapter 3).

When creating a business object, the first step is to define (name) the object and assign an appropriate description and attribute values for it. File(s) can then be checked into the object and it can be manipulated by establishing relationships, moving it from state to state and perhaps deleting or archiving it when it is no longer needed. This chapter describes the basic definition of the object and its attributes. In the next chapter, relationships, connections, states, checking files in and out, and locking objects are described in more detail.

Using Physical and Logical IDs

A physical id is a global identifier that is unique to each business object or relationship and a logical id is a global identifier shared by all members of a revision sequence. Physical and logical ids are required for VPM applications. You must upgrade your database to use these ids. See the Program Directory for this release.

Specifying a Business Object Name

When you create or reference a business object, you must give its full business object name. The full business object name must contain three elements:

```
TYPE NAME REVISION
```

Each element must appear in the order shown. If any element is missing or if the values are given in the wrong order, the business object specification is invalid.

You can also optionally specify the vault in which the business object is held. When the vault is specified in this manner, only the named vault needs to be checked to locate the business object. This option can improve performance for very large databases.

```
TYPE NAME REVISION [in VAULT]|ID [in VAULT]
```

See *Defining a Business Object* later in this chapter for more information about additional elements.

The full business object specification includes `TYPE NAME REV`: the type from which the object was created, the object name—the user-supplied identifier that is associated with the definition and identifies the object to the end user(s)—and the revision.

In the sections that follow, each of the three required elements is discussed and sample values are given.

Business Object Type

The first element in a business object specification is the object's type. Every object must have a type associated with it. When you specify the object's type, remember that it must already be defined.

If the type name you give in the business object specification is not found, an error message will display. If this occurs, use the List Type command (described in *list admintype Command*) to check for the presence and spelling of the type name. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters.

When you are assigning a type to a business object, note that all types have attributes associated with them. These attributes appear as fields when the object is accessed in Matrix Navigator. These fields can then be filled in with specific values or viewed for important information.

For example, you might assign a type of "Metallic Component" to an object you are creating. With this type, you might have four attributes: type of metal, size, weight, and strength. If you use an Attribute clause in the `add businessobject` command or modify the attributes, you can insert values that will appear whenever the object is accessed. If attributes are not specified when a business object is added or modified, the attribute defaults (if any) are used.

Business Object Name

The second element in a business object specification is the business object name. This consists of a character string that will be used to identify the business object and to reference it later. It should have meaning to you and guide you as to the purpose or contents of the business object. While the Description clause can remind you of an object's function, it is time-consuming to have to examine each object to find the one you want. Therefore, you should assign a name that clearly identifies and distinguishes the object.

You can use your exact business terminology rather than cryptic words that have been modified to conform to the computer system limitations. Collaboration and Approvals has few restrictions on the characters used for naming business objects. See *Administrative Object Names*.

When specifying an existing business object, if the name you give is not found, an error message will result. If an error occurs, use the Temporary Query command with wildcards to perform a quick search. For example, to find an object with a name beginning with the letters "HC" and unknown type and revision level, you could enter:

```
temporary query businessobject * HC* *
```

Use: the first `*` for the unknown type, the `HC*` for the name beginning with "HC", and the third `*` for the unknown revision level. The result would be all the objects beginning with "HC".

```
Product HC-430 A

Product HC-470 B
```

Business Object Revision Designator

The third element in a business object specification is the revision label or designator. The revision must be specified if the object requires the revision label in order to distinguish it from other objects with the same name and type. Depending on the object's policy, revisions may or may not be allowed. If they are not allowed or a revision designator does not exist, you must specify `""` (a set of double quotes) for MQL.

The ability (access privilege) to create revisions can be granted or denied depending on the object's state and the session context. When an object is revised, the revision label changes. This label is either manually assigned at the time the revision is created or automatically assigned if a revision sequence has been defined in the governing policy.

Revision sequences provide an easy and reliable way to keep track of object revisions. If the revision sequencing rules call for alphabetic labels, a revised object might have a label such as B or DD. If the Sequence clause in the policy definition specifies custom revision labels, you might see a label such as Unrevised, "1st Rev," "2nd Rev," and so on. In any case, the revision label you provide must agree with the revision sequencing rules. If it does not, an error message will result.

For example, the following are all valid business object specifications:

```
Component "NCR 1139" ""
Drawing "Front Elevation" 2
Recipe "Spud's Fine Mashed Potatoes" IV
"Tax Record" "Sherry Isler" "second rev"
```

The first specification has no revision designator and must be specified as such. This might be because Component types cannot be revised under the governing policy. It might also be because this is the original object that uses a sequence where the first object has no designator.

For more information about revision sequences, see *Sequence Clause*.

Object ID

When business objects are created, they are given an internal ID. As an alternative to `TYPE NAME REV`, you can use this ID when indicating the business object to be acted upon. The ID of an object can be obtained by using the print businessobject selectable "ID". Refer to *Viewing Business Object Definitions* later in this chapter for more information on select commands.

Defining a Business Object

Business objects are defined using the Add Businessobject command:

```
add businessobject BO_NAME policy POLICY_NAME [ITEM {ITEM}];
```

`BO_NAME` is the Type Name Revision of the business object.

`POLICY_NAME` is the policy that governs the business object. The Policy clause is required when creating a new business object. For more details on policies, see *Policies*.

`ITEM` is an Add Businessobject clause that provides additional information about the business object you are creating.

The complete syntax and clauses for working with business objects are discussed in *businessobject Command* in the programming reference section of this guide.

Working with Legacy Data

When migrating data from a legacy system, you need to create Business objects that represent the actual state of the data you are importing from the legacy system. This is necessary to maintain data integrity. For example, when migrating a document that has been approved in the legacy system you should set the state that correctly indicates its approved state. The business object you create

should also have the actual date/time of the creation and/or modification of the legacy data, and not the system generated date and time (that would reflect when the data was migrated).

In some cases, you may want the business objects to act as a placeholder, representing the actual objects in the external system that is constantly changing. You can update the business objects on a regular basis by resetting the current state as well as the start, end and duration values of the states. Modifying a business object this way avoids the need for promoting the object through the lifecycle, which would result in each state reflecting the date of the import/promotion and not the date from the legacy system. Each of these (current state, start date, end date, and duration) can be changed independently, with no effect on each other. If you want to maintain these values to be consistent with each other, you must change them all. There is no impact on the lifecycle of these objects if these values are not kept consistent, except for queries or webreports that depend on these values.

See *Adding Legacy Data* for the different ways you can migrate and synchronize data from external systems into the database.

Viewing Business Object Definitions

You can view the definition of a business object at any time by using the Print Businessobject command and the Select Businessobject command. These commands enable you to view all the files and information used to define the business object. The system attempts to produce output for each select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

Reserving Business Objects for Updates

When two users try to edit a business object at the same time, the modifications made by one user can overwrite the changes made by another. The same is true for business object connections. To prevent such concurrent modifications, the kernel provides the ability to mark a business object or connection as reserved and store its reservation data.

The kernel does not in itself prevent concurrent modifications. Applications using the kernel can use the reservation data to implement ways to warn or disallow users from modifying a reserved object.

When an object or connection is reserved, the kernel adds the “reserved” tag which includes a user and a timestamp. An application can be programmed such that it checks for the reserved status of a business object or connection, and if reserved, can issue a warning or disallow other users from modifying the business object or connection until it is unreserved.

Implementing reservations in an application

An application can be programmed such that no one can modify a reserved business object or connection unless it is unreserved. To query the reserved status of a business object or connection, implementors can use the following selectables:

Selectable	Description	Output
reserved	Boolean indication reserved status of a business object	True/False
reservedby	Non-empty string or the context user	Name of the person who reserved the object.
reservedstart	The date or timestamp of when the business object was reserved.	Returns the date and time of when that object was reserved.
reservedcomment	Optional comment not exceeding 254 characters from the person reserving the object.	Any comments entered.

For example:

```
print businessobject "Box Design" "Thomas" "A" select reserved
reservedby reservedstart reservedcomment;
```

The above command returns output similar to:

```
reserved = TRUE
reservedby = Jerry
reservedstart = 30-Oct-2005 10:34:10 AM
reservedcomment = "reserved from Create Part Dialog"
```

To query the reserved status of a connection use:

```
print connection 62104.18481.31626.56858 select reserved;
```

Revising Existing Business Objects

The ability to create revisions can be granted or denied depending on the object's state and the session context. For example, let's assume you are an editor working on a cookbook. A cookbook is composed of Recipe objects. Recipe objects are created by the Chef who writes the recipe, perhaps in the description field of the object. He then promotes the Recipe to the "Test" state, where he makes the dish and tastes it. At this point, he either approves it and sends it to the next state (perhaps "Submitted for Cookbook"), reassigning ownership to you (the editor), or he may want to revise it, incorporating different ingredients or varying amounts. Therefore, the "Test" state would have to allow revisions by the owner. The Recipe object could then be revised, the new revision starting in the first state of the lifecycle. Once the Recipe is approved, revisions should not be allowed; so, the "Submitted for Cookbook" state would not allow revisions.

In order to maintain revision history, the new object should be created with the revised business object command even though it is possible to create what looks like a new revision by manually assigning a revision designator with the Add or Copy Businessobject command. (However, you can add existing objects to a revision chain, if necessary. Refer to *Adding a Business Object to a Revision Sequence* for more information.) The table below shows the differences between using the

`revise businessobject` and `copy businessobject` commands discussed in *businessobject Command* in the programming reference section of this guide.

Differences/Similarities between Clones and Revisions		
Property	**Copy or Clone**	**Revision**
Attributes	Values are initially the same but can be modified as part of the clone command.	Values are initially the same but can be modified as part of the revise command.
Files	Files can optionally be copied to the Clone upon creation when the copy command specifies to do so. See *Handling Files* for details.	Files are referenced from the original until it is necessary to copy them when the revise command specifies to do so. See *Handling Files* for details.
Connections to other objects	Depends on the Clone Rules (Float, Replicate, None) set by the Business Administrator.	Depends on the Revision Rules (Float, Replicate, None) set by the Business Administrator.
Connection to Original	No implicit connection.	Implicit connection can be viewed in Revision chain.
History	The create entry shows the object from which it was cloned. If you copy an object via MQL, you can optionally include the original object's history log.	The create entry shows object from which it was revised and original shows that it was revised.

Not all business objects can be revised. If revisions are allowed, the Policy definition may specify the scheme for labeling revisions. This scheme can include letters, numbers, or enumerated values. Therefore you can have revisions with labels such as AA, 31, or "1st Rev."

If the `REVISION_NAME` is omitted, Live Collaboration automatically assigns a designator based on the next value in the sequence specified in the object's policy. If there is no sequence defined, an error message results.

If there is no defined revision sequence or if you decide to skip one or more of a sequence's designators, you can manually specify the desired designator as the `REVISION_NAME`.

Adding a Business Object to a Revision Sequence

You can use MQL to add an object to the end of an existing revision sequence using the syntax `revise bus TNR bus TNR1`.

- The first object (`TNR`) must be the last in its revision family. If not, the command will error with `Business object is not revisionable`. Therefore, you cannot insert a revision into the middle of an existing revision family.
- If the second object (`TNR1`) belongs to a revision family, the command will work, but will issue the warning `Business object being inserted will be removed from its previous revision chain`.

Creating new revisions has several side affects. When appending to a revision sequence with the MQL command, these behaviors are handled as described below:

- **Float/Replicate rules**. Ordinarily the float/replicate rules for relationships cause relationships to be moved/copied to new revisions. These rules are ignored; no relationships are added or removed to/from the inserted object, nor are any relationships added or removed to/from any of the previously existing members of the target sequence.

- **File Inheritance**. Ordinarily, a new revision of an existing object inherits all files checked into the original object. When using this interface, if the appended object already has checked in files, it does not inherit any files from the revision sequence. If the appended object has no files checked in, the behavior is controlled by the `file` keyword in MQL.
- **Triggers**. No triggers will fire.
- **History**. Revision history records will be recorded on both the new object and its previous revision (that is, both objects that are specified in the MQL command).

Major/Minor Revisions

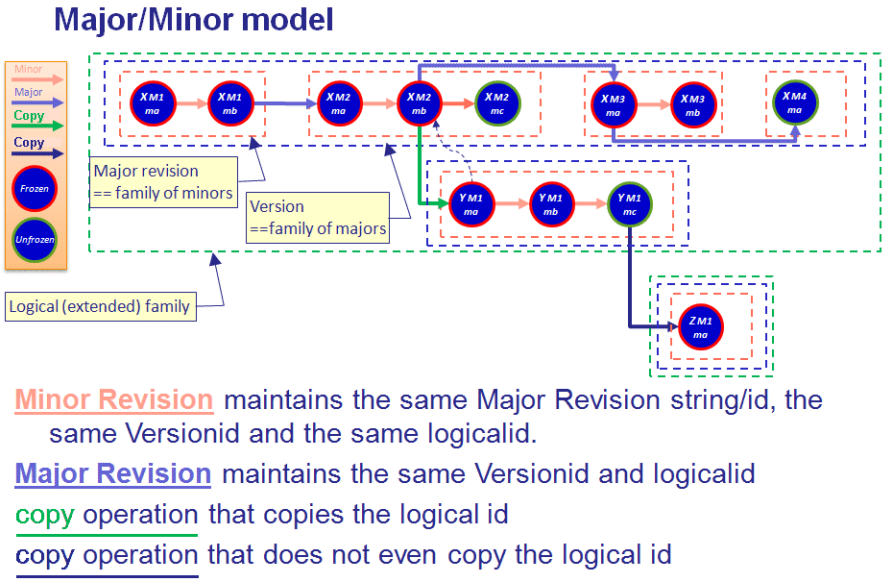
Business objects have two levels of revisioning, minor and major:

- Minor revisions are individual business objects. The Revise Minor command creates a minor revision family as a linear, ordered sequence of objects with a minor revision string that is incremented for each new minor revision.
- Major revisions are equivalent to complete minor revisions. The Revise Major command begins a new major revision (minor revision family) by creating the first minor revision in the new family. An incremented major revision string is generated for each new major revision. You can create as many new major revisions as you want by invoking Revise Major multiple times.

A new UUID called "majorid" is assigned to all objects in a major revision (all objects in the same minor revision family).

The collection of all objects created from an original object using the Revise Major/Minor commands comprises a larger family called a Version Family. A new UUID called VersionId is assigned to all objects in the version family.

The following figure shows an example of the result of a sequence of major and minor revision operations on an object. The orange arrows indicate Revise Minor operations, while the blue arrows indicate Revise Major operations.



The top row of objects shows a version family (enclosed in the dark blue dashed box) comprising four major revisions (i.e., minor revision families enclosed in orange dashed boxes). The four major revisions contain two, three, two, and one minor revisions, respectively.

The second row shows another version family with a single major revision containing three minor revisions. This version family was created through "evolution," or by a Copy operation that copies the logical ID.

The third row shows another version family with a single major revision containing one minor revision. This family was created through a New From operation that did not copy the logical ID.

Validation of MajorID/VersionID during Object Creation

Error checking of majorid/versionid is performed when a business object is added. More specifically, when performing create/clone business object operations (e.g., with the Add Bus command), if the specified majorid or versionid is already associated with other objects in the database, an error is generated. This applies only to Add Bus, not Copy or Revise.

```
add businessobject TYPE NAME REVISION policy POLICY_NAME [ITEM {ITEM}];
…
<< std::endl << " where ITEM is:"
<< std::endl << " | description VALUE |"
…
<< std::endl << " | physicalid UUID |"
<< std::endl << " | logicalid UUID |"
<< std::endl << " | versionid UUID majorid UUID |"
<< std::endl << " | majororder VALUE |"
<< std::endl << " | minororder VALUE |"
…
```

Making Connections Between Business Objects

As described in *Relationships* in Chapter 4, relationship types are created which can be used to link business objects. A *relationship type* is specified when making a *connection* or an instance of that relationship type between two business objects. One business object is labeled as the TO end and one is labeled as the FROM end. When the objects are equivalent, it does not matter which object is assigned to which end. However, in hierarchical relationships, it does matter. Live Collaboration uses the TO and FROM labels to determine the direction of the relationship.

The direction you select makes a difference when you examine or dissolve connections. When you examine an object's connections, you can specify whether or not you want to see objects that lead to or away from the chosen object. When you disconnect objects, you must know which object belongs where. Therefore, you should always refer to the relationship definition when working with connections.

Connections are also used to make associations between files, folders, and modules in DesignSync and business objects in Collaboration and Approvals, if you are using DesignSync stores for source control. These types of connections are generally made in apps which use MQL commands in implementation. For more information, see the *Business Process Services - Common User Guide* : *About DesignSync File Access*.

The commands that make and break connections between business objects are discussed in *businessobject Command* in the programming reference section of this guide. Other commands that can be used to make connections and work with connections in general are described in *Relationships* in Chapter 4.

Preserving Modification Dates

By default when connections are created or deleted (with connect bus or disconnect bus commands), the modification dates of the objects on both ends of the connection are updated, during which time they are locked. You can use the preserve option on both the mql connect and disconnect commands to avoid this update and the locking of the business objects.

Working with Saved Structures

Once you have saved a structure using the *Structure Clause* of the Expand Businessobject command, you can list, print, delete, and save it to another user's workspace using the following commands.

```
list structure;

print structure NAME;

delete structure NAME;

copy structure SRC_NAME DST_NAME [fromuser USER_NAME] [touser
USER_NAME][visible USER_NAME{,USER_NAME}] [overwrite];
```

The `print structure` command displays the results in the same manner as `expand bus` does. For example, after executing the above command (which outputs the data to the MQL

window, as well as saving it as a structure), you could execute the following to generate the output again:

```
print structure "Assigned Parts";
```

Within the print structure command you can also use select clauses on either the business object or the relationship as well as use the output/dump or terse clauses.

The `copy structure` command lets you copy structures to and from any kind of user. Including `overwrite` will replace the copied structure with any structure of the same name that was in the touser's workspace.

If an object has been disconnected or deleted, it is no longer listed as part of the structure. On the other hand, if other objects were connected since the structure was saved, they would not automatically appear in the output of the print structure command. Another expand command would need to be executed with the structure clause to update the structure.

Working with Business Object Files

A business object does not need to have files associated with it. It is possible to have business objects where the object's attributes alone are all that is required or desired. However, there will be many cases where you will want to associate external files with a business object. To make this association, you must check the files into the object. This is called file *checkin*.

Checking in a file allows other users to access a file that might not otherwise be accessible. For example, assume you have a file in your personal directory. You would like to make this file accessible to your local group and the quality assurance group. In a typical computer environment, there is no way to allow selective groups to have access while denying others. You could give the file Group Access or World Access. The Group Access takes care of your immediate group but not the quality assurance group. If you give World Access to the file, ANYONE can access it, not just quality assurance. You can overcome this problem with Live Collaboration.

The policy definition can designate when specified persons, groups, and roles have access to an object. When an object is accessible, any files that are checked into that object are also accessible. Therefore, if a group has read access to an object, they also have read access to any files checked into the object. If the policy definition for an object includes enforced locking, no checkin for that object is allowed until the lock is released, regardless if the file being checked in is going to replace the checked-out file which initiated the lock or not.

When working with checked in files, keep in mind that the copy you check in will not change if you edit your own personal copy. While you maintain the original, any edits that you make to that file will not automatically appear in Live Collaboration. The only way to have those changes visible to other users is to either check in the new version or to make the edits while you are using the 3DEXPERIENCE Platform (i.e., with Open for Edit).

Checking in files is controlled by the Checkin Businessobject command. This command and other associated commands are discussed in *businessobject Command* in the programming reference section of this guide.

Checking Out Files

Once a file is checked in, it can be modified by other users who have editing access (if the object is not locked). This means that the original file you checked in could undergo dramatic changes. As the file is modified, you may want to replace your original copy with one from Live Collaboration, or you may want to edit the file externally. This is done by *checking out* the file.

When a business object file is checked out, a copy is made and placed in the location specified. This copy does not affect the Collaboration and Approvals copy. That file is still available to other users. However now you have your own personal copy to work on.

In some situations, a person may be denied editing access but allowed checkout privilege. This means the user may not be allowed to modify the Collaboration and Approvals copy, but can obtain a personal copy for editing. This ensures that the original copy remains intact. For example, a fax template is checked in but each user can check out the template file, fill in individual information, and fax it.

Handling Large Files

The 3DEXPERIENCE Platform handles the transfer of large files (that is, files larger than 2 gb) for checkin or checkout in exactly the same way they handle smaller files. However, the larger a file is, the longer it takes to check it in.

For HTML/JSP-based applications, including the 3DEXPERIENCE apps and custom Framework programs, large file checkins require both of the following:

- Checkins must be targeted for a FCS-enabled store/location (see *Installing File Collaboration Server* for setup information.)
- Checkins must be invoked via the configurable file upload applet (see *Common Components User Guide* for setup information.)

It is recommended that you enable both FCS and the file upload applet for all implementations, even if you do not foresee working with files larger than 2 gb.

Locking and Unlocking a Business Object

A business object can be locked to prevent other users from editing the files it contains. Even if the policy allows the other user to edit it, a lock prevents file edit access to everyone except the person who applied the lock.

Locking a business object protects the object's contents during editing. When an object is opened for editing, it is automatically locked by Collaboration and Approvals, preventing other users from checking in or deleting files. However, if they have the proper access privileges, other users can view and edit attribute values and connections of the object and change its state.

A lock on an object prohibits access to the files it contains, but still allows the object to be manipulated in other ways.

But, if the checkout and edit are separate actions, the object should be manually locked. Without a lock, two people might change an object's file at the same time. With a lock, only the person who locked the object manually is allowed to check files into the object. As with an automatic lock, other users can view attributes, navigate the relationships, and even checkout an object's file. But, they cannot change the contents by checking in or deleting a file without unlocking the object.

It is possible for a locked object to be unlocked by a user with unlock access. For example, a manager may need to unlock an object locked by an employee who is out sick. See User Access in Chapter 3.

If another user has unlock privileges and decides to take advantage of them, the person who established the lock will be notified via IconMail that the lock has been removed and by which user. This should alert the lock originator to check with the *unlocker* (or the history file) before checking the file back in to be sure that another version of the same file (with the same name and format) has not been checked in, potentially losing all edits made by the unlocker.

If the object is governed by a policy which uses enforced locking, the object must be locked for files to be checked in. Users must remember to lock an object upon checkout if they intend to checkin changes, since the separate lock command will be disabled when locking is enforced.

Modifying the State of a Business Object

The following commands control the movement of a business object into or out of a particular state. A state defines a portion of an object's lifecycle. Depending on which state an object is in, a person, group, or role may or may not have access to the object. In some situations, a group should have access but is prohibited because the object has not been promoted into the next state.

Approve Business Object Command

The Approve Businessobject command provides a required signature. When a state is defined within a policy, a signature can be required for the object to be approved and promoted into the next state. You provide the signature with the Approve Businessobject command. For example, to approve an object containing an application for a bank loan, you might write this Approve Businessobject command:

```
approve businessobject "Car Loan" "Ken Brown" A
   signature "Loan Accepted"
   comment "Approved up to a maximum amount of $20,000";
```

In addition to providing the approving signature, a comment was added to provide additional information regarding the approval in the example above. In this case, the comment informs other users that the object (Ken Brown's car loan) has been approved up to an amount of $20,000. If the customer asks for more, the approval would no longer apply and the bank manager might reject it.

The Approve Businessobject command provides a single approving signature. However, the Approve Businessobject signature does not necessarily mean that the object will be promoted to the next state. It only means that one of the requirements for promotion was addressed. Depending on the state definition, more than one signature may be required.

Ignore Business Object Command

The Ignore Businessobject command bypasses a required signature. In this case, you are not providing an approving or rejecting signature. Instead you are specifying that this required signature can be ignored for this object.

In a policy definition, states are created to serve the majority of business objects of a particular type. This means that you may have some business objects that do not need to adhere to all of the constraints of the policy. For example, you might have a policy for developing software programs. Under this policy, you may have objects that contain programs for customer use and programs for internal use only. In the case of the internal programs, you may not want to require all of the signatures for external programs. Instead, you might be willing to ignore selected signatures since the programs are not of enough importance to warrant them.

Use the Ignore Businessobject command to bypass a required signature. For example, assume you have a simple inventory program to track one group's supplies. The program is highly specialized for internal use and will not be used outside the company. According to the policy governing the object (which contains the program), the company president's signature is required before the program can enter the Released state for business objects outside the company. Since no one wants to bother the president for a signature, you decide to bypass the signature requirement with the following Ignore Businessobject command. (Note that the user must have privileges to do this.)

```
ignore businessobject "Software Program" "In-house Inventory" III
   signature "Full Release"
   comment "Signature is ignored since program is for internal use only";
```

In this command, the reason for the bypass is clearly defined so that users understand the reason for the initial bypass of the signature. As time passes, this information could easily become lost. For that reason, you should include a Comment clause in the command even though it is optional.

The Ignore Businessobject command involves control over a single signature. An object is promoted to the next state automatically when all requirements are met if the state was defined with the Promote clause set to true. Bypassing the Ignore Businessobject signature does not necessarily mean that the object will meet all of the requirements for promotion to the next state. It only means that one of the requirements for promotion was circumvented. Depending on the state definition, another user or condition may be required to approve the program.

Reject Business Object Command

The Reject Businessobject command provides a required signature. In this case, the signature is used to prevent an object from being promoted to the next state.

For example, a bank manager may decide that more clarification is required before s/he will approve of car loan. S/he can enter this information by writing the following Reject Businessobject command:

```
reject businessobject "Car Loan" "Ken Brown" A
   signature "Loan Accepted"
   comment "Need verification of payoff on student loan before I'll approve
      a loan up to a maximum amount of $20,000";
```

Any other users can see the reason for the rejection. Since the signature was provided, the object cannot be promoted unless someone else overrides the signature or the reason for rejection is addressed.

The Reject Businessobject command provides a single signature. However, this signature does not necessarily mean that the object will be demoted or completely prevented from promotion to the next state. It only means that one of the requirements for promotion was denied. Depending on the state definition, another user may override the rejection.

Unsign Signature

The Unsign Signature command is used to erase signatures in the current state of an object. Any user with access to approve, reject, or ignore the signature has access to unsign the signature.

For example the command to unsign the signature "Complete" in object Engineering Order 000234 1 is:

```
unsign businessobject "Engineering Order" 000234 1 signature Complete;
```

Errors will occur under the following conditions:

- Attempts to unsign a signature not yet signed.
- Attempts to unsign all signatures if all are not signed, any that are signed; however, will be unsigned.
- Attempts to unsign a non-existent signature.
- Attempts to unsign a signature without access.

Disable Business Object Command

The Disable Businessobject command holds an object in a particular state indefinitely. When a business object is in a disabled state, it cannot be promoted or demoted from that state. Even if all

of the requirements for promotion or demotion are met, the object cannot change its state until the state is enabled again.

Use the Disable Businessobject command to disable a business object within a state:

```
disable businessobject OBJECTID [state STATE_NAME];
```

OBJECTID is the OID or Type Name Revision of the business object.

STATE_NAME is the name of the state in which you want to freeze the object. If you want to disable the current state, the State clause is not required.

For example, to disable an object containing a component design for an assembly, you could write a Disable Businessobject command as:

```
disable businessobject "Component Design" "Bicycle Seat R21" A
   state "Initial Release";
```

Labeling an object as disabled within a state does not affect the current access. This means that the designer can continue to work on the object in that state.

Enable Business Object Command

The Enable Businessobject command reinstates movement of an object. When a business object is in a disabled state, it cannot be promoted or demoted from that state. If you then decide to allow an object to be promoted or demoted, you must re-enable it using the Enable Businessobject command.

When an object is first created, all states are enabled for that object. If a manager or other user with the required authority decides that some states should be disabled, s/he can prevent promotion with the Disable Businessobject command. After the command is processed, an object will remain trapped within that state until it is enabled again.

Use the Enable Businessobject command to enable a business object within a state:

```
enable businessobject OBJECTID [state STATE_NAME];
```

OBJECTID is the OID or Type Name Revision of the business object.

STATE_NAME is the name of the state in which you want to enable the object. If you want to enable the current state, the State clause is not required.

For example, assume Component Design is disabled. All of the conditions for an Initial Release state have been met and the manager decides it is ready for promotion into the Testing state. Before the object can be promoted, however, it must be enabled. The manager can do this with the following command:

```
enable businessobject "Component Design" "Bicycle Seat R21" A
 state "Initial Release";
```

The object is enabled and available for promotion or demotion.

Enabling an object does not have an effect on the remaining states. For example, the Testing state could be disabled at the same time the Initial Release state was disabled. When the Initial Release state is enabled, the object can be promoted into the Testing state. However, once it is in this new state, it again can no longer be promoted or demoted. It will remain in the Testing state until it is enabled for that state.

Override Business Object Command

The Override Businessobject command turns off the signature requirements and lets you promote the object. Since a policy is a generic outline that addresses the majority of needs, it is possible to have special circumstances in which a user, such as a manager, may decide to skip a state rather than have the object enter the state and try to meet the requirements of the state. This is done using the Override Businessobject command.

Use The following syntax to write an Override Businessobject command:

```
override businessobject OBJECTID [state STATE_NAME];
```

OBJECTID is the OID or Type Name Revision of the business object.

STATE_NAME is the name of the state that you want to skip. If you want to override the current state, the State clause is not required.

For example, assume you have a component that has undergone cosmetic changes only. This component is needed in Final Release although the policy dictates that the component must go through Testing before it can enter that state. Since the changes were cosmetic only, the manager may decide to override the Testing state so that the users can access the component sooner. This could be done by entering a command similar to:

```
override businessobject "Component Design" "Bicycle Seat R21"
   state Testing;
```

Now, assume that the object is currently in the Initial Release state and a promotion would place it in the Testing state. How does the Override Businessobject command affect the object when it is promoted? The object will be promoted directly from the Initial Release state into the Final Release state.

Promote Business Object Command

The Promote Businessobject command moves an object from its current state into the next state. If the policy specifies only one state, an object cannot be promoted. However, if a policy has several states, promotion is the means of transferring an object from one state into the next.

The order in which states are defined is the order in which an object will move when it is promoted. If all of the requirements for a particular state are met, the object can change its state with the Promote Businessobject command.

```
promote businessobject OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the business object.

For example, the following command would promote an object containing an application for a bank loan:

```
promote businessobject "Car Loan" "Ken Brown" A;
```

An object cannot be promoted if:

- Its current state is disabled.
- The signature requirements were not met or overridden (ignored).
- There are no other states beyond the current state.

Demote Business Object Command

The Demote Businessobject command moves an object from its current state back into its previous state. If the policy has only one state or is in the first state, an object cannot be demoted. However, if a policy has several states, demotion is the means of transferring an object backward from one state into the previous state.

If an object reaches a state and you determine that it is not ready for that state, you would want to send the object back for more work. This is the function of the Demote Businessobject command:

```
demote businessobject OBJECTID;
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

For example, assume you have a component that has undergone cosmetic changes only. The manager decides to send it into Final Release without sending it through testing. Now the manager finds out that the new paint trim might weaken the plastic used in the seat. Therefore, the manager decides to demote the object back into the Testing state. This could be done by entering a command similar to:

```
demote businessobject "Component Design" "Bicycle Seat R21" A;
```

When using the Demote Businessobject command, an object cannot be demoted if:

- The current state is disabled.
- The signature requirements for rejection were not met or overridden (ignored).
- There are no other states prior to the current state.

Working with Relationship Instances

Once you have defined relationship types, connections or instances of a relationship type can be made between specific business objects. Refer to *Making Connections Between Business Objects* for the MQL commands: Connect Businessobject and Disconnect Businessobject. Other commands that can be used to make connections and work with connections in general are described in the following section.

Connections are also used to make associations between files, folders and modules in DesignSync and business objects in Collaboration and Approvals, if you are using DesignSync stores for source control. These types of connections are generally made in 3DEXPERIENCE apps, which use MQL commands in implementation. For more information, see the *Common Components User Guide* : *About DesignSync File Access*.

Connections can be accessed by MQL in two ways:

- by specifying the business objects on each end

Or:

- by specifying its connection ID.

The first method will work only if exactly one of a particular relationship type exists between the two listed objects. Therefore, if connections are to be programmatically modified, the second method, using connection IDs, is the safer approach.

Defining a Connection

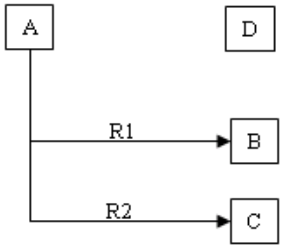
There are two ways of creating a new instance of a relationship: the Connect Businessobject command or the Add Connection command. Use the Add Connection command to make various types of connections between either:

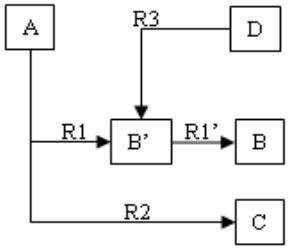
- 2 business objects
- a business object and a connection
- 2 connections

One business object or connection is labeled as the TO or TOREL end and one is labeled as the FROM or FROMREL end. When the objects are equivalent, it does not matter which object is assigned to which end. However, in hierarchical relationships, it does matter. Collaboration and Approvals use the TO and FROM labels to determine the direction of the relationship.

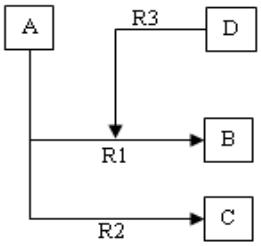
The direction you select makes a difference when you examine or dissolve connections. When you examine an object's connections, you can specify whether or not you want to see objects that lead to or away from the chosen object. When you disconnect objects, you must know which object belongs where. Therefore, you should always refer to the relationship definition when working with connections. To disconnect objects, see *Making Connections Between Business Objects*.

Creating connections that have a connection on at least 1 end eliminates the need to create dummy business objects when modeling schema. For example, if a connection can not be used as an end point, to connect object D to the R1 connection in the diagram below, you would have to create an extra business object and an extra connection so that D is logically tied to R1.





Directly connecting the object to the connection is a streamlined modeling approach resulting in reduced storage requirements and enhanced performance levels.



You can connect an object or connection to or from a connection if you have to/from connect access on the object or connection. Likewise, you can delete (disconnect) or modify the connection with appropriate access on the object or connection that is on one end of it.

You cannot use Matrix Navigator to create connections that have a connection on one or both ends. This must be done using MQL or Studio Customization Toolkit programs.

Add Connection Command

Use the Add Connection command to create new connections. You can specify either a business object or a connection at either end of the connection

```
add connection NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the connection.

`ADD_ITEM` is an Add Connection clause which provides more information about the connection you are creating.

The complete syntax and clauses for working with connections are discussed in *connection Command* in the programming reference section of this guide.

Viewing Connection Definitions

You can view the definition of a connection at any time by using the Print Connection command and the Select Connection command. These commands enable you to view all the files and information used to define the connection. The system attempts to produce output for each select clause input, even if the connection does not have a value for it. If this is the case, an empty field is output

6 Working with Workspace Objects

This chapter discusses all the different types workspace objects.

In this section:

Queries

A *query* is a search on the database for objects that meet the specified criteria. The query is formulated by an individual and, in MQL, it must be saved for subsequent evaluation. A user has access only to queries created during a session under her or his own context. It is then run or *evaluated* and Live Collaboration finds the objects that fit the query specification. The found objects are displayed in Collaboration and Approvals or listed in MQL. If the found objects are often needed as a group, they can be saved in a set which can be loaded at any time during an Collaboration and Approvals or MQL session under the same context.

There are two steps to working with queries:

- *Define* either a saved query that you want to use again, or a temporary query to be used only once for a quick search.
- *Evaluate* the query. The query is processed and any found objects can be put into a set.

Temporary queries allow you to perform a quick search for objects you need only once. In this case you don't have to first save the query itself as an object. For example, you might want to modify a particular object that is named HC-4....., but you have forgotten its full name or capitalization. You could perform a temporary search (without saving the actual query) using "HC-4*" to find all objects that have a name beginning with the letters "HC-4". From the resulting list, you could enter the correct name in your modify command.

Saved queries allow you to find, for example, all drawings created for a particular project. You can save the results of the query in a set. As the project proceeds, you can also name and save the query itself to use again to update the set contents. Or you might want to repeatedly search for any objects having a particular attribute or range of attributes. Saved queries provide a way of finding all the objects that contain the desired information.

Defining a Query

To define a saved query from within MQL, use the Add Query command:

```
add query NAME [user USER_NAME] {ITEM};
```

`NAME` is the name you assign to the query.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a query for another user. If not specified, the query is part of the current user's workspace.

`ITEM` specifies the characteristics to search for.

The complete syntax and clauses for working with queries are discussed in *query Command* in the programming reference section of this guide.

Sets

A *set* is a logical grouping of business objects created by an individual user. Sets are often the result of a query made by a user. For example, the user may want to view all drawings created for a particular project or find information about objects having a particular attribute or range of attributes. While sets can be manually constructed based on the needs and desires of the individual, queries are a fast means of finding related business objects.

The contents of a set can be viewed at any time and objects can be added or deleted from a set easily. However, a user has access only to sets created while in session under his/her context.

Understanding Differences

It is important to realize that sets are not the same as connections between business objects. Connections also group business objects together in a window for users to analyze; however, connections are globally known links rather than local links.

Business Object Connection	Set
Created only if the policy permits.	Created without any special privileges.
Available to view by all users who have access to the objects.	Available to view at any time by the person who created the set.
Valuable to all users who have access to the objects.	Valuable only within the context of the person who created it.

Defining a Set

Sets are often the result of a query. To save the contents of a query in a set, see *Evaluate Query*.

To manually define a set from within MQL, use the Add Set command:

```
add set NAME [user USER_NAME][ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the set.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a set for another user. If not specified, the set is part of the current user's workspace.

`ADD_ITEM` is an Add Set clause that provides more information about the set you are creating.

The complete syntax and clauses for working with sets are discussed in *set Command* in the programming reference section of this guide.

Filters

Filters limit the objects or relationships displayed in browsers to those that meet certain conditions previously set by you or your Business Administrator. For example, you could create a filter that would display only objects in a certain state (such as Active), and only the relationships connected *toward* each object (not *to and from*). When this filter is turned on, only the objects you needed to perform a specific task would display.

From Matrix Navigator browsers, filters that limit the number of objects that display can be very useful. Each user can create her/his own filters from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar filters consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Filters can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. Once the filters are defined, individual users can turn them on and off from the browsers as they are needed.

In the Matrix Navigator browsers, filters display on the Filters tab page within the Visuals Manager window, in the Filter bar and in the View menu.

It is important to note that filters are Personal Settings that are available and activated only when context is set to the person who defined them.

Defining Filters

To define a new filter from within MQL, use the Add Filter command:

```
add filter NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the filter.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a filter for another user. If not specified, the cue is part of the current user's workspace.

`ADD_ITEM` specifies the characteristics you are setting.

The complete syntax and clauses for working with filters are discussed in *filter Command* in the programming reference section of this guide.

Cues

Cues control the appearance of business objects and relationships inside any browser. They make certain objects and relationships stand out visually for the user who created them.

This appearance control is based on conditions that you specify such as attribute value, current state, or lateness. Objects and relationships that meet the criteria may appear in any distinct color, line, or font style.

You can save a cue and not make it active. This allows for multiple or different sets of conditions to be used at different times or as a part of different Views. See “Using View Manager” in the *Matrix Navigator Guide*. Cues are set using the Visuals Manager and saved as a query in your personal settings. They can be activated from the Visuals Manager Cue tab or from the View menu.

From Matrix Navigator browsers, cues that highlight the appearance of certain objects can be very useful. Each user can create her/his own cues from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar cues consistently, it may be easier to create them in MQL, then copy the code to each user’s personal settings (by setting context).

Cues can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. Once the cues are defined, individual users can turn them on and off from the browsers as they are needed.

In the Matrix Navigator browsers, they display on the Cues tab page within the Visuals Manager window and in the View menu.

It is important to note that cues are all Personal Settings that are available and activated only when context is set to the person who defined them.

Defining a Cue

From browsers, unique cues can be very useful when interacting with many objects. Each user can create her/his own cues from Matrix Navigator (or MQL). However, if your organization wants all users to see a basic set of similar cues consistently, it may be easier to create them in MQL, then copy the code to each user’s personal settings (by setting context).

To create a new cue from within MQL, use the Add Cue command:

```
add cue NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the nameyou assign to the cue.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a cue for another user. If not specified, the cue is part of the current user’s workspace.

`ITEM` specifies the characteristics you are setting.

The complete syntax and clauses for working with cues are discussed in *cue Command* in the programming reference section of this guide.

Inquiries

Inquiries can be evaluated to produce a list of objects to be loaded into a table in a JSP application. In general, the idea is to produce a list of business object ids, since they are the fastest way of identifying objects for loading into browsers. Inquiries include code, which is generally defined as an MQL `temp query` or `expand bus` command, as well as information on how to parse the returned results into a list of OIDs.

Defining an Inquiry

Business Administrators can create new inquiry objects if they have the Inquiry administrative access. To create a new inquiry from within MQL, use the Add Inquiry command:

```
add inquiry NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the nameyou assign to the inquiry.

`ITEM` specifies the characteristics you are setting.

The complete syntax and clauses for working with inquiries are discussed in *inquiry Command* in the programming reference section of this guide.

7 Extending an Application

This chapter discusses concepts on administrative objects used for extending applications.

Programs

A *program* is an object created by a Business Administrator to execute specific commands. Programs are used:

- In format definitions for the edit, view, and print commands.
- As Action, Check, or Override Event Triggers, or as actions or checks in the lifecycle of a policy.
- To run as *methods* associated with certain object types. (Refer to *Types* in Chapter 4, for the procedure to associate a program with a type.)
- In Business Wizards, both as components of the wizard and to provide the functionality of the wizard.
- To populate attribute ranges with dynamic values.
- In expressions used in access filters, where clauses and configurable tables.

Many programs installed with the Framework include Java code, which are invoked while performing operations with apps. This type of program is defined as *java*. The majority of your programs should be Java programs (JPO), particularly if your users are accessing the 3DEXPERIENCE Platformwith a Web browser.

Some programs might execute operating system commands. This type of program is *external*. Examples are programs such as a word processor or a CAD program which can be specified as the program to be used for the edit, view, and print commands in a format definition.

Other programs might use only MQL/Tcl commands (although this technology is older, and Java programs written with the Studio Customization Toolkit will perform better, particularly in a Web environment.) For example, a check on a state might verify the existence of an object using an MQL program.

Some programs may require a business object as the *context* or starting point of the commands. An example of this is a program that connects a business object to another object.

Java Program Objects

A Java Program Object (JPO) contains code written in the Java language. JPOs provide the ability to run Java programs natively inside the kernel, without creating a separate process and with a true object-oriented integration ‘feel’ as opposed to working in MQL. JPOs allow developers to write Java programs using the Studio Customization Toolkit programming interface and have those programs invoked dynamically.

When running inside the Live Collaboration kernel, programs share the same context and transaction as the thread from which they were launched. In fact, Java programs run inside the same Java Virtual Machine (JVM) as the kernel.

JPOs are also tightly integrated with the scripting facilities of the 3DEXPERIENCE Platform. Developers can seamlessly combine MQL, Tcl, and Java to implement their business model. However, while Tcl will continue to be supported and does offer a scripting approach which can makes sense in some cases, the Java language brings several advantages over Tcl:

- Java is compiled, and therefore offers better run-time performance
- Java is object-oriented
- Java is thread safe

Java code must be contained in a class. In general, a single class will make up the code of a JPO. However, simply translating Tcl program objects into Java is not the goal. This would lead to many small classes, each containing a single method. The very object-oriented nature of Java lends itself to encapsulating multiple methods into a single class. The goal is to encapsulate common code into a single class.

A JPO can be written to provide any logical set of utilities. For example, there might be a JPO that provides access to Administration objects that are not available in the Studio Customization Toolkit. But for the most part, a JPO will be associated with a particular Business Type. The name of the JPO should contain the name of the Business Type (but this is certainly not required).

It is the responsibility of the JPO programmer to manually create a JPO and write all of the methods inside the JPO. Keep in mind that JPO code should be considered server-side Java; no Java GUI components should ever be constructed in a JPO.

For more details about writing JPO code, see the *Configuration Guide* : *About Java Program Objects (JPOs)*.

Defining a Program

A program is created with the Add Program command:

```
add program NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the program.

`ADD_ITEM` is an Add Program clause which provides additional information about the program you are creating.

The complete syntax and clauses for working with programs are discussed in *program Command* in the programming reference section of this guide.

Using Programs

For information on using programs in an implementation, see the *Configuration Guide* : *About Programs*. It contains key information including:

- compiling programs
- extracting and inserting JPO code
- passing arguments
- Java options

The sections that follow include some basic MQL syntax for the above functionality.

Compile command

To force compilation before invoking a JPO method, use `compile program`. This is useful for bulk compiling and testing for compile errors after an iterative change to the JPO source.

```
compile program PATTERN [force] [update] [COMPILER_FLAGS];
```

When a JPO is compiled or executed, any other JPOs which are called by that JPO or call that JPO must be available in their most recent version. The compile command includes an update option which will update the requested JPO's dependencies on other JPOs that may have been added, deleted, or modified.

Execute command

You can run a program with the execute program command:

```
exec program PROGRAM_NAME [-method METHOD_NAME] [ARGS]
[-construct ARG];
```

where:

ARGS is zero or more space-delimited strings.

The `-construct` clause is used to pass arguments to the constructor. ARG is a single string. If more than one argument needs to be passed to the constructor, the `-construct` clause can be repeated as many times as necessary on a single command.

Extract command

Extracting the Java source in the form of a file out to a directory is useful for working in an IDE. While in the IDE a user can edit, compile, and debug code. The `extract program` command processes any special macros and generates a file containing the Java source of the JPO. If no source directory is given, the system uses ENOVIA_INSTALL/java/custom (which is added to `MX_CLASSPATH` automatically by the install program).

```
extract program PATTERN [source DIRECTORY]
```

In order to use the extract feature, the JPO name must follow the Java language naming convention (i.e., no spaces, special characters, etc.). Only alphanumeric printable characters as well as '.' and '_' are allowed in class names in the JPO.

Insert command

After testing and modifying Java source in an IDE, it is necessary to insert the code back into JPOs in the database. The `insert program` command regenerates special macros in the Java source as it is placed back into a JPO (reverse name-mangling). If the JPO does not exist, the `insert` command creates it automatically.

```
insert program FILENAME | DIRECTORY;
```

For example:

```
insert program matrix/java/custom/testjpo_mxJPO.java
```

OR

```
insert program matrix/java/custom/
```

The later will insert all the .java files in the specified directory.

History

Collaboration and Approvals provides a history for each business object, detailing every activity that has taken place since the object was created.

An object's historical information includes:

- The type of activity performed on the object, such as create, modify, connect, and so on.
- The user who initiated the activity.
- The date and time of the activity.
- The object's state when the activity took place.
- Any attributes applicable when the activity took place.

You can add a customized history through MQL to track certain events, either manually or programmatically.

History records can be selected when printing or expanding business object, set, or connection information using the MQL `select` clause. In addition, when printing all information about a business object, set, or connection, history records can be excluded.

System administrators only can purge the history records of a business object or connection via MQL. In addition, *all* history records of a business object or connection can be deleted with one command.

History can be turned off in a single session by a System Administrator for the duration of the session or until turned back on. In addition, if an implementation does not require history recording, or requires only custom history entries, Collaboration and Approvals "standard" history can be disabled for the entire system. Turning history recording off can improve performance in very large databases, though certain standards may require that it is turned on.

History entries larger than 255 characters are truncated to 255 characters. This includes custom history entries as well as Collaboration and Approvals history entries. This means that history logs for the modification of long description or string attribute fields may be truncated.

When objects are created and then immediately modified within a trigger, the timestamp is often identical. When this happens, the modify event may be logged before the create event, although both will have the same timestamp.

The syntax and clauses for working with history are discussed in *history Command* in the programming reference section of this guide.

Administrative Properties

Ad hoc attributes, called Properties, can be assigned to an administrative object by business administrators with Property access. Properties allow links to exist between administrative definitions that aren't already associated. There are two kinds of properties:

- *User properties*, which can be created by users to suit their needs. They can apply to all administrative objects, including workspace objects.
- *System properties*, which come with Collaboration and Approvals. These properties are used internally to implement certain kinds of administrative objects. For example, toolsets point to their programs via a property; views point to their components in the same way.

Properties may be useful for developers who are integrating Collaboration and Approvals to other application programs. However, the typical Business Administrator may never have the need to use properties.

Properties can be created, modified, displayed, and deleted only through MQL. However, MQL can be embedded in programs, `where` clauses and `select` clauses, making properties available to a broader audience.

Defining a Property

Properties can be created and attached to an object at the same time using the `add property` command. A property must have a name and be "on" an object. It can, optionally, define a link to another administrative object using the "to" clause. This command, therefore, takes two forms, with and without the "to" clause.

```
add property NAME on ADMIN_TYPE ADMIN_NAME [system] [to
ADMIN_TYPE ADMIN_NAME [system]] [value VALUE];
```

`NAME` is the name of the new property.

`ADMIN_TYPE` is the keyword for an administrative or workspace object:

association	group	policy	site	view
attribute	index	program	store	wizard
command	inquiry	query	table	
cue	location	relationship	tip	
filter	menu	role	toolset	
form	page	rule	type	
format	person	set	vault	

`ADMIN_NAME` is the name of the administrative object instance.

The `to ADMIN_TYPE ADMIN_NAME` is optional.

`system` is used only when adding properties on/to system tables.

`VALUE` is a string value of the property. The "value" clause is optional. The value string can contain up to 2gb of data.

The complete syntax and clauses for working with properties are discussed in *property Command* in the programming reference section of this guide.

Applications

You can design your application's data model such that you can mark part or all of it as private or protected. The data you mark private cannot be accessed from any other project while the data you mark protected can only be viewed and not modified. If your company is working on a top secret government project, you will need to mark the data connected to this project as private to avoid any accidental viewing or modification of data. You can do this by assigning an owning application to your project.

An *application* is a collection of administrative objects (attributes, relationships and types) defined by the Business Administrator and assigned to a project. It acts as a central place where all application dependent associations to these other administration objects are defined. The application members (the types, relationships etc) can have different levels of protection (private, protected or public). This protection also extends to the objects governed by them. For example, if you mark a relationship as protected, all connections of that type will also get marked as protected.

The advantage of assigning an owning application to a project is that it ensures data integrity. All modifications to the data are handled by the owning application code which performs all necessary checks to ensure complete data integrity. This prevents any unintended mishandling and possible corruption of data. Thus, when an Collaboration and Approvals product (or any custom application) tries to access data marked private or protected by connecting to the Server, it has to specify the name of the owning application containing that data. You can specify an application name only in the Studio Customization Toolkit.

In MQL and Business Modeler, a person can be assigned an owning application.

Defining an Application

An application is created with the Add Application command:

```
add application NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the application.

`ADD_ITEM` is an Add Application clause which provides additional information about the application you are creating.

The complete syntax and clauses for working with applications are discussed in *application Command* in the programming reference section of this guide.

Dataobjects

Dataobjects are a type of workspace object that provide a storage space for preference settings and other stored values for users. Apps will use them for cached values for form fields, as well as to add personalized pages to channels in Powerviews. Refer to *Channels* in Chapter 8 for more information.

Settings stored in dataobjects are not limited in length.

Defining a Dataobject

Dataobjects can be created in MQL only.

To create a new dataobject, use the Add dataobject command:

```
add dataobject NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the dataobject.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a dataobject for another user. If not specified, the dataobject is part of the current user's workspace.

`ADD_ITEM` further defines the dataobject.

The complete syntax and clauses for working withdataobjects are discussed in *dataobject Command* in the programming reference section of this guide.

Webreports

A webreport is a workspace object that can be created and modified in MQL or using the WebReport class in the Studio Customization Toolkit. Webreports are used to obtain a set of statistics about a collection of business objects or connections. The administrative definition of a webreport includes:

- search criteria which specifies the full set of objects to be examined.
- one or more groupby criteria which specify how to organize the objects into groups
- one or more data expressions to be calculated on each group. These are expressions suitable for evaluating against a collection of business objects – such as count, average, maximum, minimum, etc.

Once a webreport is created it can be evaluated to produce a webreport result, which consists of both the organized set of data values and objects for the subgroups. It can be saved to the database (with or without the corresponding business objects) if desired. Webreport results can also be archived and a webreport can store any number of archived results as well as a single result referred to as “the result”. Webreports are used mainly by the Business Metrics component of Business Process Services, but can be used in custom applications as well.

Defining a Webreport

To define a webreport from within MQL use the add webreport command:

```
add webreport NAME [user USER_NAME][ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the webreport.

`ADD_ITEM` is an add webreport clause that provides additional information about the webreport.

The complete syntax and clauses for working with webreports are discussed in *webreport Command* in the programming reference section of this guide.

Evaluating Webreports

When a webreport is evaluated, it can either report the results back to the caller, save them in the database, or both. When results are saved, so is the following information:

- the date/time evaluated;
- the time it took to evaluate;
- the name of the person who ran it (the context user).

Also when results are saved, they overwrite any previously saved results unless those results have been archived. Any number of results can be archived.

Webreport XML Result

There is a limitation on the size of the XML result string for a webreport on a non-Oracle database. If the XML result string exceeds 2097152 characters, an error may occur.

Since the webreport XML result is summarizing the data values for all of the subgrouping implied by the webreport definition, there are two cases where this XML result can get somewhat lengthy, and these should be avoided on non-Oracle databases:

- If any single groupby expression takes on a very large number of distinct values across the objects in a searchcriteria, such as if a groupby is defined in terms of an unconstrained attribute (no range value definitions).

  Also, groupby=owner could take on 1000’s of values, and groupby=id will result in each object being in its own subgrouping.
- If the webreport has a large number of groupbys. The total number of cells (or subgroupings) in a webreport will be equal to N1 * N2 * Nk, where N1 represents the distinct values the first groupby expression takes on, N2 represents the distinct values the second groupby expression takes on, etc.

8 Modeling Web Elements

This chapter discusses concepts on administrative objects used for modeling web elements.

Commands

Business Administrators can create new command objects if they have the Menu administrative access. Commands can be used in any kind of menu in a JSP application. Commands may or may not contain code, but they always indicate how to generate another Web page. Commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it.

Commands can be role-based, that is, only shown to particular users. For example, a number of commands may only be available when a person is logged in as a user defined as Administrator. When no users are specified in the command definitions, they are globally available to all users.

Defining a Command

To define a command from within MQL use the Add Command command:

```
add command NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the nameyou assign to the command.

You cannot have both a command and a menu with the same name.

`ADD_ITEM` is an Add Command clause that provides additional information about the command.

The complete syntax and clauses for working with commands are discussed in *command Command* in the programming reference section of this guide.

Using Macros and Expressions in Configurable Components

Many strings used in the definition of configurable Components (such as label values, hrefs, and settings) can contain embedded macros and select clauses. The ${} delimiters identify macro names. Macros are evaluated at run-time. Macros for configurable components are available for directory specification. Some existing macros are also supported (refer to *Supported Macros and Selects* for more information).

Some strings can also include `select` clauses which are evaluated against the appropriate business object at run-time. The $<> delimiters identify select clauses. Because the select clauses will generally use symbolic names, the clauses will be preprocessed to perform any substitutions before submitting for evaluation. The following example shows a macro being used in the href definition and another macro being used in the Image setting, as well as a select clause being used in the label definition of a tree menu (associated with a LineItem object):

```
MQL<2>print menu type_LineItem;
menu type_LineItem
  description
  label '$<attribute[attribute_EnteredName].value>'
  href '${SUITE_DIR}/emxQuoteLineItemDetailsFS.jsp'
  setting Image value ${COMMON_DIR}/iconSmallRFQLineItem.gif
  setting Registered Suite value SupplierCentralSourcing
  children
    command SCSAttachment
    command SCSAttributeGroup
    command SCSHistory
```

```
    command SCSSupplierExclusion
    command SCSUDA
  nothidden
  property original name value type_LineItem
  property installed date value 02-28-2002
  property installer value MatrixOneEngineering
  property version value Verdi-0-0-0
  property application value Sourcing
  created Thu Feb 28, 2002 11:12:34 AM EST
  modified Thu Feb 28, 2002 11:12:34 AM EST
```

The following example shows a typical business object macro being used in the label definition of a tree menu (associated with a Company object):

```
MQL<3>print menu type_Company;
menu type_Company
  description
  label '${NAME}'
  href '${SUITE_DIR}/emxTeamCompanyDetailsFS.jsp'
  setting Image value ${COMMON_DIR}/iconSmallOrganization.gif
  setting Registered Suite value TeamCentral
  children
    command TMCBusinessUnit
    command TMCLocation
    command TMCPeople
  nothidden
  property original name value type_Company
  property installed date value 02-28-2002
  property installer value MatrixOneEngineering
  property version value Verdi-0-0-0
  property application value TeamCentral
  created Thu Feb 28, 2002 11:31:57 AM EST
  modified Thu Feb 28, 2002 11:31:57 AM EST
```

When using a macro, surround it with quotes to ensure proper substitution if a value contains spaces.

Supported Macros and Selects

The following sections provide lists of macros used in the configuration parameters of the administrative menu and command objects found in the Framework. These menu and command objects are used for configuring the menus/trees in the apps that use them. These are the only macros currently supported for use in any dynamic UI component.

Directory Macros

The following table provides the list of directory specific macros used in the configuration setting.

Directory Macros	
Macro Name	**Description**
${COMMON_DIR}	To substitute the “common” directory below “ematrix” directory. The substitution is done with reference to any application specific directory and it is relative to the current directory.

Directory Macros	
${ROOT_DIR}	To substitute the “ematrix” directory. The substitution is done with reference to any application specific directory below “ematrix” and it is relative to the current directory.
${SUITE_DIR}	The macro to substitute the application specific directory below “ematrix” directory. The substitution is done based on the “Suite” to which the command belongs. and it is relative to the current directory.

Select Expression Macros

Select expression macros are defined as $<SELECT EXPRESSION>, where the select expression can be any valid MQL select command. Select expression macros can be used in labels for configurable components and in expression parameters. These expressions are evaluated at runtime against the current business object ID and relationship ID that is passed in. Some examples include:

- $<TYPE>
- $<NAME>
- $<REVISION>
- $<attribute[attribute_Originator].value>
- $<attribute[FindNumber].value>
- $<from[relationship_EBOM].to.name>

Menus

Business Administrators can create new menu objects if they have the Menu administrative access. Menus can be used in custom Java applications. Before creating a menu, you must define the commands that it will contain, since commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it. Menus can be designed to be toolbars, action bars, or drop-down lists of commands.

Creating a Menu

To define a menu from within MQL use the Add Menu command:

```
add menu NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the menu.

You cannot have both a command and a menu with the same name.

`ADD_ITEM` is an Add Menu clause that provides additional information about the menu.

The complete syntax and clauses for working with menus are discussed in *menu Command* in the programming reference section of this guide.

Channels

Channels are essentially a collection of commands. They differ from menus in that they are not designed for use directly in an application, but are used to define the contents of a *portal*. Channels and portals are installed with the Framework and used in apps to display PowerView pages, but may also be created for use in custom Java applications.

The use of the term "Portal" here refers to the administration object, and not to the broader internet definition described in JSR 168.

Business Administrators can create channel objects if they have the portal administrative access. Since commands are child objects of channels, commands are created first and then added to channel definitions, similar to the association between types and attributes. Likewise, channels are created before portals and then added to portal objects. Changes made in any definition are instantly available to the applications that use it.

Channels created in MQL can optionally be defined as a user's workspace item.

Creating a Channel

To define a channel from within MQL use the add channel command:

```
add channel NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the channel.

You cannot have both a channel and a portal with the same name.

`USER_NAME` can be included if you are a business administrator with person/group/role access defining a channel for another user. This user is the item's "owner." If the user clause is not included, the channel is a system item.

`ADD_ITEM` is an add channel clause that provides additional information about the channel.

The complete syntax and clauses for working with channels are discussed in *channel Command* in the programming reference section of this guide.

Portals

A *portal* is a collection of *channel*s, as well as the information needed to place them on a Web page. Some portals are installed with the Framework and used in apps to display PowerView pages, but they may also be created for use in custom Java applications. Business Administrators can create portal objects if they have the portal administrative access.

The use of the term "Portal" in this chapter refers to the administration object, and not to the broader internet definition described in JSR 168. In headings in the documentation we use the terms Live Collaboration portal and Public portal when a discernment needs to be made.

Before creating a portal, the channels that it will contain must be defined. Channels are child objects of portals — channels are created first and then added to portal definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it.

Portals created in MQL can optionally be defined as a user's workspace item.

Creating a Portal

To define a portal from within MQL use the add portal command:

```
add portal NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the portal.

`USER_NAME` can be included if you are a business administrator with person/group/role access defining a channel for another user. This user is the item's "owner." If the user clause is not included, the portal is a system item.

`ADD_ITEM` is an add portal clause that provides additional information about the portal.

The complete syntax and clauses for working with portals are discussed in *portal Command* in the programming reference section of this guide.

Tables

Tables can be defined to display multiple business objects and related information. Each row of the table represents one business object. Expressions are used to define the columns of data that are presented about the business objects in each row. When you define a table, you determine the number and contents of your table columns.

There are two kinds of tables:

- *A User* table is a user-defined template of columns that can be used when objects are displayed in the Matrix Navigator (Details browser mode only), or in MQL with the Print Table command.

  User tables are created in MQL, or in Matrix Navigator's Visuals Manager and displayed when Matrix Navigator is in details mode. In Matrix Navigator, tables that users create are available from the Views/Tables menu. This enables viewing different information about the same object with a single mouse-click.

  Tables can be added as part of the definition for a customized View. Each time that View is activated, the Table defined for it displays.

  Users can save Table definitions by name (as personal settings) to turn on or off as needed. A single table may become part of several Views, being activated in one, and available in another. Refer to the *Matrix Navigator Guide : Using View Manager* for more information on Views.

  From Matrix Navigator browsers, tables that present information in a familiar format can be very useful. Each user can create her/his own tables from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar tables consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).
- *A System* table is an Administrator-defined template of columns that can be used in custom applications. These tables are available for system-wide use, and not associated with the session context. Each column has several parameters where you can define the contents of the column, link data (href and alt), user access, and other settings. For example, a user could click on a link called Parts to display a system table containing a list of parts. The other columns in the table could contain descriptions, lifecycle states, and owners.

  System tables are created by business administrators that have Table administrative access, and are displayed when called within a custom application.

  System table columns can be role-based, that is, only shown to particular users. For example, the Parts table might have a Disposition Codes column that is shown only when a person is logged in as a user defined as a Design Engineer. Or a user defined as a Buyer might be shown a column in a table that is not seen by a Supplier user. When no users are specified in the command and table definitions, they are globally available to all users.

When business objects are loaded into a table, Collaboration and Approvals evaluate the table expressions for each object, and fills in the table cells accordingly. Expressions may also apply to relationships, but these columns are only filled in when the table is used in a Navigator window. You can sort business objects by their column contents by clicking on the column header.

Creating a Table

To define a table from within MQL use the Add Table command:

```
add table NAME user USER_NAME [ADD_ITEM {ADD_ITEM}];
```

Or

```
add table NAME system [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to table.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a table for another user. I

`system` refers to a table that is available for system-wide use, and not associated with the session context.

You must include either the user or system keyword.

`ADD_ITEM` is an Add Table clause which provides additional information about the table.

The complete syntax and clauses for working with tables are discussed in *table Command* in the programming reference section of this guide.

Forms

A *form* is a window in which information related to an object is displayed. The Business Administrator designs the form, determining the information to be presented as well as the layout. Forms can be created for specific object types, since the data contained in different types can vary greatly. In addition, one object type may have several forms associated with it, each displaying different collections of information. When a user attempts to display information about an object by using a form, Collaboration and Approvals only offer those forms that are valid for the selected object.

Expressions can be used in the form's field definitions to navigate the selected object's connections and display information from the relationship (attributes) or from the objects found at their ends. Forms can be used as a means of inputting or editing the attributes of the selected object. However, attributes of related objects cannot be edited in a form.

A special type of form called a Web form can be created for use in custom applications. Each field has several parameters where you can define the contents of the field, link data, user access, and other settings.

You must be a Business Administrator to define a form, and have form administrative access.

Defining a Form

Use the Add Form command to define a form:

```
add form NAME [web] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the form you are creating.

`web` is used when creating a "Web form." This distinguishes forms to be used in HTML/JSP applications from those used in the Studio Modeling Platform and PowerWeb.

`ADD_ITEM` is an Add Form clause which provides additional information about the form.

The complete syntax and clauses for working with forms are discussed in *form Command* in the programming reference section of this guide.

9 Maintenance

This chapter discusses database maintenance issues.

Maintaining the System

The duties of the System Administrator are to maintain and troubleshoot the system. The System Administrator should also be the on-site point of contact for all software updates, revisions, and customer support requests. Specific duties include:

- *Controlling System-wide Settings*
- *Validating the Live Collaboration Database*
- Maintaining and monitoring clients (See the *Administration Guide : Maintaining and Monitoring Clients*.)
- Monitoring server diagnostics (See the *Administration Guide : Diagnostic Commands*.)
- *Developing a Backup Strategy*

Other responsibilities are covered in these chapters:

- *Building the System Basics* in Chapter 2
- *Working with Import and Export* in Chapter 10

Controlling System-wide Settings

Password requirements can be set for the entire system, as described in *System-Wide Password Settings*. Triggers may be turned on or off with the triggers off command. In addition, System Administrators can control certain other settings that affect the system as a whole using the set system command:

```
set system SYSTEM_SETTING;
```

The specific settings are discussed below.

The complete syntax for setting the system as well as the List and Print System commands are discussed in *set system Command* in the programming reference section of this guide.

Case-Sensitive Mode

Collaboration and Approvals by default is case-sensitive. However, when using Oracle Enterprise Edition, Oracle Standard Edition, or SQL Server, Collaboration and Approvals can be configured to operate in case-insensitive mode.

Oracle Standard Edition is not available for 64-bit operating systems. Oracle Enterprise Edition is recommended for maximum scalability and performance.

Collaboration and Approvals *must remain case-sensitive when using DB2 databases. DB2 is case-sensitive only.*

The schema must be at version 10.5 or higher to turn `casesensitive` off.

Oracle setup

New and existing Oracle databases must be set up to use a function-based index, if you want to work in a case insensitive environment.

To configure Oracle to work in a case-insensitive mode

1. Modify the Collaboration and Approvals user to add the QUERY REWRITE system privilege:
   ```
   grant query rewrite to Matrix;
   ```
2. In the init<SID>.ora file, add the following lines:
   ```
   QUERY_REWRITE_ENABLED=TRUE
   QUERY_REWRITE_INTEGRITY=TRUSTED
   ```
3. You must configure Collaboration and Approvals to turn off case-sensitivity, as described in *Live Collaboration setup*.

SQL Server Setup

By default, SQL Server is supported as case-insensitive. Case sensitivity is set by configuring the character collation set of the underlying database.

To configure SQL Server to work in case-sensitive mode

1. Execute the following SQL to get the available collations. Case-sensitive collations will have "CS" in the name.
   ```
   select * from fn_helpcollations()
   ```
   For example, Latin1_General_CS_AS is a case-sensitive Latin1 collation.
2. For a new database, use the selected collation to configure the database:
   ```
   create database db_name collate collation
   ```
   Where db_name is the name of your database and collation is the selected collation.

   For an existing database, use the selected collation to configure the database:
   ```
   alter database db_name collate collation
   ```
   Where db_name is the name of your database and collation is the selected collation.
3. Verify that case-sensitivity is turned On by executing the following commands in MQL. (You should do a `clear all` before executing these commands):
   ```
   set context user creator;
   set system casesensitive on;
   print system casesensitive;
   CaseSensitive=On
   ```
4. Close MQL and restart it.

It is very important that MQL be closed and restarted after setting the `system casesensitive` setting to On.

Live Collaboration setup

Live Collaboration/Oracle databases must be at schema Version 10.5 or higher to be able to turn case sensitivity off. For databases older than 10.5, you must run the MQL upgrade command with Version 10.5 or higher. You also must convert unique constraints to unique indices in Oracle.

There is the real possibility that duplicate records will exist if you have used the database before turning off the `casesensitive` setting. For example, in a case sensitive environment, you could have users named "bill" and "Bill", which would cause unique constraint violations if an attempt was made to make this database case insensitive (add unique indices). The MQL command `validate unique` is provided to identify these conflicts so that they can be resolved prior to turning off the setting. Once resolved, you then turn off the `casesensitive` system setting, which defaults to on, and reindex the vaults to switch to unique indices.

You can run the `validate unique` *command against pre-10.5 databases using MQL 10.5 or higher to identify conflicts before upgrading.*

To make a database case-insensitive

1. In MQL, run `validate unique` to identify conflicts and resolve any issues by changing some names or deleting unneeded items. For example, the output might show:

   ```
   Duplicate person 'Bill'
   Duplicate program 'test'
   Duplicate state 'Open' in policy 'Incident'
   Duplicate signature 'Accept Quality Engineer Assignment' on state
   'Assign' in policy 'Incident'
   Duplicate signature 'Accept Quality Engineer Assignment' on state
   'Assign' in policy 'Incident'
   Duplicate business type 'CLASS'
   Duplicate business object 'RequestReport' 'Unassigned' '1'
   Duplicate business object 'Document' 'Financials' 'Q32003'
   Duplicate business object 'Test Suite' 'Vault' ''
   Duplicate business object 'Milestone' 'Covers For Manuals' ''
   Duplicate business object 'Task' 'Regression testing' '0'
   ```

   You should find all of these objects (in Matrix Navigator, Business or System) and resolve them by deleting or changing the name of one of the duplicates. Note that duplicate signatures may be reported more than once. This is dependent on the number of unique signers on the signature for all actions (approve, reject, ignore).

   You may also find Person workspace objects such as:

   ```
   Duplicate table 'Cost PRS' owned by ganley
   Duplicate BusinessObjectQuery 'a' owned by coronella
   Duplicate VisualCue 'committed' owned by maynes
   Duplicate ObjectTip 'Description' owned by liu
   Duplicate Filter 'feature' owned by zique
   Duplicate BusinessObjectSet 'Current Software' owned by powers
   Duplicate ProgramSet 'New Bug' owned by oconnor
   ```

   These must be resolved by the user specified as the owner.

   All duplicates, including administrative objects, business objects, and workspace objects must be resolved before preceding.

2. Identify and resolve existing attribute and policy definitions to be sure the rules you set are what you want. See *Collaboration and Approvals with Casesensitive Off* for details.

3. In MQL, run `set system casesensitive off;`

4. Reindex all vaults to create case insensitive indices. For very large vaults this may take several hours. You can run `validate index vault VAULTNAME;` for information on what will occur. Refer to *Index Vault* for more information.

If duplicates exist an error occurs during the vault index.

5. Calculate or update statistics on all tables by executing the following MQL command:

   ```
   <mql> validate level 4;
   ```

   The validate level 4 command should be run after any substantial data load operation, and periodically thereafter, in order to update the statistics. Refer to *Validating the Live Collaboration Database* for details.

Live Collaboration is now ready to use in case-insensitive mode.

Adaplets

Adaplets generate SQL independently of Live Collaboration, and they may be configured to be either case sensitive or insensitive. To make an adaplet operate in case insensitive mode, add the following line to the adaplet mapping file:

```
item casesensitive false
```

Without this line, adaplets will continue to behave in a case-sensitive fashion. Note that adding this line to an adaplet mapping file will cause Live Collaboration to generate SQL against the foreign database that wraps all arguments inside of an 'upper()' expression. This alone does not assure the foreign database was designed and indexed in a way that makes case-insensitive operations viable.

The foreign database may need its own configuration to run in a case insensitive manner.

If the adaplet has case-sensitivity turned on and is in extend or migrate mode, then types must use mapped ids. Refer to the *Adaplet Programming Guide* for more information.

Collaboration and Approvals with Casesensitive Off

In general when user (or program) input that corresponds to a string in the database, a case insensitive search is performed (with case sensitivity turned off). If a match is found, the user input is replaced with the database string and the processing continues.

For example, selectables and the references within square brackets of select clauses are not case sensitive when in case insensitive mode. However, regardless of what you put inside the square brackets, in most cases the system returns the name as it is stored in the database. For instance, with a type called “test” that has an attribute “testattr”, the following command returns the output shown below.

```
print type test select attribute[TESTATTR};
  attribute[testattr]=TRUE;
```

Also, when case sensitivity is turned off, the query operators “match” and “matchcase” become indistinct, as are “!match” and “!matchcase.”

Some exceptions apply and are discussed below. Before turning case-sensitivity on, custom code should be checked to see if any routines check user input against database strings. These routines may need to be reworked to achieve the expected results.

Square Brackets

As stated above, in most cases the system returns the name of a selectable as it is stored in the database, regardless of what you put inside the square brackets. Actually, this applies only to references to administration objects that can be searched for in “Business” or “System”; when in square brackets these will return the database name of the object. However, things like file names, state names and signature names are returned as entered in the square brackets. For example, consider the following:

An object has an attribute Att1 and a checked in file name File.txt:

```
MQL<6>temp query bus * a2 * select format.file[file.txt]
format.file[file.txt].name attribute[att1].value;
businessobject A a2 0
    format.file[file.txt] = TRUE
```

(note that the spelling of the filename in the brackets is not corrected.)

```
format.file[file.txt].name = File.txt
```

(but, the name of the file is output as recorded in the database)

```
attribute[Att1].value = 2
```

(and the attribute name (an admin type) is corrected.)

One exception to this rule is properties on administration objects. Properties on administration objects always come back with the database name of the property.

Attribute Range Values

Before turning off case sensitivity, you should check all attributes for ranges that may disobey the rules of case insensitivity, such as having 1 range values for initial capitalization and another all lower case (for example “Pica” and “pica” for Units attribute). In this example, one of these ranges should be deleted. In a case-insensitive environment, the equals and match operators are identical; that is “Ea” is the same as “EA” and so entering the attribute value as “Ea” will succeed even if the range value is set to “EA”.

Revision Fields

When creating objects using a policy that defines an alphabetic revision sequence, the revision field is case sensitive, regardless of the system setting (that is, Collaboration and Approvals follows the revision rule exactly). Be sure the rules are defined appropriately.

However, queries on the revision field do follow the rules for the system `casesensitive` setting. So if your query specifies “A” for the revision field with case sensitivity turned off, objects that meet the other criteria are found that have both “A” and “a” as their revision identifier.

File names

File names remain case sensitive, regardless of the case sensitivity setting. So, for example, if the object has a file “FILE.txt” checked in, the following command will fail:

```
checkout bus testtype testbus1 0 file file.txt;
```

And the following will succeed:

```
checkout bus testtype testbus1 0 file FILE.txt;
```

For checkins, if you check in a file called “FILE.txt” and then check in another file called “file.txt”, there will be two separate files checked in (using append).

User Passwords

You cannot have distinct objects with the same spelling but different capitalization (that is, “Bill” and “bill” are interpreted as the same thing). However, user passwords remain case sensitive. For example, a user named “Bill” with a password of “secret”, can set context with:

```
mql<> set context user Bill pass secret;
```

or:

```
mql<> set context user bill pass secret;
```

but not with:

```
mql<> set context user bill pass Secret;
```

Adaplet usage

When an object name is explicitly specified in the name field of a query, objects returned via an adaplet are shown with the name in the case as entered in the query, and not necessarily as it exists in the database. The same is true when explicitly specifying object names to be exported — adaplet objects are referenced in the export file with the name in the case as entered.

Updating Sets With Change Vault

When an object's vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data
- The original business object is deleted from the "old" vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object's vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless. To address this issue, the following functionality can optionally be added to the change vault command:

- Add the object clone from the new vault to all IconMail messages and user sets in which the original object was included.

Since this additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails, it is not part of the function by default, but business administrators can execute the following MQL command to enable/disable this functionality for system-wide use:

```
set system changevault update set | on  | off |;
```

For example, to turn the command on for all users, use:

```
set system changevault update set on;
```

Once this command has been run, when users change an object's vault via any application (that is, Studio Modeling Platform or Web applications, or custom Studio Customization Toolkit applications), all IconMails that reference the object are fixed, as well as user's sets.

Database Constraints

Some versions of Oracle have a bug that limits performance and concurrency when an operation uses a column that has a foreign key constraint and that column is also not indexed. Such columns can cause deadlocks and performance problems. For systems that use foreign keys extensively, this leads to a trade off that must be made between performance, storage, and use of foreign keys. You can use the set system constraint command to tailor the schema for one of three possible modes of operation::

```
set system constraint |none                      |;
                      |index [indexspace SPACE] |
                      |normal                    |
```

- Normal

  The normal system setting results in a schema that has foreign keys that are not indexed; that is, foreign key constraints are configured as they have been in pre-version 10 releases. This is the default setting. Databases using this setting are subject to the Oracle concurrency bug regarding non-indexed foreign keys.

- Index

  The index setting results in an Oracle index being added to every column that is defined as a foreign key, and has no existing index. Overhead in both storage and performance is added since updates to foreign keys also require updates to their corresponding index. This option

should be used in development and test environments, where there is substantial benefit in the enforcement of foreign key constraints, and the negative storage/performance impact will not affect large numbers of users.

When issuing the command to add indices to the system, you can specify the tablespace to use for the indexing operation.

- None

  This option improves concurrency by removing non-indexed foreign key constraints, and no additional Oracle index is required. This option eliminates the concurrency problem of foreign keys, and in fact, further improves system performance and scalability by eliminating the low-level integrity checks performed at the database level. This option should be used once an application has been thoroughly tested and is rolled out to a large-scale production environment.

When the system is set with a constraint mode, not only are the changes made to the relevant columns, but also the setting is stored in the database so that all future operations including creation of new tables, running the index vault command, and upgrade will use the selected mode.

Note that these options only affect approximately a dozen columns (among all tables) that have a foreign constraint but not an index.

Time Zones from the Database

You can use the GMT time setting in the database for your servers by running the following command:

```
set system dbservertimezonefromdb on;
```

When set, the system gets the time in GMT from the db server. GMT can also be determined based on the local time and time zone of the server object defined by the System administrator.

MX_DECIMAL_SYMBOL and System Decimal Symbol

Collaboration and Approvals has two distinct controls for the handling of period '.' or comma ',' as the decimal symbol for real numbers:

- The "set system decimal" setting determines how real values must be formatted to pass them to an Oracle database.
- The MX_DECIMAL_SYMBOL environment setting controls the format in which real numbers are returned as values in print statements, queries, expands, and selectables. This affects the strings returned by both MQL commands and Java ADK calls. The final display format can be further adjusted by application preferences and/or browser settings.
  This setting has no effect on the form of real numbers passed to database APIs, nor does it have to be the same as the "set system decimal" character.

The decimal separator mechanism works as follows:

- When accepting a real number as input to the database, the kernel replaces the decimal separator in real-number inputs with the "set system decimal" character before passing them along to the database server.
- When reading a real number from the database and returning it to the calling application, the kernel substitutes the decimal separator in real numbers with the MX_DECIMAL_SYMBOL character.

Because the "set system decimal" character controls the form in which you interact with the database server, this setting *must* be synchronized with the Oracle setting for NLS_LANG so that real numbers are passed in a form that is expected by the database interfaces.

Use the following MQL command to set the decimal character that is expected by the database server (according to the NLS_LANG setting):

```
set system decimal CHARACTER;
```

where `CHARACTER` can be . or , [period or comma].

For example, to set the system decimal to a comma, use:

```
set system decimal ,;
```

When setting the system decimal character, be sure to use the setting that is implied by the database's NLS_LANG setting. The default setting is period '.'. So, if the database setting for NLS_LANG is `AMERICAN_AMERICA.WE8ISO8859P15`, Oracle expects a period for the decimal symbol (as indicated by the territory setting of AMERICA), and Live Collaboration makes the necessary conversion, once the following command is run:

```
set system decimal .;
```

When exporting business objects, MX_DECIMAL_SYMBOL influences the format of real numbers written to the export file. Therefore, the user who imports the file should set MX_DECIMAL_SYMBOL in the same way, or errors will occur.

The command `set system decimal CHARACTER;` *is not supported for use with databases other than Oracle. For non-Oracle databases, the data is always stored with '.' but displayed based on the desktop client's MX_DECIMAL_SYMBOL setting, or in Web based implementations the locale of the browser.*

For more information on configuring Live Collaboration for multiple language support, see the following topics in the *Administration Guide*:

- *Localizing Live Collaboration*
- *Language Support*
- *Configuring a Japanese Web Environment*

Allowing Empty Strings for Object Names

The following system-wide setting is available so that System Administrators can enable/disable the creation of objects with empty name fields in MQL for all user sessions permanently:

```
set system emptyname [on|off]
```

By default, the setting is off, which means that empty names are not allowed. MQL and any other program code will issue an error if any attempt to make a business object have an empty name is made. It will also cause a warning to be issued if a query specifies an empty string (“ ”) for the name. (The query will not error out and will not abort any transaction going on.) If the setting is on, the system allows empty names for objects in MQL only.

Setting History Logging for the System

The following system-wide setting is available to enable/disable history for all user sessions permanently:

```
set system history [on|off]
```

By default, history is on. When turned off, custom history records can be used on the operations where history is required, since it will be logged regardless of the global history setting.

Adaplet Persistent IDs

Adaplet object IDs are saved in the database by default so that they are persistent between sessions. This capability is no longer dependent on the usage mode (readonly, readwrite, migrate, extend) of the adaplet. To disable the feature, persistence may be turned off with the following command:

```
set system persistentforeignids off;;
```

Refer to the *Adaplet Programming Guide* for details.

Privileged Business Administrators

By default, a business administrator can change context to any person without a password. This is to allow administrators and programs to perform operations on behalf of another user. A System Administrator can disable this system-wide “back door” security risk by issuing the following command:

```
set system privilegedbusinessadmin off;
```

After this command has been run, business administrators need a password when changing context to another person. Only system administrators can change context to any other person without a password.

A System Administrator can re-enable business administrators to change context without a password using:

```
set system privilegedbusinessadmin on;
```

System Tidy

In replicated environments, when a user deletes a file checked into an object (via checkin overwrite or file delete), by default all other locations within that store maintain their copies of the now obsolete file. The file is deleted only when the administrator runs the `tidy store` command.

You can change the system behavior going forward so that all future file deletions occur at all locations by using:

```
set system tidy on;
```

Since this command changes future behavior and does not cleanup existing stores, you should then sync all stores, so all files are updated. Once done, the system will remain updated and tidy, until and unless system tidy is turned off.

Running with system tidy turned on may impact the performance of the delete file operation, depending on the number of locations and network performance at the time of the file deletion.

Validating the Live Collaboration Database

The MQL `validate` command enables you to check the correctness and integrity of the Live Collaboration database. See the *Server Installation Guide : Upgrading the Database after Installing a New Version of Software* for complete syntax and clauses for working with the MQL validate command.

Correct command

The MQL correct command can be used by System Administrators as directed by Dassault Systèmes Technical Support, to correct anomalies caused by older versions of the product. You must be certain that no users are logged in or can log in while correct is running.

Confirm that no users are connected, using the sessions command or Oracle tools, and temporarily change the bootstrap so that no one can login while correct is running.

There are several forms of the command. Each form is described in the sections that follow.

Correct Vault

The correct vault command can be used to validate or correct the following relationship issues in a database:

1. "Stale" relationships - Relationships that reference a business object that does not exist.

   Since there is no way to determine which business object should be referenced, the relationship is deleted when the correct command is issued with "fix".

   A sample of the output generated when fixing this issue is shown below:

```
// Missing to/from businessobjects
From vault Zephyr to vault Zephyr
From vault Zephyr to vault KC
remove stale relationship
  type: BrokenRel
  from: 8286.42939.11584.38392
  to: BrokenRel child1 0
remove stale relationship
  type: BrokenRel
  from: 8286.42939.11584.38392
  to: BrokenRel child1 0
From vault KC to vault Zephyr
From vault KC to vault KC
Correct finished in 0.12 second(s)
Total of 2 problem(s) encountered and fixed
```

   If in validate mode, the last line would say:

```
Total of 2 problem(s) encountered and validated
```

2. extra "to" end - A cross vault relationship has a valid "to" side object, but there is no corresponding "from" object.

   The relationship is deleted when the correct command is issued with "fix" as there is insufficient information to reconstruct the relationship - its attributes and history are missing.

A sample of the output generated when fixing this issue is shown below:

```
// Missing entry in the lxRO table of the "from" end of the rel
From vault Zephyr to vault Zephyr
From vault Zephyr to vault KC
From vault KC to vault Zephyr
remove extra relationship 'to' end
     type: BrokenRel
     from: BrokenRel child1 0
     to:   BrokenRel TO 0
From vault KC to vault KC
Correct finished in 0.12 second(s)
Total of 1 problem(s) encountered and fixed
```

3. missing “to” end - A cross vault relationship has a valid “from” side object, but there is no corresponding “to” object.

   Missing “to” end’s are recreated when the database is corrected. This is possible because only the relationship pointers need to be re-established. The attribute and history for the relationship is stored on the “from” side.

   A sample of the output generated for this case is shown below:

```
// Case 3
// Missing entry in the lxRO table of the 'to' end of the rel
From vault Zephyr to vault Zephyr
From vault Zephyr to vault KC
add missing relationship 'to' end
   type: BrokenRel
   from: BrokenRel FROM 0
    to:   BrokenRel child1 0
From vault KC to vault Zephyr
From vault KC to vault KC
Correct finished in 0.11 second(s)
Total of 1 problem(s) encountered and fixed
```

Cases (1) and (2), by their nature, may result in data being removed from the database. Case (3) is an additive step; such cases do not result in the loss of information.

It should be noted that you can validate and correct adaplet vaults used in extend or migrate mode; read or readwrite adaplet vaults are not supported.

Syntax

The syntax of the correct vault command is:

```
correct vault VAULTNAME [to VAULTNAME] [type REL1{,REL2}] [verbose]
[fix] [validate];
```

Unless “fix” is specified, the command will run in “validate” mode.

If you include both “fix” and “validate,” an error will occur asking you to specify just one of the options. If you use “correct validate” without any arguments, all relationships’ from objects are checked and a report is generated on stdout that lists all problems found.

Each argument is described below.

vault VAULTNAME [to VAULT]

Limits the scope of the function to a single vault. All relationships that include any object in VAULTNAME are checked. You can also specify “all” for correct to go through all the vaults in the database.

When “to VAULT” is included, correct looks only at connections between VAULTNAME and VAULT.

type REL1{,REL2}

Inspects only relationship types listed. If not specified, all the relationship types in the system will inspected for problems.

verbose

Causes the program to output the SQL needed for repair to stdout. Set to false by default.

fix

Fixes the database. Set to false by default.

validate

Validates the database. Reports broken relationships but no corrections are made. Set to true by default.

Transactions

The correct vault command operates in two passes. The first pass is a quick scan through the database without performing any row level locks. The second pass gathers detailed information on relationships reported as “suspect” during the first pass.

During concurrent use of the system, it is entirely possible that the validate function on a vault may report errors that do not actually exist in the database since it is operating without the benefit of locks (the data many be changing while it is being examined). However, during the second pass, any suspicious data is fully re-qualified using row level locks, making the function safe to use while the system is live.

Note that with any general purpose application that uses locks, a deadlock situation may occur if other applications are locking the same records in an inconsistent fashion. Deadlocks are highly unlikely while using correct vault since the data that is being locked (e.g. a 'stale' relationship) is generally unusable by other applications. However, if deadlock should occur, simply reissue the correct vault command and allow it to run to completion.

This two pass transaction model is only used by the correct vault command. While "correct vault" is designed to run while in concurrent use, it is recommended that for all system administrator operations like this one, it be issued when there are no other users logged on.

Whenever any form of the correct command is executed in fix mode, you receive the following warning and prompt for confirmation:

```
Correct commands may perform very low level modifications to your
database.
Use the 'sessions' command to check that no users are logged in.
It is strongly recommended that you change bootstrap password to insure
that no users log in during correct process, and turn verbose on to
record corrective actions.
```

```
You should also consult with MatrixOne support to insure you follow
appropriate procedures for using this command.

Proceed with correct (Y/N)?y
```

For all correct commands, output can be redirected to a file using standard output.

Correct Attribute

To verify the integrity of attributes and types, each type in the database can first be checked to ensure that no attribute data or attribute range values are missing. The following command will return a list of objects of the specified TYPE that have missing data:

```
MQL<> correct attribute validate type TYPE;
```

where TYPE is a defined type in Live Collaboration. If corrections must be made, MQL will provide the business object type, name, and revision, as well as what attribute data is missing, as follows:

```
Business object TYPE NAME REVISION is missing attribute ATTRIBUTE_NAME
```

Be sure to run this command for each type in the database.

If any business objects are returned by the `correct attribute validate` command, those business objects must be fixed with the following command:

```
MQL< > correct attribute modify type TYPE_NAME attribute
ATTRIBUTE_NAME vault VAULT_NAME;
```

Where TYPE_NAME is the name of a business type to be corrected. When this command is completed all business objects that have missing attributes will have those attributes back. The value of the attribute will be the default value.

Where ATTRIBUTE_NAME is the name of the attribute that is missing from some objects.

Where VAULT_NAME is name of the vault in which business objects of type TYPE_NAME will be examined and corrected if needed.

Run this command for each vault in the database. Only those business objects that were missing the attribute will be modified. The added attribute on these objects will be initialized to its default value, and MQL will output confirmation messages similar to:

```
count of BOs of type 'TYPE'  in vault 'VAULT_NAME' = 1
   processing attribute = 'ATTRIBUTE_NAME'
        business obj '51585.1271.52275.16291' having db id '-869056605' with type
'TYPE_NAME'  is missing attr 'ATTRIBUTE_NAME'
            Total objects processed = 1
```

Correct Relationship

Correct relationship is used to correct relationship attribute data:

```
mql<> correct relationship;
```

Attributes on relationships that connect objects in different vaults are stored in the from-side vault. This command removes attributes if they exist in the to-side vault.

Correct Set

Sets can be validated either across the entire database, or on a server by server basis using the following:

```
MQL<> correct set validate [server SERVER_NAME];
```

If invalid sets are found you can delete them with:

```
MQL<> correct set fix [server SERVER_NAME];
```

Correct State

To verify the integrity of all states in the database or on a single vault use:

```
MQL<> correct state validate [vault VAULT_NAME];
```

Without the vault clause all states in the database are checked. If you include the vault clause, only that vault is validated. You can then fix the states using:

```
MQL<> correct state [vault VAULT_NAME];
```

Developing a Backup Strategy

Because there are a number of factors that can cause a database failure, including power loss, hardware failure, natural disasters, and human error, it is important that you develop both a backup and a recovery plan to protect your database. It is not enough that you *develop* a recovery plan, however. You must also test that recovery plan to ensure that it is adequate before your data is compromised. Finding out that your recovery plan is inadequate after you have already lost your data will not do you much good. Also, testing of the recovery plan may indicate changes you need to make to your backup strategy.

Inventories of all stores should be performed nightly as part of the backup procedure. Refer to Inventory Store for details.

Working with Indices

Live Collaboration stores each attribute and "basic" property of an object in a separate row in the database, allowing flexible and dynamic business modeling capabilities. (Basic properties include type, name, revision, description, owner, locker, policy, creation date, modification date, and current state). This has the side effect of requiring extra SQL calls (joins) when performing complex queries and expands that specify multiple attribute or basic values. To improve performance of these operations, you can define an *index*. In fact, test cases show marked improvement on many types of queries and expands when an index is in place.

Queries and expands will choose the "best" index to perform the operation. If 2 indices are equally qualified to perform the operation, the first one found will be used.

An *index* is a collection of attributes and/or basics, that when enabled, causes a new table to be created for each vault in which the items in the index are represented by the columns. If a commonly executed query uses multiple attributes and/or basics as search criteria, you should consider defining an index. Once enabled, searches involving the indexed items generate SQL that references the index table instead of the old tables, eliminating the need for a join.

A query that doesn't use the first specified item in an index will not use that index. Therefore, when creating an index, specify the most commonly used item first.

When an index is created or items are added or removed from it, the index is by default disabled. The new database tables are created and populated when the index is enabled. This step can be time-consuming; however, it is assumed that an index will be enabled by a system administrator once when created or modified, and will remain enabled for normal system operation.

For expands, Live Collaboration looks for and uses an index that is associated with relationships; that is, an index that has a relationship attribute as the first item in it.

Creating an index is only one way of optimizing performance. A properly tuned database is critical. While the *Installation* and *MQL Guides* provide some guidelines and procedures, refer to your database documentation for tuning details.

Write operations of indexed items occur in 2 tables instead of just 1 and so a performance penalty is paid. This has a negligible effect in cases where end users may be modifying a few values at a time, but you should disable indices prior to performing bulk load/update operations. Re-enabling the indices after the bulk update is more efficient than performing the double-bookkeeping on a value-by-value basis during the bulk update.

Considerations

Before creating indices consider how the indexed items are used in your implementation. Some pros and cons are shown below

PROS	CONS
Excellent for reads.	Write performance is impacted (data is duplicated).
Query execution runtimes are significantly reduced	More disc space is required (for duplicate data).
Table joins are significantly reduced.	Greater potential for deadlock during write operations. (more data in one row as opposed to separate rows within separate tables).
	Item order in the definition of the index is important, since if the operation doesn't use the first item, the index is skipped.

Also, keep in mind that attributes that have associated rules cannot be included in an index.

Defining an Index

An index is created with the add index command:

```
add index NAME [unique] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the index.

`ADD_ITEM` is an add index clause which provides additional information about the index.

The complete syntax and clauses for working with indices are discussed in *index Command* in the programming reference section of this guide.

Enabling an Index

An index must be enabled before it is used to improve performance. The process of enabling an index generally takes 0.01 second per object in the database. Some rough processing times that can be expected are shown in the table below:

Number of objects in DB	Time to enable an Index
500,000	1.5 hours
5,000,000	14 hours

To optimize the performance, validating the tables is recommended. You should disable an index before performing bulk loads or bulk updates.

Creating and enabling indices are not appropriate actions to be performed within explicit transaction boundaries, particularly if additional operations are also attempted before a commit.

Validating an Index

Since up-to-date database statistics are vital for optimal query performance, after enabling an index you should generate and add statistics to the new database tables. This is assuming statistics are already up-to-date for all other tables. Refer to *Validating the Live Collaboration Database* for more information.

Using the index as Select Output

You can use the index[] selectable on business objects and relationships to retrieve the attribute and basic values directly from the index table. This is roughly N times faster than using attribute[] where N is the number of attributes in the index. For example, if you routinely run a query/select like the following:

```
temp query bus Part * * select attribute\[Part_Number\]
attribute\[Quantity\] attribute\[AsRequired\]
attribute\[Process_Code\] attribute\[Function_Code\] current dump |
output d:/partlist.txt;
```

You could create an index called Parts containing all of the selectables listed and use the following query instead:

```
temp query bus Part * * select index\[Parts\] dump | output d:/
partlist.txt;
```

The temp query that uses the select index is roughly 6 times faster than using the query that selects all 6 items separately.

Select index is similar to selecting items with a few differences:

- Select index returns values for all items in the index, where select item returns just one. If an attribute does not apply to a type or relationship, the returns is null.
- String attributes longer than 251 characters will be truncated to 251 characters. Generally these are defined as multiline attributes, which cannot be included in an index anyway.
- Descriptions longer than 2040 characters will be truncated to 2040 characters.

Index Tracing

Tracing can be enabled to detect queries/expands that can benefit by using a new or modified index. When enabled, messages are displayed whenever a query or expand is executed that fails to find an associated index. A message is output for each item in the query or expand that *could* be in an index (an attribute or basic property). The intent of this tracing is to highlight queries and expands that use indexable items, none of which are the first item in any index (so no index is used to optimize the operation).

To enable the index trace messages, use the MQL command:

```
trace type INDEX on;
```

or set the environment variable:

```
MX_INDEX_TRACE=true
```

When this trace type is enabled, trace messages will be displayed in the format:

```
No index found for attribute[NAME1],attribute[NAME2],BASIC3…
```

The items in the trace message may or may not be in a defined index. However, even if they are part of an enabled index, they are not the first item in it, and so the index is not usable for that query or expand.

A developer can conclude that adding at least some of the listed fields to a new or existing index (making one of the items first) may improve system performance.

10 Working with Import and Export

This chapter discusses concepts about exporting and importing administrative and business objects.

In this section:

Overview of Export and Import

Administrative definitions, business object metadata and workflows, as well as checked-in files, can be exported from one root database and imported into another. Exporting and importing can be used across schemas of the same version level, allowing definitions and structures to be created and “fine tuned” on a test system before integrating them into a production database.

When you export a business object, you can save the data to a *exchange* file or to an XML format file that follows the Matrix.dtd specification. An exchange file is created according to the exchange Format. This format or the XML format must be adhered to strictly in order to be able to import the file.

The Exchange Format is subject to change with each version.

Exporting

Before exporting any objects, it is important to verify that MQL is in native language mode. Exporting in the context of a non-native language is not supported.

Export Command

Use the Export command to export the definitions of a database. The complete export command for administrative objects is as follows:

```
export ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]| into | file FILE_NAME
                                                             | onto |
[exclude EXCLUDE_FILE] [log LOG_FILE] [exception EX_FILE];
```

The complete syntax and clauses for working with export are discussed in *export Command* in the programming reference section of this guide.

XML Export Requirements and Options

Schema and business objects may optionally be exported in XML format. The resulting XML stream follows rules defined in the XML Document Type Definition (DTD) file (named “ematrixml.dtd”) that is installed with Live Collaboration in the /XML directory. This DTD file is referenced by all exported Live Collaboration XML files. In order to interpret and validate Live Collaboration XML export files, an XML parser must be able to access the eMatriXML DTD file. This means that the DTD file should reside in the same directory as the exported XML files and be transferred along with those files to any other user or application that needs to read them.

Using MQL, you can export objects to XML format in either of two ways:

- Insert an XML clause in any Export command for exporting administrative or business objects
- Issue an XML command to turn on XML mode as a global session setting. In this mode, any Export commands you enter, with or without the XML clause, will automatically export data in XML format.

Be aware of the following restrictions related to exporting and importing in XML format:

- When exporting programs to XML, make sure the code does not contain the characters “]]>”. These characters can be included as long as there is a space between the bracket and the greater sign.
- Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as those created with the ESC key, should not be used for ANY field, including business and administrative object names, description fields, program object code, or page object content.
- Due to an xerces system limitation, the form feed character “^L” is not supported by the import command. If they are included in the file you are trying to import, you will receive an error similar to:

  ```
  System Error: #1500127: 'file' does not exist
  businessobject Document SLB EH713116 AE failed to be imported.
  System Error: #1600067: XML fatal error at (file '/rmi_logs/GP2_tmp/
  T5011500_4x.xml', line 6292824, char 2): Invalid character (Unicode:
  0xC)
  import failed.
  ```

The workaround is to go to the line number in the XML file and remove the offending character.

The CDATA termination character string]]> is replaced by]Inserted_by_ENOVIA]Inserted_by_ENOVIA> in XML export files. XML Import files will do back replacement. Outside software reading, XML files should handle this replaced string.

XML Clause

Use the XML clause as an OPTION_ITEM in any Export command to export administrative or business objects in XML format. To ensure that the XML file(s) reside in the same directory as the XML DTD, you can export files to the XML folder (in your ENOVIA_INSTALL directory) where the DTD is installed. Alternatively, you can specify another location in your Export command and copy the DTD into that directory. Use the `into` clause in your Export command and specify a full directory path, including file name. For example:

```
export person guy xml into file
c:\enoviaV6R2011\studio\xml\person.xml;
```

XML Command

Use the XML command to turn XML mode on or off as a global session setting. The complete XML command syntax is:

```
xml [on|off];
```

Omitting the on/off switch causes XML mode to be toggled on or off, depending on its current state. XML mode is off by default when you start an MQL session.

For example, to export a person named “guy” using an XML command along with an Export command, you can enter:

```
xml on;
export person guy into file
c:\enoviaV6R2011\studio\xml\person.xml;
```

If you need to export several objects to XML format, using the XML command to turn on XML mode first can eliminate the need to re-enter the XML clause in each Export command as well as the possibility of an error if you forget.

XML Output

The XML export format typically generates a file 2 to 3 times larger than the standard export format. To conserve space, subelements are indented only if verbose mode is turned on. Indentation makes output more readable but is not required by an XML parser. Some XML viewers (like Internet Explorer 5.0) will generate the indentation as the file is parsed.

The following example shows standard export output when you export a person named “guy” during an MQL session. While relatively compact, this output is not very intelligible to a user.

```
MQL<1>set context user creator;
MQL<2>export person guy;
!MTRX!AD! person guy 8.0.2.0
guy Guy ""
 "" "" "" "" 0 0
 1 1 1 0 1 0 1 "" "" *
 1111101111111100111110
 0 1 .finder * GuyzViewTest 0 * FileInDefaultFormat 1 ""
 0 0
 0 0
 0
 0
 0
 de3IJEE/JIJJ.
 ""
 0
 0
 0
 1
 Test""
1
Description  description 1
0 0 0 0 70 21 0 0 1 1
 1
 0 0
 0
 0 0
person guy successfully exported.
!MTRX!END
export successfully completed
```

The next example shows XML output when you use the Export command, with XML mode turned on, to export a person named “guy” during a continuation of the same MQL session. While more intelligible to a user, this code creates a larger output file than the standard export format.

```
MQL<3>xml on;
MQL<4>verbose on;
MQL<5>export person guy;
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- (c)MatrixOne, Inc., 2000 -->
<!DOCTYPE ematrix SYSTEM "ematrixml.dtd">
<ematrix>
  <creationProperties>
    <release>8.0.2.0</release>
    <datetime>2000-05-05T17:57:19Z</datetime>
    <event>export</event>
    <dtdInfo>&ematrixProductDtd;</dtdInfo>
  </creationProperties>
  <person id="0.1.31721.46453">
    <adminProperties>
      <name>guy</name>
    </adminProperties>
    <fullName>Guy</fullName>
    <fullUser/>
    <businessAdministrator/>
    <systemAdministrator/>
```

```
<applicationsOnly/>
<passwordChangeRequired/>
<access>
  <all/>
</access>
<adminAccess>
  <attributeDefAccess/>
  <typeAccess/>
  <relationshipDefAccess/>
  <formatAccess/>
  <personAccess/>
  <roleAccess/>
  <associationAccess/>
  <policyAccess/>
  <programAccess/>
  <wizardAccess/>
  <formAccess/>
  <ruleAccess/>
  <siteAccess/>
  <storeAccess/>
  <vaultAccess/>
  <serverAccess/>
  <locationAccess/>
</adminAccess>
<queryList count="1">
  <query>
    <name>.finder</name>
    <queryStatement>
      <vaultPattern>*</vaultPattern>
      <typePattern>GuyzViewTest</typePattern>
      <ownerPattern>*</ownerPattern>
      <namePattern>FileInDefaultFormat</namePattern>
      <revisionPattern>1</revisionPattern>
    </queryStatement>
  </query>
</queryList>
<password>de3IJEE/JIJJ.</password>
<tableList count="1">
  <table>
    <name>Test</name>
    <columnList count="1">
      <column>
        <label>Description</label>
        <expression>description</expression>
        <usesBusinessObject/>
        <geometry>
          <xLocation>0.0</xLocation>
          <yLocation>0.0</yLocation>
          <width>70.0</width>
          <height>21.0</height>
          <minWidth>0.0</minWidth>
          <minHeight>0.0</minHeight>
          <autoWidth/>
          <autoHeight/>
        </geometry>
        <editable/>
      </column>
    </columnList>
  </table>
```

```
    </tableList>
  </person>
</ematrix>
```

Importing

When migrating objects or entire databases, it is important to import in the correct order:

1. Import all administrative objects first.
2. Import all business objects.
3. Import workspaces from the same exported ASCII data file as was used to import Persons. Refer to *Importing Workspaces* for more information.
4. Import properties of the administrative objects that have them. The same file that was used to import the administrative objects should be used. Refer to *Importing Properties* for more information.

For more migration strategies refer to *Migrating Databases*.

Import Command

Use the Import command to import administrative objects from an export file to a database. The export file must be in Exchange Format or in an XML format that follows the Matrix.dtd specification.

```
import [list] [property] ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]...];
```

The complete syntax and clauses for working with import are discussed in *import Command* in the programming reference section of this guide.

Importing Servers

If a server object is imported (via MQL or Oracle) into a different Oracle instance or user than it was exported from, the username and password settings of the server must be modified before distributed access is available. This is particularly important if an upgrade will follow the import, since all servers must be accessible to the machine doing the upgrade.

Importing Workspaces

Workspaces contain a user's queries, sets, and iconmail, as well as all Visuals. Workspaces are always exported with the person they are associated with. However, when Persons are imported, the workspace objects are not included. This is because they may rely on the existence of other objects, such as Types and business objects, which may not yet exist in the new database. Workspaces must be imported from the same .mix file that was used to import persons. For example:

```
import workspace * from file admin.mix;
```

Or:

```
import workspace julie from file person.mix;
```

Importing Properties

Properties are sometimes created to link administrative objects to one another. Like workspaces, properties are always exported with the administrative object they are on. Import is somewhat

different, however, since a property may have a reference to another administrative object, and there is no way to ensure that the referenced object exists in the new database (administrative objects are sometimes exported and then imported in pieces). So a command to import administrative objects is issued, the specified objects are created first, and then the system attempts to import its properties. If the "continue" modifier is used, the system will get all the data it can, including system and user properties. But to ensure that all properties are imported, even when the administrative data may have been contained in several files, use the `import property` command.

For example, data can be exported from one database as follows:

```
export attrib * into file attrib.exp;
export program * into file program.exp;
export type * into file type.exp;
export person * into file person.exp;
```

Then the administrative objects are imported:

```
import attrib * from file attrib.exp;
import program * from file program.exp;
import type * from file type.exp;
import person * from file person.exp;
```

Finally, workspaces and any "missed" properties are imported. Note that properties may exist on workspace objects, so it is best to import properties after workspaces:

```
import workspace * from file person.exp;
import property attrib * from file attrib.exp;
import property program * from file program.exp;
import property type * from file type.exp;
import property person * from file person.exp;
```

Importing Index objects

When importing an Index that includes attributes, these attributes must already exist in the new database — that is, you should import the attributes first. For example:

```
import attribute A from file /temp/export.exp;
import attribute B from file /temp/export.exp;
import index * from file /temp/export.exp;
```

If the export file contains a lot of other admin data and you want to "import admin *", you can use import options to avoid errors when the attributes are processed. For example, if you know you want the objects in the export file to supersede any objects of the same type/name in the database use the following:

```
import admin * overwrite from file /temp/export.exp;
```

If you want objects already in the database to remain unchanged, use:

```
import admin * commit 1 continue from file /temp/export.exp;
```

Extracting from Export Files

Sometimes export files contain more information than you want imported. When this is the case, the extract command can be used to create a new file containing only the specified information of the original file.

```
extract |bus OBJECTID     | [OPTION_ITEM [OPTION_ITEM]]from file FILENAME |into| file NEW;
        |ADMIN ADMIN_NAME|                                                 |onto|
```

Migrating Databases

When migrating entire databases or a large number of objects, it is important to import in the following order:

1. Import all administrative objects first. For example:

```
import admin * from file admin.mix;
```

2. Import all business objects. For example:

```
import bus * * * from file bus.mix;
```

3. Import workspaces from the same exported ASCII data file as was used to import Persons. For example:

```
import workspace * from file admin.mix;
```

Or:

```
import workspace julie from file person.mix;
```

Workspaces contain a user's queries, sets, TaskMail, and IconMail, as well as all Visuals. Workspaces are always exported with the person with which they are associated. However, when Persons are imported, the workspace objects are not included. This is because they may rely on the existence of other objects, such as Types and business objects, which may not yet exist in the new database.

4. Import properties of administrative objects.

When administrative objects are imported, Live Collaboration imports all the objects without their properties and then goes back and imports both system and user properties for those objects. If administrative objects were exported in pieces, such as all Types in one file, all Persons in another, then properties should be explicitly imported with the `import property` command. Refer to *Importing Properties* for more information.

Migrating Files

When migrating business objects, there are three options for its files:

1. By default, both file metadata and actual file contents are written to the export file for business objects exported with their revision chain.
Live Collaboration UUencodes the file and writes it to the export data file, along with business object metadata. Pointers to the files are guaranteed because the file is recreated during import. In this case, any file sharing is lost, (as when revisions use the same file list—each revision in the chain gets its own copy of the file).

2. Add the `!file` clause to not include any file metadata or content.

3. Use the `!captured` clause to not include captured store file content.
This option writes only the fully qualified path of checked in files in the data file, along with business object metadata.

When migrating databases, most often the `!captured` option is recommended. This will facilitate the process, in that the .mix files will not be as large, and therefore will not require as much disk space or processing time to complete the migration. Once the import is complete, the objects will point to the appropriate files in the same location.

The key to keeping the file pointers accurate is keeping the store path definitions consistent. For example, let's say the database from which we are exporting has a captured store named "Released Data Store." The path of this store is defined as "/company/released." To maintain pointer consistency when using `!captured`, the new database must also have a defined captured store "Released Data Store" with the same path definition.

If objects are to be deleted and then re-imported, use the `!captured` option, but be sure to tar off any captured store directories before deleting any objects. Once the objects are deleted, the directories restored, and the objects imported, the files will be associated with the appropriate objects.

Migrating Revision Chains

Files may be shared among revisions of a business object. This "file sharing" concept was introduced to minimize storage requirements and is primarily used with captured stores. However, this does mean special attention is required when exporting and importing. Consider the following:

If the entire revision chain is not going to be exported and/or imported into a new database, then the use of `!captured` may result in lost files. For example, if the business object Assembly 123 0 has three files checked in, and is then revised to Assembly 123 1, this new revision shares the three files with the original. As long as both are exported and imported, `!captured` can be used (in fact, should be used to avoid file duplicating). However, if just Assembly 123 1 is imported into a new database, then NO file is imported, because Assembly 123 1 inherits files from the previous revision, which is not in the database yet. If you import Assembly 123 0 later, Assembly 123 1 shows files.

Also, if you want to import business objects that have revisions and put them into a different vault, you *must* use a map file. If you attempt to use the `from vault` clause with the `to vault` clause, errors will occur.

Comparing Schema

This section describes how to compare two schemas to determine the differences between them. A *schema* is all the administrative objects needed to support an app. Use schema comparison to compare:

- Different versions of the same schema to manage changes to the schema.
- Two schemas from different databases so you can merge the schemas (for example, merge a checkout system with a production system).

The process of comparing schema involves two main steps:

1. Create a baseline sample of one schema using XML export. See *Creating a Baseline*.
2. Analyze the differences between the baseline sample and the other schema (or a later version of the same schema) using the compare command. You specify the administrative types (attributes, relationships, etc.) and a name pattern (for example, all attributes beginning with "Supplier") to compare. Each compare command outputs a single log file that contains a report. The report lists the administrative objects in the baseline export file that have been added, deleted, and modified. See *Comparing a Schema with the Baseline*.

If your goal is to merge the two schemas by making the necessary changes to one of the schemas (sometimes called "applying a delta"), you can make the changes manually or by writing an MQL script file that applies the changes to the target schema.

Scenario

Suppose you need to determine the changes that have occurred in a checkout database versus what continues to exist in the production database. In this case, you may want to create a baseline of both databases, and use each to compare against the other. One report would be useful to find out what has changed in the checkout database. The other report would be useful to determine what it would take to apply those changes to the production database.

Creating a Baseline

The first step for comparing two schemas is to establish a baseline for analysis by sampling one of the schemas. You create a baseline by exporting administrative objects to XML format. You can use any option available for the export command to create the baseline export files. For information on options and more details about the export command, see *Export Command*.

Use the following guidelines to perform the export.

- Start MQL using a bootstrap file that points to the database containing the schema for which you want to create the baseline.
- There are two ways to produce an XML export file: toggle on XML mode and issue a normal export command, or issue a normal export command but include the XML keyword. For example:

  ```
  xml;
  export ADMIN_TYPE TYPE_PATTERN into file FILENAME;
  ```

  Or the equivalent:

  ```
  export ADMIN_TYPE TYPE_PATTERN xml into file FILENAME;
  ```
- It's best to create separate a export file for each administrative type and to keep all the objects of a type in one file. For example, export all attributes to file attributes.xml, all relationships to relationship.xml, etc. This keeps the baseline files to a reasonable size, and also lets you compare specific administration types, which makes it easy to produce separate reports for each administration type. If you need to identify subsets of objects within an export file to focus the analysis, you can do so using the compare command.
- The compare command requires that the ematrixml.dtd file be in the same directory as the export files. Therefore, you should create the export files in the directory that contains the dtd file or copy the dtd file into the directory that contains the export files. If you don't specify a path for the export file, Live Collaboration creates the file in the directory that contains mql.exe file.

The following table shows examples of export commands that export different sets of administrative objects. All the examples assume the XML mode is not toggled on and that the ematrixml.dtd file is in the directory d:\Matrix\xml.

To export:	Use this command
all attributes	export attribute * xml into file d:\enoviaV6R2011\studio\xml\attributes.xml
all attributes that begin with the prefix "Supply"	export attribute Supply* xml into file d:\enoviaV6R2011\studio\xml\SupplyAttributes.xml
all administrative objects that begin with the prefix "mx" (usually better to keep all objects of a type in separate files)	export admin mx* xml into file d:\enoviaV6R2011\studio\xml\mxApp.xml

Comparing a Schema with the Baseline

After creating the baseline for one of the schemas, the second step is to analyze the differences between the baseline and the second schema, and generate a report that lists the differences. The MQL compare command performs this step. The syntax for the compare command is shown below. Each clause and option in the command is explained in the following sections.

```
compare ADMIN_TYPE TYPE_PATTERN [workspace] from file FILENAME [use [map FILENAME]
[exclude FILENAME] [log FILENAME] [exception FILENAME]];
```

When issuing the command, make sure you start MQL using a bootstrap file that points to the database that you want to compare with the schema for which you created the baseline.

The compare command analyzes only administrative objects and ignores any business object instances in the baseline export file.

Where unordered lists are used, their order resulting from a query is not guaranteed. This may potentially yield false differences during compare. Unexpected differences reported on unordered list items should be verified by examining the log or command output.

ADMIN_TYPE TYPE_PATTERN Clause

The `ADMIN_TYPE` clause specifies which administrative types to compare. Valid values are:

admin	group	person	server	vault
association	index	policy	site	wizard
attribute	inquiry	program	store	
command	location	relationship	table	
form	menu	role	type	
format	page	rule	user	

The value `admin` compares all administrative types of the name or pattern that follows. For example, the clause “admin New*” compares all administrative objects with the prefix “New.” All other ADMIN_TYPE values compare specific administrative types. For example, to compare all policies, you could use:

```
compare policy * from file
d:\enoviaV6R2011\studio\xml\policy.xml;
```

To compare objects of a particular type whose names match a character pattern, include the pattern after the `ADMIN_TYPE` clause. For example, to compare only relationships that have the prefix “Customer”, you could use:

```
compare relationship Customer* from file d:\enoviaV6R2011\studio\xml\relationship.xml;
```

workspace Option

The workspace option applies only when comparing administrative types that can have associated workspace objects: persons, groups, roles, or associations. When you add the keyword

"workspace" to the compare command, any workspace objects (tips, filters, cues, toolsets, sets, tables, and views) owned by the persons, groups, roles, or associations being compared are included in the comparison operation.

For example, suppose you compare the Software Engineer role and the only change for the role is that a filter has been added. If you don't use the workspace option, the compare operation will find no changes because workspace objects aren't included in the comparison. If you use the workspace option, the comparison operation will report that the role has changed and the filter has been added.

from file FILENAME Clause

The from file `FILENAME` specifies the path and name of the existing XML export file. This file contains the baseline objects you want to compare with objects from the current schema. The ematrixml.dtd file (or a copy of it) must be in the same directory as the baseline XML file.

use FILETYPE FILENAME option

Use the `use` *keyword once for any optional files specified in the compare command: map files, exclude files, log files, or exception files. If more than one file type is to be used, the* `use` *keyword should be stated once only.*

The `use FILETYPE FILENAME` option lets you specify optional files to be used in the compare operation. FILETYPE accepts four values:

- `log`

  The use log FILENAME option creates a file that contains the report for the compare operation. The compare command can be issued without identifying a log file, in which case just a summary of the analysis is returned in the MQL window—total number of objects analyzed, changed, added, deleted. The log file report lists exactly which objects were analyzed and describes the differences. More information appears in the report if verbose mode is turned on. For more information about the report, see *Reading the Report*.

  An efficient approach is to run the compare command without a log file to see if any changes have occurred. If changes have occurred, you could turn on verbose mode, re-run the compare command and supply a log file to capture the changes.

- `map`

  A map file is a text file that lists administrative objects that have been renamed since the baseline file was created. The map file maps names found in the given baseline file with those found in the database (where renaming has taken place). Use the map file option to prevent the compare operation from finding a lot of changes simply because administrative objects had their names changed. The map file must use the following format:

```
ADMIN_TYPE OLDNAME NEWNAME
ADMIN_TYPE OLDNAME NEWNAME
```

  `OLDNAME` is the name of the object in the baseline export file. `NEWNAME` is the name of the object in the current schema (the schema you are comparing against the baseline file). Include quotes around the names if the names include spaces. Make sure you press Enter after each NEWNAME so each renamed object is on a separate line (press Enter even if only one object is listed). For example, if the Originator attribute was renamed to Creator, the map file would contain this line:

```
attribute Originator Creator
```

If no map file is specified, the compare operation assumes that any renamed objects are completely new and that the original objects in the baseline were deleted.

- `exclude`

  An exclude file is a text file that lists administrative objects that should not be included in the comparison. The exclude file must use the following format:

```
ADMIN_TYPE NAME
ADMIN_TYPE NAME
```

  `NAME` is the name of the administrative object that should be excluded in the compare operation. Make sure you press Enter after each NAME so each excluded object is on a separate line (press Enter even if only one object is listed). Wildcard patterns are allowed. Include quotes around the names if the names include spaces. For example, if you don't want to compare the Administration Manager role, the exclude file would contain this line:

```
role "Administration Manager"
```

- `exception`

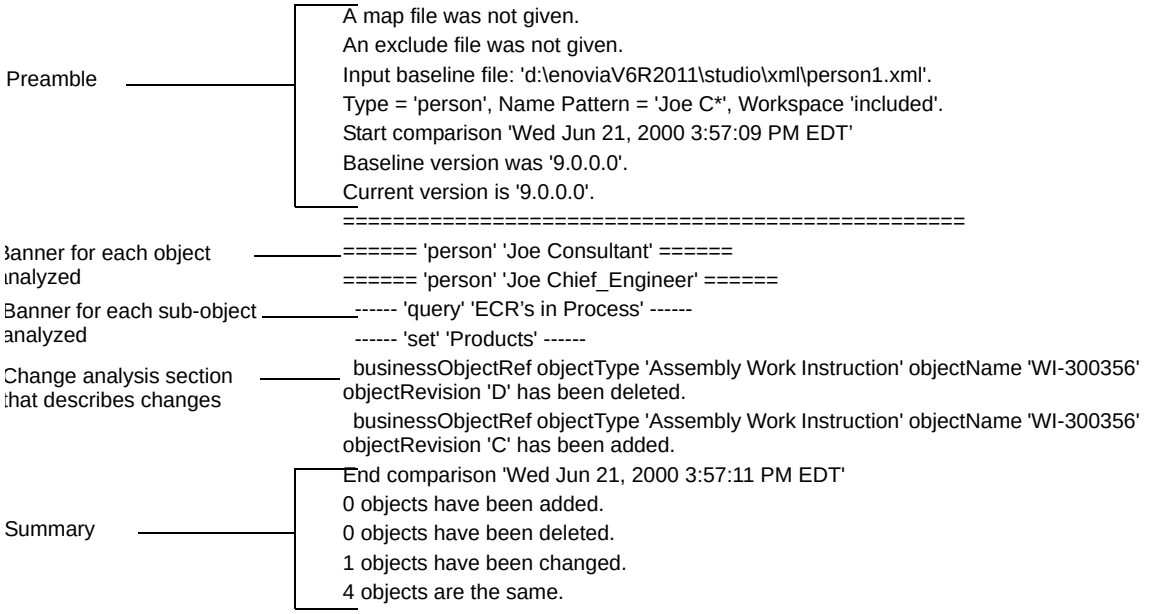
  The use exception FILENAME option creates a file that lists objects that could not be compared. If a transaction aborts, all objects from the beginning of that transaction up to and including the "bad" object are written to the exception file.

FILENAME is the path and name of the file to be created (log and exception files) or used (map and exclude files). If you don't specify a path, Live Collaboration uses the directory that contains the mql.exe file.

Reading the Report

If you specify a log file in the compare command, the compare operation generates a report that lists all objects analyzed and the changes. The report format is simple ASCII text. The report contains enough information to enable an expert user to write MQL scripts that apply the changes to a database.

Below is a sample of a report with the main sections of the report indicated. Each section of the report is described below.



Preamble—Lists the clauses and options used in the compare command, the time of the operation, and software version numbers.

Object banner—Each object analyzed is introduced with a banner that includes the object type and name wrapped by "=" characters.

Sub-object banner—Each sub-object analyzed for an object is listed under the object banner. The sub-object banner includes the sub-object type and name wrapped by "-" characters. In the above example, only one object, Joe Chief_Engineer, has sub-objects, which in this case is a query and a set.

Change Analysis—Following the banner for each object and sub-object analyzed, there are four possibilities:

1. If no changes are found, then no analysis lines appear. The next line is the banner for the next object/sub-object or the summary section.
2. If the object (sub-object) has been added, then the following line appears: "Has been added."
3. If the object (sub-object) has been deleted, then the following line appears: "Has been deleted."
4. If the object (sub-object) has been changed, there are three possibilities:

   a) If a field has been added, then the following line appears: "FIELD has been added."

   b) If a field has been deleted, then the following line appears: "FIELD has been deleted."

   c) If a field has changed, then the following line appears: "FIELD has been changed."

   where FIELD is in the following form: FIELDTYPE ['FIELDNAME'] [SUBFIELDTYPE ['FIELDNAME']] ...

   and FIELDTYPE identifies the type of field using tags found in the ematrixml.dtd file, and FIELDNAME identifies the name of the field when more than one choice exists.

   The best way to identify the field that has changed is to traverse the XML tree structure, looking at element names (tags) and the value of any name elements (placed in single quotes) along the way. Use the ematrixml.dtd file as a roadmap. Element names never have single quotes around them, and values of name elements always have single quotes around them. This should help parsing logic distinguish between the two.

   Here are some sample messages that would appear in the change analysis section if the compare operation finds that an object has changed (possibility 4):

```
    frame 'Change Class' has been added.
    typeRefList 'Change Notice' has been deleted.
    field fieldType 'select' has been deleted.
    widget 'ReasonForChange' multiline has been changed
    widget 'ReasonForChange' validateProgram programRef has been
changed.
    businessObjectRef objectType 'Assembly Work Instruction'
objectName 'WI-300356' objectRevision 'D' has been deleted.
```

Summary—The final section of the report contains a timestamp followed by the same summary that appears in the MQL window.

Verbose Mode

Turn on verbose mode to see more details in the report for changed objects/sub-objects (possibility 4 in the above description). To see these additional details, make sure you turn on verbose mode before issuing the compare command.

Verbose mode does not produce additional information if an object has been added (possibility 2 in the above description) or deleted (possibility 3). To get more information about an added object, use a print command for the object. To gather information about a deleted object, look at the XML export file used for the baseline.

When the operation finds changes to objects, verbose mode adds text as follows:

- For possibility 4a (field has been added), the keyword “new” appears followed by VALUE.
- For possibility 4b (field was deleted), the keyword “was” appears followed by VALUE.
- For possibility 4c (field was changed), the keywords “was” and “now” appear, each followed by VALUE.

  where VALUE is either the actual value (in single quotes) or a series of name/value pairs (where the value portion of the name/value pair is in single quotes).

Here are some sample messages that would appear in the change analysis section if the compare operation finds that an object has changed (possibility 4) and verbose mode is turned on:

```
field fieldType 'select' has been added.
   new absoluteX '0' absoluteY '0' xLocation
      '382.500000' yLocation '36.000000' width
      '122.400002' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor 'red' backgroundColor ''
      fieldValue 'name' fontName 'Arial Rounded MT
      Bold-10' multiline '0' editable '0'

field fieldType 'select' has been deleted.
   was absoluteX '0' absoluteY '0' xLocation
      '382.500000' yLocation '36.000000' width
      '122.400002' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor 'red' backgroundColor ''
      fieldValue 'name' fontName 'Arial Rounded MT
      Bold-10'

width has been changed.
   was '795.0'
   now '792.0'

field fieldType 'label' has been changed.
   was absoluteX '0' absoluteY '0' xLocation
      '191.250000' yLocation '110.000000' width
      '252.449997' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor '' backgroundColor ''
      fieldValue 'Maximum Distance Between Centers:
      'fontName ''
   now absoluteX '0' absoluteY '0' xLocation
      '191.250000' yLocation '108.000000' width
      '252.449997' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor '' backgroundColor ''
      fieldValue 'Maximum Distance Between Centers:
      ' fontName ''
```

Comparing Person objects

Live Collaboration does not include the default users creator and guest when you export all person objects with:

```
MQL< >export person * xml into file /temp/person.xml;
```

So if you then compare this exported file to another schema, even if the same Person objects exist, the compare output will show that 2 objects have been added. For example:

```
MQL<7>compare person * from file /temp/person.xml use log person.log;
2 objects have been added.
0 objects have been removed.
0 objects have been changed.
172 objects are the same.
```

The log will show

```
....
====== 'person' 'guest' ======
Has been added.
====== 'person' 'creator' ======
Has been added.
```

Examples

Below are two example MQL sessions. The first MQL session shows two baseline export files being created.

```
Matrix Query Language Interface, Version 9.0.0.0
Copyright (c) 1993-2000 MatrixOne, Inc.
All rights reserved.
MQL<1>set context user Administrator;
MQL<2>xml on;
MQL<3>export program A* into file d:\enoviaV6R2011\studio\xml\program1.xml;
MQL<4>export person * into file d:\enoviaV6R2011\studio\xml\person1.xml;
MQL<5>quit;
```

Line <2> places the session into XML mode. All subsequent export commands will generate export files in XML format.

Line <3> exports all programs (including wizards) that start with “A”.

Line <4> exports all persons.

The second MQL session shows several comparisons being performed using the baseline export file person1.xml. It is assumed changes have occurred in the database, or the session is being performed on a different database.

```
MQL<1>set context user Administrator;
MQL<2>compare person “Joe C*” from file
d:\enoviaV6R2011\studio\xml\person1.xml;
0 objects have been added.
0 objects have been removed.
0 objects have been changed.
5 objects are the same.
MQL<3>compare person “Joe C*” workspace from file
d:\enoviaV6R2011\studio\xml\person1.xml;
0 objects have been added.
0 objects have been removed.
1 objects have been changed.
4 objects are the same.
MQL<4>verbose on;
MQL<5>compare person “Joe C*” workspace from file
d:\enoviaV6R2011\studio\xml\person1.xml use log
d:\enoviaV6R2011\studio\xml\person1w.log;
0 objects have been added.
0 objects have been removed.
1 objects have been changed.
4 objects are the same.
compare successfully completed.
```

Line <2> compares all person objects that start with “Joe C” with the baseline file person1.xml. Since no log file is specified, no report is generated. (Not having a log file would typically be done to see if anything has changed.) The summary message states that none of the 5 objects analyzed have changed.

Line <3> performs the same compare but also includes workspace items assigned to the persons. The results now show that there has been a change. To view the changes, a report must be generated.

Line <4> turns on verbose mode.

Line <5> performs the previous compare but also gives a log file to place the report into. The resulting report can be found in *Reading the Report*.

This portion of the MQL window shows a continuation of the previous session. Here, the baseline export file program1.xml is used for several more comparisons. The sections that follow show the contents of the files used in the compare commands.

```
MQL<6>verbose off;
MQL<7>compare program A* from file d:\enoviaV6R2011\studio\xml\program1.xml
use log d:\enoviaV6R2011\studio\xml\program1.log map
d:\enoviaV6R2011\studio\xml\program1.map;
2 objects have been added.
0 objects have been removed.
3 objects have been changed.
11 objects are the same.
compare successfully completed.
MQL<8>verbose on;
MQL<9>compare program A* from file d:\enoviaV6R2011\studio\xml\program1.xml
use log d:\enoviaV6R2011\studio\xml\program2.log map
d:\enoviaV6R2011\studio\xml\program1.map;
2 objects have been added.
0 objects have been removed.
3 objects have been changed.
11 objects are the same.
compare successfully completed.
MQL<10>compare program A* from file
d:\enoviaV6R2011\studio\xml\program1.xml use log program3.log map
d:\enoviaV6R2011\studio\xml\program1.map exclude
d:\enoviaV6R2011\studio\xml\program1.exc;
2 objects have been added.
0 objects have been removed.
2 objects have been changed.
```

Line <7> performs a compare on all program objects that start with the letter “A” but also includes a map file that identifies a rename of one of the program objects. See *program1.map* and *program1.log*. Notice that “use” is only used once even though two files are used (a log file and a map file).

Line <9> performs the same compare as in Line <7> but with verbose mode turned on. Look at the two reports (*program1.map* and *program2.log*) to see the difference between verbose off and on.

Line <10> performs the same compare but adds an exclude file that eliminates two program objects from the analysis (thus leading to two fewer objects mentioned in the results). See *program1.exc* and *program3.log*.

program1.map

```
program “Add Task” “Add Task Import”
```

program1.log

```
Map file 'd:\enoviaV6R2011\studio\xml\program1.map' successfully read.
An exclude file was not given.
Input baseline file: 'd:\C:\enoviaV6R2011\studio\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:49 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
==================================================
====== 'program' 'Add Component (As-Designed)' ======
description has been changed.
====== 'program' 'Add Assembly (As-Designed)' ======
====== 'program' 'Add Purchase Requisition' ======
====== 'program' 'Add Event' ======
====== 'program' 'AttributeProg' ======
====== 'program' 'A' ======
====== 'program' 'A1' ======
====== 'program' 'A2' ======
====== 'program' 'Add ECR' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Reason For Change' ------
  width has been changed.
  widget 'ReasonForChange' multiline has been changed.
  widget 'ReasonForChange' validateProgram programRef has been
changed.
  widget 'postECR Input1label4' fontName has been changed.
  widget 'Reason For Changelabel5' widgetValue has been changed.
  ------ 'frame' 'Product Line' ------
  widget 'Product Linelabel4' has been added.
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  widget 'Product Linelabel4' has been deleted.
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
frame 'Change Class' has been added.
====== 'program' 'Add Assembly' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Make vs Buy' ------
  widget 'MakevsBuy' validateProgram programRef has been changed.
  ------ 'frame' 'Part Family' ------
  ------ 'frame' 'Assembly Description' ------
  ------ 'frame' 'Target Parameters' ------
  ------ 'frame' 'Status Feedback' ------
'program' 'Add Task' mapped to 'Add Task Import'
====== 'program' 'Add Task Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
```

```
  ------ 'frame' 'Key Task Name' ------
  ------ 'frame' 'Task Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add Note' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'prepreStatus Feedback' ------
  ------ 'frame' 'preStatus Feedback' ------
====== 'program' 'Add Operation' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Operation Name' ------
  ------ 'frame' 'Operation Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add ECR Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Change Class' ------
  ------ 'frame' 'Reason For Change' ------
  ------ 'frame' 'Product Line' ------
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Audioplay' ======
Has been added.
====== 'program' 'Add Task' ======
Has been added.
=====================================================
End comparison at 'Wed Jun 21, 2000 2:13:51 PM EDT'.
2 objects have been added.
0 objects have been deleted.
3 objects have been changed.
11 objects are the same.
```

program2.log

```
Map file 'd:\enoviaV6R2011\studio\xml\program1.map' successfully read.
An exclude file was not given.
Input baseline file: 'd:\enoviaV6R2011\studio\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:36 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
=====================================================
====== 'program' 'Add Component (As-Designed)' ======
description has been changed.
  was 'Matrix Prof Services: Contains settings for the program to
create and connect and a new object with AutoName logic.'
  now 'Matrix Professional Services: Contains settings for the program
to create and connect and a new object with AutoName logic.'
====== 'program' 'Add Assembly (As-Designed)' ======
====== 'program' 'Add Purchase Requisition' ======
====== 'program' 'Add Event' ======
====== 'program' 'AttributeProg' ======
====== 'program' 'A' ======
```

```
====== 'program' 'A1' ======
====== 'program' 'A2' ======
====== 'program' 'Add ECR' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Reason For Change' ------
  width has been changed.
    was '380.0'
    now '360.0'
  widget 'ReasonForChange' multiline has been changed.
    was '0'
    now '1'
  widget 'ReasonForChange' validateProgram programRef has been
changed.
    was 'NameCheck2'
    now 'NameCheck'
  widget 'postECR Input1label4' fontName has been changed.
    was 'Arial-bold-14'
    now 'Arial-bold-10'
  widget 'Reason For Changelabel5' widgetValue has been changed.
    was 'Enter The Stock Disposition:'
    now 'Enter Stock Disposition:'
  ------ 'frame' 'Product Line' ------
  widget 'Product Linelabel4' has been added.
    new absoluteX '0' absoluteY '0' xLocation '198.900009' yLocation
'72.000000' width '160.650009' height '24.000000'
      autoWidth '0' autoHeight '0' border '0' foregroundColor ''
backgroundColor ''
        widgetType 'label' widgetNumber '100002'
        widgetValue 'Enter Product Line' fontName 'Arial-bold-10'
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  widget 'Product Linelabel4' has been deleted.
    was absoluteX '0' absoluteY '0' xLocation '198.900009' yLocation
'72.000000' width '160.650009' height '24.000000'
      autoWidth '0' autoHeight '0' border '0' foregroundColor ''
backgroundColor ''
        widgetType 'label' widgetNumber '100002'
        widgetValue 'Enter Product Line' fontName 'Arial-bold-10'
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
frame 'Change Class' has been added.
====== 'program' 'Add Assembly' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Make vs Buy' ------
  widget 'MakevsBuy' validateProgram programRef has been changed.
    was 'NameCheckTest'
    now 'NameCheck'
  ------ 'frame' 'Part Family' ------
  ------ 'frame' 'Assembly Description' ------
  ------ 'frame' 'Target Parameters' ------
  ------ 'frame' 'Status Feedback' ------
'program' 'Add Task' mapped to 'Add Task Import'
====== 'program' 'Add Task Import' ======
  ------ 'frame' 'Master Frame' ------
```

```
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Task Name' ------
  ------ 'frame' 'Task Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add Note' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'prepreStatus Feedback' ------
  ------ 'frame' 'preStatus Feedback' ------
====== 'program' 'Add Operation' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Operation Name' ------
  ------ 'frame' 'Operation Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add ECR Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Change Class' ------
  ------ 'frame' 'Reason For Change' ------
  ------ 'frame' 'Product Line' ------
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Audioplay' ======
Has been added.
====== 'program' 'Add Task' ======
Has been added.
=====================================================
End comparison at 'Wed Jun 21, 2000 2:13:39 PM EDT'.
2 objects have been added.
0 objects have been deleted.
3 objects have been changed.
11 objects are the same.
```

program1.exc

```
program “Add ECR”
program “Add Note”
```

program3.log

```
Map file 'd:\enoviaV6R2011\studio\xml\program1.map' successfully read.
Exclude file 'd:\enoviaV6R2011\studio\xml\program1.exc' successfully
read.
Input baseline file: 'd:\enoviaV6R2011\studio\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:28 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
=====================================================
====== 'program' 'Add Component (As-Designed)' ======
description has been changed.
  was 'Matrix Prof Services: Contains settings for the program to
```

```
create and connect and a new object with AutoName logic.'
  now 'Matrix Professional Services: Contains settings for the program
to create and connect and a new object with AutoName logic.'
====== 'program' 'Add Assembly (As-Designed)' ======
====== 'program' 'Add Purchase Requisition' ======
====== 'program' 'Add Event' ======
====== 'program' 'AttributeProg' ======
====== 'program' 'A' ======
====== 'program' 'A1' ======
====== 'program' 'A2' ======
====== 'program' 'Add Assembly' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Make vs Buy' ------
  widget 'MakevsBuy' validateProgram programRef has been changed.
    was 'NameCheckTest'
    now 'NameCheck'
  ------ 'frame' 'Part Family' ------
  ------ 'frame' 'Assembly Description' ------
  ------ 'frame' 'Target Parameters' ------
  ------ 'frame' 'Status Feedback' ------
'program' 'Add Task' mapped to 'Add Task Import'
====== 'program' 'Add Task Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Task Name' ------
  ------ 'frame' 'Task Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add Operation' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Operation Name' ------
  ------ 'frame' 'Operation Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add ECR Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Change Class' ------
  ------ 'frame' 'Reason For Change' ------
  ------ 'frame' 'Product Line' ------
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Audioplay' ======
Has been added.
====== 'program' 'Add ECR' ======
Has been excluded.
====== 'program' 'Add Task' ======
Has been added.
====== 'program' 'Add Note' ======
Has been excluded.
================================================
End comparison at 'Wed Jun 21, 2000 2:13:29 PM EDT'.
2 objects have been added.
0 objects have been deleted.
```

```
2 objects have been changed.
10 objects are the same.
```

Reference Commands

This section provides reference information specific to *MQL*.

In this section:

- *General Syntax* is provided.
- Each MQL command is described. Conceptual information is provided earlier in this guide.
- *Common Clauses* are explained once and referenced throughout this section.

General Syntax

The remainder of this chapter presents the syntax for commands that are common to many MQL features (*items* such as Vault, Person, Group, etc.).

Command Syntax Notation

Here are some general rules when reading syntax in this manual.

- In the syntax definition of the command, words in lowercase are keywords. Examples are: add, print, help. When typed into MQL, they may consist of mixed lower and upper case.
- Items in UPPER CASE refer to categories. Examples are NAME, VALUE. When typed into MQL, appropriate substitutions of actual names or values must be used.

Note that names/values are case sensitive.

- Items in square brackets, [], are optional.
- Items in braces, {}, may appear zero or more times.
- Items between vertical lines are options of which one must be chosen. That is:
  | item_1 | item_2 | item_3 | means choose one of the items.
- MQL commands are entered as a free form list of words separated by one or more blanks, tabs, or newlines.
- Command lines should end with a semicolon(;). However, a command is also assumed ended when two consecutive newlines are entered.
- Text appearing after a pound sign(#), up to and including the next newline, is ignored as a comment
- NAMES, VALUES, etc. must be quoted (single or double) when they have embedded blanks, tabs, newlines, commas, semicolons, or pound signs.
- Any keyword that accepts a list of VALUES (eg. keyword [VALUE {,VALUE}]) may also be specified separately for each value. For example:
  attribute first,second,third
  or
  attribute first
  attribute second
  attribute third
- Most keywords may be abbreviated to three characters (or the least number that will make them unique).

Item Commands

Use these commands to create, copy, modify, print or delete an instance of an item. See below for a list of items. Each item may have slightly different so it is important to also refer to the detailed description of each item found later in this chapter.

User Level

Business Administrator

Syntax

```
[add|copy|modify|print|delete]ITEM ITEM_NAME [CLAUSES
{SUBCLAUSES}];
```

- `ITEM_NAME` is the name of the item.
- `CLAUSEs` provide additional information about the item.
- `SUBCLAUSEs` provide additional information about the item.

Add Item

This command will create an instance of an item. The Add command generally has many clauses and, in some cases, subclauses specific to each item.

Syntax:

```
add ITEM ITEM_NAME [ADD_ITEM {ADD_ITEM}];
```

`ITEM` can be any of the following:

application	association	attribute
businessobject	channel	command
connection	cue	dataobject
dimension	expression	filter
form	format	group
icon	index	inquiry
interface	menu	person
location	portal	program
policy	query	relationship
property	rule	set
role	store	table
site	vault	webreport
type		

- `ITEM_NAME` is a unique name for that kind of item.
- `ADD_ITEMs` are applicable fields for each item with defined values.

Copy Item

This command will clone a definition, rename it, and make the specified modifications.

Syntax:

```
copy ITEM SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- ITEM can be any of the items listed for the Add Item command above.
- SRC_NAME is the original (source) item name.
- DST_NAME is the new (destination) item name.
- MOD_ITEMs are applicable fields for each item with new values.

Modify Item

This command will make the specified changes to an existing definition.

```
modify ITEM NAME [MOD_ITEM {MOD_ITEM}];
```

- ITEM can be any of the items listed for the Add Item command above.
- NAME is a unique name for that kind of item.
- MOD_ITEMs are applicable fields for each item with new values. Refer to the specific item chapters for more information.

Print Item

The Print Item command prints the item definition to the screen allowing you to view it. When a Print command is entered, MQL displays the various clauses that make up the definition.

```
print ITEM NAME SELECT;
```

- ITEM can be any of the items listed for the Add Item command above.
- NAME is the specific instance of the item.
- SELECT lets you specify data to present about the item being printed.

Delete Item

This command enables you to delete an item when it is no longer needed in the Live Collaboration environment. Because deleting certain items can affect other elements of Live Collaboration, use it carefully. See the Delete section of each item chapter for a discussion of the impact of the deletion.

```
delete ITEM NAME;
```

- ITEM can be any of the items listed for the Add Item command above.
- ITEM can be any of the following
- NAME is the specific instance of the item.

Help Item

The Help Item command is always available while you are in the MQL interactive mode. Help commands produce a summary of all commands applicable to the item with the appropriate syntax. See *Important MQL Commands* in Chapter 1 for details.

list admintype Command

You can use the List Admintype command in its simplest form to list all instances of an administrative type (including workspace objects) that are currently defined. It is useful in confirming the existence or exact name of an object you want to modify. Additional clauses allow you to limit the objects listed, include selectable data about each object listed, and/or provide definition for the output format.

User Level

Business Administrator

Syntax

The 2 forms of the List Admintype command are:

```
list ADMIN_TYPE [ modified after DATE ] [DUMP [RECORDSEP]] [tcl] [output
FILENAME];
```

```
list ADMIN_TYPE [ NAME_PATTERN [where EXPRESSION][SELECT] ] [DUMP [RECORDSEP]]
[tcl] [output FILENAME];
```

Definitions	
ADMIN_TYPE	This can be any type of administrative object you can create. This includes workspace objects as well. When specifying a workspace object, you can optionally include the [user USERNAME] clause to indicate the person, group, role, or association, for which you want to list the workspace objects. Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.
DATE	This must be in the format specified in .ini file. You cannot include a NAME_PATTERN when using the modified after clause.
NAME_PATTERN	This can be a single name or a comma-delimited list of expressions that can include wildcard characters. You cannot include a modified after DATE clause when using NAME_PATTERN.
SELECT	This specifies data to present about each item being listed. You must include a NAME_PATTERN when using SELECT.
DUMP	This allows you to format the printed information.
RECORD_SEP	This specifies a separator character for the select output of each object listed.
tcl	This returns the results in Tcl list format.
FILENAME	This identifies a file where the print output is to be stored.

Each of these clauses is explained in detail in the sections that follow.

Name pattern

- NAME_PATTERN can be a comma-delimited list of expressions that can include the wildcard characters '*' or '?', where * matches any string of characters, and ? matches any single character. For example, if this command:

  ```
  list person a*
  ```

  produces this list: abadi,abami,adadi,adami,adams,apkarian,ata

  then:

Command	Produces:
`list person a?ami`	adami,abami
`list person ada?i`	adami,adadi
`list person a*i`	adami,adadi,abami,abadi,atai
`list person ad*,a*n`	adami,adams,apkarian,adadi

Email Pattern

This command returns a list of persons with email addresses that match the provided pattern:

```
list person email EMAIL_ADDRESS_PATTERN;
```

Where Clause on User List Commands

The "list person", "list group", "list role", and "list interface" commands allow Where clauses to refine the list returned. The Where clause comes after the name pattern, and before any Select clauses. For example:

```
list role * where ancestor==ProjectX;
```

This evaluates to true for data allowed for a user's Project.

Select Clause

This clause lets you specify data to present about each item being listed. You must include a NAME_PATTERN when using the Select clause (and therefore cannot use a `modified after DATE` clause with it). The result of the following query would be a list of attributes beginning with "mx," showing the name and type. For example:

```
MQL<n>list attribute mx* select type;
attribute type   mxsysInterface
    type = string
```

Another form of the Select clause is:

```
list type TYPE select attribute[].name;
```

An abstract type or relationship given in the selectable is substituted with the derived type or relationship. For example:

```
MQL<n>list type AB* select attribute[].name;
business type   ABSTRACT PART
    attribute[Effectivity Date].name = Effectivity Date
    attribute[Estimated Cost].name = Estimated Cost
    attribute[Lead Time].name = Lead Time
    attribute[Material Category].name = Material Category
    attribute[Target Cost].name = Target Cost
    attribute[Unit of Measure].name = Unit of Measure
    attribute[Weight].name = Weight
```

The result of a query using the Select clause is typically in the key-value format. For example:

```
MQL<n>list type Ac* select attribute[].name;
business type   Access Request
    attribute[Reason for Request].name = Reason for Request
    attribute[Grant Expiration Date].name = Grant Expiration Date
    attribute[Comments].name = Comments
    attribute[Extension Date].name = Extension Date
    attribute[Originator].name = Originator
    attribute[Include Self].name = Include Self
business type   Actual Transaction
    attribute[Transaction Date].name = Transaction Date
    attribute[Transaction PON].name = Transaction PON
    attribute[Transaction Amount].name = Transaction Amount
```

Generally, the "key" part of the output is expanded by way of substitution or filled with values that match the selectable. The same is true for input specifications using '*' or "[]". This in-place substitution, although it provides a complete and accurate answer to queries, can be a problem in client applications because the input may not always match the resultant key set, since "`attribute[].name`" does not match "`attribute[ATTR_NAME].value`". To avoid substitution, applications were forced to build multiple queries with precise definitions so that the result could be easily processed. In V6R2013x and later, you can use `[!|not]substitute` to force the value of the key in the result to be consistent with the input. For example:

```
MQL<n>list type Ac* select !substitute attribute[].name;
business type   Access Request
    attribute.name = Reason for Request
    attribute.name = Grant Expiration Date
    attribute.name = Comments
    attribute.name = Extension Date
    attribute.name = Originator
    attribute.name = Include Self
business type   Actual Transaction
    attribute.name = Transaction Date
    attribute.name = Transaction PON
    attribute.name = Transaction Amount
```

The select without substitution option works for all commands that support selectables, including:

```
list ADMIN_TYPE [SELECT] ...;
print ADMIN_TYPE [SELECT] ...;
print businessobject T N R [SELECT] ...;
print connection ID [SELECT] ...;
temp query businessobject T N R [SELECT] ...;
expand bus T N R [SELECT_BUS] ... [SELECT_REL] ...;
```

After DATE Clause

This clause can be used to "query" based on modified dates. For example:

```
list attribute modified after DATE;
```

- where `DATE` follows the format specified in the .ini file.

Listing a family of related objects

The following command can be used to access the contents of the mxFamily table:

```
list family [majorid UUID {,UUID,...}] [before TIMESTAMP]
[after TIMESTAMP] [selector NAME];
```

For example:

```
list family after TIMESTAMP; // lists all family entries made after a
                             // specific time
list family majorid ID1,ID2,...; // lists all family entries for each
                                 // of the specified IDs
list family selector BSF; // lists all BSF entries in the family table
```

All list options may be combined with other options (e.g., before/after may be combined for a particular date range).

Administrative Object Names

You can use your exact business terminology rather than cryptic words that are modified to conform to the computer system limitations.

There are few restrictions on the characters used for naming administrative objects. Names are case-sensitive and spaces are allowed. You can use complete names rather than contractions, making the terminology in your system easier for people to understand.

The maximum length of administrative object name and revision strings varies for different databases. The maximum length for SQL Server is 127 characters, whereas for Oracle and DB2 it is 255 *bytes*. The number of bytes per character depends on the database setting for character encoding. Thus, for Oracle/DB2, the character count will be fewer than for SQL Server if some characters are multibyte. The same is true for all string values stored in the database. See the Syntax section in *businessobject Command*.

Avoid using characters that are programmatically significant to Live Collaboration, MQL, URLs, and associated apps. These characters include:

```
/ \ | * ^ ( ) [ ] { } = < > $ # % & ! @ ? " ; : ,' §
```

Leading and trailing spaces are ignored in administrative object name and revision strings.

Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as

those created with the ESC key, should not be used for ANY fields, including business and administrative object names, description fields, program object code, or page object content.

Avoid giving any administrative object a NAME that can be confused with a keyword when parsing MQL command input. For example, a Role named “All” is misinterpreted if you try to use the role name in the definition of an access rule (or state access definition), since “all” is the keyword that allows all access rights.

Adding History to Administrative Objects

Administrative objects are objects in the database that represent the definition of application data models (for example, business types, relationship types, attributes, policies, or formats). Object history is a set of records of changes that have been made to an object. History records can be added to administrative objects automatically or explicitly. This capability enables apps and System Administrators or Business Administrators to record changes made to administrative objects for purposes of both maintaining an audit trail and identifying the source of a change.

About History Records

History records are added to administrative objects as a result of any of the following events:

- Object creation creates a history object marked “create.”
- Object modification creates a history object marked “modify.”
- Explicit use of the history keyword in MQL commands for adding or modifying administrative objects creates additional history records marked “custom.” The history keyword allows a free-text string describing the nature of some change to the object.

For each history entry, a persistent record is stored in the database that contains the following information:

- The label “create,” “modify,” or “custom,” as described above
- An integer index that specifies the order in which the history record was created
- The user (current context) who performed the create or modify operation
- The date and time to the nearest second
- A descriptive string of up to 2,040 characters

For custom history that is explicitly added with the `history` keyword in add or modify admintype commands, the descriptive string is the string provided in the command.

For create or modify records that the system generates automatically, the descriptive string consists of the first 2,040 characters of the MQL command that creates the change. If the change is made through the business executable, the MQL command that is recorded as the descriptive string is the same as that generated for business scripting.

The MQL `import` or `export` command reads or writes, respectively, history records along with any administrative object.

Command-Line Interface for History Records

The business graphical user interface does not support the display of history records. These are available only through the command-line interface.

To add a history record with the add command, use the following format:

```
add <admintype> NAME history STRING;
```

To add a history record with the modify command, use the following format:

```
modify <admintype> NAME history STRING;
```

History records are appended to the end of the output for print commands such as:

```
print <admintype> NAME;
```

The output format of print commands for admin objects with history records is as follows:

```
history index:1 type:custom user:creator date:Day Mmm dd, yyyy
hh:mm:ss AM/PM ZONE text:xxx
```

In the above, <admintype> can be one of only the following types:

- `attribute`
- `channel`
- `command`
- `dimension`
- `form`
- `format`
- `group`
- `index`
- `inquiry`
- `menu`
- `page`
- `person`
- `policy`
- `portal`
- `program`
- `relationship`
- `resource`
- `role`
- `rule`
- `site`
- `table` (system tables only - history does not apply to user tables)
- `type`

The history keyword is not available for user-owned workspace objects, such as the following:

- `cue`
- `filter`
- `query`
- `set`
- `table`
- `tip`
- `toolset`

Printing History for Administrative Objects

The print command for administrative objects supports a history selectable to request the printing of history records only. This command has the following syntax:

```
print <admintype> NAME select history;
```

The format of the output of such a print command is as follows:

```
history index:1 type:custom user:creator date:Day Mmm dd, yyyy
hh:mm:ss AM/AM ZONE text:xxx
```

There are no subselects for the history selectable. In other words, it is not possible to select subsets of history records.

History can be added only to databases that have been upgraded to V6R2011x level or later. Print commands that request history for objects in pre-V6R2011x databases return no records.

If history records exist, you can print information about the objects without printing their history using either of these methods:

- Use the print command with a select clause to select specific pieces of information, but do not include the history selectable:

  ```
  print <admintype> select ...
  ```

- Use the print command with the history selectable negated to print the non-formatted, complete admintype definition, but with history suppressed:

  ```
  print <admintype> !history
  ```

Adding Multiple History Records

You can add multiple history records to the same or different objects in a transaction. Date and time are calculated at the time the history addition is requested. If a transaction involving history addition is aborted, the history record is not written.

History with MQL Import/Export Commands

MQL import or export commands read or write, respectively, history records along with any administrative object.

Localization of History Text Strings

The text string input for a history record is treated the same way as any other string attribute, specifically:

- In a language-specific (in other words, non-UTF8) configuration, it is treated according to the system LANG setting.
- In a multilingual UTF8 server environment, it is treated as a UTF8 string.

Extended History Records

By default, history records are truncated for description and string attribute changes. You can change this behavior by using the following system command:

```
set system truncatehistory on|off;
```

When `truncatehistory` is on, the text is truncated at 254 bytes. This is the default behavior.

By turning the `truncatehistory` option off, you can store complete history records up to 3MB in the database. History records that exceed a predetermined length are stored in and retrieved from a separate database table. History records are written to either the lxHistory table or the lxDescription table based on length and the `truncatehistory` setting.

Once a history record has been truncated, turning `truncatehistory` off will not restore the information. Conversely, turning `truncatehistory` on will not truncate history records that have already been stored.

Common Clauses

Active Clause

The Active clause is used to indicate that an object is active or not active (!active). The default is active.

Description Clause

There may be subtle differences between items; the Description clause points out the differences to the user.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the item in a chooser in the GUI applications. Although you are not required to provide a description, this information is helpful when a choice is requested.

Dump Clause

You can specify a general output format with the Dump clause. The Dump clause specifies that you do not want to print the leading field names. For example, without the Dump clause you might get:

```
MQL<30>list format mx* select modified id;
format   mxThumbnail Image
    modified = 1/28/2006 2:07:16 AM
    id = 0.1.35873.42707
format   mxSmall Image
    modified = 1/28/2006 2:07:16 AM
    id = 0.1.35873.53610
format   mxLarge Image
    modified = 1/28/2006 2:07:16 AM
    id = 0.1.35890.47596
format   mxImage
    modified = 1/28/2006 2:07:16 AM
    id = 0.1.35891.14732
```

With the Dump clause, the data is easier to parse:

```
MQL<29>list format mx* select modified id dump;
1/28/2006 2:07:16 AM,0.1.35873.42707
1/28/2006 2:07:16 AM,0.1.35873.53610
1/28/2006 2:07:16 AM,0.1.35890.35949
1/28/2006 2:07:16 AM,0.1.35890.47596
1/28/2006 2:07:16 AM,0.1.35891.14732
```

You can also specify which character you want to use to separate the fields in the output:

```
dump ["SEPARATOR_STR"]
```

`SEPARATOR_STR` is a character or character string that you want to appear between the field values. It can be a tab, comma, semicolon, carriage return, etc. If you do not specify a separator string value, the default value of a comma is used. If tabs or carriage returns are used, they must be enclosed in double quotes (“ ”).

When using the Dump clause, all the field values for each object are printed on a single line unless a carriage return is the separator string.

Hidden Clause

An item can be marked “hidden” so that it does not appear in the chooser in the 3DEXPERIENCE Platform. Users who are aware of the hidden item’s existence can enter the name manually where appropriate. Hidden objects are accessible through MQL.

Icon Clause

Icons help users locate and recognize items by associating a special image with the item. You can assign a special icon to the new administrative object or use the default icon. The default icon is used when in view-by-icon mode in Business Modeler. Any special icon you assign is used when in view-by-image mode of Business Modeler.

For example, you may want to use a company logo for a Vault that contains objects related to doing business with that company. Also, you could use an image of a tax form for a Tax Form Type.

When assigning a unique icon, you must use a GIF image file.To write an Icon clause, you need the full directory path to the icon you are assigning. For example, the following command assigns a calendar icon to the 2007 Records Vault:

```
add vault “2007 Records”
icon $MATRIXHOME/pixmaps/app/calendar2007.gif;
```

The gif file must be accessible while defining the item, but then it is stored in the database.

GIF filenames should not include the @ sign, as that is used internally by ENOVIA Live Collaboration.

Icons can be checked out of the objects that contain them. For syntax, see *Retrieving the Image*.

In Vault Clause

When specifying business objects in MQL commands, the `In Vault` clause is generally available. This clause improves the performance of such transactions as modify, delete, and connect, since it is no longer necessary to search all vaults to find the object to act upon. This is particularly true in a distributed database.

Output Clause

This clause provides the full file specification and path to be used for storing the output of the List command. For example, to store information in an external file called sales.txt, you might write a command similar to:

```
list type Sales* select name description format dump
output c:\mydocs\sales.txt.
```

Owner Clause

This clause is used to define the owner of an object or search for objects owned by a particular user.

The Owner clause is optional when creating an object. If you do not include one, MQL will assume the owner is the current user, which is defined by the present context. This means that the System and Business Administrators can create objects for other users by first setting the context to that of the desired user and then creating the desired objects, or by using the Owner clause:

```
owner USER_NAME
```

The owner of an object can be assigned special access in the policy. The user who is assigned ownership of an object has access privileges defined in the Owner subclause of the policy. That user can be an individual, a group, or a role.

If the user name you give in the Owner clause is not found, an error message will result. If that occurs, use the List User command to check for the presence and spelling of the user name. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters.

For example, the following object definition assigns the role “Software Developer” as the owner of a business object titled “Graphics Display Routine:”

```
add businessobject “Computer Program” “Graphics Display Routine” A
   policy “Software Development”
   description “Routine for displaying information on a color monitor”
   owner “Software Developer”;
```

In this example, the administrator may not have a specific name of a user to assign to the business object. Therefore, the name of a role is used. All persons assigned the role of Software Developer can access the object as the owner. At a later time, a Person can be reassigned as the owner according to the reassign access rules specified in the governing policy.

The owner clause is also used to search for objects owned by a particular user or to assign items to particular user:

```
owner NAME_PATTERN
```

`NAME_PATTERN` is the owner(s) you are searching for or the owner(s) for which you are creating an item.

The Owner clause uses wildcard characters in a way similar to the Businessobject clause.You can also use multiple values to define the pattern. You can search for both the first and last name of an owner.

Property Clause

Integrators can assign ad hoc attributes, called Properties. Properties allow associations to exist between administrative or workspace definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administrative or workspace object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties. In order to use the Property clause you must have administrative Property access.

Recordseparator Clause

This clause allows you to define which character or character string you want to appear between the dumped output of each object when the expand command requests information about multiple objects.

```
recordseparator ["SEPARATOR_STR"]
```

`SEPARATOR_STR` is a character or character string that replaces the end-of-line (\n) ordinarily inserted at the end of each object's dumped output. It can be a tab, comma, semicolon, etc. If tabs are used, they must be enclosed in double quotes (" ").

Select Clause

This clause lets you specify data to present about each item being listed. You must include a NAME_PATTERN when using the Select clause (and therefore cannot use a `modified after DATE` clause with it). For example:

```
list attribute mx* select type;
```

The result would be a list of all attributes beginning with "mx," showing the name and type.

Select commands enable you to view a listing of the field names that make up the business object definition. Each field name is associated with a printable value. By selecting and listing the field names that you want, you can create a subset of the object's contents and print only that subset. The system attempts to produce output for each Select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

Select can be used as a clause of the `print businessobject` and `expand businessobject` commands (See *Expand Business Object* for more information), as well as in query where clauses (See *Using Select Clauses in Queries*).

Using Select and Related Clauses

Frequently, implementation code creates or modifies a businessobject or connection and then immediately needs to fetch new information from it. The select modifier is available so that data can be retrieved as output from the creation (including minorrevise and copy) or modification command itself, removing the necessity to make another, separate server call to get it, thereby improving performance. In the case of add/modify businessobject, the specified selects are applied to the businessobject being created/modified. The output produced is identical to a corresponding print bus command.

Refer to *Select commands* for syntax details on this and its related clauses.

If these selects are used within a user-defined transaction, they will return data representing the current uncommitted form of the object/connection. The data is collected and returned after all triggers associated with the command have been executed, unless a transaction is active and a

trigger is marked deferred, since in this case, the trigger is not executed until the transaction is committed, as shown below:

```
1) Larger transaction is started with any number of commands before the add/
modify/connect command with the trigger.
    2) add/modify/connect transaction is started.
        3) Normal processing of access checking occurs.
        4) If it exists, the Check Trigger is fired.
        5) If Check blocks, then transaction aborts.
           If not, the Override Trigger is fired if it exists.
    6) The Event transaction is then committed regardless of an override or
normal       activity.
    7) If the Event Trigger has a non-deferred Action Program, it is executed.
    8) If the add/modify/connect command included a Select clause, the data is
collected and returned at this point, even though the creation/modification has
not been committed.
9) The larger transaction is committed if all activities and triggers succeed.
10) Finally, if the larger transaction successfully commits, and there is a
deferred    Action Program, it is now triggered. If the larger transaction
aborts, deferred    programs are not executed.
```

Presumably, work would be done using the selected and returned data, between steps 8 and 9 above.

All Add Businessobject clauses provide information about the business object being defined. Only the Policy clause is required. The next sections describe each Add Businessobject clause.

Tcl Clause

MQL `print`, `list`, `expand`, and `temp query` commands that include selects accept an optional `tcl` clause to signal that results should be returned in Tcl list format. Use the Tcl clause after the Dump clause (if used) and before the Output clause to return select results in Tcl list format. This output from MQL select commands is parsed using the built-in list handling features of Tcl.

Additionally, logic has been added that guarantees an entry in the results list for every given *legal* select item. This “padding” ensures that when the Tcl clause is included, the results list can be parsed properly. If illegal select items are given, there are no corresponding entries in the results list. Commands that don’t include the Tcl clause create output as before without this “padding.”

Since the Tcl list structure is already curly brace delimited, any given separator characters (specified with the `dump` or `recordseparator` clauses) are ignored when generating Tcl output. However, for readability, newlines are used as record separators between objects. For instance, in an `expand bus` command that includes selects and the Tcl clause, all selected items for one object are included on one line, and a new line is started for each connected object. Newlines are considered simple white space by Tcl, and so cause no parsing problems.

To avoid issues with Tcl list operations, the three special characters ‘{’, ‘}’, and ‘\’ are escaped by the ‘\’ character wherever they occur in text appearing between curly braces.

This clause must follow the `Dump` clause, if used, and precede the `Output` clause. In temp queries, it must also follow any `recordseparator` clause used. For example:

```
print bus TYPE NAME REVISION [SELECT] [dump [RECORDSEP]] [tcl] [output
FILENAME];
```

```
print set SETNAME [SELECT] [dump [RECORDSEP]] [tcl] [output FILENAME];
list ADMINTYPE NAME_PATTERN [SELECT] [dump [RECORDSEP]] [tcl] [output
FILENAME];
expand bus TYPE NAME REVISION [SELECT] [dump RECORDSEP] [tcl] [output
FILENAME];
expand set SETNAME [SELECT] [dump RECORDSEP] [tcl] [output FILENAME];
temp query TYPE_PATTERN NAME_PATTERN REV_PATTERN QUERY_EXP [SELECT]
[dump [RECORDSEP]] [recordseparator SEPARATOR_STR] [tcl] [output
FILENAME];
```

Notice that when expanding both business objects and sets and including the Dump clause, a RECORDSEP is required. However, when the Tcl clause is also included, the separator is ignored. For justification, refer to Output Example number 11.

Output Format

The output from each Tcl select command is consistent with the expected MQL output, except that is uses Tcl list format. The results for each object are wrapped in a list, even when one object is returned, as is the case with the `print` command. Within each object record is a header section followed by a sequence of select result items, like so:

```
{ object1-header-data    select-result-items }
{ object2-header-data    select-result-items }
    :
```

Header Data

The format of the header data depends on the actual command.

- `Print`, `List`, and `Temporary Query` Commands
  business object:`{business object} {TYPE} {NAME} {REV}`
  connection object:`{connection} {ID}`
  administration:`{TYPE} {NAME}`
  set: no set header - each member uses:`{member business object} {TYPE} {NAME} {REV}`
- `Expand` Command
  `{LEVEL} {REL_NAME} {FROM/TO} {TYPE} {NAME} {REV}`

Select Results Data

Each item in the select-result-items is in the format:

```
{{select-item} results-list}
```

where each item in the results-list is:

```
{{name} {value}}
```

or the results-list is simply **{ }** if an empty result.

Using the Dump Clause

When the Dump clause is specified, most of the header is dropped, the **{select-item}** is dropped, and just **{value}** is given for each item in the results-list.

Some commands (like "expand") require a RECORDSEP to be included after the "dump" keyword. When used with the Tcl clause, this character is still required, but it is ignored.

Output examples

1. A simple print business object with Tcl clause.

```
MQL<2>print bus Assembly SA-300356 0 select id owner locker tcl;
{business object} {Assembly} {SA-300356} {0} {{id} {{id}
{20083.46775.20193.6352}}} {{owner} {{owner} {Joe Product-Manager}}}
{{locker} {{locker} {bucknam}}}
```

It may appear that there is some redundancy in the output in the select-result-items section. The first occurrence of an item (for example owner, above) matches the given select item and is followed by name/value pairs that resulted from the select item. Often the name field is redundant. However, when the select item results in many returned items (for example attribute.value or from.id) the name field gives added details. Refer to example 3 below.

2. The same print bus command including the Dump clause:

```
MQL<3>print bus Assembly SA-300356 0 select id owner locker dump tcl;
{{20083.46775.20193.6352}} {{Joe Product-Manager}} {{bucknam}}
```

3. A business object Print with a select that returns multiple items:

```
MQL<4>print bus Assembly SA-300356 0 select from.id tcl;
{business object} {Assembly} {SA-300356} {0} {{from.id}
{{from[Documentation].id} {20083.46775.30402.24363}}
{{from[Analysis].id} {20083.46775.30631.62767}} {{ from[Analysis].id}
{20083.46775.30631.41948}} {{from[Plan].id} {20083.46775.30632.60059}}
{{from[Required Tools].id} {20083.46775.30639.24886}} {{from[BOM-As
Designed].id} {20083.46775.30663.9546}} {{from[BOM-As Designed].id}
{20083.46775.6577.56182}}}
```

4. The same print of multiple items with the Dump clause, eliminating headers:

```
MQL<5>print bus Assembly SA-300356 0 select from.id dump tcl;
{{20083.46775.30402.24363} {20083.46775.30631.62767}
{20083.46775.30631.41948} {20083.46775.30632.60059}
{20083.46775.30639.24886} {20083.46775.30663.9546}
{20083.46775.6577.56182}}
```

5. A temporary query with simple select in Tcl list output (with newline between object records):

```
MQL<6>temp query bus Assembly SA* 0 limit 5 select id locker tcl;
{businessobject} {Assembly} {SA-300.125} {0} {{id} {{id}
{20083.46775.31292.44899}}} {{locker} {{locker} {}}}
{businessobject} {Assembly} {SA-300127} {0} {{id} {{id}
{20083.46775.20133.58276}}} {{locker} {{locker} {bucknam}}}
{businessobject} {Assembly} {SA-300195} {0} {{id} {{id}
{20083.46775.20117.54372}}} {{locker} {{locker} {bucknam}}}
{businessobject} {Assembly} {SA-300315} {0} {{id} {{id}
{20083.46775.20173.48444}}} {{locker} {{locker} {}}}
{businessobject} {Assembly} {SA-300356} {0} {{id} {{id}
{20083.46775.20193.6352}}} {{locker} {{locker} {bucknam}}}
```

6. The same temp query with Dump clause:

```
MQL<7>temp query bus Assembly SA* 0 limit 5 select id locker dump tcl;
{Assembly} {SA-300.125} {0} {{20083.46775.31292.44899}} {{}}
{Assembly} {SA-300127} {0} {{20083.46775.20133.58276}} {{bucknam}}
{Assembly} {SA-300195} {0} {{20083.46775.20117.54372}} {{bucknam}}
{Assembly} {SA-300315} {0} {{20083.46775.20173.48444}} {{}}
{Assembly} {SA-300356} {0} {{20083.46775.20193.6352}} {{bucknam}}
```

7. A temp query selecting attributes and their values:

```
MQL<8>temp query bus * * * limit 1 select current revision
attribute.value tcl;
```

```
{businessobject} {NC Program} {GH02456} {A} {{current} {{current}
{Planned}}} {{revision} {{revision} {A}}} {{attribute.value}
{{attribute[MachineType].value} {Machine Center}} {{attribute[Process
Type].value} {Unknown}} {{attribute[Written By].value} {}}
{{attribute[File Suffix].value} {.TAP}}}
```

8. The same temp query with Dump clause:

```
MQL<9>temp query bus * * * limit 1 select current revision
attribute.value dump tcl;
{NC Program} {GH02456} {A} {{Planned}} {{A}} {{Machine Center}
{Unknown} {}{.TAP}}
```

9. An expand set with newlines for each object:

```
MQL<10>expand set "A1 - To Do" limit 5 select bus id tcl;
{1} {Documentation} {to} {Drawing} {SA-300356} {A} {{id} {{id}
{20083.46775.30402.28967}}}
{1} {Analysis} {to} {Cost Analysis} {DA-3001356-1} {A} {{id} {{id}
{20083.46775.65320.27011}}}
{1} {Analysis} {to} {Design Analysis} {DA-3001356-1} {C} {{id} {{id}
{20083.46775.26552.19182}}}
{1} {Plan} {to} {Assembly Process Plan} {SA-300356} {0} {{id} {{id}
{20083.46775.62627.22040}}}
{1} {Required Tools} {to} {Setup Instructions} {SA-300356} {A} {{id}
{{id} {20083.46775.17589.27146}}}
```

10. The same expand set with Dump clause:

```
MQL<11>expand set "A1 - To Do" limit 5 select bus id dump : tcl;
{1} {Documentation} {to} {Drawing} {SA-300356} {A}
{{20083.46775.30402.28967}}
{1} {Analysis} {to} {Cost Analysis} {DA-3001356-1} {A}
{{20083.46775.65320.27011}}
{1} {Analysis} {to} {Design Analysis} {DA-3001356-1} {C}
{{20083.46775.26552.19182}}
{1} {Plan} {to} {Assembly Process Plan} {SA-300356} {0}
{{20083.46775.62627.22040}}
{1} {Required Tools} {to} {Setup Instructions} {SA-300356} {A}
{{20083.46775.17589.27146}}
```

11. An expand bus with Dump clause but no RECORDSEP:

```
MQL<12>expand bus Vehicle M60000 0 from recurse to 1 select rel id dump
tcl;
1tclDOCUMENTATIONtcltotclManualtclM60001tcl0tcl974.54590.24670.436
1tclDesigned Part QuantitytcltotclFront
AxletclM66000tcl0tcl974.54590.42602.58431
```

12. A typical Tcl program:

```
set output [mql expand set "A1 - To Do" select bus id dump : tcl]
set count [llength $output]
foreach row $output {
  set level [lindex $row 0]
  set relation [lindex $row 1]
  set tofrom [lindex $row 2]
  set busobj [lrange $row 3 5]
  set busid [join [lindex $row 6]]
  puts "\[level: $level\] relationship\[$relation\] $tofrom $busid"
}
```

With results similar to:

```
[level: 1] relationship[Documentation] to 20083.46775.30402.28967
[level: 1] relationship[Analysis] to 20083.46775.65320.27011
[level: 1] relationship[Analysis] to 20083.46775.26552.19182
[level: 1] relationship[Plan] to 20083.46775.62627.22040
[level: 1] relationship[Required Tools] to 20083.46775.17589.27146
[level: 1] relationship[BOM-As Designed] to 20083.46775.30337.38798
[level: 1] relationship[BOM-As Designed] to 20083.46775.29481.42882
[level: 1] relationship[Product Spec to BOM] from
20083.46775.32142.28396
:
```

Visible Clause

This clause specifies other existing users who can read the workspace item with MQL list, print, and evaluate commands. The MQL copy command can be used to copy any visible item to your own workspace.

The syntax is:

```
visible USER_NAME{,USER_NAME};
```

Separate users with a comma, but no space.

Where Clause

This clause searches for selectable business object properties. This involves examining each object for the presence of the property and checking to see if it meets the search criteria. If it does, the object is included in the query results; otherwise, it is ignored.

While the Where clause can test for equality, it can test for many other characteristics as well. The syntax for the Where clause details the different ways that you can specify the search criteria, using familiar forms of expression:

```
where "QUERY_EXPR"
Or
where 'QUERY_EXPR'
```

- `QUERY_EXPR` is the search criteria to be used.

In general, the syntax for a query expression is in one of the following forms:

`(QUERY_EXPR)`
`ARITHM_EXPR RELATIONAL_OP QUERY_EXPR`
`BOOLEAN_EXPR`
`If-Then-Else`
`Substring`

Each form is described further in the sections that follow. Refer to *Usage Notes* for quereies for advice on structuring queries.

Query Expression (QUERY_EXPR)

This form simply means that a query expression can be contained within parentheses. Although not required, parentheses can add to readability and are useful when writing complex expressions. Parentheses can be used to group subclauses of the `where` clause, but not as start/end delimiters.

Comparative Expressions

This form of a Boolean expression considers comparisons such as greater than, less than, and equality. You have two arithmetic expressions and a relational operator:

```
ARITHM_EXPR RELATIONAL_OP ARITHM_EXPR
```

`ARITHM_EXPR` is an arithmetic expression that yields a single value. This value may be numeric, a character string, a date, or a Boolean. Arithmetic expressions are further described below.

`RELATIONAL_OP` is a relational operator. Relational operators and the comparison they perform are summarized in the table Relational Operators (RELATIONAL_OP) below.

When writing comparative expressions, you can have any mixture of arithmetic expressions. Since all arithmetic expressions yield a single value, they are interchangeable as long as they are of the same type. You cannot mix different value types in the same comparison expression. If you try to mix types, an error message will result.

You cannot compare values of different types because some operators do not work with some values. For example, testing for an uppercase or lowercase match does not make sense if you are working with numeric values. Even when you have values of the same type, you must be sure to use a relational operator that is appropriate for the values being compared. If the operator is incorrect for the values being compared, an error message will result.

Arithmetic Expressions (ARITHM_EXPR)

Use the following syntax:

```
ARITHM_EXPR BINARY_ARITHMETIC_OP ARITHM_EXPR
```

ARITHM_EXPR can be a selectable field name that yields a numeric value, an arithmetic operand (value), or another arithmetic expression. While arithmetic expressions include all data types, arithmetic expressions apply only to Integer or Real value types.

BINARY_ARITHMETIC_OP is one of four arithmetic operators:

- Plus sign (+) for addition
- Minus sign (-) for subtraction
- Asterisk (*) for multiplication
- Slash (/) for division

Arithmetic expressions can be written three ways:

FIELD_NAME	Uses the value contained within the named field for each object. This field name and its notation can be found by using the Print Businessobject Selectable command (see *Select Clause*.)
VALUE	Uses the value that you provide.
ARITH_EXPR	Performs one or more arithmetic operations to arrive at a single numeric value.

These forms allow you to write comparative expressions such as:

`attribute[Units] eq Inches`	Compares the values of the Units attribute to see if it is equal to Inches. If it is, the object is included with the query output.
`"attribute[Product Cost]" > "attribute[Maximum Cost]"`	Compares the value of the Product Cost attribute with the value of the Maximum Cost attribute. If the Product Cost exceeds the Maximum Cost, the object will be included in the query output.
`("attribute[Parts In Stock]" - 10) < ("attribute[Parts Needed]" + 5)`	Evaluates the results of each arithmetic expression and checks to see if the first result is less than the second. If it is, the object is included in the query output.

In the last comparative expression, all three forms of an arithmetic expression are used. This expression compares the results of two arithmetic expressions. One value in each arithmetic expression is a field name (Parts In Stock or Parts Needed) and one is a value supplied by you (10 or 5). Before the comparison can take place, each arithmetic expression must be evaluated and reduced to a single value. Then the two values can be compared.

*Be sure to include a space on either side of each arithmetic operator (+, -, *, /) to correctly separate it from the operands.*

Relational Operators (RELATIONAL_OP)

Relational operators can be used to compare values of all data types unless specified otherwise.

Operator	Operator Name	Function

`==`
`eq`
`EQ`
	is equal to	The first value must be equal to the second value. When comparing characters, uppercase and lowercase are not equivalent.

`!=`
`neq`
`NEQ`
	is not equal to	The first value must not match the second value. When comparing characters, uppercase and lowercase are not equivalent.

`<`
`lt`
`LT`
	is less than	The first value must be less than the second value. This comparison is not normally used with Boolean data types. When comparing dates, an older date has a lesser value.

`>`
`gt`
`GT`
	is greater than	The first value must be greater than the second value. This comparison is not normally used with Boolean data types. When comparing dates, the more recent date has the greater value.

`<=`
`le`
`LE`
	is less than or equal to	The first value must be equal to or less than the second value. This operator is not used with Boolean data types.

`>=`
`ge`
`GE`
	is greater than or equal to	The first value must be greater than or equal to the second value. This operator is not used with Boolean data types

`~~`
`smatch`
`SMATCH`
	string match	The general pattern of the first value must match the general pattern of the second value. The value can be included anywhere in the string. With this operator, character case is ignored so that “redone” is considered a match for “RED*.”

`!~~`
`nsmatch`
`NSMATCH`
	not string match	The general pattern of the first value must not match the general pattern of the second value. The value can be included anywhere in the string. With this operator, character case is ignored. For example, a first value of “Red Robbin” and a second value of “rE* rO*” would result in a FALSE comparison since the two are considered a match regardless of the difference in uppercase and lowercase characters.

By default, Live Collaboration *is case sensitive, but this can be disabled by System administrators. When case sensitivity is turned off, the following 4 case sensitive operators behave identically to their string match counterparts.*		

`~=`
`match`
`MATCH`
	case sensitive match	The pattern of the first value must match the pattern of the second value. The value can be included anywhere in the string. This includes testing for uppercase and lowercase characters. For example, “Red Robbin” is not a sensitive match for the pattern value “re* ro*” because the uppercase “R” values will not match the pattern’s lowercase specification.

`!~=`
`nmatch`
`NMATCH`
	case sensitive not match	The pattern of the first value must not match the pattern of the second value. The value can be included anywhere in the string. For example, if the first value is “Red*” a second value of “red” would produce a true result because the lowercase “r” is not an exact match to the first value’s uppercase “R.”

Operator	Operator Name	Function
The following operators are used for searches on description or string attribute fields that contain more than 254 bytes of data. They do not have a symbol to represent them, and can be used only by typing in the Where clause dialog. Refer to Searching Based on Lengthy String Fields for more information.		
`matchlong`	case sensitive match for long data	Contains the specified value, in either the lxStringTables or the lxDescriptionTables. The search is case sensitive. To find all objects with the word “language” in the attribute Comments, and to ensure that both tables are checked, use `attribute [Comments] matchlong "language"` in the Where clause. Since the query is case sensitive, the query will not find objects with “Language” or “LANGUAGE” in the Comments field.
`nmatchlong`	case sensitive not match for long data	Does not contain the specified value, in either the lxStringTables or the lxDescriptionTables. The “n” is for not match. The query is case sensitive. To find all objects that do not contain the word “Language” in the attribute Comments, use `attribute [Comments] nmatchlong "language"` in the Where clause. Since the query is case sensitive, it will find objects with “Language” or “LANGUAGE” in the Comments field.
`smatchlong`	string match for long data	Contains the specified value, in either the lxStringTables or the lxDescriptionTables. The search is NOT case sensitive. To find all objects with the word “Language” in the attribute Comments, and to ensure that both tables are checked, use `attribute [Comments] smatchlong "language"` in the Where clause. Since the query is not case sensitive, the query will find objects with “language”, “Language” or “LANGUAGE” in the Comments field.
`nsmatchlong`	not string match for long data	Does not contain the specified value, in either the lxStringTables or the lxDescriptionTables. The search is NOT case sensitive. The “n” is for not match. To find all objects that do not contain the word “language” in the attribute Comments, enter “language” for the value. Because the query is not case sensitive, it would not find objects with the word written as “LANGUAGE” or “Language” in the Comments attribute, as well as those that did not contain the word at all.

To find objects where a particular property is non-blank, use a double asterisk. For example, if you want be sure that all objects in the result of your query contain a description, use:

```
where '(description=="**")';
```

Matchlist Expressions

Expressions can use two keywords, `matchlist` and `smatchlist`, which enable you to specify a list of strings on the right-side of these keywords. These work the same as the `match` and `smatch` keywords except that an additional operand is used as a separator character for the list of strings. The format is:

```
matchlist 'STRING_LIST' ['SEPARATOR_CHAR']
smatchlist  'STRING_LIST' ['SEPARATOR_CHAR']
```

where

STRING_LIST is the list of strings to be compared

SEPARATOR_CHAR defines the character that separates the list of strings. If no separator character is given, the first right-hand operand is treated exactly as match and smatch treat it, that is, as one string.

Following are some examples:

```
temp query bus Errata * * where "current matchlist 'Open,Test' ','";
temp query bus Errata * * where "attribute[Priority] matchlist '0,1,2'
','";
temp query bus Errata * * where "attribute[Notes] smatchlist
'*Yin*,*Williams*'','";
```

The last query will find all Errata in which the names “Yin” or “Williams” are included in the Notes section, and the search will be case-insensitive.

Note that in the above examples, a comma is specified as the separator character in the second right-hand operand. The first right-hand operand is treated as a list of strings delimited by this separator character.

In matchlist *and* smatchlist *expressions, the following cannot be used as a separator character:* * . \ *[that is: asterisk, period, or backslash]*

If either left or right operand is a Select clause, the result of evaluating the operand is a list of strings. In all other cases, the evaluation of the operand just gives a single string. match, smatch, matchlist, and smatchlist are evaluated by a pair-wise comparison of the two lists. If any comparison yields true, then the result is true. Otherwise, it is false.

Searching Based on Lengthy String Fields

The Oracle database stores most string attribute values in the lxStringTable for the object’s vault. However, lxStringTable cannot hold more than 254 bytes of data. When a string attribute’s value is larger than this limit, the data is stored in the descriptions table (lxDescriptionTable), and a pointer to this table is placed in the lxStringTable.

When performing an “includes” search (using match operators: match, match case, not match, not match case) on string attribute values, the Live Collaboration searches on both lxDescription and lxString tables only when the attribute involved is of type “multiline.” Also, if you use the equal operators (==, !=) and give a string of more than 254 bytes to be equal to, Live Collaboration checks the values in the lxDescriptionTable only.

To search on description or other string attribute values for given text, and to force the search of both tables, you can use the “long” match operators. These operators can be used in any expression (including the Where clause entry screen) but are not shown as options in the query dialog’s Where Pattern in either the desktop or the Web version of this Live Collaboration.

Alternatively, you can include the .value syntax in the Where clause, as shown below:

```
attribute[LongString].value ~~ “matchstring”
```

Boolean Expressions: BOOLEAN_EXPR

This form of query expression means that the query expression can be either a single arithmetic expression or a selectable business object property that yields an arithmetic expression. For example, assume you want to find a list of honor students. The criteria for honor roll may be described as:

```
where 'attribute[“Grade Point Average”]>=3.8'
```

When this clause is processed, Live Collaboration looks for all objects that have this attribute and includes the value of “Grade Point Average.” If the attribute is not found within the object definition or if the attribute value returns false when the Where clause is evaluated, the object is excluded from the query output. To include a selectable object property in a Where clause, you must use its proper name and notation.

You can obtain this by using the Print Businessobject Selectable command (see *Select Clause*.) This command prints a list of all field names that can be used and indicates how they must be written.

A Boolean expression contains one or more comparisons. These comparisons return a value of TRUE, FALSE or UNKNOWN. For example, UNKNOWN might be returned as a string by a string attribute. It could also be returned by a Program, since Programs can be invoked in Expressions and return a string.

Boolean expressions, whose values can be True, False, or Unknown, should not be confused with Boolean type attributes, whose values can be only True or False.

As with TRUE and FALSE, mixed case (Unknown) and lowercase (unknown) are allowed.

The syntax for a Boolean expression uses two basic forms:

```
QUERY_EXPR BINARY_BOOLEAN_OP QUERY_EXPR
```

```
UNARY_BOOLEAN_OP QUERY_EXPR
```

Note that when using boolean operands in expressions, you must specify ==TRUE or ==FALSE in the expression. The operand by itself will always evaluate to TRUE when used in conjunction with other SQL convertible query fields.

The first form of the Boolean expression assumes you have two Boolean values whose relationship you want to compare. While there are only three Boolean operators, there are multiple ways that these operators can be specified. Refer to the table below.

When specifying a Boolean value in this form, follow the same syntax rules as for specifying a query expression. You can create long lists of Boolean expressions. For example, you could write Where clauses such as:

```
where 'type==Student && attribute[”Grade Point Average”]==4.0'
```

```
where 'current==“Initial Testing” && attribute[compound]==steel &&
attribute[weight] < 2.5'
```

When MQL processes these clauses, it obtains the Boolean values for each field name and then evaluates the Boolean relationship. If the results of the evaluation are true, the object is included in the query output. If false, it is excluded.

Boolean “and” can be represented: `AND, and, &&`.

Boolean “or” can be represented: OR, or, `||`.

Boolean “unknown” can be represented: UNKNOWN, unknown.

The following table shows the results for Boolean relationships:

Boolean Relationship	Results
TRUE and TRUE	TRUE
TRUE and FALSE	FALSE
TRUE and UNKNOWN	UNKNOWN
FALSE and UNKNOWN	FALSE
TRUE or TRUE	TRUE
TRUE or FALSE	TRUE
TRUE or UNKNOWN	TRUE
FALSE or UNKNOWN	UNKNOWN

The second form of the Boolean expression assumes that you have a single Boolean value. This value (represented by `QUERY_EXP`) can be operated upon to yield yet another Boolean value. The unary Boolean operator is the NOT operator. This operator causes the Live Collaboration to use the opposite value of the current Boolean value. For example, with the expression `NOT`(True), the final value of the Boolean expression would be False. The unary operator has three different notations:

```
NOT
not
!
```

!UNKNOWN yields a result of UNKNOWN

The one you use is a matter of preference.

Complex Boolean Expression

With the addition of comparative expressions, query expressions can become very complicated. You can specify any amount of search criteria. Only if all the criteria is met will the object be included.

The more criteria you list, the more difficult it is to maintain readability. Remember that you can use parentheses to help readability.

When MQL encounters a complex query expression, it uses the following rules to evaluate it:

1. All arithmetic expressions are evaluated from left to right and innermost parentheses to outermost.
2. All comparative expressions are evaluated from left to right and innermost parentheses to outermost.
3. All AND operations are evaluated from left to right and innermost parentheses to outermost.
4. All OR operations are evaluated from left to right and innermost parentheses to outermost.

For example, you might use the following query to find all Drawings connected to Assembly types that have not been released and have connections from Markups.

```
print query relationship;
   query relationship
      business * * *
      vault *
      owner *
   where '(type==Drawing)
      && (relationship[Drawing].to.type==Assembly)
      && (current==Released)
      && (relationship[Markup].from.type==Markup)';
```

Note that the type can be specified as part of the business object or in a boolean expression.

If-Then-Else

If-then-else logic is available for Expressions. The syntax is:

```
if EXPRESSION1 then EXPRESSION2 else EXPRESSION3
```

The `EXPRESSION1` term must evaluate to TRUE, FALSE, or UNKNOWN.

If the `EXPRESSION1` *term evaluates to UNKNOWN, it is treated as TRUE.*

The if-then-else expression returns the result of evaluating `EXPRESSION2` or `EXPRESSION3` depending on whether or not `EXPRESSION1` is TRUE or FALSE.

Note that only one of `EXPRESSION2` or `EXPRESSION3` is evaluated. So if the expressions have side-effects (which can happen since expressions can run programs), these effects will not occur unless the expression is evaluated.

```
eval expr ' if (attribute[Actual Weight] > attribute[Target Weight])
  then ("OVER") else ("OK")' on bus 'Body Panel' 610210 0;
```

Substring

The substring operator works within an expression to provide the ability to get a part of a string; the syntax is:

```
substring FIRST_CHAR LAST_CHAR EXPRESSION
```

The substring operator works as follows:

- The FIRST_CHAR and LAST_CHAR terms must evaluate to numbers that are positive or negative, and whose absolute value is between 1 and the number of characters in the string returned by EXPRESSION.
- The numbers returned by these terms indicate a character in the string returned by EXPRESSION.
- The characters are counted so that ‘1’ refers to the first character. A negative number indicates the character found by counting in the reverse direction. So ‘-1” refers to the last character.
- The substring operator returns the part of the string returned by EXPRESSION consisting of the characters from the FIRST_CHAR character to the LAST_CHAR character, inclusive.
- If FIRST_CHAR evaluates to a character that is after the character indicated by LAST_CHAR, an empty string is returned.

To obtain the last 4 characters of a 10-character phone number, use:

```
eval expression 'substring -4 -1 attribute[Phone Number]' on bus Vendor 'XYZ Co.'
0;
```

Where Clause and Patterns Criteria on Selectables

You can include additional criteria (Where clause and name patterns) to refine the returned list of items when using the person.assignment, parent, and ancestor selectables. The syntax allowed in brackets follows the conventions set by similar functionality for the “relationship” keyword on business objects.

- If the string in brackets has a bar character (|), the text before the bar is treated as a pattern (string with wildcard and/or a comma-separated list) and the text after the bar is treated as an expression that acts as a Where-clause.
- If the string in brackets does not have a bar character, the entire string is treated as a pattern.

For example:

```
temp query bus Part * * where
'context.user.assignment[r*|isarole].name==organization'
```

This returns all Part objects that have their organization set to any roles assigned to the context user that begin with 'r'.

application Command

Description

An *application* is a collection of administrative objects (attributes, relationships and types) defined by the Business Administrator and assigned to a project. It acts as a central place where all application dependent associations to these other administration objects are defined.

For conceptual information on this command, see *Applications* in Chapter 7.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]application NAME {CLAUSE};
```

- NAME is the name you assign to the application. For information on name length, see *Administrative Object Names*.
- CLAUSEs provide additional information about the application.

Add Application

An application is created with the Add Application command:

```
add application NAME [ADD_ITEM {ADD_ITEM}];
```

ADD_ITEM provides additional information about the application:

description VALUE
version VALUE
[!|not]hidden
{memeber ADMIN access ACCESS_VALUE}
property NAME [to ADMINTYPE NAME] [value STRING]

Where ADMIN is an Add Application sub-clause and can be one of the following administration objects:

Attribute
Type
Relationship

Where ACCESS_VALUE is one of the following options:

```
public
protected
private
```

All these clauses are optional. You can define an application by simply assigning a name to it. If you do, ENOVIA Live Collaboration uses the default application icon. If you do not want the default values, add clauses as necessary.

Version Clause

This clause provides additional information about the application you are defining. Application developers can use this information to classify versions or for other purposes. For information on name length, see *Administrative Object Names*.

Member Clause

This clause lets you add members to the application. Application members can be Attributes, Relationships or Types that are already defined within the database. To each application member you can also assign the following levels of protection:

- private
- protected
- public

The default level of protection is public.

In the following example, an application Catia is created. The application member of Catia is a type CTDrawing with the default protection level of protected.

```
add application Catia member type CTDrawing access
protected;
```

Any type that you assign to an application must have at least protected (read) access to its ancestors in the inheritance tree. A derived type does not inherit protection level from its parent. Therefore, it must be assigned a protection level that is needed to access the parent definition. For example, you can assign the child (derived type) a protection level of public when the protection level of the parent is protected. But you cannot do the same when the protection level of the parent is set to private because it would not allow the child to access the parent definition.

A user who does not have access to a protected type still has toconnect and todisconnect access on objects of this type. However, fromconnect and fromdisconnect access are not allowed.

Copy Application

After an application is defined, you can clone the definition with the Copy Application command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy application SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- SRC_NAME is the name of the application definition (source) to copied.
- DST_NAME is the name of the new definition (destination).
- MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Application

After an application is defined, you can change the definition with the Modify Application command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify application NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the application you want to modify.
- MOD_ITEM is the type of modification you want to make.
- You can make the following modifications. Each is specified in a Modify Application clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Application Clause	Specifies that…
name NEW_NAME	The current application name changes to that of the new name entered.
description VALUE	The description is changed to the VALUE given.
version value	The version is changed to the VALUE given.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
add property NAME [value STRING]	The named property is added.
add property NAME to ADMIN [value STRING]	The named property is added to the ADMIN type.
add member ADMIN access ACCESS_VALUE	Modify the application members (they can be Attributes, Types or Relationships) and/or modify the level of protection for the application members.
add property NAME [value STRING]	The named property is added.
remove property NAME [value STRING]	The named property is removed.
remove property NAME to ADMIN [value STRING]	The named property is removed from the ADMIN type.

Modify Application Clause	Specifies that…
`remove member ADMIN access ACCESS_VALUE`	Remove application members.
`property NAME [value STRING]`	The named property is modified.
`property NAME to ADMIN [value STRING]`	The named property assigned to the ADMIN type changes to the value entered.
`member ADMIN access ACCESS_VALUE`	The current ADMIN definition and the access value changes to the value entered.

As you can see, each modification clause is related to the arguments that define the application.

Delete Application

If an application is no longer required, you can delete it with the Delete Application command:

```
delete application NAME;
```

- `NAME` is the name of the app that you want to delete.
- Searches the list of applications. If the name is found, that application is deleted. If the name is not found, an error message results.

association Command

Description

Associations are a collection of groups, roles, or other associations.

For conceptual information on this command, see *Controlling Access* in Chapter 3.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]associaton NAME {CLAUSE};
```

- NAME is the name of the association you are creating. All associations must have a named assigned. When you create an association, you should assign a name that has meaning to both you and the user. This name cannot be shared with any other user types (groups, roles, persons, associations). For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the attribute.

Add Association

Associations are created and defined with the MQL Add Association command:

```
add association NAME [ADD_ITEM {ADD_ITEM}];
```

ADD_ITEM provides more information about the association you are creating. While none of the clauses are required to make the association usable, they are used to define the association's relationship to existing groups, roles, and associations, as well as provide useful information about the association.

The Add Association clauses are:

description VALUE
definition DEF_ITEM
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Where DEF_ITEM is:

USER_ITEM [{OPERATOR_ITEM USER_ITEM}]
DEF_ITEM must be enclosed within quotes.

Where USER_ITEM is:

[!]GROUP_NAME
[!]ROLE_NAME
[!]ASSOCIATION_NAME

Definition Clause

This clause defines which groups(s), role(s), and association(s) are included in the new association. The definition must be enclosed in quotes.

```
definition "USER_ITEM [{OPERATOR_ITEM USER_ITEM}]"
```

- USER_ITEM is the previously defined name of a group, role or association. An association definition can consist of all possible combinations of group(s), NOT-group(s), role(s), NOT-role(s), association(s), and NOT- association(s). Any USER_ITEM containing embedded spaces must be enclosed in quotes. Since the entire definition must also be enclosed in quotes, use single quotes for the USER_ITEM components and double quotes for the entire definition, or vice versa.
- OPERATOR_ITEM can be either **&&** (to represent AND) or || (to represent OR).
- Checks the association definition before accepting it as a valid definition. AND commands are evaluated before OR commands (that is, AND has a higher order of precedence). This order cannot be changed. The operator OR signifies the end of one expression and the beginning of another.

For example, the following definition:

```
Admin and Production or Manager and Services
```

is evaluated as:

Admin and Production	or	Manager and Services

To create a definition using the AND operator

The AND operator requires that the person be defined in each group, role, or association that you include in the AND definition. For example:

```
definition "Engineering && Management"
```

To satisfy this definition, the person must be defined in *both* the Engineering Group *and* in the Management Role.

To create a definition using the OR operator

The OR operator allows you to include persons defined in any of the groups, roles, or associations included in the OR definition. For example:

```
definition "Engineering || 'Senior Management'"
```

To satisfy this definition, the person must be defined in *either* the Engineering Group *or* in the Senior Management Role.

Notice that single quotes are used for the two word name "Senior Management," and that there is no space between the name and either single quote. A space between the second single quote and the double quote is optional.

To create a definition using Not Equal

To exclude certain role(s) or group(s) or association(s) from the definition, you can use the Not Equal operator in the definition. Place an exclamation point (!) in front of whichever group/role/association you want to exclude. Be sure there is no space between the exclamation point and the name of the group/role/association. For example:

```
definition "!Engineering && Management"
```

To satisfy this definition, the person must be defined in the Management role, but *not* in the Engineering Group.

To create a definition using multiple operators

You can use any combination of role(s) and group(s) and association(s) using AND and OR operators and Equal and Not Equal.

You can, for example, create a definition such as "A member of the Products Group AND a member of the Engineering Group OR a member of the Design Group OR a Vice President but NOT a member of the Marketing Group." For example:

```
definition "!Marketing && Products && Engineering || Design ||'Vice President'"
```

Copy Association

After an association is defined, you can clone the definition with the Copy Association command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy association SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the association definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Association

After an association is defined, you can change the definition with the Modify Association command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify association NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the association you want to modify.
- MOD_ITEM is the type of modification to make. Each is specified in a Modify Association clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Association Clause	Specifies that…
name NEW_NAME	The current association name is changed to the new name.
description VALUE	The current description, if any, is changed to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
definition DEF_ITEM	The definition is changed to the new definition specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the association.

For example, assume you want to alter the definition for the Drivers First Shift association to reflect the addition of a new group. You might write this command:

```
modify association "Drivers First Shift"
   definition "RR#6 && 'West Side' && RR#15";
```

This command redefines the groups that are included in the association.

When modifying an association:

- The roles, groups, or associations that you assign to the association must already be defined within the database. If they are not, an error will display when you try to assign them.
- Remember that altering the group or role access affects all persons included in the association that contains those groups or roles. If it is a singular case of special access, you may want to assign that person to the policy directly or define a role that is exclusively used by the person in question.

Delete Association

- If an association definition is no longer required, you can delete it with the Delete Association command:

```
delete association NAME;
```

- NAME is the name of the association to be deleted.
- After this command is processed, the Live Collaboration searches the list of associations. If the name is not found, an error message is displayed. If the name is found, the association is deleted and any linkages to that association are dissolved.

Example

For example, to delete the association named “Tree Specialists,” enter the following MQL command:

```
delete association “Tree Specialists”;
```

After this command is processed, the association is deleted and you receive an MQL prompt for another command.

attribute Command

Description

An *attribute* is any characteristic that can be assigned to an object or relationship. Objects can have attributes such as size, shape, weight, color, materials, age, texture, and so on.

For conceptual information on this command, see *Attributes* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|convert|copy|check|delete|list|modify|print]attribute NAME
{CLAUSE};
```

- NAME is the name you assign to the attribute. Attribute names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the attribute.

Add Attribute

Before types and relationships can be created, the attributes they contain must be defined as follows:

```
add attribute NAME type TYPE [ADD_ITEM {ADD_ITEM}];
```

Specifying the attribute type is required, including one of the following values in brackets:

```
type {binary|date|integer|string|real|boolean}
```

ADD_ITEM provides additional information about the attribute you are adding. The Add Attribute clauses are:

default VALUE
description VALUE
rule NAME
maxlength VALUE
owner ADMIN
range RANGE_ITEM {range RANGE_ITEM}
TRIGGER PROG_NAME [input ARG_STRING]
dimension DIMENSION_NAME

format FORMAT
keepformat
[!|not]multivalue
[![not]rangevalue
[!|not]multiline
[!|not]hidden
[!|not]resetonclone
[!|not]resetonrevision
property NAME [value STRING]
property NAME to ADMIN [value STRING]

Type Clause

The Type clause is always required. It identifies the type of values the attribute will have. An attribute can assume several different types of values. When determining the attribute type, you can narrow the choices by deciding if the value is a number or a character string.

Once a type is assigned to an attribute, it cannot be changed. The user can associate only values of that type with the attribute.

Type	Description
`binary`	Binary attributes must be passed in as a string. Binary attribute values are stored in the LXBINARY table. The following rules apply to binary types of attributes. • Binary attributes support up to 4000 bytes of data. If multi-byte values are passed in as a string, the 4000 bytes will not be consistent. In such cases, check in the data as a file. • You can change any existing string attributes to binary. All rows from the IxString table for the attribute type are moved to the LXBINARY table. • Dimensions are not supported for binary attributes. • Multival attributes cannot be binary. • Binary attributes cannot be part of an index or a uniquekey. • Interface types are supported on binary attributes. • You can define binary attributes as being owned by a type. • The transition commands are not supported when used in the context of binary attributes.
`String`	One or more characters. These characters can be numbers, letters, or any special symbols (such as $ % ^ * &). Although numbers can be part of a character string, you cannot perform arithmetic operations on them. To perform arithmetic operations, you need a numeric type (Integer or Real). String attributes (as well as description fields) have a limit of 2,048 KB. If you expect to handle more data in an attribute, consider re-designing the schema to store the data in a checked-in file instead.
`Boolean`	A value of `TRUE` or `FALSE`.
`Real`	A number expressed with a decimal point (for example, 5.4321). The range of values it can assume depends on the local system architecture. Both 32-bit and 64-bit floating point numbers are supported. Since real values are stored in the name format of the local system architecture, the precision and range of values varies from architecture to architecture. To obtain the exact range for your system, see your system manual.
`Integer`	A whole number whose range is defined by the local architecture. All CPU architectures with the exception of signed 8-bit integers are supported. Depending on the system architecture, an attribute with the integer type can assume a value within the following ranges: unsigned integer — 8 bits — 0 to 255 signed integer — 16 bits — -32,768 to 32,767 unsigned integer — 16 bits — 0 to 65,535 signed integer — 32 bits — -2,147,483,647 to 2,147,483,647 unsigned integer — 32 bits — 0 to 4,294,967,295

Type	Description

`Date and Time`	Any collection of numbers or reserved words that can be translated into an expression of time. Times and dates can be expressed using the formats listed below. This includes the year, month, day, hour, minute, and/or second. Abbreviations or full-words are acceptable for the day of the week and month.
	In the following example, the day of the week is optional.
	`Wed Feb 15, 1999`
	Another way to enter this date is:
	`2/15/99`
	The time of day. In the following example, the meridian and time zone information are optional:
	`01:30:00 PM EST`
	Another example is:
	`13:30:00`
	When you enter both the date and time, the time should follow the date. For example:
	`February 1, 1999 12:52:30 GMT`
	The actual date/time is calculated based on the current time and date obtained from your system clock. The date range is January 1, 1902 to December 31, 2037.
	See the *Administration Guide* : *Configuring Date and Time Formats* for details.
	For any attribute with a date format, even if you have a date setting in your initialization file (enovia.ini) and you input any date without a year, the date is accepted and converted to Sat Jan 01, 0000, 12:00:00 AM. Dates should be input as the full date, including the year.

About Floating Point Precision/Output

For a floating point number F, the number of digits of accuracy maintained, and the number of digits printed depends on the magnitude of the absolute value of F.

Absolute Value	Accuracy	Output Possibilities
< 1.0	13 digits exact	“0.” + leading zeros
	14th digit rounded	+ maximum of 14 nonzero digits
>= 1.0	14 digits exact	no leading zeros
	15th digit rounded	+ maximum of 15 nonzero digits + 0's up to 15th digit + non-significant digits +".0"

In the case where ABS(F) < 1.0, there are never any digits printed beyond the maximum of 14.

But for large numbers, it may be necessary to pad if the number of digits before the decimal point exceeds 15. These non-significant digits are subject to precision inaccuracies of the operating system, and may include random nonzero digits, as is demonstrated in the examples below by A digitsBig3 0.

Examples:

```
add bus A digitsBig1 0 policy A vault Standards;
add bus A digitsBig2 0 policy A vault Standards;
add bus A digitsBig3 0 policy A vault Standards;
add bus A digitsBig4 0 policy A vault Standards;
add bus A digitsSmall1 0 policy A vault Standards;
add bus A digitsSmall2 0 policy A vault Standards;
add bus A digitsSmall3 0 policy A vault Standards;
add bus A digitsSmall4 0 policy A vault Standards;
```

```
mod bus A digitsBig1 0 r1 1.234567890123459999;
mod bus A digitsBig2 0 r1 123456789.0123459999;
mod bus A digitsBig3 0 r1 1234567890123459999.0;
mod bus A digitsBig4 0 r1 12345678901234599990000000000.0;
temp query bus A digitsBig* * select attribute[r1] dump ' ';
A digitsBig1 0 1.23456789012346
A digitsBig2 0 123456789.012346
A digitsBig3 0 1234567890123460100.0
A digitsBig4 0 12345678901234600000000000000.0

mod bus A digitsSmall1 0 r1 0.1234567890123459999;
mod bus A digitsSmall2 0 r1 0.000123456789012345999 9;
mod bus A digitsSmall3 0 r1 0.00000000000001234567890123459999;
mod bus A digitsSmall4 0 r1
0.000000000000000000000001234567890123459999;
temp query bus A digitsSmall* * select attribute[r1] dump ' ';
A digitsSmall1 0 0.12345678901235
A digitsSmall2 0 0.00012345678901235
A digitsSmall3 0 0.00000000000012345678901235
A digitsSmall4 0 0.00000000000000000000000012345678901235
```

Default Clause

The Default clause defines a default value for the attribute.

When a business object is created and the user does not fill in the attribute field, the default value is assigned. When assigning a default value, the value you give must agree with the attribute type. If the attribute should contain an integer value, you should not assign a string value as the default.

For example, assume you want to define an attribute called PAPER_LENGTH. Since this attribute will specify the size of a sheet of paper, you defined the type as an integer. For the default value, you might specify 11 inches or 14 inches, depending on whether standard or legal size paper is more commonly used.

As another example, assume you are defining an attribute called Label Color. If the most common color is yellow, you might define the attribute with this command:

```
add attribute "Label Color"
   default "yellow", type string;
```

If the user does not assign a value for the Label Color, “yellow” is assigned by default.

Reset on Clone or Revision Flag

You can determine whether an attribute value should be reset to its default value when a business object or connection is cloned or revised. An attribute’s default value is defined using the Default clause. Set `resetonclone` to True to reset the attribute value to its default value in new business object and connection clones. Set `resetonminorrevision` to True to reset the attribute value to its default value in new business object and connection revisions. Set either flag to False to retain the current attribute value in new business object and connection clones and revisions. By default, the current attribute value is retained.

The float and replicate options for cloning and revising connections will still reset the attribute value to its default value when the appropriate flag is set to True.

Maximum Length Clause

Attributes defined as type String can also contain a maximum length clause. This clauses allows you to define the maximum number of characters the attribute's value can contain. This clause is very useful in enabling greater interoperability with other systems where attribute length is also constrained. After the maximum length is defined, an error will occur if you attempt to create or modify an attribute with a greater number of characters.

If the maximum length definition of an attribute is 0, existing attributes can have values of unlimited length. If the maximum length of a string is modified for an attribute, and the attribute value is greater than or equal to 1020 bytes and violate the maximum length, the modifying of maximum length will produce an error with the offending value's object ids. Otherwise, modifying an attribute's maximum length allows existing values to be retained.

By default maximum length is defined as zero (0) where zero means an unlimited number of characters. The number you define for the maximum length does not take into account the different character encoding types (UTF8, UTF16, etc.).

*SQL Server restricts the size of the index to 900 bytes. For example, if you try to create two string attributes with length 255, since they are nvarchar columns, the size of the index for string attributes is calculated as 255*2 (for Unicode), which would go over the limit. You can use the* `maxlength` *clause with the Add Attribute command to restrict the size so that indexing is right.*

Dimension Clause

A dimension is an administrative object that provides the ability to associate units of measure with an attribute, and then convert displayed values among any of the units defined for that dimension. The Dimension clause of the Add Attribute command enables you to specify a dimension to use for real or integer type attributes.

```
add attribute NAME dimension DIMENSION_NAME;
```

The Dimension clause can be used only with attributes of type real or integer.

Rule Clause

Rules are administrative objects that define specific privileges for various users. The Rule clause enables you to specify an access rule to be used for the attribute.

```
add attribute NAME rule RULENAME;
```

Range Clause

The Range clause defines the range of values the attribute can assume. This range provides a level of error detection and gives the user a way of searching a list of attribute values. If you define an attribute as having a specific range, any value the user tries to assign to that attribute is checked to determine if it is within that range. Only values within the defined range are allowed.

When writing a Range clause, use one of the following forms depending on the types and amount of range values:

```
range RELATONAL_OPERATOR VALUE
range PATTERN_OPERATOR PATTERN
range between VALUE {inclusive|exclusive} VALUE {inclusive|exclusive}
range program PROG_NAME [input ARG_STRING]
```

If you have an attribute with a default value that is outside the ranges defined (such as "Undefined") and you try to create a business object without changing the attribute value, the default value against the ranges is not checked. A check trigger could be written on the attribute to force a user to enter a value.

Each form is described below. The method and clauses you use to define a range are a matter of preference. Select the clauses that make the most sense to you and then test the range to be sure it includes only valid values.

Range Compared with a Relational Operator

This form offers greater flexibility in defining the range of possible values:

```
range RELATIONAL_OPERATOR VALUE
```

A relational operator compares the user's value to a set of possible values. Choose from these relational operators:

Operator	Operator Name	Function
=	is equal to	The user value must equal this value or another specified valid value (given in another Range clause). When comparing user-supplied character values to the range value, uppercase and lowercase letters are equivalent.
!=	is not equal to	The user-supplied value must not match the value given in this clause. If it matches, the user is notified that the value is invalid. When comparing user-supplied character values to the range value, uppercase and lowercase letters are equivalent.
<	is less than	The user value must be less than the given range value. If the user value is equal to or greater than the range value, it is not allowed.
<=	is less than or equal to	The user value must be less than or equal to the given range value. If the user value is greater than the range value, it is not allowed.
>	is greater than	The user value must be greater than the given range value. If the user value is equal to or less than the range value, it is not allowed.
>=	is greater than or equal to	The user value must be greater than or equal to the given range value. If the user value is less than the range value, it is not allowed.

Depending on the relational operator you use, you can define a range set that is very large, or a set that contains a single value. When you use a relational operator, the value provided by the user is compared with the range defining value. If the comparison is true, the value is allowed and is assigned to the attribute. If the comparison is not true, the value is considered invalid and is not allowed.

For example, assume you want to restrict the user to entering only positive numbers. In this case, you could define the range using either of the following clauses:

```
range > -1
Or:
range >= 0
```

If the user enters a negative number (such as -1), these commands are false (-1 is not greater than -1 and is not greater than or equal to zero). Therefore the value is invalid.

You may have an attribute with a few commonly entered values but that can actually be any value. To provide the user with the ability to select the commonly entered values from a menu, but also allow entry of any value, you would:

- Add the ranges of **Equal** values for the common values.
- Add a range of **Not Equal** to `xxxxx`.

This will allow any value (except `xxxxx`) and also provide a list from which to choose the common values.

When defining ranges for character strings, remember that you can also perform comparisons on them. By using the ASCII values for the characters, you can determine whether a character string has a higher or lower value than another character string. For example “Boy” is less than “boy” because uppercase letters are less than lowercase letters and “5boys” is less than “Boy” because numbers are less than uppercase letters. For more information on the ASCII values, refer to an ASCII table.

But what would you do if the attribute value had a second part that was to start with the letters REV? The next form of the Range clause is available for this reason.

Range Compared with a Pattern (Special Character String)

This form, used exclusively for character strings, uses a special character string called a *pattern*.

```
range PATTERN_OPERATOR PATTERN
```

Patterns are powerful tools for defining ranges because they can include *wildcard* characters. Wildcard characters can be used to represent a single digit or a group of characters. This allows you to define large ranges of valid values.

For example, you can define an attribute’s range as “DR* REV*” where the asterisk (*) is a wildcard representing *any* character(s). This range allows the user to enter any value, as long as the first half begins with the letters “DR” and the second half begins with the letters “REV”.

When using a pattern to define a range, you must use a pattern operator. These operators compare the user's value with the pattern range. All the pattern operators allow for wildcard comparisons. Two of them check the user's entry for an exact match (including checks for uppercase and lowercase).

Operator	Operator Name	Function
`match or ~=`	match	The user's character string must match the exact pattern value given, including uppercase and lowercase letters. For example, "Red Robin" is not a sensitive match for the pattern value "re*ro*" since the uppercase R's do not match the pattern.
`!match or !~=`	not match	The user's character string must not match the exact pattern value. For example, if the user enters "Red" when the pattern value is "red", the value is allowed; "Red" is not an exact match to "red".
`smatch or ~~`	string match	The user's character string must match only the general pattern value, independent of case. Case is ignored so that "RED" is considered a match for "red".
`!smatch or !~~`	not string match	The user's character string must not match the general pattern value, independent of case. For example, assume the range pattern is defined as "re*ro*". The value "Red Robin" is not allowed, although "red ribbon" is allowed. That is because the first value is a pattern match (regardless of case difference) and the second is not.

When the user enters a character string value, these operators compare that value to the defined range pattern. If the comparison results in a true value, the user value is considered valid and is assigned to the attribute. If the comparison is false, the user value is considered out of range and is not valid.

Multiple Ranges

When defining the range, remember that you can use more than one Range clause. More than one clause may be required.

```
range between VALUE {inclusive|exclusive} VALUE {inclusive|exclusive}
```

For example, if you have a situation where the attribute value can be any uppercase letter except for I or O (since these can be confused with numbers), you must define multiple ranges to exclude those two letters:

```
range < I
range between J inclusive N inclusive
range > O
```

Ranges Defined Within a Program

Allows you to use a program to define range values. This allows the flexibility to change range values depending on conditions. Use the range program clause to specify the name of the program to execute. For example:

```
range program SetRanges;
```

In this example, the name of the program to execute is SetRanges.

You can define arguments to be passed into the program. Your program could change the attribute range depending on the argument passed. For example:

```
range program SetRanges attrang1;
```

When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2. . . etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”, . . . etc.

Programs should return the list of choices in the variable that has the same name as the program name, which can be obtained through argument “0.” Be sure to use the `global` keyword on your `set env` command when passing the list back.

See *Runtime Program Environment* for additional information on program environment variables.

The following is output from a MQL session. The attribute type ‘designerName’ produces the choices ‘Tom,’ ‘Dick,’ ‘Harry,’ ‘Larry,’ ‘Curly,’ and ‘Moe.’ Note that an attribute type can have any number of ranges, but only one range can be of type program.

```
MQL<1>print attr designerName;
attribute designerName
 type string
 description
 default Tom
 range = Tom
 range = Dick
 range = Harry
 range uses program nameRange
 not multiline
MQL<2>print prog nameRange;
program nameRange
 mql
 description
 code 'tcl;
eval {
 # set the event
 set event [mql get env EVENT]
 # set the choices
 set names {Larry Curly Moe}
 # set the output variable (arg 0 is this program's name)
 set output [mql get env 0]
# test event, and either generate choices, or test value
 if { $event == "attribute choices"} {
    # note that choices are returned in a global RPE variable
    mql set env global $output $names
 } else {
    # assume that it is safe to unset global RPE variable during value
test
    mql unset env global $output
    # set the value
    set value [mql get env ATTRVALUE]
```

```
    # test the value
   if {[lsearch -exact $names $value] == -1} {
     # value not in list, return non-zero
     exit 1
   } else {
     # value in list, return zero
     exit 0
   }
 }
}
```

The following macros are available to Range Programs:

- **Owning Item information macros**. There are 3 scenarios:
  - If the attribute is “owned” by a business object, the macros available are OBJECT, TYPE, NAME, and REVISION.
  - If a relationship instance “owns” the attribute, the RELID macro is provided.
  - During query formulation the owner is unknown so no owner macros are available.
- **Invocation Information.** INVOCATION will always equal “range”.
- **Event information**. EVENT and possibly ATTRVALUE. For example:
  - When the range program is being asked to produce all legal values, EVENT will equal “attribute choices”.
  - When the range program is being asked to check the legality of a given value, EVENT will equal “attribute check” and the ATTRVALUE macro will also be provided.
- **Attribute information**. ATTRNAME and ATTRTYPE. For example:

  ATTRNAME=designerName

  ATTRTYPE=String
- **Basic information**. VAULT, USER, TRANSACTION, HOST, APPLICATION, and LANGUAGE.

For additional information about macros, see the *Configuration Guide : Appendix Macros* in the online documentation.

Trigger Clause

Event Triggers provide a way to customize Live Collaboration behavior through Program Objects. Triggers can contain up to three Programs, (a check, an override, and an action program) which can all work together, or each work alone. Attributes support the use of triggers on the modify event. For more information on Event Triggers, refer to the *Configuration Guide : Trigger Event Macros* in the online documentation. The syntax for the trigger clause is:.

```
trigger modify {action|check|override} PROG_NAME [input ARG_STRING];
```

For example, to assign a check trigger called OnApprove, you would use:

```
trigger modify check OnApprove;
```

In this example, the name of the program to execute is OnApprove.

You can define arguments to be passed into the program. For example:

```
trigger modify check OnApprove input ChngChk;
```

In this example, the argument passed into the OnApprove program is ChngChk.

When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,… etc.

Multiline Clause

If you define the attribute type value as “string,” you can specify the format of the data entry field. Use the `multiline` clause if you want the data entry field to consist of multiple lines. If this clause is specified, the text wraps to the next line as the user types. The text box is scrollable.

If `multiline` is not specified or if `!multiline` (notmultiline) is specified, then the data entry field consists of a single line. If the amount of text exceeds the size of the field shown, the line of text scrolls to the left as the user is typing.

History Clause

The `history` keyword adds a history record marked “custom” to the attribute that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Attribute

After an attribute is defined, you can clone the definition with the Copy Attribute command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy attribute SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the attribute definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the attribute that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Attribute

After an attribute is defined, you can change the definition with the Modify Attribute command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify attribute NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the attribute you want to modify.
- `MOD_ITEM` is the type of modification you want to make.

- You cannot alter the Type clause. If you must change the attribute type, delete the attribute and add it again using a new Type clause.

Modify Attribute Clauses

You can make the following modifications to an existing attribute definition. Each is specified in a Modify Attribute clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Attribute Clause	Description
`name NAME`	The current attribute name changes to the new name entered.
`default VALUE`	The current default value, if any, is set to the value entered.
`description VALUE`	The current description value, if any, is set to the value entered.
`dimension DIMENSION_NAME`	Adds an existing dimension DIMENSION_NAME to an integer or real attribute. *When adding dimensions to attributes that already belong to business object instantiations, refer to Applying a Dimension to an Existing Attribute for information on converting the existing values to the required normalized value.*
`keepformat`	Keeps the current attribute format.
`format FORMAT`	Changes the attribute format to the specified FORMAT.
`icon FILENAME`	The image is changed to the new image in the file specified.
`owner ADMIN`	Changes the attribute owner to the specified ADMIN object.
`[!|not]multivalue`	Changes the attribute to be `multivalue` or not. Only existing `singlevalue` attributes can be modified to become `multivalue`. If the attribute is already a `rangevalue`, an error will be thrown. See *Multi-Value Attributes* for details.
`[!|not]rangevalue`	Changes the attribute to specify a range values or not. See *Multi-Value Attributes* for details.
`[!|not]hidden`	Changes the option to specify that the object is hidden, or when using the prefix '`!`' or "`not`", the object is not hidden.

Modify Attribute Clause	Description
[!|not]resetonclone	Resets the attribute value to its default value in new business object and connection clones, or when using the prefix '!' or "not", the current attribute value is retained in new business object and connection clones.
[!|not]resetonminorrevision	Resets the attribute value to its default value in new business object and connection revisions, or when using the prefix '!' or "not", the current attribute value is retained in new business object and connection revisions.

Modify Attribute Clause	Description
`maxlength VALUE`	The maximum number of characters the attribute's value can contain. If you modify the maximum length definition for an attribute to a number less than the current number, existing attributes will be maintained with their longer values and new attributes will need to follow the new maximum length.

`add |dimension DIMENSION_NAME |range RANGE_ITEM |TRIGGER PROG_NAME |rule NAME |keepformat |format FORMAT |property NAME [value STRING] |property NAME to ADMIN [value STRING]`
 	
where `RANGE_ITEM` is one of:
`|| = | != | < | > | <= | >= | VALUE |`
`|| smatch|!smatch|match|!match | PATTERN |`
`| between VALUE|inclusive|VALUE|inclusive||`
`|exclusive| |exclusive||`
`| program PROG_NAME [input ARG_STRING] |`
 	
where `TRIGGER` is:
`trigger EVENT_TYPE | action |`
`| check |`
`| override |`
where `EVENT_TYPE` is: `| modify |`
where `ADMIN` is: `| TYPE_NAME |`
	Adds the named dimension to an integer or real attribute.
	Adds the specified Range clause to the existing list of Range clauses. This Range clause must obey the same construction and syntax rules as when defining an attribute.
	Chnages the trigger program to PROG_NAME. TRIGGER is the type of trigger program (action, check, or override) and PROG_NAME is the name of the program that replaces the current trigger program.
	"modify" is the only supported trigger event for attributes.
	Adds the named access rule.
	Adds the named property.
	Adds the named property to the specified administrative object.

`remove | dimension DIMENSION_NAME | range RANGE_ITEM | TRIGGER | rule NAME | keepformat | format | property NAME | property NAME to ADMIN |`	Removes the dimension DIMENSION_NAME from an integer or real attribute. Removes the specified Range clause from the existing list of Range clauses. The given Range clause must exist, or an error message is displayed. The specified trigger program is removed. PROG_TYPE is the type of trigger program (`check`, `override`, or `action`). Removes the named access rule. Removes the named property. Removes the named property from the specified administrative object.

Modify Attribute Clause	Description
`history STRING`	Adds a history record marked “custom” to the attribute that is being modified. The STRING argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

As you can see, each modification clause is related to the clauses and arguments that define the attribute. For example, you would use the Default clause of the Modify Attribute command to modify the Default clause that you used in the Add Attribute command.

Assume you have the following attribute definition:

```
attribute Units
   description "Measuring units"
   type string
   default "Linear Feet"
   range = Quarts
   range = "Linear Feet";
```

Now you have decided to use the attribute for measuring distances only. While it might be faster to define a new attribute, you could modify this attribute with the following command:

```
modify attribute Units
   description "Units for Measuring Distances"
   remove range = Quarts
   add range = "Linear Yards";
```

After this command is processed, the attribute definition for the Units attribute is:

```
attribute Units
   description "Units for Measuring Distances"
   type string
   default "Linear Feet"
   range = "Linear Feet"
   range = "Linear Yards";
```

Check Attribute

Any attribute can be checked to find out whether a specified value is within the given range for that attribute:

```
check attribute NAME VALUE [businessobject TYPE NAME REV] [connection RELID];
```

`NAME` is the name of the attribute.

`VALUE` is the value you want to check.

Because an attribute’s Range Program can produce different legal values depending on the state or settings of the owning business object or relationship instance, the business object TYPE NAME REV or RELID can be used to specify the owner of the attribute:

RELID is the id of the relationship instance that owns the attribute.

For example, to find out if “3” is a legal value for an attribute called “Priority,” you could issue the following command:

```
check attribute Priority 3;
```

If it is not a legal value, the following is returned:

```
'3' violates range for attribute 'Priority'
```

Convert Attribute

The convert command requires system administrator privileges.

```
convert attribute NAME to unit UNITNAME [commit N]
[output FILENAME] searchcriteria SEARCHCRITERIA
```

If the `searchcriteria` contains instances that have already been converted to:

- the unit specified in the search criteria, then no conversion is done.
- a different unit than specified in the search criteria then the conversion is done and the input unit is updated to `UNITNAME`

This command also makes an entry to the object’s history. Triggers will be disabled during the command execution.

The number `N` supplied with the `commit` modifier controls how many objects or connections are processed before committing the transaction.

The `output` modifier logs information to the specified `FILENAME`.

To convert stored values to the default units

1. Log into MQL as an administrative user.
2. Apply the dimension to the attribute:
3. Type `Y`.

Applying a Dimension to an Existing Attribute

You can add a dimension to attributes of type integer or real, whether or not the attributes already belong to instantiated business objects. The existing value stored for that attribute will be treated as the normalized value.

However, if the existing value is a unit other than what the dimension uses as its default, you need to convert those values. For example, an existing attribute Length has stored values where the business process required the length in inches. You want to add a Length dimension to the Length attribute, and that dimension uses centimeters as the default. If you just add the dimension to the attribute, the existing values will be considered as centimeters without any conversions.

As another example, the Length attribute can be used with multiple types. Some types may use the attribute to store lengths in millimeters, while others store lengths in meters. Because there is only one attribute Length that represents two kinds of data, the dimension cannot be applied without normalizing the business types on the same unit.

Use the convert attribute command to convert the attribute to 1 unit of measure.

```
mod attribute ATTRIBUTE_NAME dimension DIMENSION_NAME;
```

Replace `ATTRIBUTE_NAME` with the attribute name and `DIMENSION_NAME` with the dimension name.

Specify the existing units of the attribute

```
convert attribute ATTRIBUTE_NAME to unit UNIT_NAME on temp query bus
"TYPE" * *;
```

This command applies the unit `UNIT_NAME` to the attribute, which causes the conversion from the existing value with the applied units to normalized values.

The system displays this message:

```
Convert commands will perform mass edits of attribute values.
You should consult with MatrixOne support to ensure you follow
appropriate procedures for using this command.
Proceed with convert (Y/N)?
```

If the system message includes the following warning,

```
WARNING: The search criteria given in the convert command includes objects
that currently have unit data. This convert command will overwrite that
data.
```

the data for this attribute has already been converted and you should not continue (type N).

Changing the Default (Normalized) Units of a Dimension

The Weight attribute included in the Framework, is defined to have a dimension containing units of measure with the default units set to grams. If your company uses the Weight attribute representing a different unit, you have these options:

- Convert existing values to use the new dimension with its default as described in Applying a Dimension to an Existing Attribute above.
- Change the Dimension's default units, offsets, and multipliers as described in this section

---

After applying a dimension, you can only change the default units of measure before any user enters any data for the attribute. After values are stored in the asentered field, you cannot change the default units of measure.

To Change a Dimension's Default Units

1. Using Business Modeler, locate the dimension and open it for editing.
2. Click the **Units** tab.
3. For the unit you want to use as the default:
   - **a)** Highlight the Unit.
   - **b)** Change the multiplier to 1.
   - **c)** Change the offset to 0.
   - **d)** Check the **Default** check box.
   - **e)** Click **Edit** in the Units section.

4. For all other units:
   a) Highlight the Unit.
   b) Change the multiplier to the appropriate value.
   c) Change the offset to the appropriate value.
   d) Click **Edit** in the Units section.
5. Click **Edit** at the bottom of the dialog box.

Delete Attribute

If an attribute is no longer required, you can delete it by using the Delete Attribute command:

```
delete attribute NAME;
```

`NAME` is the name of the attribute to be deleted.

Searches the list of defined attributes. If the name is found, that attribute is deleted. If the name is not found, an error message is displayed.

For example, to delete the Shipping Address attribute and the Label attribute, enter the following two commands:

```
delete attribute "Shipping Address";
delete attribute Label;
```

After these commands are processed, the attributes are deleted and you receive an MQL prompt for another command.

businessobject Command

Description

Business objects form the body of Collaboration and Approvals. They contain much of the information an organization needs to control. Each object is derived from its previously-defined type and governed by its policy.

For conceptual information on this command, see *Manipulating Data* in Chapter 5.

User Level

Business Administrator

Syntax

```
[add|print|copy|revise|modify|expand|connect|disconnect|checkin|
checkout|lock|unlock|delete|purge] businessobject BO_NAME
policy POLICY_NAME {CLAUSE};
```

- BO_NAME is the Type Name Revision of the business object.
  The maximum length of business object name and revision strings varies for different databases. The maximum length for SQL Server is 127 characters, whereas for Oracle and DB2 it is 255 *bytes*. The number of bytes per character depends on the database setting for character encoding. Thus, for Oracle/DB2, the character count will be fewer than for SQL Server if some characters are multibyte. The same is true for all string values stored in the database. See also *Administrative Object Names*.
- POLICY_NAME is the policy that governs the business object. The Policy clause is required when defining a new business object. For more details on policies, see *Policy Clause*.
- CLAUSEs provide additional information about the business object.

In addition to the syntax above, you can also control the state of a business object. See *Business Object State*.

Add Business Object

Business objects are defined using the Add Businessobject command:

```
add businessobject BO_NAME policy POLICY_NAME [ITEM {ITEM}]
[SELECT [DUMP] [RECORDSEP] [tcl] [output FILENAME]];
```

- BO_NAME is the Type Name Revision of the business object.
- ITEM is an Add Businessobject clause:

description VALUE
vault VAULT_NAME
owner USER_NAME

organization USER_NAME
project USER_NAME
state STATE_NAME | schedule | DATE | actual |
originated DATE
modified DATE
current STATE_NAME
ATTRIBUTE_NAME VALUE
physicalid UUID
logicalid UUID

Adding Legacy Data

If you are adding legacy data from external systems into the database, three clauses are provided to migrate and synchronize the data.

The Current clause of the Add Businessobject is used to create business objects that are already beyond the first state of their policies. The Originated clause of the Add Businessobject command is used to maintain origination dates from the legacy system. Lastly, the Modified clause of the Add Businessobject command is used to maintain modification dates from the legacy system.

If you are migrating legacy data without these clauses, see *Modifying Legacy Data* to make these changes with the Modify Businessobject command.

Policy Clause

This clause assigns a policy to the business object. A policy controls a business object. It specifies the rules that govern access, approvals, lifecycle, versioning and revisioning capabilities, and more. If there is any question as to *what you can do* with a business object, it is most likely answered by looking at the object's policy. Specifically, a policy defines the following information:

- The types of objects the policy will govern.
- The types of formats that are allowed for file checkin and the default format.
- Where and how checked-in files are managed.
- How revisions will be labeled.
- The number and order of each object state.
- The restrictions, if any, associated with each object state.

Since this information is required for a business object to be usable, a Policy clause must be included in the business object definition. If you are unsure about the policy, you should examine the policy definition with the Print Policy command (see *Policies* in Chapter 4).

The policy specified must be defined to govern the type of business object being created.

If the policy name you give in this clause is not found, an error message results. If that occurs, use the List Policy command to check for the existence and spelling of the policy name.

For example, the following object definition assigns the “Software Development” policy to a business object titled “Graphics Display Routine:”

```
add businessobject Routine “Graphics Display Routine” 1
      policy “Software Development”;
      description “Routine for displaying information on a color monitor”
```

After this command is processed, the business object will be created and the “Software Development” policy will control who can access the object and the object’s lifecycle states.

Image Clause

This clause associates a special image, called an ImageIcon, with a business object. While it is optional, it can assist a user in locating a desired object. For example, you may have several different components, blueprints, or designs within a vault. By having an ImageIcon of each item associated with its object name, a user can easily locate the object s/he desires by using the associated ImageIcon as a guide.

For example, you might specify a GIF file of a drawing of each object as their ImageIcons. Then, as you were scanning a set of business objects using the Matrix Navigator (not MQL), you could easily identify a particular object. This would be true even if the objects had names such as “J391” or “X19.”

While ImageIcons are very similar to icons, they require more display time and probably should be displayed only at specific times. Also, while an icon is associated with items such as policies, types, and formats, ImageIcons are associated only with objects and persons. An object may or may not have an ImageIcon associated with it. Excluding an ImageIcon from this object will not affect other objects that are created with the same type, policy, state, and owner. You can even have two revisions of an object with different ImageIcons since each revision represents a different object.

The GIF file needs to be accessible only during definition using the Add or Modify Businessobject command. Once an accessible file in the correct image type is used in the Businessobject command, ENOVIA Live Collaboration will read the image and store it with the object definition in the database. Therefore, the physical icon files do not have to be accessible for every session. (If the file is not accessible during definition, ENOVIA Live Collaboration will be unable to display the image.)

To write an Image clause, you need the full directory path to the ImageIcon you are assigning. For example, the following command assigns an ImageIcon of the actual vehicle to the business object:

```
add businessobject “Energy Efficient Vehicle” “Solar Vehicle X29” A
   policy “Solar Vehicle”
   description “Uses traditional batteries”
   image $MATRIXHOME/demo/vehicleX29.gif;
```

You can view an image with the View menu options or by setting the Session Preferences options in Matrix Navigator.

Specifying redundant icons adds redundant information in the database that requires more work to display. When the default icon is desired, do not specify it.

The Image clause of the Add Businessobject command is optional unless there is a need to distinguish between objects of the same type. If you do not specify an image, the default icon of the type is used.

Retrieving the Image

If you associate an image with a business object using the Image clause, you can retrieve the image as a GIF file using the Image command. The GIF file is placed in the ENOVIA_INSTALL directory, unless overridden by the optional clause, directory.

```
image businessobject OBJECTID [directory DIRNAME] [file FILENAME] [verbose];
```

`OBJECTID` is the `OID` or Type Name Revision of the business object instance from which you want to get the image.

`DIRNAME` is the full pathname where you want to save the image file. If no pathname is specified, the file is saved in the `$MATRIXHOME` directory.

`FILENAME` is the name to be given to the image `.gif` file. If the file name is not specified, the file is given the name of the object with a `.gif` extension.

Using `verbose` prints the file name on the screen.

For example:

```
image bus Assembly 12345 0;
```

This retrieves the image of object Assembly 12345 0 and saves it to the `$MATRIXHOME` directory with the name `12345.gif`.

Vault Clause

This clause specifies the name of the vault where the object will reside. If you include a Vault clause in your definition, it must be placed after the Policy clause and before any other Add Businessobject clauses.

The Live Collaboration must know which vault is associated with the business object. This element is optional because if not explicitly stated. The current vault (as set in the current context) is a default value. If the object you are creating should not be in the current vault, you must include the vault's name in the object's definition.

For example, assume you are in a vault named "Car Loans." You decide to create an object with the specification "Car Loan" "Alan Broder." You could do this by entering the following command:

```
add businessobject "Car Loan" "Alan Broder" A
   policy "Bank Loans";
```

Since a vault is not specified, the current vault, "Car Loans," of iftg to store the business object named "Alan Broder." But what if Mr. Broder also had a mortgage with the same company and you were in the "Mortgages" vault instead of the "Car Loans" vault? You would have to include a Vault clause:

```
add businessobject "Car Loan" "Alan Broder" A
   policy "Bank Loans"
   vault "Car Loans";
```

In many ways, vaults resemble and operate like directories on computers. Your context is similar to your default directory. If the object (like a file) is not in the default directory, you must include the directory as part of the object name.

Owner, Organization, and Project Clauses

In addition to the primary owner, you can define up to two additional owners for a business object or connection: `organization` and/or `project`. This may be important for applications that use access models based on projects and organizations. You must have ChangeOwner privileges to set these owners.

Unlike the primary owner for whom special privileges can be set in a policy (using “owner”), no privileges are set for the additional owners, except by virtue of their user name (that is, policies do not support the notion of generic ORGANIZATION or PROJECT access on every object governed by the policy).

If you do not specify the security context, the following security context is set for you:

- For create and clone, project and organization are set from the log in security context.
- For revise, project and organization are set from the previous revision.

You can also use `altowner1` and `altowner2` to assign alternate owners. An alternate owner can be a person, group, role, or an association. In the following example, the business object is created and `altowner1` is set as a role.

```
add businessobject Document PROSpecification A policy
Documentation owner JerryC altowner1 SpecWriter;
```

State Clause

The lifecycle of a business object is assigned as part of the policy and consists of a series of object states. A state identifies a stage in the lifecycle of an object. Depending on the type of object involved, the lifecycle might contain only one state or many states. The states control the actual object instances and specify what can be done with an object after it is created. Each state defines who will have access to the object in that state, what type of access is allowed, whether or not the object can be revised, whether or not files within the object can be revised, and the conditions required for changing state.

This clause is optional and can be used to specify the scheduled target date for this object in a particular state. It is not necessary to assign a date for any particular state. There may be situations where the change from one state to another may occur at any time. For example, if automobile insurance is contained within an object, the object might change when an accident occurs. Since accidents are not usually planned, no schedule date can be assigned. If, however, you have a state for policy renewal, the date can be scheduled since you know when the policy renewal is due.

Note that this is a target date only. The object will not automatically enter that state on the given date. The object can only be promoted after all required conditions are met—this date will not influence those conditions. The date is intended for informational purposes only.

Actual dates are saved for each state in an object's its lifecycle — and are the dates the object is promoted to that state. These dates are generally system-generated, but can be adjusted with this command for data migration purposes, for cases where the data being imported is from an external source that supports the concept of states with associated actual dates. Since the command is intended for use within a program object, it is up to the program to handle any errors. An error will result if the context user is not a business administrator with Schedule access to the object, or if the date is not specified in the required formats.

When specifying the name of the state to which you want to assign a date, you must make sure it is a valid name. If the state name given in the State clause is not found in the policy you are assigning to the object, an error message will result. If that occurs, use the Print Policy command to check for the existence and spelling of the state name.

When specifying the date within a State clause, you must use the date/time formats defined in your initialization file. If nothing is set there, the defaults are used. See the *Administration Guide* : *Configuring Date and Time Formats*.

For example, the following object definition will schedule the object to be in the "Re-evaluation and Renewal" state on June 30, 2002.

```
add businessobject "Homeowner's Insurance" "Hebron Family" B
   policy "Primary Homeowners"
   description "Homeowner's Insurance for Hebron's Primary Residence"
   state "Re-evaluation and Renewal" schedule "June 30, 2002";
```

In this example, you want to remind the agent of when the homeowner's insurance policy should be renewed. By inserting this date, an agent might promote the object to the "Re-evaluation and Renewal" state even if the date is June 26 (since it is close enough to the target date of June 30th.)

Originated Clause

When migrating legacy data, you are likely to want to maintain the originated date from the legacy system. This clause allows you to assign an originated date. Without this clause, the originated date for objects is the date read from the system.

For example:

```
add businessobject "Document" "Manual" 0 originated 2/2/02;
```

You must specify the date using the date/time formats specified in your initialization file. If nothing is set there, the defaults are used. See the *Administration Guide* : *Configuring Date and Time Formats*.

The actual creation date and time are logged in the create history record of the object. The originated date specified shows up in the Originated field in the Basics dialog in Matrix Navigator. It is also returned with the originated selectable.

The originated date can only be changed with the Modify Businessobject Originated command.

Modified Clause

When migrating legacy data, you are likely to want to maintain the modified date from the legacy system. This clause allows you to assign a modification date. Without this clause, the modified date is the same as the creation date.

For example:

```
add businessobject "Document" "Manual" 0 modified 3/2/02;
```

You must specify the date using the date/time formats specified in your initialization file. If nothing is set there, the defaults are used. See the *Administration Guide* : *Configuring Date and Time Formats*.

The modification date specified shows up in the Modified field in the Basics dialog in Matrix Navigator. It is also returned with the modified selectable.

Unlike the date specified with the originated clause, the modified date will change as normal if modifications are made after object creation.

Current Clause

When migrating legacy data, you are likely to want to create business objects that are already beyond the first state of their policies; for instance, you might want to add Parts that are already released. The Current clause of the Add Businessobject command allows you to create an object in any state of its governing policy. Without this clause, users create business objects in the first state of their governing policies.

Since the Current clause identifies the state of the policy for the business object, it must be specified after the Policy clause.

For example:

```
add businessobject "Document" "Book" A policy
"Documentation" current "Approved";
```

If the state "Approved" is not specified in the policy "Documentation", an error message results.

Promote triggers associated with the state specified do not fire since the object is created in and not promoted to the current state. The create trigger does fire as usual.

Attribute Clause

This clause allows you to assign a specific value to one of the object's attributes. As stated earlier, you must assign a type to any object being created. An object's type may or may not have attributes associated with it. If it does, you can assign a specific value to the attribute using the Attribute clause.

If you are unsure of either of the `ATTRIBUTE_NAME` or the `VALUE` to be assigned, you can use the Print Type and Print Attribute commands, respectively.

For example, assume you are defining an object of type "Shipping Form" by using the following Add Businessobject command:

```
add businessobject "Shipping Form" "Lyon's Order" A
   policy "Shipping";
   description "Shipping Form for the Lyon's Order"
```

Only the attributes associated with an object's type can be assigned values for the instance.

If you were to examine the definition for the type "Shipping Form," you might find it has three attributes associated with it called "Label Type," "Date Shipped," and "Destination Type." When this type is assigned to the object called "Lyon's Order," these attributes are automatically associated with the object. It is up to you as the user to assign values to the attributes.

If you do not assign values, they remain blank or the default is used if there is one. To assign values, you can insert an Attribute clause into the object definition.

When you are specifying an attribute value, be sure the value is in agreement with the attribute definition. In other words, only integer values are assigned to integer attributes, character string values are assigned to character string attributes, and so on. Also, if the attribute has a range of valid values, the value you give must be within that range.

You can use the Print Attribute command to examine an attribute's definition. For example, assume you are unsure of the definition of "Label Type." If you examine the attribute definition, you might see that it is a character string value with no predefined range. With this knowledge, you can insert an Attribute clause to define a value similar to the following:

```
"Label Type" "Overnight Express"
```

String attributes (as well as description fields) have a limit of 2,048 KB. If you expect to enter more data, consider checking in a file instead.

If you want to assign values to each of the three attributes, you can write the object definition as:

```
add businessobject "Shipping Form" "Lyon's Order" A
   policy "Shipping"
   description "Shipping Form for the Lyon's Order"
   "Label Type" "Overnight Express"
   "Date Shipped" 12/22/1999
   "Destination Type" "Continental U.S.";
```

With this definition, each attribute associated with the type "Shipping Form" will have a specific value and those values can be viewed whenever the "Lyon's Order" object is accessed.

Physical ID and Logical ID Clauses

Physical and logical IDs are required for VPM applications. A physical ID is a global identifier that is unique to each business object or relationship. A logical ID is a global identifier shared by all members of a minor revision family.

These clauses are available only if your database has been upgraded to use physical and logical IDs. For details on upgrading to use these IDs, see the *Installation Guide : Upgrade Command* .

Print Business Object

You can view the definition of a business object at any time by using the Print Businessobject command and the Select Businessobject command. These commands enable you to view all the files and information used to define the business object. The system attempts to produce output for each Select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

```
print businessobject OBJECTID [select SELECTABLE] [DUMP
["SEPARATOR_STR"]] [output FILENAME] [sortattributes];
```

```
print businessobject selectable;
```

The Print Businessobject command is similar to using the object inspector in Matrix Navigator.

When this command is processed, MQL displays all information that makes up the named business object's definition. This information appears in alphabetical order according to names of the object's fields. For example, you could obtain the definition for the "Shipping Form" "Lyon's Order" A business object by entering:

```
print businessobject "Shipping Form" "Lyon's Order" A;
```

Unlike the other Print commands, the Print Businessobject command uses four optional Print command clauses:

- *Select Clause*
- *Dump Clause*

- *Tcl Clause*
- *Output Clause*

The optional forms of the Print Businessobject command enable you to specify the information you want to retrieve from a business object. This subset is created by identifying the desired fields of the business object. The field contents can be prepared for formatting and stored in an external file. But which fields can you print and how do you specify them? The Print Businessobject Selectable command addresses these questions. This command enables you to view a listing of field choices and specify those choices, as described in the following sections.

Reviewing the List of Field Names

The first step is to examine the general list of field names. This is done with the command:

```
print businessobject selectable;
```

The Selectable clause is similar to using the ellipsis button in the graphical applications—it provides a list from which to choose.

When the command above is processed, MQL lists all selectable fields with their associated values. This command might produce a listing such as:

```
business object selectable fields:
 name
 description
 revision
 originated
 modified
 lattice.*
 owner.*
 grant.*
 grantor.*
 grantee.*
 granteeaccess
 granteesignature
 grantkey
 policy.*
 type.*
 attribute[].*
 default.*
 format[].*
 current.*
 state[].*
 revisions[].*
 previous.*
 next.*
 first.*
 last.*
 history[].*
 relationship[].*
 to[].*
 from[].*
 toset[].*
 fromset[].*
 exists
 islockingenforced
 vault.*
 locked
 locker.*
 id
 method.*
 search.*
```

Notice that some of the fields are followed by square brackets and/or a "`.*`". The asterisk denotes that the field can be further expanded, separating the subfields with a period. If only one value is associated, the name appears without an asterisk. For example, the Name and Description fields each have only a single value that can be printed. On the other hand, a field such as type has other items that can be selected. If you expand the Type field, you might find fields such as `type.name`, `type.description`, and `type.policy`. This means that from any business object, you could select a description of its type and find out other valid policies for it:

```
print businessobject Drawing 726590 B select type.description type.policy;
```

The above may output something like this:

```
business object Assembly 726590 B
   type.description = Assembly of parts
   type.policy = Production
   type.policy = Production Alternative
```

All items that can be further expanded have a default subfield that is used when that field is specified alone. For example, selecting type is the same as selecting type.name, because the default for types is the type name. For more information, see the *Configuration Guide* : *Appendix: Selectables*.

The use of square brackets in the above selectable list indicates one of two things:

- The assigned name may also be included as part of the selection. For example, an object generally has several attributes. To select a particular attribute, you must give the assigned name of the attribute as part of the field name.

```
print bus Component 45782 A select attribute[Color];
```

  If you select attribute without specifying anything, a list of all attributes with their values are returned. The same is true for state, format, relationship, etc.

- With the `from` and `to` selectables only, you can provide a comma delimited list of patterns that may include wildcard characters, as a means of filtering on relationship names and connected object types. For example:.

```
print bus Guide "Owners Manual" 2004 select
to[N*,S*].from.name   to[N*,S*].from.revision;
```

This command might return something like this:

```
business object Guide Owners Manual 2004
    to[New].from[Chapter].name = Chapter 2
    to[New].from[Photo].name = '2004 Red Camry'
    to[Same].from[Chapter].name = Chapter 4
    to[New].from[Chapter].revision = 5
    to[New].from[Photo].revision = 1
    to[Same].from[Chapter].revision = 2
```

For more information on selecting information on related items, refer to *Expand Business Object*.

Selecting Field Names for Printing

Once you have identified names of the fields that can be printed, you can select them.

```
print OBJECT_ID select FIELD_NAME {FIELD_NAME};
```

`FIELD_NAME` can be a string, list of strings, or wildcard character (*) that represents a printable field value associated with a business object. Each field name obeys the following syntax:

```
FIELD_NAME [ASSIGNED_NAME_PATTERN].SUBFIELD_PATTERN
```

`ASSIGNED_NAME_PATTERN` is the name assigned to the field when it is created. This name, when required, must be within square brackets.

SUBFIELD_PATTERN is the specification of another field. This field is a part of the larger field specification. All subfield names are preceded by a period and often correspond to the clauses that make up a field definition.

For example:

This select command is added as a clause to the Print Businessobject command:

```
print businessobject Drawing 726596 B select name description type.name;
```

This yields:

```
business object Drawing 726596 B
   name = 726596
   description = Piston Assembly
   type.name = Drawing
```

Notice that the fields appear in the order they were specified in the Select clause.

Sortattributes Clause

When printing business objects, the Sortattributes clause causes the command to list attributes in the same order as in the Live Collaboration.

Connection Selectables

The frommid and tomid selectables on the Print Connection command allow you to print information on the FROM and TO ends of a relationship.

For example, the following commands create a set of objects with relationships that connect objects to objects, objects to relationships, relationships to objects, and relationships to relationships. They then execute print/query commands with the corresponding selectables.

```
#
#  Creates this structure:
#
#  A1 ------rx------> A2 ------rx------> A3 ------rx------> A4
#   \       |    \
#    \      |     \
#    o2r   r2r   r2o
#      \    |       \
#       \   |        \
#  B1 ------rx------> B2

set context user creator;
add vault unit1;
add attribute string-u type string;
add attribute rstring-u type string;
add type tx attribute string-u;
add policy px type tx state one state final;
add rel rx from type tx to  type tx attribute rstring-u;
add rel r2o from rel rx to type tx attribute rstring-u;
add rel o2r from type tx to rel rx attribute rstring-u;
add rel r2r from rel rx to rel rx attribute rstring-u;
```

```
add bus tx A1 0 policy px vault unit1 string-u A1-attr;
add bus tx A2 0 policy px vault unit1 string-u A2-attr;
add bus tx A3 0 policy px vault unit1 string-u A3-attr;
add bus tx A4 0 policy px vault unit1 string-u A4-attr;
add bus tx B1 0 policy px vault unit1 string-u B1-attr;
add bus tx B2 0 policy px vault unit1 string-u B2-attr;

connect bus tx A1 0 rel rx to tx A2 0 rstring-u A1-A2;
connect bus tx A2 0 rel rx to tx A3 0 rstring-u A2-A3;
connect bus tx A3 0 rel rx to tx A4 0 rstring-u A3-A4;
connect bus tx B1 0 rel rx to tx B2 0 rstring-u B1-B2;
add connection o2r from tx A1 0 torel bus tx B1 0 to tx B2 0 rel rx
rstring-u A1-rel;
add connection r2o fromrel bus tx A1 0 to tx A2 0 rel rx to tx B2 0
rstring-u rel-B2;
add connection r2r fromrel bus tx A1 0 to tx A2 0 rel rx torel bus tx
B1 0 to tx B2 0 rel rx rstring-u rel-rel;

# Fromrel, torel show any relationships on the to/from end of "this"
relationship.
# Frommid, tomid show any relationships that have "this" relationship
to relationships
# allow subselects to get data on the rels they map to.
print bus tx A1 0 select from.attribute.value from.torel
from.torel.attribute.value from.torel.from.name from.torel.to.name;
query connection type rx,r2o,o2r,r2r select fromrel
fromrel.attribute[rstring-u] torel torel.attribute[rstring-u];
query connection type rx,r2o,o2r,r2r select frommid
frommid.attribute[rstring-u] tomid tomid.attribute[rstring-u];

# Fromall, toall show both businessobjects and rels (with a B or R
prefix)
# They
query connection type rx,r2o,o2r,r2r select fromall toall;


# Cleanup
del bus tx A1 0 ;
del bus tx A2 0 ;
del bus tx A3 0 ;
del bus tx A4 0 ;
del bus tx B1 0 ;
del bus tx B2 0 ;
del rel rx;
del rel r2o;
del rel o2r;
del rel r2r;
del policy px;
del type tx;
```

```
##########################################################################
#####
# HF-105472V6R2012_

tcl;
set rxId ""
set o2rId ""
set obj2Id ""
set lIds [split [mql print bus tx A1 0 select id from.id from.to.id]
\n]
foreach sId $lIds {
  if {[string first "\[rx\].id" $sId] > 0} {
    set rxId [string trim [lindex [split $sId =] 1]]
  }
  if {[string first "o2r" $sId] > 0} {
    set o2rId [string trim [lindex [split $sId =] 1]]
  }
  if {[string first "\[rx\].to.id" $sId] > 0} {
    set obj2Id [string trim [lindex [split $sId =] 1]]
  }
  set r2oId [mql print bus tx B2 0 select to\[r2o\].id dump]
}
puts "rxId=$rxId\no2rId=$o2rId\nobj2Id=$obj2Id\nr2oId=$r2oId"

# Bad command - changing TO from obj to rel
mql start trans;
mql mod connection $rxId torel $o2rId
mql abort trans;

# Bad command - changing TO from rel to obj
mql start trans;
mql mod connection $o2rId to $obj2Id
mql abort trans;

# Bad command - changing FROM from obj to rel
mql start trans;
mql mod connection $rxId fromrel $o2rId
mql abort trans;

# Bad command - changing FROM from rel to obj
mql start trans;
mql mod connection $r2oId from $obj2Id
mql abort trans;

# Cleanup
del bus tx A1 0;
del bus tx A2 0;
del bus tx A3 0;
del bus tx A4 0;
```

```
del bus tx B1 0;
del bus tx B2 0;
del rel rx;
del rel r2o;
del rel o2r;
del rel r2r;
del type tx;

########################################################################
#
#  GENERATE XML OUTPUT FROM EXPAND COMMAND
#
expand bus tx A1 0 xml from recurse to all select bus attribute.value
select rel type attribute.value frommid.attribute.value
frommid.to.type frommid.to.name frommid.to.revision frommid.torel
frommid.torel.attribute.value frommid.torel.from.name
frommid.torel.to.name output testExpand.xml;
```

The output of the above commands is as follows:

```
MQL<100>add type tx attribute string-u;
MQL<101>add policy px type tx state one state final;
MQL<102>add rel rx from type tx to  type tx attribute rstring-u;
MQL<103>add rel r2o from rel rx to type tx attribute rstring-u;
MQL<104>add rel o2r from type tx to rel rx attribute rstring-u;
MQL<105>add rel r2r from rel rx to rel rx attribute rstring-u;
MQL<106>
add bus tx A1 0 policy px vault unit1 string-u A1-attr;
MQL<107>add bus tx A2 0 policy px vault unit1 string-u A2-attr;
MQL<108>add bus tx A3 0 policy px vault unit1 string-u A3-attr;
MQL<109>add bus tx A4 0 policy px vault unit1 string-u A4-attr;
MQL<110>add bus tx B1 0 policy px vault unit1 string-u B1-attr;
MQL<111>add bus tx B2 0 policy px vault unit1 string-u B2-attr;
MQL<112>
connect bus tx A1 0 rel rx to tx A2 0 rstring-u A1-A2;
MQL<113>connect bus tx A2 0 rel rx to tx A3 0 rstring-u A2-A3;
MQL<114>connect bus tx A3 0 rel rx to tx A4 0 rstring-u A3-A4;
MQL<115>connect bus tx B1 0 rel rx to tx B2 0 rstring-u B1-B2;
MQL<116>add connection o2r from tx A1 0 torel bus tx B1 0 to tx B2 0
rel rx rstring-u A1-rel;
MQL<117>add connection r2o fromrel bus tx A1 0 to tx A2 0 rel rx to tx
B2 0 rstring-u rel-B2;
MQL<118>add connection r2r fromrel bus tx A1 0 to tx A2 0 rel rx torel
bus tx B1 0 to tx B2 0 rel rx rstring-u rel-rel;
MQL<119>
print bus tx A1 0 select from.attribute.value from.torel
from.torel.attribute.value from.torel.from.name from.torel.to.name;
business object  tx A1 0
    from[o2r].attribute[rstring-u].value = A1-rel
    from[rx].attribute[rstring-u].value = A1-A2
    from[o2r].torel = rx
```

```
    from[o2r].torel.attribute[rstring-u].value = B1-B2
    from[o2r].torel.from.name = B1
    from[o2r].torel.to.name = B2
MQL<120>query connection type rx,r2o,o2r,r2r select fromrel
fromrel.attribute[rstring-u] torel torel.attribute[rstring-u];
relationship rx
relationship rx
relationship rx
relationship rx
relationship o2r
    torel = rx
    torel.attribute[rstring-u] = B1-B2
relationship r2o
    fromrel = rx
    fromrel.attribute[rstring-u] = A1-A2
relationship r2r
    fromrel = rx
    fromrel.attribute[rstring-u] = A1-A2
    torel = rx
    torel.attribute[rstring-u] = B1-B2
MQL<121>query connection type rx,r2o,o2r,r2r select frommid
frommid.attribute[rstring-u] tomid tomid.attribute[rstring-u];
relationship rx
relationship rx
relationship rx
    frommid = r2o
    frommid = r2r
    frommid[r2o].attribute[rstring-u] = rel-B2
    frommid[r2r].attribute[rstring-u] = rel-rel
relationship rx
    tomid = o2r
    tomid = r2r
    tomid[o2r].attribute[rstring-u] = A1-rel
    tomid[r2r].attribute[rstring-u] = rel-rel
relationship o2r
relationship r2o
relationship r2r
MQL<122>query connection type rx,r2o,o2r,r2r select fromall
fromall.attribute[rstring-u] toall toall.attribute[rstring-u];
relationship rx
    fromall = B2384.50535.15736.55350
    toall = B2384.50535.15736.54777
relationship rx
    fromall = B2384.50535.15736.54777
    toall = B2384.50535.15736.32988
relationship rx
    fromall = B2384.50535.15736.51305
    toall = B2384.50535.15736.55350
```

```
relationship rx
    fromall = B2384.50535.15736.63113
    toall = B2384.50535.15736.6841
relationship o2r
    fromall = B2384.50535.15736.51305
    toall = R2384.50535.15736.45609
relationship r2o
    fromall = R2384.50535.15736.51178
    toall = B2384.50535.15736.6841
relationship r2r
    fromall = R2384.50535.15736.51178
    toall = R2384.50535.15736.45609
```

Revisions Selectable

The `revisions[ ].revindex` selectable is similar to other revision related selectables in that it returns a revision in the revision sequence. It returns revisions in between the first and last revisions. In the selectable you can specify the business object revision by its index. If the index is not available, this selectable will work like the `next` selectable and return no value.

Method and Program Selectables

The select keywords method and program can be used wherever select keywords for business objects can be used. They can be used in both print bus and in expressions evaluated against business objects. The syntax is:

```
program [PROGRAM_NAME ARG1 ARG2 …]
```

Or:

```
method [METHOD_NAME ARG1 ARG2 …]
```

where `PROGRAM_NAME` is the name of a program

where `METHOD_NAME` is the name of a method

Zero or more arguments can be included as indicated. The total number of characters in brackets is limited to 128.

The `program` keyword allows non-method programs to be run.

- They can use the same macros as a method.
- The arguments are space-delimited.
- The program/method name must be quoted if it contains spaces.
- Arguments containing spaces must be quoted or enclosed in {}.
- If the program exits with a non-zero exit code, the select clause returns an empty string as a value.

The select clause returns a string whose value is specified by the program. The program specifies it by placing the string in the GLOBAL RPE variable whose name equals the program name. This is consistent with the use of a program to load a single valued text widget, except for having to use `global`.

Note that as a select clause, this capability can be used in any of the following ways:

- `print bus T N R select owner current program[NAME] attribute[ATT] dump`
- to load Tcl variables via: `set sVar [mql print bus ....]`
- as part of a where clause in a Query or Cue
- as the expression to be provided in a Tip
- as a column definition.

Execute Selectable

You can use the select keyword `execute` in place of `program` to pass selectables as arguments to Tcl programs. `Execute` is a faster alternative to `program` because it passes selectable values directly into programs. For example, you can write:

```
temp query bus * * * where 'escape execute[X "arg0"
attribute\[Z\]]==TRUE'
```

And then in the program X have:

```
tcl;
eval {
set arg0 ${1};
set data ${2};
...
return ${result};
}
```

The keyword `escape` in the above argument allows selectables to be passed in the bracketed portion of the `execute` selectable. There is no need to call `get env` or `set env global`. Arguments specified with execute in this manner are pre-evaluated and the values are passed directly into the program. The return value is collected from Tcl output. Performance is improved by eliminating time spent fetching data from the Runtime Program Environment (RPE) or putting data as output back into the RPE.

.generic subselect

The kernel provides a number of APIs that output fields for dates, times, and real numbers according to global environment variables (MX_NORMAL_DATETIME_FORMAT, MX_DECIMAL_SYMBOL, etc.) set by an Administrator. Since there are many formats possible to output these fields, a generic format can now be used to ensure consistent date, time, and real number format regardless of the global settings. To use the generic format, add the ‘generic’ subselect or suffix to a selectable e.g. attribute[Cost].generic or modified.generic.

The generic format for dates and times corresponds to JavaSimpleDateTimeFormat specified by the following date and time pattern: yyyy/MM/ dd'@'HH:mm:ss:z. For real numbers, the generic format returns a dot (.) as a decimal separator.

The ‘generic’ subselect can be added to almost any selectable which returns date/time or real numbers and can be used in both the Java APIs as well as the MQL APIs. It should not be used in where clauses or access filters.

```
print businessobject OBJECTID select modified.generic dump;
```

Copy Business Object

After a business object is added and its values defined, you can copy or clone the object with the Copy Businessobject command. This command lets you duplicate a business object's defining clauses and change some of the values at the same time.

While grants are not a defining clause, when business objects are copied, any and all grants are also copied to the new object. You cannot revoke the accesses within the copy businessobject command.

You might first view the object's contents using the Print Businessobject commands described in *Print Business Object*. Then you can use the Copy Businessobject clauses listed below to modify the values you want to change. You should recognize these clauses since they are the same in the Add Businessobject command.

```
copy businessobject OBJECTID [!file] to NAME REVISION
[history] [MOD_ITEM {MOD_ITEM}] [SELECT [DUMP] [RECORDSEP]
[tcl] [output FILENAME]];
```

- `OBJECTID` is the `OID` or Type Name Revision of the existing business object.
- `NAME` is the name for the new business object. If you use the same type, name and revision as the original business object, an error will occur and the business object will not be copied.
- `REVISION` is the revision label or designator. If a revision is desired, it should be specified as the full revision string (major-minor) as specified in the governing policy (see *Major/Minor Revisions* in Chapter 5). Otherwise, it is `""` (a set of double quotes).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the Modify Business Object table below for a complete list of possible modifications.
- Any of these values can be changed as you copy the business object. Note that you only need to include clauses for the values that you want to change. If you do not make any changes, the values remain the same as the original business object with the new name you have assigned.

Handling Files

If you want to clone an object without including the original files in the new object, use the following syntax:

```
copy bus OBJECTID [!file] to NEWNAME REVISION [Vault];
```

The placement of the !file clause must follow the above exactly or an error will occur.

Since files are included by default, include the !file clause and they won't be copied. For example:

```
copy businessobject "Shipping Form" "Lyon's Order" 5 !file
to "Ryan's Order" 1;
```

Including History

By default, when a new object is created with the copy command (or via the Studio Customization Toolkit or GUI), the only history record it has is similar to:

```
history = create - user: creator time: Mon Sep 12, 2005 10:28:58 PM EDT
state: one  revisioned from: T1 1 0
```

To include the history of the original object in MQL, use:

```
copy businessobject BO_NAME to NAME REVISION history;
```

After copying the history records from the original object to the cloned object, the additional create history record will be appended indicating that it is a copy (revisioned from), as shown above.

Since the command is intended for use within a program object, it is up to the calling program to handle any errors.

Revise Business Object

To revise a business object, use the Revise Businessobject command:

```
revise businessobject OBJECTID [to REVISION] [!file] [SELECT
[DUMP] [RECORDSEP] [tcl] [output FILENAME]] [major|minor];
```

- `OBJECTID` is the `OID` or Type Name Revision of the object you want to revise.
- `REVISION` is the revision designator to be assigned. If a revision is desired, it should be specified as the full revision string (major-minor) as specified in the governing policy (see *Major/Minor Revisions* in Chapter 5).
- Refer to *Using Select and Related Clauses* for additional details.

In order to maintain revision history, the new object should be created with the revised business object command even though it is possible to create what looks like a new revision by manually assigning a revision designator with the Add or Copy Businessobject command. (However, you can add existing objects to a revision chain, if necessary. Refer to *Adding a Business Object to a Revision Sequence* in Chapter 5 for more information.) The table below shows the differences between using the `revise businessobject` command and the `copy businessobject` command.

Not all business objects can be revised. If revisions are allowed, the Policy definition may specify the scheme for labeling revisions. This scheme can include letters, numbers, or enumerated values. Therefore you can have revisions with labels such as AA, 31, or “1st Rev.”

If the `REVISION` is omitted, a designator is automatically assigned based on the next value in the sequence specified in the object’s policy. If there is no sequence defined, an error message results.

If there is no defined revision sequence or if you decide to skip one or more of a sequence’s designators, you can manually specify the desired designator as the `REVISION`.

For example, assume you entered the following command:

```
revise businessobject “Shipping Form” “Lyon’s Order” 5;
```

Since no revision designator is given, the new business object may be called the following (assuming the revision scheme of 1, 2, 3 ...):

```
“Shipping Form” “Lyon’s Order” 6
```

If you want to skip over the next revision number and assign a different number, you can do so as:

```
revise businessobject “Shipping Form” “Lyon’s Order” 5 to
10;
```

This command will produce a new business object labeled:

```
"Shipping Form" "Lyon's Order" 10
```

Now the revised business object has a revision designator equal to ten.

Note that you cannot use a designator that has already been used. If you do, an error message will result.

Handling Files

If you want to revise an object without including the original files in the new object, use the following syntax. See above

To add an object to an existing revision sequence, use the Revise Businessobject command:

```
revise businessobject OBJECTID bus NEWOBJECTID [file];
```

`OBJECTID` is the `OID` or Type Name Revision of the last object in a revision chain.

`NEWOBJECTID` is the `OID` or Type Name Revision of an existing object that is not already part of any revision sequence.

The optional `file` keyword applies only if the new object has no files checked in, in which case it specifies whether files should or should not be inherited from the revision sequence to which it is being added. The default is `!file`, so no files are inherited. This flag is ignored if `NEWOBJECTID` contains files.

Creating Major and Minor Revisions

Using the *major* or *minor* keyword with the Revise command specifies whether to create a new major revision or a new minor revision, respectively. Creating a new major revision starts a new minor revision family.

```
revise bus T N R [major|minor] ... ;
```

If neither the *major* nor *minor* keyword is specified, then a new minor revision is created and existing revisions become minor revisions.

Revise Major/Minor commands affect several basic business object properties, as shown in the following table. Shading indicates new properties that were added in V6R2013.

Property	Type	Revise minor	Revise major	Storage
PhysicalId	UUID	Changed	Changed	mxIdent
LogicalId	UUID	Same	Same	mxIdent
MajorId	UUID	Same	Changed	**mxIdent**
MajorRevision	String	Same	Incremented	lxBO.lxRev
VersionId	UUID	Same	Same	**mxIdent**

Property	Type	Revise minor	Revise major	Storage
MinorRevision	String	Incremented	Reset	lxBO.lxRev
MajorOrder	Integer	Same	Incremented	**mxIdent**
MinorOrder	Integer	Incremented	Reset	**mxIdent**
Published	Boolean	From policy/state	From policy/state	**mxIdent**

If an object policy does not have a minor revision sequence defined, then Revise Minor produces an error. If an object policy does not have a major revision sequence defined, then Revise Major produces an error. If an object policy has both major and minor revision sequences defined, the majorrevision and minorrevision strings are calculated as in the table above, and the two strings are joined with the delimiter defined in the policy and stored in the existing revision field. See policy Command, *Majorsequence and Minorsequence Clauses*.

Storage of major/minor in the existing revision field allows the Type-Name-Revision combination to continue to be the default uniqueness criteria without requiring an update to the out-of-the-box table that governs TNR uniqueness. It also allows all commands that reference objects by TNR criteria to continue to work as before. For objects that have both major and minor revision strings, the evision field is of the form `major-minor`, where the dash represents the delimiter specified in the policy.

Changing policy on a business object is allowed as long as the new policy does not change a delimiter or add/remove either a major or minor sequence to/from the existing policy. Legacy policies with no sequences defined are considered to have a minorsequence, which is empty, and no majorsequence.

Several new selectables for business objects were added in V6R2013:

- Existing selectables (*revision*, *revisions*,*first*, *last*, *next*, and *previous*) continue to work for existing revision sequences as before, until the policy is modified to define major and minor revision sequences.
- Parallel minor-revision-related selectables (*minorrevision*, *minorrevisions*, *firstminor*, *lastminor*, *nextminor*, *previousminor*) were added to continue supporting existing functionality even in the presence of mixed major/minor revisions.
- Parallel major-revision-related selectables (*majorrevision*, *majorrevisions*, *firstmajor*, *lastmajor*, *nextmajor*, *previousmajor*) were added to support major revisions.
- Additional business object selectables (*ispublished*, *lastpublished*, *isbestsofar*, *bestsofar*) were added to report the published and best-so-far (BSF) status of business objects.

Modify Business Object

After a business object is added, you can change it with the Modify Businessobject command. This command lets you add or remove defining clauses or change some of their values.

```
modify businessobject OBJECTID [MOD_ITEM {MOD_ITEM}]
[SELECT [DUMP] [RECORDSEP] [tcl] [output FILENAME]];
```

- OBJECTID is the OID or Type Name Revision of the business object you want to modify.
- MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Businessobject clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Businessobject Clause	Specifies that…
`description VALUE`	The current description, if any, is changed to the value entered.
`vault vault_NAME [update set]`	The business object is moved from the current vault to the new vault specified here. See *Changing a Business Object's Vault*.
`name NAME [revision REVISION]`	The name, and optionally the revision, of the business object are changed to the name and revision specified here. The revision should be specified as the full revision string (major-minor) as specified in the governing policy (see *Major/Minor Revisions* in Chapter 5).
`owner USER_NAME`	The owner of the business object is changed to the owner specified here.
`add ownership ORGANIZATION PROJECT [for COMMENT] [as ACCESS]`	Adds an ownership to the business object. See *Multiple Ownerships* and *Adding or Removing Business Object Ownerships*.
`remove ownership ORGANIZATION PROJECT [for COMMENT] [as ACCESS]`	Removes an ownership from the business object. See *Multiple Ownerships* and *Adding or Removing Business Object Ownerships*.
`add ownership bus|rel OWNERSHIP_ID`	Adds an ownership that is inherited from another object or relationship. See *Ownership Inheritance* and *Adding Ownership Inheritance*.
`grant USER access ACCESS{,ACCESS} [signature true|false] {grant USER access ACCESS{,ACCESS} [signature true|false]} [key KEY]`	The specified access to the object is granted to the user(s) specified. See *Granting Business Object Access*.
`revoke all|grantee|grantor [USER]] {grantee|grantor [USER]} [key KEY]`	Access that has previously been granted to another user is revoked. See *Revoking Business Object Access*.
`policy POLICY_NAME`	The policy of the business object is changed to the policy specified here. Acquired signatures are retained if they match new signature requirements. The current state of the business object is retained if that state is a state of the new policy. If you change an object's policy you may also inadvertently change the store used by the object. In this case, newly checked in files will be kept in the new policy's store. If the object contained it's own files before the policy was changed, they will stay in the old store, until/unless they are replaced during a subsequent checkin. However, the old store will still be used in the case where another object contains a reference to files in the object that now uses a different store. When the time comes for the reference to become an actual file (as when the file list changes between the 2 objects) the file copy is made in the same store the original file is located in.
`move [format FORMATNAME] from BO_NAME FILES_TO_COPY;`	The specified FILES_TO_COPY are removed from the from BO_NAME to the business object being modified. Refer to *Moving Files* below for more information.

Modify Businessobject Clause	Specifies that…
`state STATE_NAME | schedule | DATE` `| actual |`	The date associated with the beginning of a state is changed as specified here.
`rename [format FORMATNAME] [[!|not]propagaterename] file FILENAME TOFILENAME`	The specified FILENAME is changed to TOFILENAME. Refer to *Renaming Files* below for more information.
`reserve [user USER] [comment COMMENT];`	To attach reservation data to an object: username, date, optional comment. The user clause can be included only by system administrators to reserve a connection on behalf of someone else. The Reserve access mask controls the changing of the reserve status of an object or connection. See *Modify Connection*.
`unreserve`	To attach reservation data to an object: username, date, optional comment. The user clause can be included only by system administrators to unreserve a connection on behalf of someone else. The unreserve access mask controls the changing of the reserve status of an object or connection. See *Modify Connection*.
`!reserve`	To attach reservation data to an object: username, date, optional comment. The user clause can be included only by system administrators to reserve a connection on behalf of someone else. The reserve access mask controls the changing of the reserve status of an object or connection. See *Modify Connection*.
`type TYPE_NAME`	The type associated with the business object is changed as specified here. Attributes of the business object which are part of the new type definition are retained. Attributes which are not part of the new type definition are deleted. Any new attributes are added with their default value.
`ATTRIBUTE_NAME VALUE`	The attribute value associated with the business object is changed as specified here.
`add history VALUE [comment VALUE]`	A history Tag and a Comment can be specified with this clause. The Tag is the `VALUE` you enter after `add history` as a name for the custom history entry. A user/time/date/current stamp is automatically applied when this command executes. The `comment` is the `VALUE` (or text) you enter to describe the history item you are adding.
`delete history [ITEM {ITEM}]`	System administrators only can purge the history records of a business object. See *Delete History* for details.
`originated DATE`	Business Administrators only can modify the originated basic property of a business object. DATE must be in the format specified in the environment. Refer to the *Installation Guide* for details. There currently is no history or trigger event associated with this event.
`modified DATE`	Business Administrators only can modify the modified basic property of a business object. DATE must be in the format specified in the environment. Refer to the *Installation Guide* for details. There currently is no history or trigger event associated with this event.

As you can see, each modification clause is related to the clauses and arguments that define the business object. When you modify a business object, you first name the object to be changed (by type, name, and revision) and then list the modification using the appropriate clauses.

Reserving a Business Object

To mark the business object or connection as reserved:

```
modify businessobject "Box Design" "Thomas" "A"
reserve comment "reserved from Create Part Dialog";
```

The businessobject "Thomas" of type "Box Design" and revision "A" is reserved.

When a reservation is made, the following information is sent for storage in the database:

- Person reserving the object.
- Time when the object is reserved.
- Comments entered when the reservation is made.

The comment string is optional and has a 254 character limit. Comments longer than 254 characters are truncated and only the first 254 characters are stored in the database.

You can use reservation data to control concurrent modification in two ways:

- Code controlling the display of an editing page may check the reserved status beforehand, and reject the editing request if the object is reserved.
- Define access rules to control any access. See definition of any reserve/no reserve/context reserve/inclusive reserve under *policy Command* and *rule Command*.

The data is written to the database only upon transaction commit. The reserved status of a business object or connection is true only if a user has reserved it and has committed the transaction containing the modify reserve command.

System administrators can include the user clause to reserve an object on behalf of another user:

```
modify businessobject "Box Design" "Thomas" "A"
reserve user Sue comment "reserved from Create Part Dialog";
```

The businessobject "Thomas" of type "Box Design" and revision "A" is reserved for Sue by the super user (a user that must be defined as a system administrator).

The privilege to reserve or unreserve an object can be specific in accesss rules using the reserve and unreserved access mask. For more information, see *policy Command* and *rule Command*. Unreserving a business object or connection by providing the wrong OID or Type Name Revision will result in an error message. Unreserving a business object or connection that is not reserved will result in a warning.

Changing a Business Object's Vault

When an object's vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data.
- The original business object is deleted from the "old" vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object's vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless. To address this issue, you can fix sets at the time the change vault is performed (in MQL only). For example:

```
modify bus Assembly R123 A vault Engineering update set;
```

The additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails, which is why you must explicitly ask the system to do this.

Business administrators can execute the following MQL command to enable/disable this functionality as the default behavior for system-wide use:

```
set system changevault update set | on | off |;
```

See *Controlling System-wide Settings* in Chapter 9 for more information.

A business object can appear to have two id's if it moves from one vault to another. However, there is only one real id (in the current vault).

The record in the old vault is kept to redirect to the record in the new vault to avoid the following errors for programs that may continue to operate on the object with the old id to complete their task:

```
'business object stale' and 'business object id not found'
```

Query for the object via `Type Name Rev Vault` will not return these errors.

The old records are cleaned during the original vault cleanup.

Granting Business Object Access

Any person can grant accesses on an object to any other user as long as the person has "grant" access. The grantor is allowed to delegate all or a subset of his/her accesses on the current state.

The MQL command and Studio Customization Toolkit methods for granting business objects allow users to:

- grant an object to multiple users
- have more than one grantor for an object
- grant to any user (person/group/role/association)
- use a key to revoke access without specific grantor/grantee information

Users cannot grant the grant access itself. The intent is to provide just 1 level of delegation. Although including grant in the list of accesses will not fail, the grantee of the grant access will not be able to grant, unless s/he already has grant access.

For details on accesses, see *Controlling Access* in Chapter 3.

Granting Access to Multiple Users

You can use one `modify bus` command to grant an object to multiple grantees with the same or different accesses.

To grant an object to multiple grantees, each with different accesses use::

```
modify bus OBJECTID grant USER access ACCESS{,ACCESS} [signature true|false]
{grant USER access ACCESS{,ACCESS} [signature true|false]};
```

- `OBJECTID` is the `OID` or Type Name Revision of the object you want to modify.
- `USER` is any person, group, role or association;
- `ACCESS` is any of the different accesses that need to be provided to the grantee. (For details, see the *Accesses* in Chapter 3.) At least one ACCESS must be specified, or none are given. The ! can be used with any access in conjunction with the keyword `all` to provide all privileges the delegator has, except those specified.

For example:

```
modify bus Assembly SA-4356 A grant Engineering access all, !checkin;
```

This command will provide Engineering with all accesses except checkin. However, the Engineering group will only be able to perform operations for which the grantor has access (with the exception of checkin). “All” is evaluated just like other access items, and exceptions to “all” must delimited with a comma.

The optional `signature` clause is used to provide the same access on signatures for the business object that the Grantor has. The default is `false`.

For example:

```
mod bus Part Sprocket 3 grant Jo access read signature false
grant Jackie access read,modify signature true
grant Jules access read,modify,checkin,checkout;
```

The above `mod bus` grants access to the object to three grantees: `Jo`, `Jackie`, and `Jules`. User `Jo` has only read access and cannot sign for the grantor. User `Jackie` has read and modify access and can sign for the grantor. User `Jules` has read, modify, checkin, checkout and cannot sign for the grantor.

- To grant an object to more than one grantee with the same accesses and signature privilege use:

```
mod bus TYPE NAME REVISION grant USER{,USER} access ACCESS{,ACCESS} [signature
true|false];
```

For example:

```
modify bus Part Sprocket 3 grant Jo,Jackie,Jules access
read,modify signature false;
```

Subsequent `modify bus` commands with the grant clause will overwrite, (not add to) the granted accesses and users *only* if the grantor and grantee match an existing grantor/grantee pair on the business object. If either the grantor or grantee of the pair does not match an existing pair, an additional set of accesses will be granted for the object.

Granting Access Using Keys

Profile management tasks in apps typically perform all grants from the same grantor, which makes identifying/modifying individual grants very difficult for the programmer. When granting business objects via MQL (or MQLCommand), it is possible to specify an identifier, or key, for each individual grant. Grants can then be revoked by referencing this key with or without the grantor/grantee information. To specify a for a grant, use the following:

```
modify bus TYPE NAME REV grant USER access MASK key KEY;
```

For example:

```
modify bus Assembly 123 0 grant stacy access all key Buyer;
```

Revoking Business Object Access

You can use several approaches to revoke granted access to a business object:

- If you have Revoke access, you can remove all granted access on a business object with:

```
mod bus TYPE NAME REVISION revoke all;
```

  For example:

  ```
  mod bus Part Sprocket 3 revoke all;
  ```

- You can revoke all accesses granted by a specific grantor with:

```
mod bus TYPE NAME REVISION revoke grantor USER;
```

  For example:

  ```
  mod bus Part Sprocket 3 revoke grantor Jo;
  ```

  The above command removes all grants made by grantor `Jo`. This command completes successfully if the context user is `Jo`, or if the context user has Revoke access; otherwise, an error is generated.

- You can revoke all accesses granted to a specific grantee with:

```
mod bus TYPE NAME REVISION revoke grantee USER;
```

  For example:

  ```
  mod bus Part Sprocket 3 revoke grantee Jackie;
  ```

  This will revoke all grants where Jackie is the grantee. This command executes successfully if the current context is `Jackie` or if the current context has Revoke access; otherwise, an error is generated.

- You can specify both grantee and grantor to revoke the grant made between them with:

```
mod bus TYPE NAME REVISION revoke grantor USER grantee USER;
```

  For example:

  ```
  mod bus Part Sprocket 3 revoke grantor Jo grantee Jackie;
  ```

  This will revoke the grant between grantor `Jo` and grantee `Jackie`. If the context user is `Jo`, or if the context user has Revoke access, this command will succeed; otherwise, an error will be generated.

- To revoke the grant without specific grantor/grantee information, you can specify an identifier, or key, defined when the grant was made:

```
mod bus TYPE NAME REVISION revoke key KEY;
```

  In addition, the key is selectable as follows:

  ```
  print bus|set TYPE NAME REV select grantkey;
  expand bus|set TYPE NAME REV select grantkey;
  ```

  The following example demonstrates the use of keys to grant and revoke access, and includes MQL `print bus` output on the object:

  ```
  MQL< >mod bus Assembly 123 0 grant bill access read,checkin,show key
  Supplier;
  MQL< >mod bus Assembly 123 0 grant bill access
  read,checkin,checkout,show key SupplierEngineer;
  MQL< >mod bus Assembly 123 0 grant stacy access all key Buyer;
  MQL< >print bus Assembly 123 0 select grant.*;
  business object  Assembly 123 0
      grant[creator,bill].grantor = creator
      grant[creator,bill].grantor = creator
  ```

```
grant[creator,stacy].grantor = creator
grant[creator,bill].grantee = bill
grant[creator,bill].grantee = bill
grant[creator,stacy].grantee = stacy
grant[creator,bill].granteeaccess = read,checkin,show
grant[creator,bill].granteeaccess = read,checkin,checkout,show
grant[creator,stacy].granteeaccess = all
grant[creator,bill].granteesignature = FALSE
grant[creator,bill].granteesignature = FALSE
grant[creator,stacy].granteesignature = FALSE
grant[creator,bill].grantkey = Supplier
grant[creator,bill].grantkey = SupplierEngineer
grant[creator,stacy].grantkey = Buyer
```

Since a unique name (key) has been provided for each grant, the grants can be revoked using this key, rather than relying on the grantor, grantee pair, which may not be -- and in this example is not -- unique. For example:

```
MQL< >mod bus Assembly 123 0 revoke grant key SupplierEngineer;
```

This command removes the access granted to Bill as a SupplierEngineer, but it will leave the access granted to Bill as a Supplier. The print command output confirms this:

```
MQL< >print bus Assembly 123 0 select grant.*;
business object  Assembly 123 0
  grant[creator,bill].grantor = creator
  grant[creator,stacy].grantor = creator
  grant[creator,bill].grantee = bill
  grant[creator,stacy].grantee = stacy
  grant[creator,bill].granteeaccess = read,checkin,show
  grant[creator,stacy].granteeaccess = all
  grant[creator,bill].granteesignature = FALSE
  grant[creator,stacy].granteesignature = FALSE
  grant[creator,bill].grantkey = Supplier
  grant[creator,stacy].grantkey = Buyer
```

Be aware that Revoke access is a very powerful access. By giving a user Revoke access, you are saying that this user can revoke anyone's grants.

Moving Files

You can move files from one object to another (or another format of the same object) using the `modify bus` command, provided you have checkout access to the from object and checkin access to the to object. Moving files is equivalent to doing a checkout, delete file and then checkin to a new format or object. However, the files are not actually moved at all (the files stays in the same store - same physical machine location). The command simply changes the metadata so that the files no longer appear to belong to the original object but now belong to the new object.

```
modify businessobject BO_NAME move [format FORMATNAME] from BO_NAME FILE_TO_COPY;
```

where FILES_TO_COPY is one of:

```
format FORMATNAME

[format FORMATNAME] file FILENAME{,FILENAME}

all
```

Files are copied (appended) from the object specified in the from clause (the “from object”) to the object being modified (the “to object”). If a format is specified just after the keyword `move`, the files will be moved to that format. Otherwise, each file will keep its existing format as long as that format is available in the to object. If not, it will use the default format of the to object.

If only a format is given for the from object (part of FILES_TO_COPY), all files of that format are moved. If no format is specified, then the default format is assumed. If any of the specified files is not found with the specified or assumed format, an error is issued and no action is taken.

No triggers are fired when files are moved in this manner.

Files in DesignSync stores cannot be manipulated in this manner. The modify bus command with the move keyword will fail when attempted on a Business Object governed by a policy that uses a DesignSync store.

Renaming Files

You can rename a file that is checked into a business object without doing a file checkout and checkin replace. The renaming is done by changing the 'user visible' filename in the metadata. The actual physical file remains hashed in some store and is not touched. Renaming a file in this manner results in significant performance gain.

File renaming does not involve a file copy or file transfer operation. This feature is provided only for MQL. It cannot be used from the thick client, the PowerWeb GUI or the Studio Customization Toolkit.

The syntax is:

```
modify bus TYPE NAME REVISION rename [format FORMATNAME][!|not]propagaterename]
file FILENAME TOFILENAME
```

where the default is `propagaterename`.

Renaming Files Inherited in a Revision Sequence

During file rename, the modifier `propogaterename` controls renaming of files as follows:

- If the `propogaterename` modifier is used, subsequent revisions inheriting from this file will have the renamed file name.
- If the `not propogaterename` modifier is used, file rename is effective for the current revision only. Subsequent revisions inheriting from this file will have the old file name.

During file rename, history is added to the business object where the file is checked in:

```
history = rename file - user: <name>  time: <date-time>  state:
<state name> format: <format> file: <filename> to file :
<filename>
```

The history indicates file renaming and that metadata operation has been done on a business object. In addition, the history record is a selectable as follows:

```
print bus T N R select history.newname;
modify bus TNR delete history event newname
```

File renaming is a simple operation governed by these rules:

- `TOFILENAME` cannot be empty.
- `TOFILENAME` cannot be a name that already exists in a business object.
- `TOFILENAME` cannot contain these special characters: / \ **: * ?** " < > | **@**

Modifying Legacy Data

After migrating legacy data from an external or legacy database to Collaboration and Approvals, Business Administrators can modify the following properties of the object to reflect the same data as the external system:

- current state
- originated date
- modified date
- start, end, and actual dates of all states in the policy.
- duration values of all states in the policy

For example, a Purchase Order that has been completed in a legacy system should not be set to the initial state of a Purchase Order in Collaboration and Approvals. The imported Purchase Order should immediately be modified to a valid state of the policy to accurately reflect the complete status.

You must be a Business Administrator to make these types of modifications to business objects. These changes can also be made at the time of object creation with the add bus command.

These types of modifications are generally performed by a program, so that other events occurring due to the migration are not reflected in the dates or state set. Since these commands are intended for use within a program object, it is up to the program to handle any errors. An error will result if the context user is not a business administrator, or if the date is not specified in the required formats.

In some cases, you may want the business objects to act as a placeholder, representing the actual objects in the external system that is constantly changing. You can update the business objects on a regular basis by resetting the current state as well as the start, end and duration values of the states. Modifying a business object this way avoids the need for promoting the object through the lifecycle, which would result in each state reflecting the date of the import/promotion and not the date from the legacy system. Each of these (current state, start date, end date, and duration) can be changed independently, with no effect on each other. If you want to maintain these values to be consistent with each other, you must change them all. There is no impact on the lifecycle of these objects if these values are not kept consistent, except for queries or webreports that depend on these values.

When you modify migrated objects in this manner the following do not occur:

- Triggers do not fire
- History records are not added
- Other information associated with a state, like start date, end date, actual date, and duration are not changed
- No signature data is affected

To modify the current state of a business object, use the modify business object command.

```
modify businessobject BO_NAME [MOD_ITEM {MOD_ITEM}];
```

- BO_NAME is the business object whose state you are modifying.
- where MOD_ITEM is:

```
current NEW_STATE_NAME
```

```
state STATE_NAME STATE_ITEM
```

- NEW_STATE_NAME is the name of the state you want to show as the current state of the business object
- STATE_NAME is the name of the state you want to modify with new STATE_ITEMs.
- STATE_ITEM is a State subclause which provides additional information about the state. The different STATE_ITEM clauses are as follows:

```
schedule SCHEDULE_DATE
```

```
actual ACTUAL_DATE
```

```
start START_DATE
```

```
end END_DATE
```

```
duration N
```

- SCHEDULE_DATE is the date the object is scheduled for promotion from (or completed) this state.
- ACTUAL_DATE is date when the object was last promoted/demoted into this state.
- START_DATE is the date when the object first reached this state.
- END_DATE is the date when the object last left this state.
- duration N is the number of days the object was in this state.

See also *Organization and Project-Based Access* in Chapter 3.

Adding or Removing Business Object Ownerships

Business objects can have multiple owners. To add an ownership to or remove an ownership from a business object:

```
modify businessobject OBJECTID add|remove ownership
ORGANIZATION PROJECT [for COMMENT] [as ACCESS];
```

- ORGANIZATION is a "role" object that represents an organization.
- PROJECT is a "role" object that represents a project.
- COMMENT is a short string that is primarily used for annotation.
- ACCESS is a comma-separated list of security tokens.

Each of these fields may be specified when providing an ownership. The first three (ORGANIZATION, PROJECT, and COMMENT) together provide a unique identifier for the ownership entry. Tokens following "for" constitute the ownership comment; those following "as" constitute the access mask.

The following are some examples of usage:

```
modify bus OBJECTID add ownership "MyCompany" "MyProject"
for "ownership”;
modify bus OBJECTID add ownership "Supplier1" "MyProject"
for "Project Applicability";
modify bus OBJECTID add ownership "Supplier2" "MyProject"
as show,read,checkout;
modify bus OBJECTID remove ownership "Supplier1"
"MyProject" for "Project Applicability";
```

Adding Ownership Inheritance

Several business objects can inherit ownership from a common parent object. To specify ownership inheritance:

```
modify businessobject OBJECTID add ownership OWNERSHIP_ID;
```

The full OWNERSHIP_ID is defined as:

```
| ORGANIZATION PROJECT [for COMMENT] [as ACCESS]|
| businessobject NAME [for COMMENT]             |
| relationship OBJECT_ID [for COMMENT]          |
```

Ownership may be selected off of either business objects or relationships.

To specify ownership inheritance from a business object:

```
modify businessobject OBJECTID add ownership bus NAME [for
COMMENT];
```

To specify ownership inheritance from a relationship:

```
modify businessobject OBJECTID add ownership rel OBJECT_ID
[for COMMENT];
```

The following are some examples of usage:

```
modify bus T N R add ownership bus Part 1239291 - for "parent
  workspace";
modify bus T N R add ownership rel 3423.2366.4544.456;
modify bus T N R add ownership bus 3243.34.5365.564 as read;
```

Expand Business Object

The Expand Businessobject command enables you to view connected objects. Depending on the form of the command that you use, you can list the objects that lead to an object, from it, meet selected criteria, or are of a specific type. Then you can place the list of objects into/onto a set, or output the information to a file.

The Expand Businessobject command also allows you to select properties of either connected business objects or the relationship which links them. To expand a business object use the following syntax:

```
expand businessobject BO_NAME [from [relationship PATTERN]
[type PATTERN] to [relationship PATTERN] [type PATTERN];
```

```
fixedpoints ID{,ID}

withroots |
| activefilters [reversefilters] |
| filter PATTERN [reversefilters] |
| preventduplicates [expression EXPRESSION] |

[recurse [to | N |]] [| [ into|onto ] set NAME |]
| all |
| end |
| rel |

[SELECT_BO] [SELECT_REL]

[DUMP [RECORDSEP] [tcl][output FILENAME]

terse

limit N

size N
```

The Expand Businessobject command is similar to using the Star or Indented browser in Matrix Navigator. Use of the clauses that specify criteria is like applying a filter in one of the relationship browsers. In addition to a list of connected business objects that meet the specified criteria, the Expand Businessobject command provides information about the connections. The level number of the expansion and the relationship type of the connection are provided along with the business object name, type, and revision.

Using a size of 0 is the same as not using the Size clause; the expand is streamed using the value of MX_QUERY_PAGE_SIZE, if set. For information on query page size, refer to *Live Collaboration Administration Guide : Optional Variables*.

The following example shows basic use of the Expand Businessobject command. It tells us that there is one drawing attached to a component (with the relationship of “Drawing to”), a comment (with the relationship of “Comment from”) and a markup (with the relationship of “Markup from”). The number “1” on each line indicates the expansion level.

```
expand businessobject Drawing 726592 A;
   1 Drawing to Component 726592 A
   1 Comment from Comment 012528 1
   1 Markup from Markup 002670 1
```

When using the Expand Businessobject command you must provide a business object specification as your starting point. Depending on the keyword (and values) that follow, you will *expand* in one direction only or in both. In addition, you can use the `recurse` argument to indicate that you also want to expand the business objects connected/related to the initially specified business object.

The expansion happens one level at a time, and any relationships/business objects that do not satisfy the Where clause are removed from the list and those expansion paths are abandoned. Consider the following example:

Assembly -> Part -> Doc

`expand bus Assembly from recurse 2 where 'type == Doc'` will return nothing, since the first level expansion finds Part, which is not of type Doc, so the second expansion is not performed at all.

Each clause is described in the sections that follow.

From Clause

When you use the From clause, you expand away from the starting object. This gives you all of the related objects defined as the TO ends in a relationship.

Think of the starting object as the *tail* of the arrow—you are looking for all of the arrow *heads*.

For example, assume you have an object that contains a component used within larger assemblies. If you have the name of the component, you might want to know which objects (larger assemblies) it goes into:

```
expand businessobject Component "Mini Numeric Keypad" C
from;
```

This might produce a list of objects that could include a calculator, telephone, and VCR. Since no other criteria is specified, this command gives you ALL objects that occupy the TO end of a relationship definition.

To Clause

When the To clause is used, the named business object is used as the starting point. However, it is assumed that the object is defined as the TO end of the relationship definition. Therefore, you are looking for all objects that lead to the named object—the objects defined as the FROM ends in a relationship definition.

Using the To clause can be useful to work backward. For example, you may want to determine components that make up a particular subassembly. If you know the name of the subassembly object, you can do this by writing a command similar to:

```
expand businessobject Component "Mini Numeric Keypad" C to;
```

This might produce a list of objects that contain buttons, plastic housings, and printed circuit boards. All related objects defined as the FROM connection end are listed. This may include additional objects that you don't need. To help reduce the number of objects listed and to allow you to look in both directions, you can use the Relationship or Type clauses of the Expand Businessobject command.

Relationship Clause

This clause returns all objects connected to the starting object with a specific relationship. The command can be made to be more particular by specifying the direction of the relationship and/or the types of objects to return.

The Relationship clause is useful when you are working with an object that may use multiple relationship types. If the starting object can connect with only one relationship, this clause has the effect of listing all of the connection ends used by the starting object. It does not matter if the end is defined as a TO end or a FROM end. Only the relationship type is important.

For example, assume you have an object that contains a drawing. This drawing may use a number of relationships such as a User Manual Relationship, Design Relationship, Marketing Relationship, and Drawing Storage Relationship. You may want to examine all objects that use a particular relationship type. For example, you might want to learn about all objects that have used the drawing for marketing. To do this, you might enter a command similar to:

```
expand businessobject Drawing "Mona Lisa" 1
relationship "Marketing Relationship";
```

This command lists all objects connected to the Drawing with the Marketing Relationship relationship. It searches through all connections that use the Marketing Relationship definition for the named business object. If it finds the object, MQL displays the other connection end and identifies the end's directional relationship. It does not matter if the related objects are defined as the FROM end or the TO end of the relationship. Only the relationship type is important.

Type Clause

This clause displays all related objects of a specific object type. This clause is useful when you are working with an object that may be connected to multiple object types. (If the starting object can connect with only one object type, this form is similar to the Relationship form using a wild character pattern.) For example, assume you have an object that contains a training course. To this object you might have related objects that are of type Evaluation, Student, Materials, and Locations. You may want to examine all the Evaluation type objects to trace the course's progress in meeting student needs. You might enter a command similar to:

```
expand businessobject Course "Money & Marketing" 3 type Evaluation;
```

This command lists all related objects that have the Evaluation type. Those objects might belong to multiple relationship types (such as Professional Evaluation Relationship or Student Evaluation Relationship). It does not matter if the related objects are defined as the FROM end or the TO end of the relationship. Only the object type is important to locate and display the output objects.

If the specified type has sub-types, these are also listed. For example:

```
expand bus Assembly 12345 1 type Part;
```

might return all Component and Assembly, as well as Part objects.

Select To/From Patterns

Use `select from[].to[]` or `to[].from[]` clauses (with print bus, print connection, temp query, add query, and expand bus commands) to navigate connections and obtain information about related business objects. Two simple examples are:

- `print bus T N R select from.to.owner from.to.current;`
  This command looks at relationships pointing *from* the specified object and returns the owner and current state of the object on the TO end.
- `print bus T N R select to[Employee].from.name;`

This command looks specifically at relationships of type Employee pointing *to* the specified object and returns the name of the object on the FROM end.

Adding Relationship and business Object Patterns

You can also include a list of comma-delimited strings, which may include wildcard characters, to filter objects in two ways:

- to express the *relationship(s)* to expand on. For example:
  `print bus T N R select from[R*,N*].to.owner from.to.current;`
- to further filter by the business *type(s)* on the other end, such as:
  `print bus T N R select to[Employee].from[T*].name;`

The results are similar to those found with `expand bus`. The following example illustrates pattern use with a relationship:

```
print bus Guide "Owners Manual" 2004 select to[N*,S*].from.name
to[N*,S*].from.revision;
business object Guide Owners Manual 2004
    to[New].from[Chapter].name = Chapter 2
    to[New].from[Photo].name = '2004 Red Camry'
    to[Same].from[Chapter].name = Chapter 4
    to[New].from[Chapter].revision = 5
    to[New].from[Photo].revision = 1
    to[Same].from[Chapter].revision = 2
```

One thing to note about relationship patterns is if there are no connections that meet the criteria, such as if `to[X*].from.name` were specified, the output would be similar to:

```
business object Guide Owners Manual 2004
    to[X*] = False
```

You can add a pattern for the type, further filtering the output as follows:

```
print bus Guide "Owners Manual" 2004 select to[N*,S*].from[Ch*].name
to[N*,S*].from[Ch*].revision;
business object Guide Owners Manual 2004
    to[New].from[Chapter].name = Chapter 2
    to[New].from[Chapter].revision = 5
    to[Same].from[Chapter].name = Chapter 4
    to[Same].from[Chapter].revision = 2
```

There are two things to note about type patterns:

- Not only are the object types that meet the pattern criteria returned, but also those objects that are a derived type matching the pattern. For example, if the type Page were a sub-type of Chapter in the above example, the output would include Page type objects that were connected. This is similar to using the expand check box in a query dialog in Matrix Navigator.
- If there are no connected objects that meet the criteria, such as if `to.from[X*].name` were specified, there is no output.

Using Where Clauses to Filter Connections and Business Objects

To include more extensive filtering, you can also include Where clause criteria in the square brackets of `select from[].to[]` or `to[].from[]` clauses. The relationship or type name pattern is specified first, and then may be followed by the Where clause, the fields delimited by the "|" character. For example:

```
print bus Drawing My Drawing 0 select
   from[D*|attribute[RelCount] ~~ "*1"];
```

The Where clauses are handled exactly as `expand bus` would handle them. For example, consider a business object Drawing “My Drawing” 0 which has 3 “Drawing of” relationships coming from it, and 2 “Part of Drawing” relationships. On the “Drawing of” relationships, the attribute values for RelCount are 1, 11, and 12. The 3 “Drawing of” relationships point to objects of types Drawing, Sketch and Drawing Markup and these objects have attribute Count values of 1, 11, and 12. The following shows all of this information (and a bit more):

```
print bus Drawing “My Drawing” 0 select from.type
   from[Drawing of].attribute[RelCount]
   from[Drawing of].to.type
   from.to.attribute[Count];
 business object  Drawing "My Drawing" 0
    from[Drawing of].type = Drawing of
    from[Drawing of].type = Drawing of
    from[Drawing of].type = Drawing of
    from[Part of Drawing].type = Part of Drawing
    from[Part of Drawing].type = Part of Drawing
    from[Drawing of].attribute[RelCount] = 1
    from[Drawing of].attribute[RelCount] = 12
    from[Drawing of].attribute[RelCount] = 11
    from[Drawing of].to.type = Drawing
    from[Drawing of].to.type = Drawing Markup
    from[Drawing of].to.type = Sketch
    from[Drawing of].to.attribute[Count] = 11
    from[Drawing of].to.attribute[Count] = 1
    from[Drawing of].to.attribute[Count] = 12
    from[Part of Drawing].to.attribute[Count] = 1
    from[Part of Drawing].to.attribute[Count] = 1
```

But, if we want to get back ONLY the “Drawing of” relations with specified values of RelCount, we could use:

```
print bus Drawing "My Drawing" 0 select from[Drawing
of|attribute[RelCount] ~~ "*2"].attribute[RelCount];
business object  Drawing "My Drawing" 0
    from[Drawing of].attribute[RelCount] = 12
```

Or if we want to specify the type of other objects for “Drawing of” relations:

```
print bus Drawing "My Drawing" 0 select
   from[Drawing of].to[*|type ~~ "*up*"].type;
business object  Drawing "My Drawing" 0
    from[Drawing of].to[Drawing Markup].type = Drawing Markup
```

Or we want to specify attribute values on the other object for “Drawing of” relations:

```
print bus Drawing "My Drawing" 0 select
   from[Drawing of].to[*|attribute[Count] ~~ "*1"].attribute[Count];
business object  Drawing "My Drawing" 0
    from[Drawing of].to[Drawing Markup].attribute[Count] = 11
    from[Drawing of].to[Sketch].attribute[Count] = 1
```

Withroots Keyword

You can include the root object along with all selectables to make it easier for client apps to identify a hierarchy.

Examples Without the `withroots` Keyword	Examples With the `withroots` Keyword

`MQL<3>expand bus t2 t2-1 0 recurse to 2;`
`1 r2 to t2 t2-2 0`
`2 r2 to t2 t2-3 0`
`2 r3 to t2 t2-2-2 0`
`2 r3 to t2 t2-2-3 0`
`2 r1 from d12 d12-2 0`
`2 r1 from d11 d11-3 0`
`2 r2 from t2 t2-1 0`
	`MQL<7>expand bus t2 t2-1 0 recurse to 2 withroots;`
	`0 t2 t2-1 0`
	`1 r2 to t2 t2-2 0`
	`2 r2 to t2 t2-3 0`
	`2 r3 to t2 t2-2-2 0`
	`2 r3 to t2 t2-2-3 0`
	`2 r1 from d12 d12-2 0`
	`2 r1 from d11 d11-3 0`
	`2 r2 from t2 t2-1 0`

`MQL<16>expand bus t2 t2-1 0 recurse to 2 select bus id dump;`
`1 r2 to t2 t2-2 0 20768.23721.38088.36415`
`2 r2 to t2 t2-3 0 20768.23721.38088.39031`
`2 r3 to t2 t2-2-2 0 13576.55552.30896.29472`
`2 r3 to t2 t2-2-3 0 6384.39445.50345.7641`
`2 r1 from d12 d12-2 0 20768.23721.38088.32444`
`2 r1 from d11 d11-3 0 20768.23721.38088.24862`
	`MQL<18>expand bus t2 t2-1 0 recurse to 2 withroots select bus id dump;`
	`0 t2 t2-1 0 20768.23721.38088.35693`
	`1 r2 to t2 t2-2 0 20768.23721.38088.36415`
	`2 r2 to t2 t2-3 0 20768.23721.38088.39031`
	`2 r3 to t2 t2-2-2 0 13576.55552.30896.29472`
	`2 r3 to t2 t2-2-3 0 6384.39445.50345.7641`
	`2 r1 from d12 d12-2 0 20768.23721.38088.32444`
	`2 r1 from d11 d11-3 0 20768.23721.38088.24862`

Filter and Activefilter Clauses

The `Filter` clause of the Expand Businessobject command allows you to specify an existing filter(s) that is defined within your context to be used for the expansion. You can use wildcard characters or an exact name. For example:

```
expand bus Assembly 12345 1 filter OpenRecs;
```

In addition, you can use the Activefilter clause to indicate that you want to use all filters that are enabled within your context. For example:

```
expand bus Assembly 12345 1 activefilter;
```

Recurse Clause

Once you have a list of related objects, you may also want to expand these objects. This would be like using the There and Back feature in the Star browser of Matrix Navigator. The Recurse clause of the Expand Businessobject command expands through multiple levels of hierarchy by applying the Expand command to each business object found:

```
recurse [to | all | [preventduplicates]]
            | N   |
```

`N` is any number indicating the number of levels that you want to expand.

`all` indicates that you want to expand all levels until all related business objects are found.

Include `preventduplicates` in the Recurse clause to avoid *expanding* objects more than once. If an object is connected at more than on level, it is reported at all levels where it is connected, but expanded only once. This reduces the amount of data returned and improves performance.

It is recommended that you specify a recursion level with the Recurse clause. Recurse to all may be time-consuming. Recurse alone (without to) defaults to recurse to all.

This example shows use of the Expand Businessobject command with the Recurse (to all) clause:

```
MQL<40> expand businessobject Drawing 726592 A recurse to all;

1 Drawing from Component 726592 A
  2 As Designed from Assembly 726509 A
    3 As Designed to Component 726593 A
      4 Drawing from Drawing 726593 a
        5 Comment from Comment 123528 1
          6 Comment to Drawing 726592 A
            7 Markup from Markup 002670 1
    3 As Designed to Component 726594 A
      4 Drawing from Drawing 726594 A
        5 Markup from Markup 002590 1
    3 As Designed to Component 726595 A
      4 Drawing from Drawing 726595 A
        5 Markup from Markup 002733 1
      4 Note from Note 014258 1
        5 Note to Component 726591 B
          6 Note from Note 012499 1
          6 Photo from Photo 017012 1
```

```
6 Photo from Photo 117012 1
6 Drawing from Drawing 726591 C
  7 Markup from Markup 002715 1
6 As Designed from Assembly 726596 B
  7 As Designed to Component R0056-48 A
  7 As Designed from Assembly 726590 A
    8 As Designed to Component R0045-62 B
    8 As Designed to Component 23348S C
    8 As Designed to Component W2236-8S A
    8 Comment from Comment 013070 1
    8 Drawing from Drawing 726590 B
    8 Layout from Layout 100265 F
    8 Analysis from Stress Analysis 100345 D
      9 Specification from Specification 2278 C
        10 Specification Change from Spec. Change Notice 00247 1
        10 Specification Change from Spec. Change Notice 00252 1
  7 Drawing from Drawing 726596 B
```

This example shows use of the Expand Businessobject command with the Recurse (to all preventduplicates) clause.

```
MQL<15> expand businessobject Part MyPart A from recurse to all preventduplicates;

1 As Designed to Widget MyWidget A
  2 As Designed to Component MyComponent A
    3 As Designed to Component Component007 A
1 As Designed to Component Component007 A
  2 Drawing from Drawing MyDrawing A
    3 Specification from Specification MySpecification A
      4 As Designed to  Part MyPart A
```

In this example, object Component Component007 A is connected at multiple levels. It is reported at all levels where it appears but is expanded only once.

Where to Use - recurse to end and recurse to relationship

Many times it is important to get a list of all the objects connected at certain end points in a structure. For example, part of the process for analyzing an ECR is to determine which Products are affected by the Parts listed in the ECR for change. Getting a list of all these products helps determine which Change Boards need to be informed of the change.

You can use the following syntax to navigate from any point in a structure (in either direction) to a “leaf” without examining the nodes in between:

```
expand bus [| from |] relationship PATTERN [recurse to | end |]
           | to   |                                  | relationship |
```

- PATTERN can include wildcard characters and include more than 1 relationship name or pattern, separated by a comma but no space.

These commands use a hierarchical query to perform a recursive expansion with a single SQL command, making the navigation extremely efficient, with only the end-nodes returned to the server. However, this means that the intermediate nodes in the expansion cannot be examined in any way. If a Where clause or filter is specified in the expansion, it is only applied to the end-nodes.

When `show` access is enabled, it is checked on the root of the expansion and *only the children that are returned*. In normal expands, show access is also checked on intermediate nodes, and may block access to some branches in the structure. Because of this, recurse to end/rel could potentially return more results than the leaf-nodes in a corresponding recurse to all.

All supported databases generate hierarchical/recursive SQL to implement recurse to end/recurse to relationship functionality.

Recurse to End

You can include Recurse To End with Expand Bus so that the system navigates directly to all end nodes of the structure. You could use this form of the command to identify “end items” in a BOM structure. For example, to find all references of Part MyWidget 1 following relationships of type EBOM or BOM, use:

```
expand bus Part MyWidget 1 to rel EBOM,BOM recurse to end;
```

Recurse to Relationship

Use recurse to relationship to only return those relationships (and their target objects) whose relationship type matches the last relationship in the relationship pattern. For example, to expand the structure following both EBOM and BOM relationships, but returning only those objects connected with the BOM relationship, use:

```
expand bus Part MyWidget 1 to rel EBOM,BOM recurse to rel;
```

Select operations may be applied normally on results from the new expand operations.

The Expand limit is applied only to the returned nodes of the expansion.

DB2 does not automatically detect cycles that cause infinite loops. It terminates if the SQL recurses MX_MAX_RECURSE_TO_END_LEVEL levels deep (set to 50 by default). Previously, this would have executed a normal expand, but returned only the leaf/end objects.

Use Cases

The following scenarios can be addressed using the recurse to end/rel commands:

1. Determine if a Part is ultimately used in a Product. A trigger or check may be run that traverses up a BOM structure looking for at least 1 product before allowing a Part to be promoted to the release state. This ensures that all Parts that are released will be picked up by the ERP planning which is always done at the Product level.
2. Find all Leaf Nodes (Components) in an EBOM. Get a list of actual end items in a BOM Structure. These end items represent the total number of unique parts in a system.
3. Find all Leaf Nodes (Components) in an EBOM that are “purchased”. The indication for “purchased” may be a different type (relationship or object type) or it may just be an attribute value on the item.
4. Report how many Parts in a structure have Drawings. This information can be used along with the total number of parts in a structure to calculate/report on the percent complete of the design.

Examples

Consider this structure of alternating relationships:

```
expand bus Part A1-v1 0 rel EBOM,BOM,"Manufactured by" from
recurse to all;
1  EBOM  to  Part  B1-v1  0
  2  BOM  to  Part  C1-v1  0
    3  EBOM  to  Part  D1-v1  0
      4  BOM  to  Part  E1-v1  0
```

To recurse through both EBOM,BOM relationships, but only report the end-items, use recurse to end:

```
expand bus Part A1-v1 0 rel EBOM,BOM from recurse to end;
1  BOM  to  Part  E1-v1  0
```

To recurse through both EBOM, BOM relationships, but only report relationships of type BOM, use recurse to rel, placing the BOM relationship last in the relationship list.

```
expand bus Part A1-v1 0 rel EBOM,BOM from recurse to rel;
1  BOM  to  Part  C1-v1  0
1  BOM  to  Part  E1-v1  0
```

Use cases 3 and 4 relate to cases where the "terminating condition" is when you expand on a node that has a relationship of a different type coming from it. In case 3 (assuming "purchased" items are indicated by a relationship "Purchased From" that connects to the vendor), one could find them with:.

```
expand bus Assembly MyAssembly 1 from rel
"EBOM,BOM,Purchased From" recurse to rel select rel from.id;
```

The above would traverse all 3 relationship types in the from->to direction and return the relationships, and vendors, but the "select rel from.id" would return the object ID on the FROM end of the "Purchased From" relationships — namely the items in the BOM which are purchased.

In case 4, you could use the same approach to look for "Specification" relationships that connect BOM elements with drawings:.

```
expand bus Assembly MyAssembly 1 from rel
"EBOM,BOM,Specification" recurse to rel select rel from.id;
```

The objects returned by this expand would be the drawings themselves, but the "select from.id" would similarly return the object IDs of the BOM elements on the FROM ends of the Specifications relationships - namely, the items that point to the drawings.

In either of the above cases, if you really wanted the vendors or the drawings themselves, you could get the object ID on the TO end of the appropriate relationships by using "select rel *to*.id".

Set Clause

Once you have a list of related objects, what do you do with them? In some cases, you will simply search for a particular object and will not need to reference the output object again. In other cases, you may want to capture the output business objects and save them for another time. That is the purpose of the Set clause of the Expand Businessobject command.

When the Set clause is included within the Expand Businessobject command, the related objects are placed within a set and assigned a set name. This set may already contain values or it may be a new set created for the purpose of storing the output.

When it is an existing set, the previous values are either replaced or appended, depending on the keyword you use. If the Set clause begins with the keyword `INTO`, the existing set contents are discarded and only the current output is saved. If the Set clause begins with the keyword `ONTO`, the new output is added onto the existing set contents. (This is the same as when working with queries, described in *Queries* in Chapter 6.)

The Set clause is optional. It can be used in conjunction with the other Expand Businessobject clauses according to the following guidelines:

- You can specify a set (with the Set clause) *OR* select properties for output (with the Select clause) but not both in the same command.
- You can specify that the results are saved in a set *OR* request that output be saved in a file (with dump/output clauses), but not both in the same command.

Structure Clause

If you expand an object using the `filter, from, to, relationship`, and/or `type` clauses of the Expand Businessobject command, you have defined a *structure* that is not necessarily easy to define again. If you want to reuse this structure you can save it to your workspace using the `Structure` clause. For example:

```
expand bus Assembly 12345 1 from relationship "As Designed" filter Part structure
"Assigned Parts";
```

A structure stores all the information—including objects, relationships, and relative level number — necessary to recreate the printed output. The Structure clause is optional. It can be used in conjunction with any of the other Expand Businessobject clauses, including select, output, and set.

If an object has been disconnected or deleted, it is no longer listed as part of the structure. If the deleted or disconnected object was expanded as part of the structure, its "child" objects would also be removed from the saved structure.

On the other hand, if other objects are connected after the structure is saved, they will not automatically appear in the output of the print structure command.

Also, if a business object (and its relationships) are moved to another vault, the structure will usually still remember them. However, if an object is moved to a vault in which no other object in the structure currently resides, the object will not be found. If this object is the "top" object in the structure (that is, the object that was expanded to create the structure), an error will occur if one tries to do anything with the structure. If the "lost" object is a child in the structure, it and all objects under it will not be listed as part of the structure.

Another expand command would need to be executed with the Structure clause to update the structure, in any of these cases. Refer to *Working with Saved Structures* in Chapter 5.

SELECT_ BO and SELECT_REL Clauses

The Expand Businessobject command provides a means of displaying information about the connected business objects. Information contained in the connected business object can be selected as well as data on the relationship that connects the objects. It works in a similar manner on the expand command as it does on the print command. The differences are:

- a list of the selected values, one for each object or relationship that meets the criteria, is presented.
- the keyword `businessobject` or `relationship` must be used with the Select clause before the requested items so that the Live Collaboration knows from where the information is to come.

For example:

```
expand bus Assembly "MTC 1234" C select bus description select relationship id;
```

The system attempts to produce output for each Select clause input, even if the object does not have a value for it. If this is the case, an empty field is output. For information on the `Select` clause, see *Select Clause*.

When you use the Select clause to expand on both business objects and relationships, business object selectables are always listed before relationship selectables in the output, even if they appear in the opposite order in the select list.

Using a Where Clause

When expanding business objects, you can use where clauses as another means of specifying which objects to list. When using this technique, you must always insert `select bus` or `select rel` before the where clause. This tells the system whether to apply the where clause to the relationships it finds or to the objects on the other end of those relationships. It does not, however, select and print out any information about the connected objects or expanded relationships (unless another SELECT_BO or SELECT_REL clause is included.

Applying the Where Clause to Business Objects

```
mql expand bus Type_one Object_1 Rev_1
   select bus
   where ' (type == "Type two") && (1 == 1) ' dump
```

Applying the Where Clause to Relationships

Note that since the Where clause works off the rel, you must use ‘to.type’.

```
mql expand bus Type_one Object_1 Rev_1
   select rel
   where ' (to.type == "Type two") && (1 == 1) ' dump
```

Alternatively, you can let `expand bus` check the object type for you (see Rules of thumb below):

```
mql expand bus Type_one Object_1 Rev_1
   type "Type two"
   select rel
   where ' (1 == 1) ' dump
```

Note that the first command above is a bit more general, ignoring the direction of the relationship, whereas the second will only find Type two’s at the end of relationships pointing *from* Object_1 *to* the Type 2 object.

Reworking the first syntax for more specificity:

If you only want objects connected to Object_1 by the relationship Rel, add ‘rel Rel’:

```
mql expand bus Type_one Object_1 Rev_1
   rel Rel
   select bus
```

```
where ' (type == "Type two") && (1 == 1) ' dump
```

And if you only want to look at the relationship Rel coming *from* Object_1, add ‘from rel Rel’. This is equivalent to the REL syntax above:

```
mql expand bus Type_one Object_1 Rev_1
   from rel Rel
   select bus
   where ' (type == "Type two") && (1 == 1) ' dump
```

Rules for `Where` Clauses

- Always enclose the entire `Where` clause in single quotes and a space.
- Always surround operators with spaces.
- In `expand bus`, if you can safely be specific about the relationship type and direction to be expanded, always use the syntax:

  ```
  expand bus T N R <direction> rel <relnameList> select <rel-or-bus>
  where...
  ```

  because it is quicker. It will return only the correctly named and directed relationships and filter those relationships (or the business objects on the other ends for `select bus`) through the Where clause.
- If you also know which Types you are looking for, use the `type` modifier for `expand bus` rather than relegate it to the `Where` clause:

  ```
  expand bus T N R from rel Rel type "Type two"
    select rel where ' <other qualifiers on the rel> '
    select bus where ' <other qualifiers on the TO obj's> '
  ```

Terse Clause

You can specify the Terse clause so that object IDs are returned instead of type, name, and revision. This is done with the Terse clause. For example, the following command returns a list of object IDs for objects connected to the specified Part:

```
expand businessobject Part "35735" A terse;
```

Limit Clause

Since you may be accessing very large databases, you should consider *limiting* the number of objects to display. Use the Limit clause to define a value for the maximum number of objects to include in the expansion. For example, to limit the expansion to 25 objects, you could use a command similar to the following:

```
expand businessobject Part "35735" A limit 25;
```

Size Clause

You can specify a Size clause in a expand bus (and query) command to enable streaming and return the specified number of objects per page. For example, to expand a business object and view the only 50 results at a time using the streaming capability, use the following:

```
expand businessobject Part "35735" A size 50;
```

Streaming can be enabled for all expands (and temporary queries) through the MX_QUERY_PAGE_SIZE environment variable, through classes in the Studio Customization Toolkit, or at runtime by using the size clause in MQL temp query or expand bus commands.

Using a size of 0 is the same as not using the Size clause and the expand is streamed using the value of MX_QUERY_PAGE_SIZE or the classes in the Studio Customization Toolkit, if set. For information on MX_QUERY_PAGE_SIZE, see the *Installation Guide : Optional Variables*.

Enabling streaming through the Java classes usually provides the greatest improvement in performance and memory consumption.

Connect Business Object

The Connect Businessobject command links one business object to another. For example, you could have a business object that contains information about a particular course. That information might include the course content, schedule date, instructor, student list, cost, and so on. As each student enrolls in the course, you might create a business object for that student. This object might include the student's background, experience, and personal information.

After a student record object is created, it stands alone with no relationships attached to it. However, if you view the course object, you might want to see the student objects associated with it. Also, if you view the student object, you might want to see the course objects associated with that person.

Use the Connect Businessobject command to establish the relationship between the business object containing the student record and the object containing the course information:

```
connect businessobject BO_NAME relationship NAME | to   | BO_NAME [preserve]
                                                 | from |
[ATTRIBUTE_NAME VALUE {ATTRIBUTE_NAME VALUE}] [SELECT [DUMP] [RECORDSEP] [tcl]
[output FILENAME]];
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object. It can also include the `in VAULTNAME` clause, to narrow down the search.
- `NAME` is the name of the relationship type to use to connect the two named business objects. If the relationship name is not found or defined, an error message will result.
- `ATTRIBUTE_NAME VALUE` indicates attributes assigned to the relationship you are creating.

The Connect Businessobject command has two forms: TO and FROM. The form you choose depends on the placement of the two objects you are connecting. You specify which business object will be associated with the TO end and which will be associated with the FROM end. For example, to assign a student to a course, you might use the TO form of the Connect Businessobject command:

```
connect businessobject Student "Cheryl Davis" Sophomore
   relationship "Student/Course Relationship"
      to Course "C Programming Course" 1;
```

When this command is processed, it will establish a "Student/ Course Relationship" between the course object and the student object. Cheryl Davis is assigned to the FROM end and the C Programming course is assigned to the TO end. You can think of Cheryl as *leading to* the course object.

You also could think of the course as coming *from* the student object(s). In other words, you can define the same relationship using the FROM form of Connect Businessobject command:

```
connect businessobject Course "C Programming Course" 1
   relationship "Student/Course Relationship"
      from Student "Cheryl Davis" Sophomore;
```

When this relationship is established, Cheryl's object is again assigned to the FROM end and the course is assigned to the TO end.

If you are defining equivalent objects, either object could be defined as the TO end or the FROM end. However, in hierarchical relationships, direction is important. With these relationships, you should consult the relationship definition before creating the connection. Otherwise, you may have difficulty locating important objects when needed. If you assigned the wrong object to the connection end, you will have to dissolve the relationship and re-define it.

Using Select and Related Clauses

Frequently, implementation code creates or modifies a businessobject or connection and then immediately needs to fetch new information from it. The select modifier is available so that data can be retrieved as output from the creation (including revise and copy) or modification command itself, removing the necessity to make another, separate server call to get it, thereby improving performance. In the case of add/modify businessobject, the specified selects are applied to the businessobject being created/modified. In the case of connect bus, the specified selects are applied to the connection being created. The output produced is identical to a corresponding print bus or print connection command.

When you add a Select clause to a connect bus command, the select applies to the connection being created and NOT to the objects on either end.

The output produced is identical to a corresponding print bus command.

Refer to *Select Clause* for syntax details on this and its related clauses.

If these selects are used within a user-defined transaction, they will return data representing the current uncommitted form of the object/connection. The data is collected and returned after all triggers associated with the command have been executed, unless a transaction is active and a trigger is marked deferred, since in this case, the trigger is not executed until the transaction is committed, as shown below:

```
1) Larger transaction is started with any number of commands before the add/
modify/connect command with the trigger.
    2) add/modify/connect transaction is started.
        3) Normal processing of access checking occurs.
        4) If it exists, the Check Trigger is fired.
        5) If Check blocks, then transaction aborts.
           If not, the Override Trigger is fired if it exists.
    6) The Event transaction is then committed regardless of an override or
normal       activity.
    7) If the Event Trigger has a non-deferred Action Program, it is executed.
   8) If the add/modify/connect command included a Select clause, the data is
collected and returned at this point, even though the creation/modification has
not been committed.
9) The larger transaction is committed if all activities and triggers succeed.
10) Finally, if the larger transaction successfully commits, and there is a
deferred    Action Program, it is now triggered. If the larger transaction
aborts, deferred    programs are not executed.
```

Presumably, work would be done using the selected and returned data, between steps 8 and 9 above.

Disconnect Business Object

The Disconnect Businessobject command dissolves a link that exists between one business object and another. For example, a student enrolled in a class might discover that other commitments make it impossible to attend. Since the student is withdrawing from the class, you no longer need a relationship that was established between that student and the class.

Use the Disconnect Businessobject command to dissolve the relationship between the business object containing the student record and the object containing the course information:

```
disconnect businessobject OBJECTID relationship NAME [to|from] OBJECTID
[preserve];
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search.
- `NAME` is the name of the relationship type that exists between the two named business objects. If the relationship name is not found or defined, an error message will result.

The Disconnect Businessobject command has two forms: TO and FROM. The form you choose depends on the placement of the two objects you are disconnecting. In our example, the student was assigned to the FROM end and the course was assigned to the TO end. Therefore, to dissolve this relationship, you can write either of these commands:

```
disconnect businessobject Student "Cheryl Davis"
Sophomore
   relationship "Student/Course Relationship"
     to Course "C Programming Course" 1;
```

```
disconnect businessobject Course "C Programming Course"
1
   relationship "Student/Course Relationship"
     from Student "Cheryl Davis" Sophomore;
```

In the first command, the TO form is used. You are dissolving the relationship of student *to* the course. In the second command, the FROM form is used. You are dissolving the relationship *from* the student.

Regardless of the form you use, once processed, the relationship that was established between the student and the course is removed.

The direction you select makes a difference when you are dissolving relationships. If you reversed the order of the two business objects, the names would not be found. For example, assume you entered this command:

```
disconnect businessobject Student "Cheryl Davis"
Sophomore
   relationship "Student/Course Relationship"
     from Course "C Programming Course" 1;
```

This command assumes that the student is the TO end and the course is the FROM end. If the Live Collaboration searched for this relationship, a match would not be found and an error message would result. You should check to be sure that you know the correct direction before writing your Disconnect Businessobject command.

Checkin Business Object

The Checkin Businessobject command associates one or more files with a particular business object. To check a file into an object, you must first have Checkin privileges. If you do, you can check in the file by using the `checkin businessobject` command.

```
checkin businessobject OBJECTID [format FORMAT_NAME] [store STORE_NAME] [unlock]
[client|server]  [append] {FILENAME};
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object.
- `FORMAT_NAME` is the name of the format to be assigned to the files being checked in. (Formats are defined in the *Format Clause* of the Policy associated with the object Type.) If you are going to include both a format and a store, you must include the format first.

- `STORE_NAME` is the name of the store where the file(s) should be checked in. This can be used to override the default store defined in the policy that determines where a governed business object's files are kept. For example, a business applications's checkin page may be used by several different companies, each with its own file store. The logic on the page could be designed to check which company the current user is employed by, and check the file into the store assigned to that company.If you are going to include both a format and a store, you must include the format first.
- `FILENAME` is the full directory path to the files you want to associate with this business object. When including multiple filenames, separate each with a space. Filename must be specified last.

As an example of using the Checkin Businessobject command, assume you have written procedures for assembling and disassembling a telephone. You now want to associate these procedures with the telephone object. To do that, you might write a command similar to:

```
checkin businessobject Assembly "Phone Model 324" AD
   $MATRIXHOME/telephones/assemble324.doc
   $MATRIXHOME/telephones/disassemble324.doc;
```

In this example, it is assumed that the default format is for document files. If a default format is some other format, you would need to modify the above command to include the Format clause. Let's assume that the two files are standard ASCII text files. Since that is not the default format, the above command would have to be modified as:

```
checkin businessobject Assembly "Phone Model 324" AD
   format "ASCIItext"
   $MATRIXHOME/telephones/assemble324.txt
   $MATRIXHOME/telephones/disassemble324.txt;
```

If someone wants to access the files, the associated format will define the commands used to view, edit, and print the files.

Although the format identifies how the files are to be accessed, it does not necessarily mean that access is permitted. Access is controlled by the policy. Also, editing access can be controlled by the use of exclusive editing locks (see *Locking and Unlocking a Business Object* in Chapter 5).

Checkin Performance

The first MQL checkin after a machine restart, or when the OS file cache has been invalidated, may take some time because of the need to load JVM jars and DLL files. You can prevent this behavior by keeping the necessary files loaded into the OS file cache by periodically launching a small MQL command that executes a JPO in the background.

To accomplish this, follow these steps:

1. Create a C:\tmp\JVMINIT.java file with the following content:

```
import matrix.db.*;
import java.io.*;
public class ${CLASSNAME}
{
   public ${CLASSNAME} () {}
   public int mxMain(Context context, String[] args) throws Exception
   {
      return 0;
   }
}
```

2. To add the JPO to the system, run the following command under the MQL prompt within the admin user context:

```
add prog JVMINIT java file C:\temp\JVMINIT.java;
```

3. Then compile the JPO and grant execute permission on that program to the user guest:

```
compile prog JVMINIT;
mod prog JVMINIT execute user guest;
```

4. In order to run the JPO periodically in the background, you need to use a batch file. You can find a sample batch file (`mqljvminit_sample.bat`) under the *`Modeling_Studio_Install_Path`*`\`*`Platform`*`\code\bin\` directory. The content of the `mqljvminit_sample.bat` file is as follows:

```
@echo off
:beginning
start /wait /b
Modeling_Studio_Install_Path\Platform\code\bin\mql.exe -c "execute
prog JVMINIT;" -d -p
ping -n 300 127.0.0.1 > nul
goto beginning
```

   - Replace *`Modeling_Studio_Install_Path`* with your custom installation path.
   - Replace *`Platform`* with either `intel_a` (for x86 platforms) or `win_b64` (for x64 platforms).

5. To prevent a black command prompt window from appearing, you can launch the batch file with the following `mqljvminit.js` file:

```
var WshShell = new ActiveXObject("WScript.Shell");
WshShell.run("Modeling_Studio_Install_Path\\Platform\\code\\bin\\mqljv
minit.bat", 0, false)
```

   You can find a sample of this file under:
   `Modeling_Studio_Install_Path``\`*`Platform`*`\code\bin\mqljvminit_sample.js`
   Remember to rename the `mqljvminit_sample.bat` file to `mqljvminit.bat` and the `mqljvminit_sample.js` file to `mqljvminit.js`, if you are using those sample files.

There are several possible ways to launch `mqljvminit.js`:

- You can force `mqljvminit.js` to run at machine boot time by using the following Windows registry key:
  `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`
  and adding a new key MQLJVMINIT of type REG_SZ with the value:
  `Modeling_Studio_Install_Path``\`*`Platform`*`\code\bin\mqljvminit.js`
- You can also force `mqljvminit.js` to run at machine boot time by creating a shortcut for it in the Startup dicrectory:
  `C:\Documents and Settings\All Users\Start Menu\Programs\Startup`
- You can launch `mqljvminit.js` manually by simply running it from the *`Modeling_Studio_Install_Path`*`\`*`Platform`*`\code\bin\` directory.

Unlock Clause

When an object first has files checked in, it is probably not locked. However, when files are later checked out to be updated, the object should be locked to prevent other users from editing the file's contents. Checking in the file has the effect of updating the database with the latest version of the file. But typically, the object owner will want to maintain editing control over the file. For example, the owner may be editing a CAD file and the file is still undergoing changes. If the owner checks in the file, others can monitor the process even though they cannot make changes. After the edit is complete the owner would remove the lock. The owner would include the Unlock clause in the Checkin command. Alternatively, if access permitted, the lock could be removed with the `unlock businessobject` command.

If locking is enforced in the object's policy, the checkin will fail if:

- the object is not locked
- the user performing the checkin in is not the one who locked it

When an object is locked, no files can be checked in to the object. This means that attempts to open for edit, as well as checkin, will fail. Files can be checked out of a locked object, and also opened for view.

This behavior ensures that one user is not overwriting changes made by another.

Client and Server Clauses

The `Client` and `Server` clauses allow programmers to specify where files are to be located when their programs are executed from a Web client (either downloaded or run on the Collaboration Server). These clauses are used to override the defaults and are ignored when executed on the desktop client.

The default file location for checkin from the Web is the Collaboration Server. Programmers can specify `client` to have the file copied from the Web client machine instead. For example:

```
mql checkin bus Assembly Wheel 0 format ascii file \tmp\text.txt;
```

would look for the file text.txt on the server, while the following would look for it on the client:

```
mql checkin bus Assembly Wheel 0 client format ascii file \tmp\text.txt;
```

The Server clause can be specified to force the default location. For example:

```
mql checkin bus Assembly Wheel 0 server format ascii file \tmp\text.txt;
```

will yield the same results as:

```
mql checkin bus Assembly Wheel 0 format ascii file \tmp\text.txt;
```

Append Clause

The Append clause is used when the files should be added to the contents of the object. Without this flag, the checkin command will overwrite any files that are contained in the object, even if the file names are unique. For example:

```
checkin businessobject Assembly "Phone Model 324" AD
   format "ASCIItext"
   $MATRIXHOME/telephones/assemble324.doc;
checkin businessobject Assembly "Phone Model 324" AD
   format "ASCIItext"
   append $MATRIXHOME/telephones/disassemble324.doc;
```

Without the append flag, the Assembly Phone Model 324 AD would contain just one file: disassemble324.doc.

Be aware that when you use the Append clause to check in a file, and the business object already contains a file of the same name, the file is overwritten without a prompt for verification.

Store Clause

The Store clause can be used to override the default store (defined in the policy) that determines where a governed business object's files are kept. For example, a business application's checkin page may be used by several different companies, each with its own file store. The logic on the page could be designed to check which company the current user is employed by, and check the file into the store assigned to that company. For example:

```
checkin bus Part Sprocket 3 append store CAD c:\sprocket.cad;
```

Checkout Business Object

The Checkout Businessobject command enables you to obtain a personal copy of the files in a particular business object. To check a file out of an object, you must first have Checkout privileges. If you do, you can check out the files by using the Checkout Businessobject command. Otherwise, Live Collaboration prevents you from checking out the file.

```
checkout businessobject OBJECTID [lock] [server|client] [format FORMAT_NAME]
[file |FILENAME {,FILENAME}| all] [in VAULTNAME] [DIRECTORY];
```

- OBJECTID is the OID or Type Name Revision of the business object. It can also include the in VAULTNAME clause, to narrow down the search.
- FORMAT_NAME is the name of the format of the files to be checked out. (Formats are defined in the *Format Clause* of the Policy associated with the object Type.) If no format is specified, only files of the default format are checked out. If a format is specified, but no filename(s), all files of the specified format are checked out.

- `FILENAME` is a specific file (or files) that you want to check out, or specify `all` to checkout all files in the specified format.
- `DIRECTORY` is the complete directory path where you want the checked out copies. If the directory is omitted, the checked out copies are copied to the current system directory.

For example, assume you have written procedures for assembling and disassembling a telephone. After checking them in, you allow users to edit the procedures to correct errors or ambiguities. To do this, they might write a command similar to:

```
checkout businessobject Assembly "Phone Model 324" AD
   file assemble324.doc $MATRIXHOME\telephones;
```

After this command is processed, a copy of the named file will appear in the directory specified. If the directory is not specified, the files are copied to the current system directory.

If you have checked out an earlier version of a file to edit, be careful not to overwrite the external file with the new checked out file. If the same file name already exists in the target directory, no error message appears in MQL. The new file will overwrite the existing file without further warning.

When you check out a file, the checkout operation triggers multi-site file sync, if needed, and writes to the history file. The read/write operation has to complete before you can use a file.

Lock Clause

In the Checkout command, it is assumed that the file being checked out is unlocked to allow other users to edit the file's contents. To prevent other users from checking files in, lock it by including the keyword `Lock` within the Checkout command. Only the person who locks the object will be allowed to check in files.

For example, the following command locks and checks out a file:

```
checkout businessobject Assembly "Phone Model 324" AD lock;
```

Using the lock to prevent editing is useful when you are making extensive changes to an external file. While those changes are being made, you do not want to worry about any other users modifying the original file without your knowledge. When you have completed your changes, you can check the file in and remove the lock at the same time.

It is possible for a locked object to be unlocked by a user with unlock access. For example, a manager may need to unlock an object locked by an employee who is out sick.

If locking is enforced in the object's policy the object MUST be locked within the checkout command if the updated file is to be checked back in.

Server and Client Clauses

The `Server` and `Client` clauses allow programmers to specify where files are to be located when their programs are executed from a Web client (either downloaded or run on the Collaboration Server). These clauses are used to override the defaults and are ignored when executed on the desktop client.

By default, the checkout command executed from the Web stores files on the Web client machine. The Server clause allows programmers to alternatively specify the Server as the file location for the operation. For example, the following command in a Tcl program object that is run from the Web client will land the file text.txt on the client:

```
mql checkout bus Assembly Wheel 0 format ascii file text.txt \tmp;
```

while the following would look for it on the server:

```
mql checkout bus Assembly Wheel 0 server format ascii file text.txt \tmp;
```

The client clause can be specified to force the default location. For example:

```
mql checkout bus Assembly Wheel 0 client format ascii file text.txt \tmp;
```

will yield the same results as:

```
mql checkout bus Assembly Wheel 0 format ascii file text.txt \tmp;
```

Format Clause

In addition to checking out all files of the default format from the object, you can check out only specific formats of the file. Files can be checked in using multiple formats. For example, a text file may be checked in using two formats that represent two different versions of a word processing program. You can check out only the file associated with the specific format by using the Format clause.

The Format clause of the Checkout Businessobject command specifies the format of the file(s) to check out. If no format is specified, the default format is assumed. For example, the following command checks out all files associated with the 1998 word processing program.

```
checkout businessobject "Phone Book" "Boston Region" K format 1998;
```

Opening Files

Files that have been checked in to a business object using the Checkin command can be opened for either viewing or editing.

View—To open files and launch the appropriate application for viewing, use the following command:

```
openview businessobject OBJECTID [format FORMAT_NAME] [file FILENAME];
```

Edit—To open files and launch the appropriate application for editing, use the following command:

```
openedit businessobject OBJECTID [format FORMAT_NAME] [file FILENAME];
```

- OBJECTID is the OID or Type Name Revision of the business object.
- FORMAT_NAME is the file format in which the file has been checked in.
- FILENAME is the name of the file.
- For information on moving and renaming files, see *Modify Business Object*.

Checking Out without Modifying the Modification Date

Prior to V6R2013, when history was on, a Checkout operation modified the `modified` attribute of the business object by default. This behavior had some side effects on operations that were based on the `modified` attribute, such as indexation. Starting in V6R2013, the default behavior for a standard checkout is to write the checkout event to the history log without changing the business object modification date.

To revert back to the pre-V6R2013 behavior, if you are relying on the old behavior, run

```
set system fcssettings modbusoncheckouthistory on;
```

With the FCS setting `modbusoncheckouthistory` turned on, a standard checkout updates the business object modification date. To restore the V6R2013 default behavior (no update to business object modification date on checkout), run

```
set system fcssettings modbusoncheckouthistory off;
```

To check the business object modification date, run:

```
print bus T N R select modified dump;
```

To check the value of the FCS setting `modbusoncheckouthistory`, run:

```
print system fcssettings modbusoncheckouthistory;
```

Lock Business Object

A business object can be locked using either the Lock command or the Checkout command. The Checkout command is used to copy files from an object (as discussed in *Checkout Business Object*). The Lock command is used for general locking purposes.

If locking is enforced in the policy, the lock must be part of the checkout command. Attempts to use the lock command will fail.

The Lock command places a lock on a business object:

```
lock businessobject OBJECTID;
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object.
- When this command is used, MQL checks to see if you have locking privileges and if there is an existing lock on the object. If you have privileges and there is no existing lock, a lock is applied to the object. (Otherwise, an error results.) After the lock is applied, only you or someone operating in your context can edit the object contents.

For example, the following commands restrict to Barbara the editing of the “Box Design” object “Bran Cereal:”

```
set context user Barbara password "Van Gogh";
lock businessobject "Box Design" "Bran Cereal" A;
```

The first command sets the context and identifies the person placing the lock. The second command applies the lock to the object. Unless Barbara or someone with unlocking privileges removes the lock, no one else can alter the object’s contents.

Unlock Business Object

A business object can be unlocked using either the Unlock command or the Checkin command. The Checkin command is used when you are associating files with the object (as discussed in *Checkin Business Object*). The Unlock command removes a lock from a business object:

```
unlock businessobject OBJECTID [comment TEXT];
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object.
- `TEXT` is a special message sent to the original lock holder.
- When this command is used, MQL checks for a lock on the object. If there is an existing lock, and you are the user that locked the object, the lock will be removed from the object. If you are not the User that locked the object, MQL will check for Unlock access and will unlock the object only if you are entitled to do so.

After the lock is removed, other users may apply new locks to obtain exclusive editing access. For example, the following commands remove the exclusive editing lock from the Bran Cereal business object:

```
set context user Barbara password "Van Gogh";
unlock businessobject "Box Design" "Bran Cereal" A;
```

The first command sets the context and identifies the person removing the lock. This must be the same person who placed the lock on the object or someone who has unlocking privileges. The second command unlocks the object. Unless Barbara or someone with unlocking privilege removes the lock, no one else can alter the object's contents.

Under some circumstances, it may be necessary to have someone other than the lock owner remove the lock. For example, in the case of extended illness, a manager may decide to have someone else edit the object. When the manager removes the lock, the original lock owner will receive a notice (IconMail) that the lock was removed. If desired, an additional message can be sent along with the notification. This is the purpose of the Comment clause of the Unlock command.

Delete Business Object

You can remove an entire business object (including all checked in files) or you can remove single files from the business object with the Delete Businessobject command:

```
delete businessobject OBJECTID [[format FORMAT_NAME] file FILENAME{,FILENAME}];
Or
delete businessobject OBJECTID [[format FORMAT_NAME] file all];
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object to be deleted or from which files should be deleted.
- `FORMAT_NAME` is the name of the format of the files to be removed from the business object. If none is specified the default format is assumed.
- `FILENAME` indicates a checked in file to be removed from the business object. You can use the All argument (rather than `FILENAME`) to remove all checked in files for a given format.

Do not delete the persons creator, guest, or Test Everything using MQL. Deleting these objects could cause triggers, programs or other application functions not to work.

When this command is processed, the Live Collaboration searches the list of business objects. If the named business object is found, any connections it has are first removed, (so that history is updated

on the object on the other ends) and then that object is deleted along with any associated files. If the name is not found, an error message results.

For example:

```
delete businessobject Person "Cheryl Davis" 0;
```

After the Delete command is processed, the business object is deleted and any relationships with that object are dissolved. You will receive the MQL prompt for another command.

To remove a checked in file of format ASCII text, enter the following command:

```
delete businessobject Assembly "Telephone Model 324" AD
   format "ASCII Text"
   file assemble324.doc;
```

Deletion of files can occur automatically, as well, during a checkin/replace, removal of a business object, move file command, or synchronization of file stores. In a replicated environment, the delete function removes all copies of each file at each location. For more information about checked-in files, refer to *Locking and Unlocking a Business Object* in Chapter 5.

When deleting files from the last object in a revision chain, only the link to the files is deleted. However, when deleting a file/format from a business object that is not the last in a revision chain, the file is first copied to the next object in the revision chain, if a reference to that file exists. This latter case may impact performance.

Business Object State

The following MQL commands control the state of a business object:

`approve businessobject OBJECTID signature SIGN_NAME [comment VALUE];`
`ignore businessobject OBJECTID signature SIGN_NAME [comment VALUE];`
`reject businessobject OBJECTID signature SIGN_NAME [comment VALUE];`
`unsign businessobject OBJECTID signature SIGN_NAME|all [comment VALUE];`
`enable businessobject OBJECTID [state STATE_NAME];`
`disable businessobject OBJECTID [state STATE_NAME];`
`override businessobject OBJECTID [state STATE_NAME];`
`promote businessobject OBJECTID;`
`demote businessobject OBJECTID;`

- `OBJECTID` is the `OID` or Type Name Revision of the business object.
- For a detailed description of each command, see *Modifying the State of a Business Object* in Chapter 5.

Purge Business Object or Business Object List

The Purge Businessobject command is used to purge all but the last updated file of the associated business object. If the Purge businessobject command is used without a location name, it does a purge on all locations. The syntax is:

```
purge businessobject OBJECTID [location LOCATION_TARGET{,LOCATION_TARGET}];
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object.
- `LOCATION_TARGET` is one or more location names that are associated with the specified business object. Locations are separated by a comma but no space.

The keyword `businessobject` can be shortened to `bus`.

Location Clause

To specify a subset of locations that you want purged, include the `Location` clause. For example:

```
purge bus Assembly R123 A location London,Paris,Milan;
```

When listing locations, delimit with a comma but no space.

The Purge Businessobjectlist command is used to purge files associated with a list of business objects. The syntax is:

```
purge businessobjectlist BUS_OBJECT_COLLECTION_SPEC [continue]
[commit N] [location LOCATION_TARGET{,LOCATION_TARGET}];
```

- BUS_OBJECT_COLLECTION_SPEC includes:

```
    | set NAME                                                    |
    | query NAME                                                  |
    | temp set BO_NAME{,BO_NAME}                                  |
    | temp query [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM}]              |
    | expand [EXPAND_ITEM {EXPAND_ITEM}]                          |
```

You can also use these specifications in boolean combinations.

The keyword `businessobjectlist` can be shortened to `buslist`.

Additional `purge businessobjectlist` clauses are described in the sections that follow.

Continue Clause

Include the keyword `continue` if you don't want the command to stop if an error occurs. If the log file is enabled, failures are listed in the file. Refer to *trace Command* for more information. For example:

```
purge businessobjectlist set MyAssemblies continue;
```

If an error occurs when using the `Continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next business object. For this reason, when using the `Continue` clause you should also include the `Commit` clause, described below.

Commit N Clause

Include the `Commit N` Clause when purging large business object lists. The number N that follows specifies that the command should commit the database transaction after this many objects have been purged. The default is 10. For example:

```
purge buslist set MyAssemblies continue commit 20;
```

Rechecksum Business Object or Business Object List

The Rechecksum Businessobject command is used to calculate the checksums for all files owned by a business object. Typically, this command is used on migrated files. This method of creating checksums for files that were migrated and not individually checked in (and thus did not have a checksum created at checkin) assumes the files are not already corrupted. You might find it helpful to run the validate command first. Refer to *Validate Store* for more information.

Note that the `rechecksum` command applies only when `checksum` has been turned on at the store level. Refer to *Add Store* for more information.

The syntax is:

```
rechecksum businessobject OBJECTID;
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object.

The keyword `businessobject` can be shortened to `bus`.

TheRechecksum Businessobjectlist command is used to create checksums for files associated with a list of business objects. The syntax is:

```
rechecksum businessobjectlist BUS_OBJECT_COLLECTION_SPEC
[continue] [commit N];
```

- BUS_OBJECT_COLLECTION_SPEC includes:

```
    | set NAME                                                    |
    | query NAME                                                  |
    | temp set BO_NAME{,BO_NAME}                                  |
    | temp query [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM}]              |
    | expand [EXPAND_ITEM {EXPAND_ITEM}]                          |
```

You can also use these specifications in boolean combinations.

The keyword `businessobjectlist` can be shortened to `buslist`.

Additional `rechecksum businessobjectlist` clauses are described in the sections that follow.

Continue Clause

Include the keyword `continue` if you don't want the command to stop if an error occurs. If the log file is enabled, failures are listed in the file. Refer to *trace Command* for more information. For example:

```
rechecksum businessobjectlist set MyAssemblies continue;
```

If an error occurs when using the `Continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next business object. For this reason, when using the `Continue` clause you should also include the `Commit` clause, described below.

Commit N Clause

Include the `Commit N` Clause when creating checksums for large business object lists. The number N that follows specifies that the command should commit the database transaction after this many objects have had checksums created for them. The default is 10. For example:

```
rechecksum buslist set MyAssemblies continue commit 20;
```

Validate Business Object or Business Object List

The Validate Businessobject command is used to calculate the current checksums for files owned by a business object and compare those checksums with the values on record for those files.

Note that the commands for validating checksums are expensive operations. For each file to be validated, a compute checksum request is sent to the corresponding FCS and the file is scanned to compute the current checksum.

Also, the validate checksum commands are read-only operations. If a new checksum is different, it will be reported to the output file, but not stored.

The syntax is:

```
validate businessobject OBJECTID fcsdbchecksum;
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object.
- The keyword `fcsdbchecksum` specifices the type of validation to be performed.

The keyword `businessobject` can be shortened to `bus`.

TheValidate Businessobjectlist command is used to calculate the current checksums for files associated with a list of business objects and compare those checksums with the values on record for those files. The syntax is:

```
validate businessobjectlist BUS_OBJECT_COLLECTION_SPEC fcsdbchecksum;
```

- BUS_OBJECT_COLLECTION_SPEC includes:

```
    | set NAME                                                    |
    | query NAME                                                  |
    | temp set BO_NAME{,BO_NAME}                                  |
    | temp query [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM}]              |
    | expand [EXPAND_ITEM {EXPAND_ITEM}]                          |
```

You can also use these specifications in boolean combinations.

The keyword `fcsdbchecksum` specifices the type of validation to be performed.

The keyword `businessobjectlist` can be shortened to `buslist`.

channel Command

Description

Channels are a collection of commands. They differ from menus in that they are not designed for use directly in a Studio Customization Toolkit application, but are used to define the contents of a *portal*. Channels and portals are installed with the Framework and used in apps to display PowerView pages, but may also be created for use in custom Java applications.

For conceptual information on this command, see *Channels* in Chapter 8.

User Level

Business Administrator with portal administrative access.

Syntax

```
[add|copy|modify|delete]channel NAME {CLAUSE};
```

- NAME is the name you assign to the channel. Channel names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the channel.

Add Channel

To define a channel from within MQL use the add channel command:

```
add channel NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the channel you are defining. Channel names cannot include asterisks.

You cannot have both a channel and a portal with the same name.

- user USER_NAME can be included if you are a business administrator with person/group/role access defining a channel for another user. This user is the item's "owner." If the User clause is not included, the channel is a system item.
- ADD_ITEM provides additional information about the channel. The add channel clauses are:

description VALUE
label VALUE
href VALUE
alt VALUE
height VALUE
command NAME {,NAME}

```
setting NAME [STRING]
dataobject NAME [user USER_NAME]
visible USER_NAME{,USER_NAME};
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

Each clause and the arguments they use are discussed in the sections that follow.

Label Clause

This clause specifies the label to appear in the application in which the channel is assigned.

Href Clause

This clause is used to provide link data to the JSP. The Href link is evaluated to bring up another page. Many channels will not have an Href value at all. However, channels designed for the “tree” channels require an Href because the root node of the tree causes a new page to be displayed when clicked. The Href string generally includes a fully-qualified JSP filename and parameters, which can contain embedded macros and expressions for mapping to database schema. Refer to *Using Macros and Expressions in Configurable Components* in Chapter 8 for more details.

The syntax is:

```
href VALUE;
```

- `VALUE` is the link data.

Alt Clause

This clause is used to define text that is displayed until any image associated with the channel is displayed and also as “mouse over text.”

The syntax is:

```
alt VALUE;
```

- `VALUE` is the text that is displayed.

For example, you could use the following for a Tools channel:

```
alt “Tasks”;
```

Height Clause

This clause is used to indicate the height size in pixels the channel will occupy on the page. The default and minimum allowed is 260. The syntax is:

```
height VALUE;
```

- `VALUE` is the size in pixels. If you enter a value less than 260, 260 will be used.

Command Clause

This clause is used to specify existing commands to be added to the channel you are creating. Each command will be a separate tab in the channel and displayed in the order in which they are added. Separate items with a comma.

The syntax is:

```
command NAME{,NAME};
```

- `NAME` is the name of the command you are adding.
- For details on creating commands, see *Add Command*.

Setting Clause

This clause is used to provide any name/value pairs that the channel may need. They can be used by JSP code, but not by hrefs on the Link tab. Refer to *Using Macros and Expressions in Configurable Components* in Chapter 8 for more details.

The syntax is:

```
setting NAME [STRING];
```

Dataobject Clause

This clause can be used to add dataobjects to a channel. A dataobject can be used to personalize a channel.The syntax is:

```
dataobject NAME [USER_NAME];
```

Visible Clause

This clause specifies other existing users who can read the channel with MQL list, print, and evaluate commands. The MQL copy channel command can be used to copy any visible channel to your own workspace.

The syntax is:

```
visible USER_NAME{,USER_NAME};
```

Separate users with a comma, but no space.

History Clause

The `history` keyword adds a history record marked "custom" to the channel that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Channel

After a channel is defined, you can clone the definition with the Copy channel command.

A Business Administrator with channel access, can copy system channels. A Business Administrator with person access, can copy channels in any person's workspace (likewise for groups and roles). Other users can copy visible workspace channels to their own workspaces.

This command lets you duplicate channel definitions with the option to change the value of clause arguments:

```
copy channel SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the channel definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any channel of the same name belonging to the user specified in the `Touser` clause.
`history STRING`	Adds a history record marked "custom" to the channel that is being copied. The STRING argument is a free-text string that allows you to enter some information describing the nature of the copy operation. For details, see *Adding History to Administrative Objects*.

The order of the `Fromuser`, `Touser` *and* `Overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications. Note that you need specify only the fields to be modified.

Clone/Modify Channel Clause	Specifies that…
`name NEW_NAME`	The current channel name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
`icon FILENAME`	The image is changed to the new image in the file specified.
`label VALUE`	The label is changed to the new value specified.
`href VALUE`	The link data information is changed to the new value specified.
`alt VALUE`	The alternate text is changed to the new value specified.

Clone/Modify Channel Clause	Specifies that…
`height VALUE`	The height is changed to the new value specified.
[fromuser FROMUSER] touser TOUSER	For use with copy only. The current channel is owned by FROMUSER, and copied to TOUSER. If not specified, it is assumed to be current context user.
`place COMMAND1 before COMMAND2`	The named COMMAND1 is moved or added before COMMAND2. If COMMAND2 is an empty string, COMMAND1 is placed before all commands in the channel.
`place COMMAND1 after COMMAND2`	The named COMMAND1 is moved or added after COMMAND2. If COMMAND2 is an empty string, COMMAND1 is placed after all commands in the channel.
`add setting NAME [STRING]`	The named setting and `STRING` are added to the channel.
`place dataobject NAME1 [user USER] before NAME2 [user USER]`	The named dataobject is moved or added before dataobject NAME2. If NAME2 is an empty string, NAME1 is placed before all dataobjects in the channel.
`place dataobject NAME1 [user USER] after NAME2 [user USER]`	The named dataobject is moved or added after NAME2. If NAME2 is an empty string, NAME1 is placed after all dataobjects in the channel.
`remove dataobject NAME [user USER]`	The named dataobject is removed from the channel.
`remove command NAME`	The named command is removed from the channel.
`remove setting NAME [STRING]`	The named setting and `STRING` are removed from the channel.
`visible USER_NAME{,USER_NAME};`	The named user(s) have visibility to the channel.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	The history keyword adds a history record marked “custom” to the channel that is being copied. The STRING argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Channel

A Business Administrator with channel access, can modify system channels. A Business Administrator with person access, can modify channels in any person’s workspace (likewise for groups and roles). Other users can modify only their own workspace channels.

You must be a business administrator with group or role access to modify a channel owned by a group or role.

Use the Modify channel command to add or remove defining clauses or change the value of clause arguments:

```
modify channel NAME [user USER_NAME][MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the channel you want to modify. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.
- `MOD_ITEM` is any of the items in the table above.

Each modification clause is related to the arguments that define the channel. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a channel, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire command.
- If you are working from a script, group the changes together in a single modify channel command.

History Clause

The `history` keyword adds a history record marked "custom" to the channel that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Delete Channel

A Business Administrator with channel access, can delete system channels. A Business Administrator with person access, can delete channels in any person's workspace (likewise for groups and roles). Other users can delete only their own workspace channels.

You must be a business administrator with group or role access to delete a channel owned by a group or role.

If a channel is no longer required, you can delete it using the Delete channel command

```
delete channel NAME [user USER_NAME];
```

`NAME` is the name of the channel to be deleted. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.

Searches the list of defined channels. If the name is found, that channel is deleted. If the name is not found, an error message is displayed. For example, to delete the channel named "Toolbar" enter the following:

```
delete channel "Tasks";
```

After this command is processed, the channel is deleted and you receive an MQL prompt for another command.

command Command

Description

Use this command to define your own commands. Commands can be used in any kind of menu in a JSP application. Commands may or may not contain code, but they always indicate how to generate another Web page. Commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it.

For conceptual information on this command, see *Commands* in Chapter 8.

User Level

Business Administrator with Menu administrative access

Syntax

```
[add|copy|modify|delete]command NAME {CLAUSE};
```

- NAME is the name you assign to the command. Command names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the command.

Add Command

To define a command from within MQL use the Add Command command:

```
add command NAME [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the command you are defining. Command names cannot include asterisks. The name you choose is the name that will be referenced to include this command within a menu.

You cannot have both a command and a menu with the same name.

- ADD_ITEM is an Add Command clause that provides additional information about the command. The Add Command clauses are:

description VALUE
label VALUE
href VALUE
alt VALUE
code VALUE
file FILENAME

```
user [NAME {,NAME} | all]
setting NAME [STRING]
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

Label Clause

This clause specifies the label to appear in the menu in which the command is assigned. For example, many desktop applications have a File menu with options labeled “Open” and “Save.”

Href Clause

This clause is used to provide link data to the `href` HTML commands on the Web page that references the command. This field is optional, but generally an href value is included.

The syntax is:

```
href VALUE;
```

- `VALUE` is the link data.

For example:

```
href "emxForm.jsp?header=Basic Info";
```

Assigning an href to a link can create problems if a user clicks the same link twice when initiating such actions as changing an object’s state or submitting a form. The reason for this is that an href assigned to a link is considered a server request, even if it is a JavaScript command. Whenever the browser detects a server request, the browser stops processing the current request and sends the new request. Therefore, when a user first clicks on an href link, the request is processed, and typically, a JSP page starts executing. If, during this time, a user clicks the same link again, the first request is interrupted before completion and the new request is processed instead.

To avoid this scenario, you can set the href to “#” and use the `onclick` event instead. The generic code for this is:

```
<a href="#" onclick="submitForm()">
```

Alt Clause

This clause is used to define text that is displayed until any image associated with the command is displayed and also as “mouse over text.”

The syntax is:

```
alt VALUE;
```

- `VALUE` is the text that is displayed.

For example, you could use the following for a Tools command:

```
alt “Tools”;
```

Code Clause

This clause is used to add JavaScript code to the command.

The syntax is:

```
code VALUE;
```

- `VALUE` is the JavaScript code.

When commands are accessed from a JSP page, the href link is evaluated to bring up the next page. Commands only require code if the href link references JavaScript that is not provided on the JSP. The JSP must provide logic to extract the code from this field in order for it to be used. None of the commands provided by the applications or Framework use the code field.

You can add the code in the Code clause or it can be written in an external editor. If you use an external editor, use the File clause instead.

File Clause

This clause is used to specify the file that contains the code for the command when the code is written in an external editor.

The syntax is:

```
file FILENAME;
```

`FILENAME` is the name of the file that contains the code for the command.

User Clause

This clause is used to define the users allowed to see the command.

Any number of roles, groups, persons, and/or associations can be added to the command (role-based in this case includes all types of users and is not limited to only roles).

The syntax is:

```
user NAME {,NAME};
```

- `NAME` is the name of the user(s) who have access to see the command.

Or

```
user all;
```

- If the `User` clause is not included, all users are given access to the command.

Setting Clause

This clause is used to provide any name/value pairs that the menu may need. They can be used by JSP code, but not by hrefs on the Link tab.

The syntax is:

```
setting NAME [STRING];
```

For example, an image setting with the image name can be specified to display when the menu is used in a toolbar:

```
setting Image iconSmallMechanicalPart.gif;
```

History Clause

The `history` keyword adds a history record marked “custom” to the command that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Command

After a command is defined, you can clone/copy the definition. Cloning a Command definition requires Business Administrator privileges, except that you can copy a Command definition to your own context from a group, role or association in which you are defined.

This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy command SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the command definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the command below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the command that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Command

The List Command command is used to display a list of all commands that are currently defined. It is useful in confirming the existence or exact name of a command that you want to modify, since the Live Collaboration is case-sensitive.

```
list command [modified after DATE] NAME_PATTERN [select FIELD_NAME {FIELD_NAME}] [DUMP [RECORDSEP]] [tcl] [output FILENAME];
```

- For details on the List command, see *list admintype Command*.
- Use the list of all the existing commands along with the `Print` command to determine the search criteria you want to change.
- Use the Modify Command command to add or remove defining clauses and change the value of clause arguments:

```
modify command NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the command you want to modify.

- `MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Command clause, as listed in the following command. Note that you need specify only the fields to be modified.

Modify Command Clause	Specifies that…
`name NEW_NAME`	The current command name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
`icon FILENAME`	The image is changed to the new image in the file specified.
`label VALUE`	The label is changed to the new value specified.
`href VALUE`	The link data is changed to the new value specified.
`alt VALUE`	The alternate text is changed to the new value specified.
`code VALUE`	The code associated with the command is replaced by the new code specified.
`file FILENAME`	The file that contains the command code is changed to the file specified.
`add user NAME`	The named user is granted access to the command.
`add user all`	All users are granted access to the command.
`add setting NAME [STRING]`	The named setting and `STRING` are added to the command.
`remove user NAME`	The named user access is revoked.
`remove user all`	Access to the command is revoked for all users.
`remove setting NAME [STRING]`	The named setting and `STRING` are removed from the command.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the command that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the arguments that define the command. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a command, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire command.
- If you are working from a script, group the changes together in a single Modify Command command.

Delete Command

If a command is no longer required, you can delete it using the Delete Command commands

```
delete command NAME:
```

- `NAME` is the name of the command to be deleted.
- When this command is processed, the Live Collaboration searches the list of defined commands. If the name is found, that command is deleted. If the name is not found, an error message is displayed.

For example, to delete the command named “Convert,” enter the following:

```
delete command "Convert";
```

After this command is processed, the command is deleted and you receive an MQL prompt for another command.

config Command

Description

The *config* command provides configuration details.

User Level

System Administrator

Syntax

```
[print|zip] config [CLAUSEs];
```

- CLAUSEs provide additional information about the connection.

Print Config

You can use the `print config` command to output configuration messages to the console or to a file, using the following syntax:.

```
print config [ITEM] [full | !full] [output FILENAME];
```

where:

- ITEM is

  `oracleset user USERNAME password PASSWORD`
- FILENAME is the name of the file to be created, and can include the PATH. If the FILENAME specified is “mxAudit.log”, it will append the data to the file that is generated by the configuration checker at server startup time. For example:

  `print config output /app/rmi1050/logs/mxAudit.log;`

To print information from the Oracle v$parameter view:

```
print config oracleset user system password manager;
```

The `oracleset` subcommand can be used to retrieve the Oracle compatibility mode setting and determine whether the optimizer mode is enabled. It also reports other settings that influence Server stability. For more information about `v$parameter`, refer to Oracle documentation.

Output

The information returned represents what the kernel sees as it scans certain processes, and it can be used to verify that the application has been set up correctly without looking at the config files. The information is presented with the following headers:

`****Oracle Settings**** (if oracleset is specified)`

This is a subset of the V$PARAMETERS table.

`****ENOVIA Kernel JVM System Properties****`

This is a subset of Java system properties (System.getProperties)

`****ENOVIA Kernel Local Environment****`

The Live Collaboration kernel local environment shows the equivalent of typing `Set` on DOS or `environ struct` on UNIX.

`****`ENOVIA **`Default Environment Settings****`**

This is a list of settings important to kernel loading and running, such as `LD_LIBRARY_PATH` and `NLS_LANG`.

`****`ENOVIA **`Kernel Settings****`**

Live Collaboration kernel settings show output of Kernel INI in-memory values. There are name/value pairs determined by INI settings, environment variables, and the MatrixIniDefaults program. The name/value pairs listed in the `print config` output might have a "computed" value. For example, `MX_CHARSET` is used to determine whether the kernel needs to convert Java strings to the system character set. If `MX_CHARSET` is set to `UTF8`, no conversion is needed and the computed value that `print config` will print looks like:

```
MX_CHARSET=FALSE (if MX_CHARSET is set to UTF8 in the INI file)
```

******ENOVIA Jar Version******

This is the version of `eMatrixServletRMI.jar` (retrieved from `matrix.util.Version.number`).

******Current ENOVIA Live Collaboration INI Settings**** (Win32 Only)**

This is the contents of the enovia.ini file for the Live Collaboration Server.

******Operating System Kernel Settings******

This shows operating system information, such as kernel process ID, os.name, os.arch, and os.version.

******ENOVIA Live Collaboration Upgrade Validation******

This is the output of the MQL Validate Upgrade command.

Below is an example of Live Collaboration kernel JVM system properties that are output from the Print Config command:

```
2503 10/2/2003 4:56:43 PM ****eMatrix kernel JVM system
properties*****
2503 10/2/2003 4:56:43 PM java home: /usr/java/jdk1.3.1_06/jre
2503 10/2/2003 4:56:43 PM java vm specification version: 1.0
2503 10/2/2003 4:56:43 PM java vm specification vendor: Sun
Microsystems Inc.
2503 10/2/2003 4:56:43 PM java vm specification name: Java Virtual
Machine Specification
2503 10/2/2003 4:56:43 PM java vm version: 1.3.1_06-b01
2503 10/2/2003 4:56:43 PM java vm vendor: Sun Microsystems Inc.
2503 10/2/2003 4:56:43 PM java vm name: Java HotSpot(TM) Client VM
2503 10/2/2003 4:56:43 PM java specification version: 1.3
2503 10/2/2003 4:56:43 PM java specification vendor: Sun Microsystems
Inc.
2503 10/2/2003 4:56:43 PM java specification name: Java Platform API
Specification
2503 10/2/2003 4:56:43 PM java class version: 47.0
2503 10/2/2003 4:56:43 PM java class path: /usr/java/jdk1.3.1_06/lib:/
app/rmi1010/java/classes:/app/rmi1010/java/properties:/app/rmi1010/
java/lib:/app/rmi1010/java/lib/ext:/app/rmi1010/java/custom:/usr/java/
jdk1.3.1_06/lib/dt.jar:/usr/java/jdk1.3.1_06/lib/htmlconverter.jar:/
usr/java/jdk1.3.1_06/lib/tools.jar:/app/rmi1010/java/lib/
```

```
eMatrixServletRMI.jar:/app/rmi1010/java/lib/jdom.jar:/app/rmi1010/
java/lib/xalan.jar:/app/rmi1010/java/lib/xerces.jar:/app/rmi1010/java/
lib/common.jar:/app/rmi1010/java/lib/component.jar:/app/rmi1010/java/
lib/domain.jar:/app/rmi1010/java/lib/engineering.jar:/app/rmi1010/
java/lib/kcServlet.jar:/app/rmi1010/java/lib/xalanj1compat.jar
2503 10/2/2003 4:56:43 PM java version: 1.3.1_06
2503 10/2/2003 4:56:43 PM java vendor: Sun Microsystems Inc.
2503 10/2/2003 4:56:43 PM java vendor url: http://java.sun.com/
2503 10/2/2003 4:56:43 PM user name: dvlp
2503 10/2/2003 4:56:43 PM user home: /home/dvlp
2503 10/2/2003 4:56:43 PM user dir: /home/dvlp/buildCC/src/ematrix
```

Zip Config

The `zip config` command allows you to create a zip file that contains information necessary to assist in configuration troubleshooting.

```
zip config [output FILENAME];
```

Where FILENAME is the fully qualified path and name of the file to be created.

For example:.

```
zip config output /app/rmi1050/mxAudit.zip;
```

If the output clause is not included, a file named mxAudit.zip is created in the ENOVIA_INSTALL directory.

The zip file will contain the files referenced in the ENOVIA_INSTALL/mxAuditFile.txt, which is created at install time, as well as the mxAudit.log file that is created at server startup. Together the zipped up files provide a comprehensive picture of configuration settings.

Since the list of files in mxAuditFile.txt is a snapshot of what the configuration file locations were at install time, it is important to know what files are being included in the zip file. Administrators often change startup script names and locations. When the `zip config` command is issued, the command will generate the name of each file that is added to the zip file, enabling you to verify that the files being zipped are the current active files. Below is sample output from `zip config`:

```
****Begin eMatrix Zip Config****
zip config file : /app/rmi1050/logs/mxAudit.log
zip config file : /app/rmi1050/scripts/mxEnv.sh
zip config file : /app/rmi1050/java/properties/framework.properties
zip config file : /opt/jakarta-tomcat-4.1.18/bin/catalina.sh
```

connection Command

Description

Once you have defined relationship types, *connections* or instances of a relationship type can be made between specific business objects.

For conceptual information on this command, see *Manipulating Data* in Chapter 5.

User Level

Business Administrator

Syntax

```
[add|print|modify|freeze|query]connection NAME [CLAUSEs];
```

- NAME is the connection or the instance of the relationship type you are adding.
- CLAUSEs provide additional information about the connection.

The syntax and clauses for working with connections and business objects is similar. For more infomation on how to work with connections, see *businessobject Command*.

Add Connection

Use the Add Connection command to create new connections. You can specify either a business object or a connection at either end of the connection

```
add connection RELTYPE [ADD_ITEM {ADD_ITEM}];
```

- RELTYPE is the connection or the instance of the relationship type you are adding.
- ADD_ITEM is an Add Connection clause which provides more information about the connection you are creating.

The Add Connection clauses are:

to BO_NAME
from BO_NAME
torel CONNECTION
fromrel CONNECTION
ATTRIBUTE_NAME_VALUE
interface INTERFACE_NAME
relationshiprule NAME
owner USER_NAME

`organization ORGANIZATION_NAME` **Note:** `altowner1 USER_NAME` is still supported for backwards compatibility.

`project PROJECT_NAME`
Note: `altowner2 USER_NAME` is still supported for backwards compatibility.
 	
`physicalid UUID`

`logicalid UUID`

To and From Clauses

The To and From (and Torel and Fromrel) clauses of the Add Connection command define the ends of the connection. Use the To and From clauses with a business object ID. Use the Torel and Fromrel clauses with a connection ID.

All connections will occur between either two business objects, two connections, or a business object and a connection. When looking at a connection, the objects are connected TO and FROM one another. You must define the connection with one To clause and one From clause in your Add Connection command. If either clause is missing or incomplete, the connection will not be created.

Both the To and From clauses use the same set of subclauses because they define the same information about each connected object. The separate values you use to define each connection end will vary according to the type of connection you are making and the types of objects involved.

It is important to note that the rules governing the creation of business objects and connections (e.g. definitions of type, relationship type, policy, etc.) are only enforced at the time creation. These rules will be ignored during later modifications of those business objects and connections.

For example, changing the type of business object on one end of a connection will ignore the type restrictions defined on the relationship. This behavior is allowed to support a dynamic modeling environment.

Print Connection

You can print the description of a connection using the Print Connection command. For example

```
print connection OBJECTID;
```

The results of the above may be something like:

```
Relationship Drawing
   From bus obj 50402
   To bus obj 50402
   relationship[Drawing].isfrozen = FALSE
   relationship[Drawing].propagatemodifyfrom = FALSE
   relationship[Drawing].propagatemodifyto = FALSE
   history
   create - user: adami time: Thu May 22, 1997 6:29:20 PM
     from: Drawing 50402 A to: Assembly 50402 B
```

If the connection's ID is unknown, the following can also be used:

```
print connection bus OBJECTID to|from OBJECTID relationship RELTYPE;
```

`OBJECTID` is the `OID` or Type Name Revision of the business object or connection. It can also include the `in VAULTNAME` clause, to narrow down the search.

Note that the result of `print connection` with object ends specified will be ambiguous when multiple relationships between the two objects exist.

Select commands

Select commands enable you to view a listing of the field names that make up the connection definition. Each field name is associated with a printable value. By selecting and listing the field names that you want, you can create a subset of the connection's contents and print only that subset. The system attempts to produce output for each Select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

Select can be used as a clause of the `print connection` and in query where clauses (See also *Using Select Clauses in Queries*).

Reviewing the List of Field Names

The first step is to examine the general list of field names. This is done with the command:

```
print connection selectable;
```

The Selectable clause is similar to using the ellipsis button in the graphical applications—it provides a list from which to choose.

When the command above is processed, MQL lists all selectable fields with their associated values:

```
print connection selectable;
connection selectable fields :
    name
    type.*
    attribute[].*
    businessobject.*
    to.*
    toall
    tomid.*
    torel.*
    from.*
    fromall.*
    frommid.*
    fromrel.*
    history.*
    isfrozen
    propagatemodifyto
    propagatemodifyfrom
    id
```

```
    method.*
    program.*
    execute.*
    rule[].*
    originated
    modified
    context.*
    index.*
    interface.*
    expression[]
    evaluate[]
    reserved
    reservedby.*
    reservedcomment
    reservedstart
    owner.*
    organization.* or altowner1.* (see below)
    project.* or altowner2.* (see below)
```

MQL connection commands still support the `altowner1` *and* `altowner2` *keywords when specified explicitly. For example,*

```
print connection select altowner1 altowner2;
```

is supported and the result prints `altowner1 = XXX / altowner2 = YYY`.

Where those keywords cannot be specified explicitly, they are not shown but `organization` *and* `project`*, respectively, are shown instead. For example:*

```
print connection selectable;
```

shows `organization = XXX` *and* `project = YYY`.

Notice that some of the fields are followed by square brackets and/or a "`.*`". The asterisk denotes that the field can be further expanded, separating the subfields with a period. If only one value is associated, the name appears without an asterisk. For example, the Name field has only a single value that can be printed. On the other hand, a field such as type has other items that can be selected. If you expand the Type field, you might find fields such as `type.name`, `type.description`, and `type.policy`. This means that from any connection, you could select a description of its type and find out other valid policies for it. Refer to *Print Business Object* for more information about the use of selectables.

Modify Connection

After a connection is defined, you can change the definition with the Modify Connection command. This command lets you add or remove defining clauses and change the value of clause arguments.

```
modify connection CONNECTION [MOD_ITEM {MOD_ITEM}];
```

`CONNECTION` is the name or ID of the connection you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Relationship clause, as listed in the following table. Note that you only need to specify fields to be modified:

Modify Relationship Clause	Specifies that…
`to BO_NAME`	A modification is made to the business object on the TO end. See *Changing Connection Ends*.
`from BO_NAME`	A modification is made to the business object on the FROM end. See *Changing Connection Ends*.
`torel CONNECTION`	A modification is made to the connection on the TO end. See *Changing Connection Ends*.
`fromrel CONNECTION`	A modification is made to the connection on the FROM end. See *Changing Connection Ends*.
`ATTRIBUTE_NAME VALUE`	The specified attribute is changed to that of the new attribute. Attributes of connections can be modified if the user has modify access on the objects or connections on both ends of the connection.
`add interface INTERFACE_NAME`	This interface listed here is added to the connection. When you add an interface to a connection, it obtains all of the attributes defined for the interface, even those that were not defined for the connection. This allows for ad hoc attributes for connections. Moreover, when you add an interface to a connection, it gives you the ability to classify the connection by virtue of the interface hierarchy. To add or remove interfaces from connections, you must have ChangeType access on the object on the FROM end of the connection. If the FROM end is not a business object and it is a connection, the system looks for ChangeType access on the TO end of the connection. If neither end of the connection is a business object, any relationship rule defined is checked. History is recorded when you add or remove interfaces from connections. You can only add an interface to a connection of a relationship type that is added to the interface definition. For information on how to add a relationship to an interface, see *Interfaces* in Chapter 4.
`remove interface INTERFACE_NAME`	This interface listed here is removed from the connection.
`add history VALUE [comment VALUE]`	A history Tag and a Comment can be specified with this clause. The Tag is the `VALUE` you enter after `add history` as a name for the custom history entry. A user/time/date/current stamp is automatically applied when this command executes. The `comment` is the `VALUE` (or text) you enter to describe the history item you are adding.
`delete history DEL_HISTORY_ITEM`	System administrators only can purge the history records of a business object. See *Delete History* for details.
`reserve [user USER] [comment VALUE]`	To attach reservation data to an object: username, date, optional comment. The user clause can be included only by system administrators to reserve a connection on behalf of someone else. The reserve access mask controls the changing of the reserve status of an object or connection. See *Reserving a Business Object*.

Modify Relationship Clause	Specifies that…
`unreserve`	To attach reservation data to an object: username, date, optional comment. The user clause can be included only by system administrators to unreserve a connection on behalf of someone else. The unreserve access mask controls the changing of the reserve status of an object or connection. See *Reserving a Business Object*.
`!reserve`	To attach reservation data to an object: username, date, optional comment. The user clause can be included only by system administrators to reserve a connection on behalf of someone else. The reserve access mask controls the changing of the reserve status of an object or connection. See *Reserving a Business Object*.
`owner USER_NAME`	The owner of the connection is changed to the owner specified here.
`organization ORGANIZATION_NAME` (or `altowner1 USER_NAME`)	The additional owner of the connection is changed to the specified organization. You must have ChangeOwner privileges to set the owning organization.
`project PROJECT_NAME` (or `altowner2 USER_NAME`)	The additional owner of the connection is changed to the specified project. You must have ChangeOwner privileges to set the owning project.

Any combination of IDs and full specification by type name and revision (or relationship name) are allowed. When using type, name and revision you can also include in VAULTNAME.

Reserving a Connection

To mark a business object or connection as reserved:

```
modify connection ID reserve comment "reserved from Create
Part Dialog";
```

When a reservation is made, the following information is sent for storage in the database:

- Person reserving the object.
- Time when the object is reserved.
- Comments entered when the reservation is made.

The comment string is optional and has a 254 character limit. Comments longer than 254 characters are truncated and only the first 254 characters are stored in the database.

You can use reservation data to control concurrent modification in two ways:

- Code controlling the display of an editing page may check the reserved status beforehand, and reject the editing request if the object is reserved.
- Define access rules to control any access. See definition of any reserve/no reserve/context reserve/inclusive reserve under *policy Command* and *rule Command*.

The privilege to reserve or unreserve an object can be specific in accesss rules using the reserve and unreserved access mask. For more information, see *policy Command* and *rule Command*. Unreserving a business object or connection by providing the wrong OID or Type Name Revision will result in an error message. Unreserving a business object or connection that is not reserved will result in a warning.

Obtaining Connection IDs

To obtain the IDs of all connections, use the `relationship ID` selectable item. For example:

```
expand bus Assembly 50402 B select relationship id;
```

The following is a possible result that can be saved to a file for an Output clause:

```
1 Drawing from Drawing 50402 A
   id = 19.24.5.16
1 Process Plan from Process Plan 50402-1 C
   id = 19.26.6.6
```

Refer to *Expand Business Object* for more information on the `expand` command.

Changing Connection Ends

It is possible to specify a new object to replace the existing object on the end of a connection. This is comparable to using the drop connect feature, and dropping an object on top of a relationship arrow (to replace the child object) in the Navigator window.

The operation actually performs a disconnect and then a connect action, and so relies on those accesses to determine if the user can perform the action. The user must also have modify access on the connection. Use the keywords `to` and `from` or `torel` and `fromrel` on the `modify connection` commands. For example:

```
modify connection CONNECTION from|to BO_NAME;
```

`BO_NAME` is the ID or Type Name Revision of the business object.

To change the direction of the relationship, use both a 'to' and a 'from' `BO_NAME` clause. For example, assume that Assembly 50402 B is currently on the FROM end of the relationship. The following command would reverse the direction:

```
modify connection 62104.18481.14681.40078 to Assembly 50402
B from 62104.18481.13184.46733;
```

Freeze Connection

Connections can be frozen (locked) to prevent configurations from being modified. Connections that are frozen cannot be disconnected, and their attributes cannot be modified. A user would only be allowed to freeze a connection if they have freeze access on the objects on both ends.

```
freeze connection ID;
```

- ID is the ID of the connections to be modified.

When objects are cloned or revised, their existing relationships are either floated, replicated, or removed. When a frozen connection is floated, the new connection is frozen as well. Replicated connections take on the thawed state.

Thawing Connections

A frozen connection can be thawed (unlocked) by using the following syntax:

```
thaw connection ID;
```

Users must have thaw access on the objects on both the to and FROM end of the connection.

Query Connection

Connections can be queried using the Query Connection command:

```
query connection [QUERY_ITEM {QUERY_ITEM}] [SELECT] [DUMP]
[RECORDSEP] [tcl] [output FILENAME];
```

- QUERY_ITEM is a Query Connection clause.

`to BO_NAME`
`from BO_NAME`
`torelationship CONNECTION_ID`
`fromrelationship CONNECTION_ID`
`where EXPR`
`type PATTERN`
`relationship PATTERN`
`vault PATTERN`
`limit N`
`size N`
`orderby FIELD_NAME`

- `BO_NAME` is the name of the business object that make up the toend or the fromend.
- `CONNECTION_ID` id the id of the connections that make up the toend or the fromend.
- `EXPR` is the string containing the value of the where clause.
- PATTERN for is a name pattern for the relationship or vault.
- `FIELD_NAME` is the selectable you specify for sorting when using the orderby clause. You can prefix the `FIELD_NAME` with a “+” to sort in ascending order or “-” to sort in descending order. If no prefix is specified, the results are sorted in ascending order. You can use a maximum of three orderby clauses in a query connection command.

For example:

```
query connection to Part T22 0 where ATTR_NAME==”Material
Used” orderby +name orderby type;
```

For information other query connection clauses, see *Expand Business Object*.

Query connection commands cannot be saved for later evaluation as can be done with the add query command.

context Command

Description

Setting *context* identifies the user and which areas of access the current user maintains.

User level

Business Administrator

Syntax

```
[set|push|pop|print]context [CLAUSEs];
```

- CLAUSEs provide additional information about context.

Set Context

Context is controlled with the Set Context command which identifies a user by specifying the person name and vault:

```
set context [ITEM {ITEM}];
```

- `ITEM` is a Set Context clause.

The Set Context clauses provide more information about the context you are setting. They are:

`person PERSON_NAME`
`password VALUE`
`newpassword VALUE`
`vault VAULT_NAME`
`role ROLE_NAME`
`vault VAULT_NAME`

The Person clause can be replaced with the User clause. In both cases a defined person must be entered. A group or role is not allowed.

Using the `ITEM` clause `role`, Application programmers can provide a means for users to specify their role when setting context using MQL or Studio Customization Toolkit (see javadoc for specifics on Studio Customization Toolkit classes). The role has an effect on access ONLY if there are access filters in place that reference the entered role value with the context.role selectable.

A context role remains valid only for the session for which it was set. Within a session a context role can be changed anytime by setting a new context.

When resetting a context, if the user specifies a role that does not exist, then the previous context and role are retained.

For example, suppose only users with the PRODesigner role have access to project PRO objects and only users with CIMDesigner role have access to the project CIM objects. User Ted initially starts a session with the role PRODesigner to access the PRO project.

```
set context user Ted password *** role PRODesigner;
```

Now, if he wants to access the project CIM, he can reset his context and login using the CIMDesigner role.

```
set context user Ted password *** role CIMDesigner;
```

How you set the context varies based on whether the person definition includes a Password, No Password, or Disable Password clause. Each situation is described in *Working with Context* in Chapter 3.

Push Context

The `push context` command changes the context to the specified person and places the current context onto a stack so that it can be recovered by a `pop context` command. `Push context` can also be issued with no additional clauses, in which case the current context would be placed on the stack, but the current context would be unchanged. The command syntax is:

```
push context [person PERSON_NAME] [password VALUE] [vault VAULT_NAME];
```

Pop Context

The `pop context` command takes the last context from the stack and makes it current. The command syntax is:

```
pop context;
```

For example, a user needs to run a wizard that requires checkin access for a business object. Since the creator of the wizard does not know if all users who run the wizard will have checkin access, the wizard first changes context to assure checkin access:

```
push context user Taylor password runwizard vault Denver;
```

The user Taylor is a person who has checkin access. It might even be a person definition created specifically for the purpose of running wizards.

The last thing the wizard does before the program terminates is to use the `pop context` command to set context back to whatever it was before the wizard was run. The wizard creator does not have to know who is running the wizard or what the original context was.

Print Context

The print context command prints the a context definition to the screen allowing you to view it. The various clauses that make up that context definition are shown.

cue Command

Description

Cues control the appearance of business objects and relationships inside any browser. They make certain objects and relationships stand out visually for the user who created them.

For conceptual information on this command, see *Cues* in Chapter 6.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]cue NAME {CLAUSE};
```

- NAME is the name of the cue you are defining. Cue names cannot include asterisks.
- CLAUSEs provide additional information about the cue.

Add Cue

To create a new cue from within MQL, use the Add Cue command:

```
add cue NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the cue you are defining.
- USER_NAME can be included with the user keyword if you are a business administrator with person access defining a cue for another user. If not specified, the cue is part of the current user's workspace.
- ITEM specifies the characteristics you are setting.

When assigning a name to the cue, you cannot have the same name for two cues. If you use the name again, an error message will result. However, several different users could use the same name for different cues. (Cues are local to the context of individual users.)

After assigning a cue name, the next step is to specify cue characteristics (ITEM). The following are Add Cue clauses:

[!|in|notin|not]active
appliesto businessobject|relationship|all
type TYPE_PATTERN
name PATTERN
minorrevision REVISION_PATTERN
color COLOR

```
font FONT
highlight COLOR
vault PATTERN
linestyle solid|bold|dashed|dotted
order -1|0|1
owner PATTERN
where QUERY_EXPR
[!|not]hidden
visible USER_NAME{,USER_NAME};
property NAME on ADMIN [to ADMIN] [value STRING]
```

Appliesto Clause

This clause indicates that the cue applies to a business object, a relationship, or both (all). Since relationships and objects can have the same attributes, cues must specify to which they should apply.

When defining a cue for an object, you can use color and font to highlight it. When defining a cue for a relationship you can use color and line style, as described below.

Type Clause

This clause assigns a cue for a particular type of business object or relationship.

```
type TYPE_PATTERN
```

- `TYPE_PATTERN` defines the types for which you are assigning a cue.

The Type clause can include more than one type by using multiple values to define the pattern. The Type clause can also use wildcard characters.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, the Live Collaboration reads the value as the next part of the cue definition.

Name Clause

This clause assigns the names of objects to include in the cue.

```
name PATTERN
```

`PATTERN` is the names of objects to include in your cue.

The Name clause can include more than one name by using multiple values to define the pattern. The Name clause can also use wildcard characters.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, "International Business Machines"). If you include spaces, the Live Collaboration reads the value as the next part of the cue definition, as described in the Type clause.

Minorrevision Clause

This clause assigns a cue for a particular revision of business objects.

```
minorrevision REVISION_PATTERN
```

- REVISION_PATTERN defines the revision for which you are assigning a cue.

The Minorrevision clause can include more than one revision value and wildcards, as in the other clauses that use patterns. Typically, you might use a wildcard, but would not include more than one or two revisions. Adding a cue for the latest revision number can highlight only the most current business objects in a view. (To do so, you would use where clause "revision == last").

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, the Live Collaboration reads the value as the next part of the cue definition.

Color, Font, Highlight, and Linestyle Clauses

These clauses define the visual characteristics of the objects/relationships. The options for an object are color and font, while the options for a relationship are color and line style.

```
color COLOR
font FONT
highlight COLOR
linestyle solid|bold|dashed|dotted
```

COLOR is the name of the color to display for the object text and/or the relationship arrow. When COLOR is defined for highlight, this applies when the object is highlighted (selected).

You can enter a style of solid, bold, dashed, or dotted for the relationship arrow displayed between objects. This control is available only if you are applying the cue to relationships.

Vault Clause

This clause assigns a cue for business objects that are in a particular or similar vault:

```
vault PATTERN
```

- PATTERN defines the vault(s) for which you are assigning a cue.

The Vault clause can use wildcard characters. For example, the following definition displays a cue for all business objects that reside in the "Vehicle Project" vault:

```
add cue "Vault Search"
   businessobject * * *
   vault "Vehicle Project"
   owner *;
```

Order Clause

This clause defines the order in which the cue is applied in relation to other cues: before, with, or after other cues. If more than one cue can apply to an object, you need to define which cues to present. The before, with, and after ordering scheme establishes these priorities.

```
order -1|0|1
```

- `-1` is the lowest priority. These cues are applied first with any subsequent cue changes allowed.
- `0` ranks the cue in the order in which they are activated.
- `1` is the highest priority. Objects will change to these cues after any others are first applied.

Copy Cue

You can modify any cue that you own, and copy any cue to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, business administrators can change their workspace to that of another user to work with cues that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

After a cue is defined, you can clone the definition with the Copy Cue command.

If you are a Business Administrator with person access, you can copy cues to and from any person's workspace (likewise for groups and roles). Other users can copy visible cues to their own workspaces.

This command lets you duplicate cue definitions with the option to change the value of clause arguments:

```
copy cue SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the cue definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

COPY_ITEM	Specifies that…
`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any cue of the same name belonging to the user specified in the `touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Cue

Use the Modify Cue command to add or remove defining clauses and change the value of clause arguments:

```
modify cue NAME [user USER_NAME] [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the cue you want to modify. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.

MOD_ITEM is the type of modification you want to make. With the Modify Cue command, you can use these modification clauses to change a cue:

MODIFY_ITEM	Specifies that…
active	The active option is changed to specify that the object is active.
notactive	The active option is changed to specify that the object is not active.
appliesto businessobject|relationship|all	The appliesto option is changed to reflect what the cue now affects.
type TYPE_PATTERN	The type criteria is changed to the named pattern.
name PATTERN	The name criteria is changed to the named pattern.
revision REVISION_PATTERN	The revision criteria is changed to the named pattern.
color COLOR	The color is changed to the new color.
font FONT	The font is changed to the new font.
highlight COLOR	The highlight is changed to the new color.
vault PATTERN	The vault is changed to the pattern specified.
linestyle solid|bold|dashed|dotted	The linestyle is changed to the new linestyle specified.
order -1|0|1	The order is modified.
owner PATTERN	The owner is changed to the pattern specified.
where QUERY_EXPR	The query expression is modified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
visible USER_NAME{,USER_NAME};	The object is made visible to the other users listed.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

When making modifications, you simply substitute new values for the old. As you can see, each modification clause is related to the clauses and arguments that define the cue.

Although the Modify Cue command allows you to use any combination of criteria, no other modifications can be made. To change the cue name or remove the cue entirely, you must use the Delete Cue command (see *Delete Cue*) and/or create a new cue.

For example, assume you have a cue named “Product Comparisons” with the following definition:

```
cue “Product Comparisons”
   type FormulaA
   name Lace
   revision 3
   color blue
   highlight yellow
   vault “Perfume Formulas”
   owner channel,taylor;
```

To this cue, you want to add another owner for the criteria. To make the change, you would write a modify cue command similar to the following:

```
   modify cue “Product Comparisons”
      owner channel,taylor,cody;
```

This alters the cue so that it now appears as:

```
cue “Product Comparisons”
   type FormulaA
   name Lace
   revision 3
   color blue
   highlight yellow
   vault “Perfume Formulas”
   owner channel,taylor,cody
```

Delete Cue

If a cue is no longer needed, you can delete it using the Delete Cue command:

```
delete cue NAME;
```

- `NAME` is the name of the cue to be deleted. If you are a business administrator with person access, you can include the user clause to indicate another user’s workspace object.

Searches the local list of existing cues. If the name is found, that cue is deleted. If the name is not found, an error message results.

For example, assume you have a cue named “Overdue Invoices” that you no longer need. To delete this cue from your area, you would enter the following MQL command:

```
delete cue “Overdue Invoices”;
```

After this command is processed, the cue is deleted and you receive the MQL prompt for the next command.

dataobject Command

Description

Dataobjects are a type of workspace object that provide a storage space for preference settings and other stored values for users.

For conceptual information on this command, see *Dataobjects* in Chapter 7.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]dataobject NAME [CLAUSEs];
```

- NAME is the name of the dataobject you are defining. The dataobject name cannot include asterisks. When assigning a name to the dataobject, you cannot have the same name for two dataobjects. If you use the name again, an error message will result. However, several different users could use the same name for different dataobjects. (Dataobjects are local to the context of individual users.)
- CLAUSEs provide additional information about the dataobject.

Add Dataobject

Dataobjects can be created in MQL only.

To create a new dataobject, use the Add dataobject command:

```
add dataobject NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the dataobject you are defining. The dataobject name cannot include asterisks. When assigning a name to the dataobject, you cannot have the same name for two dataobjects. If you use the name again, an error message will result. However, several different users could use the same name for different dataobjects. (Dataobjects are local to the context of individual users.)
- USER_NAME can be included with the user keyword if you are a business administrator with person access defining a dataobject for another user. If not specified, the dataobject is part of the current user's workspace.
- ADD_ITEM further defines the dataobject.

The following are Add dataobject clauses:

description STRING_VALUE
type TYPE_STRING
value VALUE_STRING
[!|not]hidden

```
visible USER_NAME{,USER_NAME};
property NAME [to ADMIN] [value STRING]
```

Type Clause

This clause may be used as a way of classifying various data objects. It does not refer to Types, but can be set to any string value up to 255 characters long.

```
type TYPE_STRING
```

ValueClause

This clause can be used to hold up to 2 gb of data.

```
value VALUE_STRING
```

This is where cached values and customized pages will be stored.

Copy Dataobject

After a dataobject is defined, you can clone the definition with the Copy dataobject command.

If you are a business administrator with person access, you can copy dataobjects to and from any person's workspace (likewise for groups and roles). Other users can copy visible dataobjects to their own workspaces.

This command lets you duplicate dataobject definitions with the option to change the value of clause arguments:

```
copy dataobject SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the dataobject definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

COPY_ITEM	Specifies that…
`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any dataobject of the same name belonging to the user specified in the `Touser` clause.

The order of the `Fromuser`, `Touser` *and* `Overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Dataobject

Use the Modify dataobject command to add or remove defining clauses and change the value of clause arguments:

```
modify dataobject NAME [user USER_NAME] [MOD_ITEM
{MOD_ITEM}];
```

- NAME is the name of the dataobject you want to modify. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.
- MOD_ITEM is the type of modification you want to make. With the Modify dataobject command, you can use these modification clauses to change a dataobject:

MOD_ITEM	Specifies that…
description STRING_VALUE	The description is changed to the STRING_VALUE given.
value VALUE_STRING	The value is changed to the VALUE _STRING given.
type TYPE	The type value is changed to TYPE. This value is not a defined Type, but can be used as a way to classify dataobjects.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
visible USER_NAME{,USER_NAME};	The object is made visible to the other users listed.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

When making modifications, you simply substitute new values for the old. As you can see, each modification clause is related to the clauses and arguments that define the dataobject.

Although the modify dataobject command allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the dataobject name or remove it entirely, you must use the Delete dataobject command and/or create a new dataobject.

Delete Dataobject

If you do not need a dataobject, you can delete it using the Delete dataobject command:

```
delete dataobject NAME [user USER_NAME];
```

- NAME is the name of the dataobject to be deleted. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.

Searches the local list of existing dataobjects. If the name is found, that dataobject is deleted. If the name is not found, an error message results.

dimension Command

Description

The *dimension* administrative object provides the ability to associate units of measure with an attribute, and then convert displayed values among any of the units defined for that dimension.

For conceptual information on this command, see *Dimensions* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]dimension NAME [CLAUSEs];
```

- NAME is the name you assign to the dimension. Dimension names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the dimension.

Add Dimension

Before attributes can be defined with a dimension, the dimension must be created. You can define a dimension if you are a business administrator with Attribute access, using the add dimension command:

```
add dimension NAME [ADD_ITEM {ADD_ITEM}];
```

ADD_ITEM provides additional information about the dimension you are defining.

The Add Dimension clauses are:

description VALUE
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
unit UNITNAME [UNIT_ITEM {UNIT_ITEM}]
history STRING

Each clause and the arguments they use are discussed in the sections that follow.

Unit Clause

The Unit clauses are:

```
label UNITLABEL
multiplier MULTIPLIER
offset OFFSET
[!|not]default
property NAME [to ADMINTYPE NAME] [value STRING]
setting NAME VALUE
```

Label Clause

The Label clause provides a description of the units. For example, if the unit is F, the label could be degrees Fahrenheit.

Multiplier Clause

The Multiplier clause provides the value to multiple against the normalized value when calculating the value for units other than the default units.

Default Clause

The Default clause of the Add Unit command defines the unit as the default for the dimension. Default units are the normalized units–the units used to store the attributes value in the database and with a multiplier of 1 and offset of 0.

Settings Clause

Settings are allowed on a unit and are visible in Business Modeler.

History Clause

The `history` keyword adds a history record marked “custom” to the dimension that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Dimension

You can clone the definition with the Copy Dimension command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy dimension SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the dimension definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).

- MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The history keyword adds a history record marked “custom” to the dimension that is being copied. The STRING argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Dimension

After a dimension is defined, you can change the definition with the Modify Dimension command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify dimension NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the dimension you want to modify.
- MOD_ITEM is the type of modification you want to make.

You can make the following modifications to an existing dimension definition. Each is specified in a Modify Dimension clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Dimension Clauses	Specifies that…
name NEW_NAME	The current dimension name changes to the new name entered.
icon FILENAME	The image is changed to the new image in the file specified.
description VALUE	The current description value, if any, is set to the value entered.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
add unit UNITNAME [UNIT_ITEM {UNIT_ITEM}]	The new unit UNITNAME is added to the dimension.
remove unit UNITNAME	The unit is removed from the dimension. *You cannot remove a unit from a dimension if it is the default unit and has been applied to an attribute, or if it has any business object instantiations that use it as the input unit.*
modify unit UNITNAME [UNIT_NAME] {UNIT_ITEM}]	The unit is modified in accordance with the accompanying Unit clause.
modify unit UNITNAME system SYSTEMNAME to unit UNITNAME	For the named system association, replace the first named unit with the second named unit
modify unit UNITNAME remove property NAME to ADMIN	The named property is removed from the unit
modify unit UNITNAME remove property NAME	The named property is removed from the unit.

Modify Dimension Clauses	Specifies that…
`modify unit UNITNAME remove setting NAME`	The named setting is removed from the unit.
`modify unit remove system SYSTEMNAME to unit UNITNAME`	The named unit is removed from the named system, but not removed from the dimensions
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property of the dimension is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added to the dimension.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed from the dimension.
`history STRING`	Adds a history record marked “custom” to the dimension that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Assume you have the following dimension definition:

```
dimension Length
   description "Absolute length of item"
   unit "Meter"
   label
   multiplier 1.0
   offset 0.0
   unit "Centimeter"
   label
   multiplier .01
   offset 0.0;
```

Now you have decided to add meters as a unit of the dimension. You can modify this dimension with the following command:

```
modify dimension Length
   add unit "Meter" ;
```

After this command is processed, the dimension definition for the Units attribute is:

```
dimension Length
   description "Absolute length of item"
   unit "Meter"
   label
   multiplier 1.0
   offset 0.0
   unit "Centimeter"
   label
   multiplier .01
   offset 0.0
   unit "Foot"
   label
   multiplier 0.3048
   offset 0.0;
```

The system modifier allows units to be grouped and associated with other units within a dimension. For example, if you have a Length dimension, you can group meter, centimeter, and millimeter with a system modifier named Metric, and feet and inches with a system modifier English. When specifying a system for a unit, you also specify which units of that system you want the current unit to be converted into. When converting lengths from English to Metric, you may want inches to be converted into centimeters, while feet are converted into meters as shown in this example dimension definition:

```
dimension Length
  unit In
    label      Inches
    multiplier 1.0
    offset     0.0
    system metric to unit Cm
  unit Ft
    label      Feet
    multiplier 12.0
    offset     0.0
    system metric to unit Meter
```

Delete Dimension

If a dimension is no longer required, you can delete it by using the Delete Dimension command:

```
delete dimension NAME;
```

- `NAME` is the name of the dimension to be deleted.

You can only delete a dimension if it has not been applied to any attributes.

Searches the list of defined dimensions. If the name is found, that dimension is deleted if it has not been applied to any attribute. If the name is not found, an error message is displayed.

For example, to delete the Length dimension, enter the following command:

```
delete dimension Length;
```

After this command is processed, the dimension is deleted and you receive an MQL prompt for another command.

Example

This Add Dimension command creates the dimension Length with units “in”, as the default, then adds units for cm, ft, and meter with the appropriate multipliers. For this dimension, no offset is required,

```
add dimension Length unit in default \
  unit cm multiplier 0.393700787 \
  unit ft multiplier 12 \
  unit meter multiplier 39.3700787;
```

download Command

Description

The *download* and upload MQL commands are available for use in programs to be executed by a Web client through a Server. They allow client-side files and directories to be manipulated by server-side programs (and vice versa). See *upload Command* for information on the upload command.

Generally the download and upload commands would be used in conjunction with a wizard ‘file text box’ widget. See *Widget Subclause* for details.

User level

Business Administrator

Syntax

Use the `download` command to download files to the client that are created by wizards on the server, or when a file resides on a server that must be processed by a client-side program.

```
download sourcefile SOURCE targetfile TARGET [[!|not]delete] [[!|not]overwrite];
```

- `SOURCE` specifies the directory path and filename of the file to be downloaded from the server, relative to `WORKSPACEPATH`. Refer to *Using ${} Macros* for more information on `WORKSPACEPATH`. Note that when used in a desktop environment, the file is moved from one location to another on the same machine.
- `TARGET` specifies the full path and filename on the client. If the specified directories do not exist, the Live Collaboration creates them.
- When `delete` is specified, the source file is deleted on the server after the download. The default is TRUE.
- When `overwrite` is specified, if `TARGET` filename already exists, the downloaded file replaces it. The default is FALSE.

Example

For example:

```
download sourcefile report.doc targetfile reports/report.doc !delete overwrite;
```

export Command

Description

Administrative definitions, business object metadata, and checked-in files, can be *exported* from one root database and imported into another. Exporting and importing can be used across schemas of the same version level, allowing definitions and structures to be created and “fine tuned” on a test system before integrating them into a production database.

For conceptual information on this command, see *Working with Import and Export* in Chapter 10.

User Level

System Administrator

Syntax

Use the Export command to export the definitions of a database. The complete export command for administrative objects is as follows:

```
export ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]| into | file
FILE_NAME
                                                              | onto |
[exclude EXCLUDE_FILE] [log LOG_FILE] [exception EX_FILE];
```

ADMIN_TYPE is the administrative definition type to be exported. It can be any of the following:

admin	location	role
application	menu	rule
association	package[members] [usescustompackage|,usesallpackage]	server
attribute	page	site
channel	pathtype[xml][!archive] [continue][incremental[N]]	store
command	person[!mail][!set]	table
customevent	policy	tenant
dimension	portal	tenantadmin
form	process	type
format	product	uniquekey
group	program	vault
index	relationship	workflow
interface	report	
inquiry	resource	

The ADMIN_TYPE is followed by a TYPE_PATTERN which filters the definitions to be exported.

To export wizards, use the program ADMIN_TYPE.

- OPTION_ITEM is an Export clause which further defines the requirements of the export to be performed. It can be any of the following:

!archive	incremental [N]	xml
continue	!mail	
!icon	!set	

!mail and !set (notmail and notset) are only meaningful when exporting person objects but should not cause syntax errors for other types.

- OPTION_ITEMs can be used in any order before the Into File/Onto File clause.
- FILENAME is the name of a new or existing file to which to write the export data.

Each clause is described in the sections that follow.

Into File/Onto File

Exported data can be appended to a file or written to a new file by using the `onto file` or `into file` forms of the Export command. The `into file` form creates a new file, or overwrites an existing file of the name specified as `FILENAME`. The `onto file` form appends to an existing file.

!Mail and !Set Clauses

When exporting person objects, there is an associated workspace. The workspace contains IconMail and sets (as well as Visuals) saved by the person being exported. When exporting Persons, workspaces are exported by default, but you can omit mail and sets using the `!mail` and `!set` clauses. These clauses are valid with the `export person` and `export admin` commands only. Refer to the discussion in *Migrating Databases* in Chapter 10 for information on why you would want to exclude these items.

`!mail` *and* `!set` *(notmail and notset) are only meaningful when exporting Person objects but should not cause syntax errors for other types.*

For example, to export all persons without mail or sets, the following can be used:

```
export person * !mail !set into file person.mix;
```

Incremental Clause

Use the `Incremental` clause to export a specified number (`N`) of unarchived objects. Unarchived objects are objects that have not yet been exported, or have been changed since they were last exported. If no number is specified, all unarchived objects are exported. Without this clause, export does not look at the `archived` setting and exports all objects fitting the criteria.

Note that while it is possible to export administrative changes incrementally, it is recommended that during database migration, Business and System Administration tasks are avoided.

!Archive Clause

Use the `!archive` clause not to set the archive setting on objects being exported. Use this clause to facilitate performance of large exports that do not require an incremental export.

Continue Clause

Use the `Continue` clause to make the Live Collaboration proceed with additional exports even if an error is generated. When continue is used, it is helpful to use the `log` and/or `exception` clauses as well, so that diagnostics can be performed, and the data that caused the error is trapped.

XML Clause

Use the XML clause to export data into XML format. For details, see *XML Export Requirements and Options* in Chapter 10.

Exclude Clause

Use the Exclude clause to point to a map file that lists any objects to be excluded. It includes the `exclude` keyword and a map file name. For example:

```
export admin TEST* into file teststuf.mix exclude
nogood.txt;
```

The contents of the exclude map file for Administrative objects must follow the following format:

```
ADMIN_TYPE NAME
ADMIN_TYPE NAME
```

`ADMIN_TYPE` can be any of the administrative object types: vault, store, location, server, attribute, program, type, relationship, format, role, group, person, policy, form, association, or rule.

`NAME` is the name of the definition instance that should be excluded in the import. Wildcard patterns are allowed.

As indicated, each definition to be excluded must be delimited by a carriage return. The Exclude clause is optional.

Log FILENAME Clause

Apply this clause if you want to specify a file for error messages and details of the export process. The output is similar to using the `verbose` flag, but includes more details.

Exception FILENAME Clause

`Exception FILENAME` provides a file location where objects are written if they fail to export. The file will contain the type and name of any objects that could not be exported. This is often used with a log file so that after the diagnostics are performed from the information in the log file, the exception file can be used as a guide to know what should be exported.

Export Bus Command

Use the Export Businessobject command to export business objects from a vault or a set:

```
export bus[inessobject] BUSID [from |vault VAULT_NAME|] [OPTION_ITEM
{OPTION_ITEM}]
                                   |set SET_NAME    |
| into | file FILENAME [FILE_TYPE FILENAME [FILE_TYPE FILENAME]...];
| onto |
```

- `BUSID` is the Type, Name, and Revision of the business object. Wildcard patterns are allowed. (You cannot use OIDs with export bus).
- `VAULT_NAME` is the name of the vault from which to export the business object(s). While neither is required, you can specify either a vault or a set from which to export, but not both.
- `SET_NAME` is the name of the set from which to export the business objects. While neither is required, you can specify either a vault or a set from which to export, but not both.

- OPTION_ITEM is an Export clause which further defines the requirements of the export. It can be any of the following:

!archive	!history	!state
!captured	!icon	xml
continue	incremental [N]	.
!file	!relationship	.

- OPTION_ITEMs can be used in any order before the into|onto file clause.
- FILENAME is the name of the file in which to store the exported information.
- FILE_TYPE FILENAME offers a way to log errors and trap exceptions, as well as to exclude objects in the export.
- FILE_TYPE can be any of the following:

exclude	log	exception

Note that when FILE_TYPEs are specified on export, the use *keyword is not required. However, when used on import, it is.*

For a detailed description of items in the export business object command, refer back to *export Command*.

Excluding Information

When exporting business objects, the default is to include everything about the object. However, the OPTION_ITEMS can be used to further define the requirements of the export to be performed, by specifying information to exclude. The following clauses can be used to exclude information when exporting business objects.

!captured	!icon	!relationship
!file	!history	!state

For example, to export a single object without including history, use:

```
export businessobject Assembly "ABC 123" A !history into
file obj.mix;
```

To export a single object without including its icon, use:

```
export businessobject Assembly "ABC 123" A !icon into file
obj.mix;
```

To export a single object without any of its relationships, use:

```
export businessobject Assembly "ABC 123" A !relationship
into file obj.mix;
```

To export all objects from vault TEMP and reset the current state to the beginning of the lifecycle, use:

```
export businessobject * * * from vault TEMP !state into file
obj.mix;
```

Exclude Clause

Use the Exclude clause to point to a file that lists any objects to be excluded. For example:

```
export businessobject TEST* into file teststuf.mix exclude nogood.txt;
```

The contents of the exclude file for business objects must follow the following format:

```
businessobject OBJECTID
businessobject OBJECTID
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search.

As indicated, each business object to be excluded must be delimited by a carriage return.

Excluding Files

When exporting (or importing) business objects, the options for its files are:

- `!file` -- does not export files when exporting business objects. By default, both file data and content are exported.
- `!captured` -- does not export captured store file content. This flag affects content only, not file metadata.
- Otherwise, both file metadata and actual file contents are written to the export file. In this case any file sharing is lost (resulting in duplication of files).

If you use `!file`, you cannot use the `captured` flag because files without metadata cannot be included. (Attempts to do this will not return an error, but no file information or content will be included). Refer to *Migrating Databases* in Chapter 10 for more information on file migration strategies.

expression Command

Description

Using MQL, *expressions* can be created and saved in the database to be evaluated against a business object, a connection, or from within a webreport, against a collection of business objects or connections. Once created, saved expressions can be referenced by name in a business object or connection Select clause. Since Select clauses can be embedded in expressions, they can be used in Where clauses and access filters as well. Expression objects can also be referenced by name in webreports.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]expression NAME [CLAUSEs];
```

- NAME is the name of the expression you are defining. The name you choose is the name that will be referenced to use the expression. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the attribute.

Add Expression

To define an expression from within MQL use the add expression command:

```
add expression NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides additional information about the expression.

The add expression clauses are:

description STRING_VALUE
value VALUE
property NAME [to ADMIN] [value STRING]

Value Clause

This clause is the string that defines the expression. It can contain selects, comparisons, equations, etc. Enclose the value string in quotes if it contains any space.

Refer *Formulating Expressions for Collections* and *Evaluating Expressions* for more information and examples.

Formulating Expressions on Business objects or Relationships

If-Then-Else

If-then-else logic is available for expressions. The syntax is:

```
if EXPRESSION1 then EXPRESSION2 else EXPRESSION3
```

- The EXPRESSION1 term must evaluate to TRUE, FALSE, or UNKNOWN.

If the EXPRESSION1 *term evaluates to UNKNOWN, it is treated as TRUE.*

The if-then-else expression returns the result of evaluating EXPRESSION2 or EXPRESSION3 depending on whether or not EXPRESSION1 is TRUE or FALSE.

Note that only one of EXPRESSION2 or EXPRESSION3 is evaluated. So if the expressions have side-effects (which can happen since expressions can run programs), these effects will not occur unless the expression is evaluated.

```
eval expr' if (attribute[Actual Weight] > attribute[Target Weight])
  then ("OVER") else ("OK")' on bus 'Body Panel' 610210 0;
```

Substring

The substring operator works within an expression to provide the ability to get a part of a string; the syntax is:

```
substring FIRST_CHAR LAST_CHAR EXPRESSION
```

The substring operator works as follows:

- The FIRST_CHAR and LAST_CHAR terms must evaluate to numbers that are positive or negative, and whose absolute value is between 1 and the number of characters in the string returned by EXPRESSION.
- The numbers returned by these terms indicate a character in the string returned by EXPRESSION.
- The characters are counted so that ‘1’ refers to the first character. A negative number indicates the character found by counting in the reverse direction. So ‘-1” refers to the last character.
- The substring operator returns the part of the string returned by EXPRESSION consisting of the characters from the FIRST_CHAR character to the LAST_CHAR character, inclusive.
- If FIRST_CHAR evaluates to a character that is after the character indicated by LAST_CHAR, an empty string is returned.

To obtain the last 4 characters of a 10-character phone number, use:

```
eval expression 'substring -4 -1 attribute[Phone Number]' on bus Vendor 'XYZ Co.'
0;
```

Dateperiod

Dateperiod allows you to determine the period (year, quarter, month, week and/or day) in which a given date has occurred. You include 2 arguments with the keyword dateperiod; the first defines the period to evaluate; the second is the date in question. To define a dateperiod, you can include one or more of the following periods to inquire about:

y	Returns a 4-digit year

`q`	Returns 1 digit indicating the calendar year quarter, with possible values of 1,2,3, or 4.
`m`	Returns a 2-digit month
`w`	Returns a 2-digit week, with possible values of 01 to 53, calculated using International Standard ISO 8601 (http://www.cl.cam.ac.uk/~mgk25/iso-time.html). The first week of a new year (Week 01) is the week that has the majority of its days in the new year. Week 01 might also contain days from the previous year and the week before week 01 of a year is the last week (52 or 53) of the previous year even if it contains days from the new year. A week starts with Monday (day 1) and ends with Sunday (day 7). For example, the first week of the year 1997 lasts from 1996-12-30 to 1997-01-05; and the first week of 2005 lasts from 2005-01-03 to 2005-01-09.
`d`	Returns a 2-digit date corresponding to the day of the month
`fq##`	## is 2 digits representing a month. fq## returns a 1-digit fiscal quarter whose year is assumed to start with the indicated month.
`fy##`	## is 2 digits representing a month. fy## returns a 4 digit fiscal year where the year is assumed to start with the indicated month.

Any other characters are simply returned with the expression's output.

For example, the following expression determines in which fiscal quarter the objects it is evaluated against are due:

```
add expr QuarterlyDeliverables value 'dateperiod fq07
  attribute[DueDate]';
```

Other examples:

`dateperiod ym 2/03/04`	`returns 200402`
`dateperiod dwqy 2/03/04`	`returns 030512004`

Using Dates in Expressions

The following calculations can be performed on dates within an expression:

- subtract two dates, obtaining a number of seconds
- add or subtract a number (of seconds) to/from a date
- use the string MX_CURRENT_TIME for the current date/time

For example, the following could be used to determine how old an object is (in hours):

```
evaluate expr '(MX_CURRENT_TIME - state[Released].actual) / 3600' on bus 'Body
Panel' 610210 0
```

The following returns the average age of all objects in the set M6000-panels (in hours):

```
evaluate expr 'average ( (MX_CURRENT_TIME - state[Released].actual) / 3600)' on
set M6000-panels;
```

Formulating Expressions for Collections

Certain kinds of expressions are applicable to collections of business objects (such as Query results or sets) or connections, returning a single answer for the entire collection. These expressions are formed by using one of several keywords. They are:

`count`
`sum`
`maximum`
`minimum`
`average`
`median`
`standarddeviation (stddev or stddeviation)`
`correlation (cor)`

All but the last of these keywords is expected to be followed by one expression, its *argument*, that applies to business objects. The last one, correlation, needs to be followed by two such expressions. Alternative spellings are indicated in parentheses. For each keyword, you can use all lowercase, all uppercase, or first character uppercase followed by all lowercase.

Count

The `count` keyword takes as its argument a Where clause expression that evaluates to TRUE or FALSE. It returns an integer that is the number of items in the collection that satisfy the argument. A simple example of such an expression is “count TRUE”, which evaluates to the number of objects in a collection of business objects.

For example, when the following expression is evaluated, it indicates the number of objects of Type ‘Body Panel’ in the database:

```
eval expr 'count TRUE' on temp query bus 'Body Panel' * *;
```

Evaluating the following returns the number of objects in the set “NewBooks” whose cost is between 10 and 50:

```
add expr LowCost value 'attribute[cost]>=10 AND
attribute[cost]<=50';
eval expr 'count expression[LowCost]' on set NewBooks;
```

To do the same, but exclude children’s books, you could use the following:

```
eval expr 'count expression[LowCost]' on set NewBooks LESS
temp query bus Book * * where 'attribute[Reading
Level]==Child';
```

Or change the expression to:

```
add expr NewAdultLowCostBooks value 'attribute[cost]>=10
AND attribute[cost]<=50 AND attribute[Reading
Level]!=Child';
```

Sum

`Sum` returns a real number that represents a total of all the values of the specified attribute for all business objects in the collection. For example, when evaluated, the following returns the total of the values of the "Amount Due Employee" attribute for all business objects in the saved query "Expense Reports":

```
add expr Expenses value 'attribute[Amount Due Employee]';
eval expr 'sum expression[Expenses]' on query Expense
Reports;
OR
eval expr "'sum attribute[Amount Due Employee]' on query
Expense Reports";
```

The following commands evaluate the ratio of total price to total cost for all objects in the set "Components":

```
add expression Price value 'attribute[price]';
add expression Cost value 'attribute[cost]';
eval expr '((sum expression[Price]) / (sum
expression[Cost]))' on set Components;
```

Maximum, Minimum, Average

`Maximum` returns a real number that represents the single largest value for the specified attribute for all business objects in the collection. For example, the following checks the value contained in the "diameter" attribute of each business object in the set "o-rings," and returns whichever value is the highest:

```
add expr MaxDiameter value attribute[diameter];
eval expression 'maximum expression[MaxDiameter]' on set
o-rings;
```

`Minimum` returns a real number that represents the single smallest value for the specified attribute for all business objects in the collection.

```
eval expr "'minimum attribute[diameter]' on set "o-rings"";
```

`Average` returns a real number that represents the average of all values for the specified attribute for all business objects in the collection.

```
eval expr "'average attribute[diameter]' on set "o-rings"";
```

Median

`Median` returns a real number that represents the middle number of all values for the specified attribute for all business objects in the collection.

For example, the following shows the values, listed in numerical order, for the attribute Actual Weight for seven business objects that comprise the set FprSet:

Since the middle number of seven numbers is the fourth number, the median in this case is 25. That is the value returned for the following command:

```
eval expr "'median attribute[Actual Weight]' on set FprSet";
```

Standard Deviation

`Standard deviation`, generally used in statistical analysis, tells how closely data conforms to the mean in a set of data. There are two formulas for standard deviation; one for calculating the standard deviation given all elements of some population; another for when using a sample to get a good estimate for the population. The Live Collaboration assumes the stddev expression is evaluated over the entire population, and not just a sample.

Use `Standard deviation` to compare the values of business object attributes. The returned value is a real number.

For example, if you know the average age of all employees, you might want to know how many people are close to that age. The standard deviation will tell you, on average, how much the ages of the group differ from the mean. If the standard deviation is a small number, it could indicate that most of the people are close to the average age. If the standard deviation is a large number, it could indicate that there is a broader spread of ages.

The following example performs a standard deviation on the age attribute of all persons in the set Employees:

```
eval expr "'stddev attribute[age]' on set Employees";
```

Correlation

`Correlation`, generally used in statistical analysis, is a direct measure of the relationship between variables. It can be used to determine the relationship between attributes of an object or group of objects. The returned value (the correlation coefficient) is a real number between -1 and +1.

- If the returned value is between 0 and +1 (positive correlation), it indicates that an increase in the value of one attribute results in an increase in the value of the other (or vice-versa).
- If the returned value is between 0 and -1 (negative correlation), it indicates that an increase in the value of one attribute results in a decrease in the value of the other (or vice-versa).
- A returned value of 0 represents no relationship.

For example, the following expression can be used to check how well cost correlates with price for all objects of Type “tire frame”.

```
eval expr "'cor attribute[Cost] attribute[Price]' on temp query bus 'tire frame'
* *";
```

Evaluating Expressions

The `evaluate expression` command makes it possible to evaluate a temporary expression that is not saved. It also allows evaluation of a statistical expression against any collection of

business objects that can be defined through various operations of combining, intersecting, or subtracting the collection of business objects from one set, query, temp query, or expand command with/from another.

To evaluate an expression, use the evaluate expression command:

```
evaluate expression EXPRESSION {EXPRESSION] on ON_ITEM {on ON_ITEM} DUMP
[RECORDSEP];
```

EXPRESSION is an expression.

ON_ITEM can be any of the following clauses:

list businessobject SEARCHCRITERIA
relationship
SEARCHCRITERIA
businessobject TYPE NAME REV
relationship ID

Expressions can be evaluated against individual objects or against groups of objects. When the on clause is used, the expression will be evaluated as a single-object expression. The on clause can be repeated to get the value for multiple separate objects. For example::

```
eval expr 'attribute[string-u] + "||" + attribute[color-u]' on bus t2 t2-1 0 on
bus t2 t2-2 0;
```

You can also identify a collection of objects for the expression evaluation using the keywords set, query, temp set, temp query, or expand in the SEARCHCRITERIA. When any of these keywords are used, the expression will be evaluated as a collection, and must use one of these collection expressions:

Count, sum, maximum, minimum, average, median, standarddeviation, correlation

If you use a single-object expression and provide a searchcriteria that identifies a collection or vice-versa, there will be no output.

Expressions can be evaluated on relationships. For information on how to find relationship IDs, refer to *To and From Clauses*.

Expressions can also be evaluated on a specified business object.

Expressions can also be evaluated on a collection of business objects, as explained below.

The DUMP and recordseparator clauses can occur anywhere in the command, but the EXPRESSIONs *must be given before any* ON_ITEMs.

This command outputs the result of evaluating each expression for each ON_ITEM one after the other.

- The dump separator character (a comma, by default) separates each value for a single ON_ITEM.
- The recordseparator character (a new line, by default) separates the results of one ON_ITEM from the next.

Evaluating Saved Expressions

You can evaluate a saved expression as part of an expression string, or by using the Select Expression clause with print bus or print connection commands. For example:

```
eval expr 'count expression[SAVEDEXPRESSION]' on set myset;
print bus OBJECTID select expression[SAVEDEXPRESSION];
print connection CONNECTID select expression[SAVEDEXPRESSION];
```

You can also do this in a webreport:

```
add expression PriceToCost value 'sum ( attribute[price] ) /
sum ( attribute[cost] )';
temp webreport searchcriteria 'set Components' data object
PriceToCost;
```

Note that in this last example the expression object is being evaluated against a collection.

SEARCHCRITERIA Clause

SEARCHCRITERIA is defined recursively as one of:

`set NAME`
`query NAME`
`temp set BO_NAME{,BO_NAME}`
`temp query businessobject TYPE NAME REVISION [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM }] [casesensitive]`
`expand [businessobject] BO_NAME [EXPAND_ITEM {EXPAND_ITEM}] on SEARCHCRITERIA [EXPAND_ITEM {EXPAND_item}`
`query connection [type PATTERN] [vault PATTERN] [owner PATTERN] [where WHERE_CLAUSE] [limit N]`
SEARCHCRITERIA BINARY_OP SEARCHCRITERIA
`[( {( }]SEARCHCRITERIA [ ){ )}]`

TEMP_QUERY_ITEM is one of:

`owner NAME`
`vault NAME`
`[!|not]expandtype`
`where WHERE_CLAUSE`
`limit NUMBER`
`over SEARCHCRITERIA`
`querytrigger`

OBJECTID is the Type Name and Revision of the business object.

The values of TEMP_QUERY_ITEM have the same meaning in this context as they have for the temp query command, as described in *Temporary Query*.

EXPAND_ITEM is one of:

from
to
type PATTERN {,PATTERN}
relationship PATTERN {,PATTERN}
select businessobject
select relationship
where WHERE_CLAUSE
activefilters
reversefilters
filter PATTERN
view NAME
recurse to N [leaf]
recurse to all [leaf]

The values of EXPAND_ITEM have the same meaning in this context as they have for the expand bus command, as described in *Expand Business Object*, with a few exceptions:

- In the case of “select,” it indicates whether a subsequent ‘Where clause’ applies to business objects or to relationships. In expand bus select has an additional meaning that is not applicable here.
- Leaf can be used with recurse in a SEARCHCRITERIA expand to specify that only “leaf” nodes of the expand should be returned. What counts as a leaf node differs depending on whether “recurse to all” or “recurse to n” is issued. In the “all” case, a leaf node is one that has no children returning from the expand. With a recursion level indicated, a leaf node is one that is at that level.

  Leaf is not allowed in the expand bus command, but is allowed here.

An “expand” searchcriteria does not make sure that returned objects are unique. That is, any objects that are connected more than once to the object(s) being expanded (or recursed to) will be listed more than once. This affects webreports, eval expression, and other commands using searchcriteria. For example, the following might return a list containing duplicate entries:

```
MQL<18>eval expres "count TRUE" on ( expand bus t2 t2-1 0 );
```

To avoid the duplication, you could change the searchcriteria to the following:

```
temp query * * * over expand bus t2 t2-1 0
```

The output from evaluating two expressions, E1 and E2, on two sets, A and B, with values V1A, V2A, V1B, and V2B would look like the following (using default separators):

```
V1A,V2A

V1B,V2B
```

SEARCHCRITERIAs can be linked with binary operators, which follow simple, intuitive rules:

- and—an object is in both collections
- or—an object is in one or the other collection
- less—an object is in the result if it is in the left-hand collection but not the right-hand one.

If there is more than one binary operator in a SEARCHCRITERIA, parentheses must be used to disambiguate the clause. (There is no implied order of operations.) You must include a space before and after each parenthesis. In addition, the number of left and right parentheses must match each other. For example:

```
( set A + set B ) - set C
```

would probably evaluate differently than:

```
set A + ( set B - set C )
```

Examples:

This list of examples shows the power of the SEARCHCRITERIA concept.

The following command returns the number of objects in the set Projects:

```
eval expression "count TRUE" on set Projects;
```

The same thing could be written as follows (note the spaces around the parentheses):

```
eval expression "count TRUE" on ( set Projects );
```

The following command returns the number of objects returned by a query named “ask1” that are not Project 1029 0:

```
eval expression "count TRUE" on query ask1 less temp set Project 1029 0;
```

The following command returns the number of Jim’s open Projects that originated before the beginning of this year:

```
eval expression "count TRUE" on temp query bus Project * * owner Jim where
"originated < ’1/1/99’ and current == open";
```

The following command returns the number of open, new Projects that do not belong to Jim:

```
eval expression "count TRUE" on ( set openprojects AND set newprojects ) less
temp query bus Project * * owner Jim;
```

The following command returns the number of Projects that are either owned by Jim or included in the set “Q4 Projects”;

```
eval expression "count TRUE" on ( temp query bus Projects * * owner Jim ) OR set
"Q4 Projects";
```

Copy Expression

After an expression is defined, you can clone the definition with the copy expression command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy expression SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- SRC_NAME is the name of the expression definition (source) to be copied.
- DST_NAME is the name of the new definition (destination).
- MOD_ITEMs are modifications that you can make to the new definition. Refer to the command below for a complete list of possible modifications.
- Cloning an expression definition requires expression Business Administrator privileges. These can be granted via MQL or Business Modeler.

Modify Expression

Use the Modify expression command to add or remove defining clauses and change the value of clause arguments:

```
modify expression NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the expression you want to modify.
- MOD_ITEM is the type of modification you want to make. Each is specified in a Modify expression clause, as listed in the following command. Note that you need specify only the fields to be modified.

Modify Expression Clause	Specifies that…
`name NEW_NAME`	The current expression name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
`icon FILENAME`	The image is changed to the new image in the file specified.
`value VALUE`	The value is changed to the new value specified.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

Each modification clause is related to the arguments that define the expression. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

Delete Expression

If an expression is no longer required, you can delete it using the Delete expression commands

```
delete expression NAME:
```

- NAME is the name of the expression to be deleted.

Searches the list of defined expressions. If the name is found, that expression is deleted. If the name is not found, an error message is displayed. For example, to delete the expression named “Used Parts,” enter the following:

```
delete expression "Used Parts";
```

After this command is processed, the expression is deleted and you receive an MQL prompt for another command.

Example

The following example combines several of the capabilities available for expressions. It shows how to obtain the average number of days spent to move a feature from Open to Test for v7.0 and v7.1 through one MQL command. We divide by (3600 * 24) (the number of seconds in a day) because we want the number of days and the difference of two dates is given in seconds.

```
evaluate expression "(average (state[Test].actual -
state[Open].actual) )/ (3600 * 24)"
  on expand Product XYZ v7.0 to relationship Committed,
Candidate, Proposed type Feature
          where "current == Test or current == Closed"
  on expand Product XYZ v7.1 to relationship Committed,
Candidate, Proposed type Feature
          where "current == Test or current == Closed"
  dump "|";
```

The next example calculates the number of Features connected to v7.2 that have been promoted to Test in the past week (and have not been demoted).

```
evaluate expression "count (( MX_CURRENT_TIME -
state[Test].actual ) / (3600 * 24) < 7)"
   on expand XYZ 7.2 to relationship Committed, Candidate,
Proposed type Feature
          where "current == Test or current == Closed" dump
"|";
```

filter Command

Description

Filters limit the objects or relationships displayed in browsers to those that meet certain conditions previously set by you or your Business Administrator. For example, you could create a filter that would display only objects in a certain state (such as Active), and only the relationships connected *toward* each object (not *to and from*). When this filter is turned on, only the objects you needed to perform a specific task would display.

For conceptual information on this command, see *Filters* in Chapter 6.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]filter NAME {CLAUSE};
```

- NAME is the name of the filter you are defining.
- CLAUSEs provide additional information about the filter.

Add Filter

To define a new filter from within MQL, use the Add Filter command:

```
add filter NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

- USER_NAME can be included with the user keyword if you are a business administrator with person access defining a filter for another user. If not specified, the cue is part of the current user's workspace.
- ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the filter, you cannot have the same name for two filters. If you use the name again, an error message will result. However, several different users could use the same name for different filters. (Remember that filters are local to the context of individual users.)

After assigning a filter name, the next step is to specify the conditions (ADD_ITEM) that each object must meet in order to display when the filter is turned on (activated). The following are Add Filter clauses:

[!|in|notin|not]active
appliesto businessobject|relationship
from|to|both
type TYPE_PATTERN
name PATTERN

```
minorrevision REVISION_PATTERN
vault PATTERN
owner PATTERN
where QUERY_EXPR
[!|not]hidden
visible USER_NAME{,USER_NAME};
property NAME on ADMIN [to ADMIN] [value STRING]
```

Each clause and the arguments they use are discussed in the sections that follow.

Applies to Clause

This clause indicates that the filter applies to business objects or relationships.

Direction Clause

This clause indicates the direction of the relationships to which the filter applies: `to`, `from`, or `both`.

Type Clause

This clause assigns a filter for a particular type of business object or relationship.

```
type TYPE_PATTERN
```

`TYPE_PATTERN` defines the types for which you are assigning a filter.

The Type clause can include more than one type by using multiple values to define the pattern. The Type clause can also use wildcard characters. For example, the following definition displays a filter to display all customers and other users:

```
add filter "Customers and Other Users" type customer,user;
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (e.g., `"new customer",user`). If you include spaces, the value is read as the next part of the filter definition. For example, the following Type clause would produce an error because MQL reads the type as "new" and the next specification for the filter as "customer".

```
type new customer;
```

Name Clause

This clause assigns the names of objects to include in the filter.

```
name PATTERN
```

- `PATTERN` is the names of objects to include in your filter.

The Name clause can include more than one name by using multiple values to define the pattern. The Name clause can also use wildcard characters. For example, the following definition adds a filter to display all business objects with names that start with “Inter”, or include the letters “IBM”, or include the words “International Business Machines”:

```
add filter “IBM”
name Inter*,*IBM*,”International Business Machines”;
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, `“International Business Machines”`). If you include spaces, the value is read as the next part of the filter definition, as described in the Type clause.

Minorrevision Clause

This clause assigns a filter for a particular revision of business objects.

```
minorrevision REVISION_PATTERN
```

- `REVISION_PATTERN` defines the revision for which you are assigning a filter.

The Minorrevision clause can include more than one revision value and wildcards as in the other clauses that use patterns. Typically, you might use a wildcard, but would not include more than one or two revisions. Filtering by the latest revision number or all revisions A and B can remove much of the out-dated business objects from view. This allows you to work with only the most current objects.

The Minorrevision clause can also use wildcard characters. For example, the following definition displays a filter to display all parts of revision 1 or 2:

```
add filter “Revised Parts”
type parts
minorrevision 1,2;
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example. `“status new”,”status old”`). If you include spaces, the value is read as the next part of the filter definition.

Vault Clause

This clause assigns a filter for business objects that are in a particular or similar vault:

```
vault PATTERN
```

- `PATTERN` defines the vault(s) for which you are assigning a filter.

The Vault clause can use wildcard characters. For example, the following definition displays a filter for all business objects that reside in the “Vehicle Project” vault:

```
add filter “Vault Search”
   appliesto businessobject
   vault “Vehicle Project”
   owner *;
```

Copy Filter

You can modify any filter that you own, and copy any filter to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, you can use the `set workspace` command to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with filters that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

After a filter is defined, you can clone the definition with the Copy Filter command. Cloning a filter definition requires Business Administrator privileges, except that you can copy a filter definition to your own context from a group, role or association in which you are defined.

This command lets you duplicate filter definitions with the option to change the value of clause arguments:

```
copy filter SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the filter definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).

`COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any filter of the same name belonging to the user specified in the `touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Filter

Use the Modify Filter command to add or remove defining clauses and change the value of clause arguments:

```
modify filter NAME [user USER_NAME] [ITEM {ITEM}];
```

- `NAME` is the name of the filter you want to modify. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.
- `ITEM` is the type of modification you want to make. With the Modify filter command, you can use these modification clauses to change a filter:

Modify Filter Clause	Specifies that…
`active`	The active option is changed to specify that the object is active.
`notactive`	The active option is changed to specify that the object is not active.
`appliesto businessobject |relationship`	The appliesto option is changed to reflect what the filter now affects.

Modify Filter Clause	Specifies that…
`from|to|both`	The direction of the filter is changed.
`type TYPE_PATTERN`	The type is changed to the named pattern.
`name PATTERN`	The name is changed to the named pattern.
`revision REVISION_PATTERN`	The revision is changed to the named pattern.
`vault PATTERN`	The vault is changed to the pattern specified.
`owner PATTERN`	The owner is changed to the pattern specified.
`where QUERY_EXPR`	The query expression is modified.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`visible USER_NAME{,USER_NAME};`	The object is made visible to the other users listed.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

These clauses are essentially the same ones that are used to define an initial filter except that Add property and Remove property clauses are substituted for the Property clause. When making modifications, you simply substitute new values for the old.

Although the Modify filter command allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the filter name or remove the filter entirely, you must use the Delete filter command and/or create a new filter.

Delete Filter

If a filter is no longer needed, you can delete it using the Delete filter command:

```
delete filter NAME [user USER_NAME] ;
```

- `NAME` is the name of the filter to be deleted. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.
- Searches the local list of existing filters. If the name is found, that filter is deleted. If the name is not found, an error message results.

For example, assume you have a filter named "Overdue Invoices" that you no longer need. To delete this filter from your area, you would enter the following MQL command:

```
delete filter "Overdue Invoices";
```

After this command is processed, the filter is deleted and you receive the MQL prompt for the next command.

When a filter is deleted, there is no effect on the business objects or on queries. Filters are local only to the user's context and are not visible to other users.

form Command

Description

A *form* is a window in which information related to an object is displayed. The Business Administrator designs the form, determining the information to be presented as well as the layout. Forms can be created for specific object types, since the data contained in different types can vary greatly.

For conceptual information on this command, see *Forms* in Chapter 8.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete|print] form NAME [CLAUSEs];
```

- NAME is the name you assign to the form. Form names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the form.

Add Form

Use the Add Form command to define a form:

```
add form NAME [web] [ADD_ITEM {ADD_ITEM}];
```

- web is used when creating a "Web form." This distinguishes forms to be used in HTML/JSP applications from those used in the Studio Modeling Platform and PowerWeb.
- ADD_ITEM provides additional information about the form.

The Add Form clauses are:

[!|not]web
units [picas|points|inches]
description VALUE
rule RULENAME
color [FOREGROUND] [on BACKGROUND]
header HEADER_SIZE
footer FOOTER_SIZE
margins LEFT_MARGIN RIGHT_MARGIN

```
type TYPE_NAME {,TYPE_NAME}
size WIDTH HEIGHT
field FIELD_TYPE FIELD_DEF
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

With the exception of the Units and Size clauses, all other Add Form clauses are optional. (Units will default to picas if you do not enter a value.) Field clauses specify the field values that should be printed in the form and where. Without at least one Field clause, your form will not have much value.

Units Clause

This clause specifies the units of page measurement. There are three possible values: picas, points, or inches.

```
units picas
Or
units points
Or
units inches
```

Without a unit of measurement, the values of any given header, footer, margin, or field size are not interpretted. Because picas are the default unit of measurement, a picas value is automatically assumed if you do not use a Units clause.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character “I” may not occupy the same amount of space as the characters “E” or “O.” To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit across a form page? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

Rule Clause

Rules are administrative objects that define specific privileges for various users. The Rule clause enables you to specify an access rule to be used for the form.

```
add form NAME rule RULENAME;
```

Color Clause

This clause specifies color values used as the default foreground and background for the form.

```
add form color [FOREGROUND |on BACKGROUND |]
```

- `FOREGROUND` is the name of the color for the foreground printed information (any vertical or horizontal lines of information).
- `BACKGROUND` is the name of the color used as an overall background for the form. Note that the word `on` is required only if a background color is specified.

For a list of available colors, refer to:

- Windows— \$MATRIXHOME\lib\winnt\rgb.txt.
- UNIX— /$MATRIXHOME/lib/ARCH/rgb.txt.

  $MATRIXHOME is where Business Process Services is installed and ARCH is the UNIX platform.

Header Clause

This clause places a border at the top of the page. It specifies the number of lines, points, or inches that should be measured down from the top of the page. While inserting a Header clause defines an upper border, it does not prevent you from placing information within that border. Header clauses are often used in conjunction with page titles. You may want to place title information within the header with the values below it.

The following command creates a form named “Material Properties.” Measured in picas, the form is 80 characters wide and 60 lines long.

```
add form "Material Properties"
   units picas
   size 80 60
   header 6
```

Footer Clause

This clause is similar to the Header clause. It places a border at the bottom of the page by specifying the number of lines, points, or inches that should be measured up from the bottom of the page. While inserting a Footer clause defines a lower border, it does not prevent you from placing information within that border.

Footer clauses are often used in conjunction with display rules and page footnotes (such as page numbers or titles). When you define a footer, you are defining where a rule line can be placed. You may want to place your footnote information below that rule line. This information might consist of

summary values, orientation material (such as a page number and title), or special information (such as warning flags).

```
add form "Material Properties"
   units picas
   size 80 60
   footer 6
```

Margins Clause

This clause specifies a left and right border on each page:

```
add form margins LEFT_MARGIN RIGHT_MARGIN
```

`LEFT_MARGIN` is the number of units from the left page edge. Always specify this value first.

`RIGHT_MARGIN` is the number of units from the right page edge.

For example, assume you want to include margins in a “Material Properties” form definition:

```
add form "Material Properties"
   units picas
   size 80 60
   header 3
   footer 3
   margins 5 10;
```

In this example, the left margin is set as five characters in from the left page edge. The right margin is set as ten characters in from the right page edge. Since the page size is set as 80 characters, this means that the center working area is equal to 65 characters.

When including a Margins clause in an Add Form command, you must always specify two values even if you only want one margin. This determines which value is the left margin offset and which is the right. For example, to define only a right margin, use can use this Margins clause:

```
margins 0 5
```

The left margin is defined as the left page edge (0). The right margin is then defined as five characters in from the right page edge. This gives you a new working area of 75 characters in width.

```
add form "Label List"
   units picas
   size 40 12
   header 1
   footer 1
   margins 5 5;
```

With the inclusion of the Margins clause, you have a working area that is 30 characters wide. The left margin is at five characters from the left edge and the right margin is at five characters from the right edge. Since the right edge is at 40 characters, this definition is equivalent to saying that the right edge is at 35 characters.

Type Clause

This clause lists business types that the form is associated with. When a business object is highlighted and the Form option is selected, any forms associated with that type of business object are presented.

Size Clause

This clause defines the page dimensions of the form. This is commonly equal to standard page sizes such as 8½ by 11 inches or 8½ by 14 inches. However, you are not restricted to these sizes.

A page is a logical unit that you define. Once defined, the definition determines where to place the header, footer, and margins. But, you must define the page size to determine when one page ends and another begins.

To define a page size, you need two numeric values. One represents the width and one represents the height. Both of these values must be provided and entered according to the following syntax:

```
add form size WIDTH HEIGHT
```

If you wanted the dimensions of a standard page, you would write one of the following clauses. The clause you use depends on the units you specified in the Units clause.

`size 80 66`	Measured in picas.
`size 612 792`	Measured in points.
`size 8.5 11`	Measured in inches.

Field Clause

The Field clause specifies the values to be printed and their general placement on the page:

```
add form field FIELD_TYPE FIELD_DEF
```

FIELD_TYPE identifies the general function of the field value being defined. All Field clauses must include a Field Type subclause which must be given before any defining subclauses. When specified, the field type identifies the kind of value that will be printed:

Field Type	Specifies That
label STRING_VALUE	A printable string of characters used for headers, column headings, and separators. For example, a form title or footnote would use the Label field type.
select QUERY_WHERE_EXPR	Any selectable business object value. This allows for information to be retrieved from related objects as well as from the object to which the form is attached. Enter an expression that should appear on the form. The expression is constructed according to the syntax described in *Where Clause*. Unlike a QUERY_EXPR expression that must produce a true or false value, the QUERY_WHERE_EXPRESSION can produce any type of value. This means that you can use the name of a field that contains a non-Boolean value in the field type definition. For example, each of the following are valid Expression subclauses that define a field type value: expression DESCRIPTION expression attribute "All tests are negative" expression ‘"attribute[Base Cost]" <= "attribute[Maximum Cost]" expression ‘Blood_Test_Positive or EKG_Positive’ In the first example, the Description field value (a character string value) is used. In the second example, the value of the attribute “All tests are negative” (a Boolean value) is printed. In the third and fourth examples, the values of the relational and Boolean expressions are used for the form output. For more information on writing query expressions, see *Queries* in Chapter 6. For more information on locating and specifying field names, refer to *Manipulating Data* in Chapter 5.
graphic IMAGE_PATH	An imported graphical image, such as a logo or scanned image. Enter the directory path for the graphic file. This is the same image no matter what object is selected.
icon	The icon of the form.

- FIELD_DEF (field definition) is a subclause that provides additional information about the value to be printed. These subclauses define information such as where the values should be placed on the page, how often the field values should be printed, and test criteria to ensure that you have the correct values.

Field Definition	Specifies That
setting NAME VALUE	For use in Web forms only. Settings are general name/value pairs that can be added to a field as necessary. They can be used by JSP code, but not by hrefs on the Link tab. Also refer to “Using Macros and Expressions in Dynamic UI Components” in the *Business Modeler Guide* for more details.
user USER_NAME | all	For use in Web forms only to specify who will be allowed access to the field.
alt ALT_VALUE	For use in Web forms only to display alternate text until any image associated with the command is displayed and also as “mouse over text.”
autoheight [false|true]	When set to true, the height of the field will adjust to the amount of information displayed.
autowidth [false|true]	When set to true, the width of the field will adjust to the amount of information displayed.

Field Definition	Specifies That
`businessobject EXPRESSION`	Computable expression pertaining to business objects.
`color [FOREGROUND] [on BACKGROUND]`	The color of the form foreground (printed information) and background.
`drawborder [false|true]`	Draws a border around the output field.
`edit true | false`	Is field editable?
`font FONT_NAME`	The name of a system font that a field displaying text will use.
`[!]hidden`	Is field hidden or not (!) hidden.
`href HREF_VALUE`	For use in Web forms only to provide link data to the JSP.
`label LABEL`	Field label
`minsize MIN_WIDTH MIN_HEIGHT`	The minimum width and/or height of the field. *The mechanism used for rendering fonts on the Web differs from the one that is used for the Studio Modeling Platform. Therefore, forms that are intended for use in both environments need to be designed to accommodate these slight differences. Increasing the size of the field will fix the problem.*
`multiline true | false`	Is field multiline?
`name NAME`	Field name
`order NUMBER`	For use in Web forms only to re-order field items. When the order number of a field is set to a number less than 0, a warning is issued and the field is placed before all other fields. If you print a form from the database that has fields with negative order numbers, you will also receive a warning.
`range RANGE_HELP_HREF_VALUE`	For use in Web forms only to specify the JSP that gets a range of values and populates the field with the selected value. These values can be displayed in a popup window or a combo box.
`relationship EXPRESSION`	Computable expression pertaining to relationships.
`remove setting NAME VALUE`	For use in Web forms only to remove settings.
`remove user USER_NAME | all`	For use in Web forms only to specify who will not be allowed access to the field.
`resizeheight [false|true]`	Permits the resizing of the height value for variable sized display fields.
`resizewidth [false|true]`	Permits the resizing of the width value for variable sized display fields
`scale PERCENTAGE_VALUE`	The percentage to scale the form.
`select EXPRESSION`	Select clause

Field Definition	Specifies That
`size WIDTH HEIGHT`	The width and height size of the field.
`start XSTART YSTART`	The X and Y coordinates of the field's starting point. This is where the first character of the field value is printed. Refer also to the description of Field Starting Point below.
`update UPDATE_URL_VALUE`	For use in Web forms only to specify the URL page that should be displayed after the field is updated.

Field Starting Point

The field's starting point can be specified in one of two ways. The first is to give the absolute X and Y coordinates. The second is to give the X and Y coordinates relative to the form's header and left margin.

Absolute coordinates begin with 1,1 and are measured from the upper left corner of the page. They use the syntax:

```
@X_START_VALUE @Y_START_VALUE
```

- @ indicates that the coordinates are absolute.
- `X_START_VALUE` specifies the distance across.
- `Y_START_VALUE` specifies the distance down.

For example, you could write the following field description to place a title at the top of the form:

```
field label "Daily Customer Form For:"
   start @10 @5 size 28 1
```

This description will start printing the title string "Daily Customer Form For:" in the upper left corner of the page. Its coordinates, given in picas, indicate ten characters over and five lines down from the uppermost left corner of the page. Take care to ensure that field sizes do not conflict with other field locations.

Relative coordinates can begin with 0,0 and are specified in the same general manner as absolute coordinates:

```
X_START_VALUE Y_START_VALUE
```

- `X_START_VALUE` specifies the distance across.
- `Y_START_VALUE` specifies the distance down.
- However, with relative coordinates, the values are measured from the upper left corner of the header and left margin intersection.

For example, assume you have a form with a header of 6 and a left margin of 11. To place the same title in the same place as in the previous example, you would write the following field definition:

```
field string "Daily Customer Form For:" start 0 0 size 28 1
```

While the starting point is given as (0,0), this actually translates to an absolute starting point of (11,6). That is because the starting point for all relative coordinates is the bottom of the header (the 6th line) and the end of the left margin (11th character). When specifying the relative coordinates, you will always have to add the header and left margin values to obtain the absolute coordinates. Therefore a relative position of (28,0) translates into an absolute position of (39,6).

When specifying the starting point, you can use any combination of relative or absolute values. Absolute coordinates are useful when you want to print a title within the heading, footer, or margin areas. You cannot do this using relative coordinates.

Centering the title on the page is done by using the starting point in conjunction with the Size subclause. The Size subclause specifies the width and height of the field. First determine the number of characters required to print the title (36 characters) and then determine the amount of space remaining (80 - 36 = 44). That amount is divided in half to determine the starting row for the first field (@22). If you add the field size to this starting value, you find the starting location for the second field (@49).

Like the Start clause of the Add Form command, the size is given width first and height second. For example, a value of “Address: ” is nine pica characters long and uses one line. Therefore, its size could be expressed as:

```
size 9 1
```

All geometry subclauses must include the field size. If the size value is larger than the field value, the field value is padded with blank spaces so that the field and size values are equivalent. If the size value is smaller than the field value, the field value is truncated on the right to fit into the field size. Multiple-line text output will wrap at word boundaries if the form field contains more than one output line.

History Clause

The `history` keyword adds a history record marked “custom” to the form that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Form

After a form is defined, you can clone the definition with the Copy Form command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy form FROM_NAME TO_NAME [MOD_ITEM {MOD_ITEM}];
```

- `FROM_NAME` is the name of the form definition (source) to copied.
- `TO_NAME` is the name of the new definition (destination).
- `MOD_ITEMs` are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the form that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Form

After a form is defined, you can change the definition with the Modify Form command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify form NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the form you want to modify.
- MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Form clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Form Clause	Specifies that …
units [picas|points|inches]	The current units of measurement is changed to the new units entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
add rule NAME	The named rule is added.
remove rule NAME	The named rule is removed.
color [FOREGROUND] [on BACKGROUND]	The foreground and/or background colors are changed to the new values entered.
header HEADER_SIZE	The current header, if any, is set to the value entered.
footer FOOTER_SIZE	The current footer, if any, is set to the value entered
margins LEFT_MARGIN RIGHT_MARGIN	The left and/or right margins are changed to the new values entered.
type TYPE_NAME {,TYPE_NAME]	The type(s) associated with the form is changed to the type(s) entered.
type delete TYPE_NAME {,TYPE_NAME}	The type identified by the type name is removed from the form.
size WIDTH HEIGHT	The current page size is set to the new values given
field delete FIELD_NUMBER	The field identified by the given field number is removed from the form. To obtain the field number for a specific field, use the Print Form command. When the form definition is listed, note the number assigned to the field to delete.
field modify FIELD_NUMBER FIELD_DEF FIELD_TYPE FIELD_DEF	A field is modified (according to the field definition clauses) and placed at the end of the field list
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.
history STRING	The history keyword adds a history record marked “custom” to the form that is being modified. The STRING argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the arguments that define the form. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change. For example, use the Size clause of the Modify Form command to alter the values of the

Size clause used in the Add Form command. The only exception to this general rule involves modifying field definitions.

When modifying field definitions within an existing form, you have only two choices. You can either remove an existing field definition or you can add a new one. The Modify Form clause does not offer a way to alter the subclause values that make up a field definition. Therefore, if you are unhappy with a subclause value, you can only remove the entire field definition and replace it with one that has the desired changes in it.

New field definitions appear at the end of the form definition. While they are listed last, their placement in the form definition does not affect the placement of the form values. That is controlled by the geography and size values within the field definitions themselves.

Delete Form

If a form is no longer required, you can delete it by using the Delete Form command:

```
delete form NAME;
```

- `NAME` is the name of the form to be deleted.

Searches the list of defined forms. If the name is found, that form is deleted If the name is not found, you an error message is displayed. For example, to delete the form named “Income Tax Form,” enter the following:

```
delete form "Income Tax Form";
```

After this command is processed, the form is deleted and you receive an MQL prompt for another command.

Print Form

Use the Print Form command to print information about the attributes of a specific form, including the number and characteristics of each form field.

```
print form NAME [SELECT];
```

- `NAME` is the name of the form to be printed.
- `SELECT` specifies a subset of the list contents. For more information see the *Configuration Guide : Appendix: Selectables* in the online documentation.

Searches the list of defined forms. If the name is found, that form information is printed. If the name is not found, an error message is displayed. For example, to print details about the form named “TechTip,” enter the following:

```
print form "TechTip";
```

The following is sample output:

```
MQL<28>print form 'TechTip';

Form           TechTip
 inactive

 field# 1       label Notes:
    font         Times New Roman-bold-12
    autoheight   false
    autowidth    false
```

```
  drawborder   false
  hidden       false
  start        2 54
  size         13 2
  user         all

field# 2       select attribute[Notes]
  color        black on lemon chiffon
  font         Times New Roman-10
  autoheight   false
  autowidth    false
  drawborder   true
  multiline    true
  edit         true
  hidden       false
  start        23 54
  size         75 5
  user         all

field# 3       label Reason:
  font         Times New Roman-bold-12
  autoheight   false
  autowidth    false
  drawborder   false
  hidden       false
  start        2 44
  size         20 2
  user         all

field# 4       select "attribute[Reason]"
  color         on lemon chiffon
  font         Times New Roman-10
  autoheight   false
  autowidth    false
  drawborder   true
  multiline    true
  edit         true
  hidden       false
  start        22 44
  size         75 8
  user         all

field# 5       select description
  color         on lemon chiffon
  font         Terminal-10
  autoheight   false
  autowidth    false
  drawborder   true
  multiline    true
  edit         true
  hidden       false
  start        22 23
  size         75 19
  user         all

nothidden
created Fri Jun 22, 2001 8:16:45 PM EDT
modified Fri Jun 22, 2001 8:16:45 PM EDT  nothidden
created Wed Oct 31, 2001 2:57:09 PM EST
```

```
    modified Wed Feb 20, 2002 2:47:56 PM EST
```

Since forms have additional uses in support of dynamic UI modeling, the MQL print command suppresses the output of data that is not used. For example, if you print a form that is defined as a system object used for Web applications, the following selects will not be printed:

```
size, minsize, scale, font, minwidth, minheight, absolutex, absolutey,
xlocation, ylocation, width, and height.
```

Conversely, when printing non-Web forms, parameters used only for Web forms are suppressed from the output:

```
href, alt, range, update, and settings
```

format Command

Description

Use a *format* definition to capture information about different application file formats. A format stores the name of the application, the product version, and the suffix (extension) used to identify files. It also contains the commands necessary to automatically launch the application and load the relevant files. Formats are the definitions used to link Live Collaboration to the other applications in the users' environment.

For conceptual information on this command, see *Formats* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]format NAME [CLAUSEs];
```

- NAME is the name you assign to the format. Format names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the format.

Add Format

Format definitions are created using the MQL Add Format command. This command has the syntax:

```
add format NAME [ADD_ITEM {ADD_ITEM}]
```

- ADD_ITEM provides more information about the format. They also provide information on how a file with that format should be processed. The Add Format clauses are:

description VALUE
creator NAME
type NAME
edit PROGRAM_OBJECT_NAME
print PROGRAM_OBJECT_NAME
suffix VALUE
mime VALUE
version VALUE
view PROGRAM_OBJECT_NAME

```
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

Each clause and the arguments they use are discussed in the sections that follow.

Creator and Type Clauses

The Creator and Type fields are Macintosh file system attributes (like Protection and Owner on UNIX systems). They should not be confused with users or types. The following is an example Creator clause of the Add Format command:

```
creator 'MPSX'
```

The following is an example Type clause of the Add Format command:

```
type 'TEXT'
```

This would identify a script file created by the Macintosh toolserver. Both fields are four bytes in length and are generally readable ASCII. If you specify a value for only one of the two clauses, the other clause assumes the same value. The values for creator and type are registered with Apple for each Macintosh application. When a file is checked out to a Macintosh, these attribute settings will be applied. If Macintoshes are not used, the fields can be left blank.

View, Edit, and Print Clauses

Theses clauses specify the program to use to view (open for view), edit (open for edit), or print files checked into the format. When you specify the program, you are actually specifying the name of the program object that represents the program.

For Windows platforms, if you want to open files for view, edit, or print based on their file extensions and definitions in the Windows Registry, you can leave out the corresponding clause. For example, by default Windows uses MS Paint to open files with a file extension of .bmp. Keep in mind that each user's PC contains its own Windows Registry database, which is editable; the databases are not shared between computers. If you want to provide a more complex and flexible format that will use the file association mechanism of windows, refer to *Format Definition Example Program*.

Program Object Requirements

To be used in a format definition, a program object definition must include these characteristics:

- The Needsbusinessobject clause must be true.
- The Code clause must contain the command needed to execute the program and the syntax for the command must be appropriate for the operating system.
- The Code clause should end with the $FILENAME macro so the program opens any file. Enclose the macro in quotes to ensure that files with spaces in their names are opened correctly.

For more information on defining program objects for use in a format definition, see *Code Clause*.

Syntax

The View, Edit, and Print clauses of the Add Format command use this syntax:

```
view PROGRAM_OBJECT_NAME
edit PROGRAM_OBJECT_NAME
print PROGRAM_OBJECT_NAME
```

For example, the following is a sample format definition for CADplus, a computer aided design system:

```
add format CADplus
   description "CADplus Computer Aided Design System"
   version 10
   suffix ".cad"
   view CADview.exe
   edit CADedit.exe;
```

After this format is defined, the Live Collaboration can open a file checked in with this format using CADview for viewing or using CADedit for editing.

Suffix Clause

This clause specifies the default suffix for the format. If an object is selected that contains no files, “open for edit” generates the name of the file from the object name. The Live Collaboration attempts to open a file with that name and the default format suffix.

Assume you want to add a note to a business object. You might use the TextTYPE or BestBooks word processing programs to create the note. TextTYPE uses a default file suffix of `.text` for document files and BestBooks uses `.bb`. These suffixes enable users to quickly identify file types. For example:

```
add format "TextTYPE"
   description "For documents created with TextTYPE"
   version 3.1
   suffix ".tex";
add format "BestBooks"
   description "For documents created with BestBooks"
   version 6.0
   suffix ".bb";
```

After these definitions are made, any file that uses a TextTYPE format will have a suffix of `.tex` and any file that uses a BestBooks format will have a suffix of `.bb`.

The suffix specified in the Format is not used in the launching mechanism—the file itself is passed to the operating system and its extension (or suffix) is used to determine what application should be opened.

Mime Clause

You can specify the MIME (Multi-Purpose Internet Mail Extension) type for a format. MIME types are used when files are accessed via a Web browser. To specify a MIME type, use the Mime clause in the format definition:

```
creator VALUE
```

- VALUE is the content type of the file. The format of VALUE is a type and subtype separated by a slash. For example, text/plain or text/jsp.

The major MIME types are application, audio, image, text, and video. There are a variety of formats that use the application type. For example, application/x-pdf refers to Adobe Acrobat Portable Document Format files. For information on specific MIME types (which are more appropriately called “media” types) refer the Internet Assigned Numbers Authority Web site at http://www.isi.edu/in-notes/iana/assignments/media-types/. The IANA is the repository for assigned IP addresses, domain names, protocol numbers, and has also become the registry for a number of Web-related resources including media types.

To find the MIME types defined for a particular format, use the following command:

```
print format FORMAT_NAME select mime;
```

Version Clause

This clause identifies the version number of the software required to process the file. The software version is useful when tracking files created under different software releases. Upward and downward compatibility is not always assured between releases. If you install a new software release that cannot process existing files, you can create a new format for the new release and leave the old format in place. The old format automatically references the older version of the software while the new format references the new version.

The version clause does not check the version number against the software you are using. You can enter any value. However, you should use the actual version number or identifier if possible. For example:

```
add format ASCII version Standard;
```

```
add format “TextTYPE” version 3.1;
```

History Clause

The history keyword adds a history record marked “custom” to the format that is being added. The STRING argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Format

After a format is defined, you can clone the definition with the Copy Format command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy format SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM};
```

- SRC_NAME is the name of the format definition (source) to copied.
- DST_NAME is the name of the new definition (destination).
- MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the format that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Format

After a format is defined, you can change the definition with the Modify Format command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify format NAME [MOD_ITEM] {MOD_ITEM};
```

- `NAME` is the name of the format you want to modify.
- `MOD_ITEM` is the type of modification you want to make.

There are different types of modifications you can make. Each modification is specified in a Modify Format clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Format Clause	Specifies that…
`description VALUE`	The current description, if any, is changed to the value entered.
`creator NAME`	For Macintosh only, the current creator name is changed to the value entered.
`edit PROGRAM_OBJECT_NAME`	The program object that represents the program to use to open the business object file for editing or modification.
icon FILENAME	The image is changed to the new image in the field specified.
`name NEW_NAME`	The current format name is changed to that of the new name entered.
`print PROGRAM_OBJECT_NAME`	The program object that represents the program to use to print the business object file.
`suffix VALUE`	The default file suffix specified is used when creating new files.
`type NAME`	For Macintosh only, the current type name is changed to the value entered.
`mime VALUE`	The MIME type for the format, which is used when a file is accessed via a Web browser.
`version VALUE`	The version number is set for the software processing a file with this format.
`view PROGRAM_OBJECT_NAME`	The program object that represents the program to use to open the business object file for viewing.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.

Modify Format Clause	Specifies that…
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the format that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

As you can see, each modification clause is related to the clauses and arguments that define the format. For example, the following command changes the name and version of the format named “TextTYPE Version 9.1”:

```
modify format "TextTYPE Version 9.1"
   name "TextTYPE Version 10"
   version 10.0;
```

When modifying a format, remember the question of upward and downward compatibility between software versions. Since all files with the defined format are effected by the change, you should test sample files or read the release notes to determine whether or not old files will be negatively effected. If they will be, you may want to create a new format for the new software version rather than modify the existing format definition.

In some cases, the suffix will be different for documents created in a new release of the application software. Therefore, a separate format is required (at lease until all files are updated).

Delete Format

If a format is no longer required, you can delete it with the Delete Format command:

```
delete format NAME;
```

- `NAME` is the name of the format to be deleted.

Searches the list of formats. If the name is not found, an error message is displayed. If the name is found and there are no files with that format in the database, the format is deleted. If there are files that use that format within the database, they must be reassigned or deleted from the business object before you can remove the format from the format list.

For example, delete the TextTYPE Version 9.1 format, enter the following MQL command:

```
delete format "TextTYPE Version 9.1";
```

After this command is processed, the format is deleted and you receive an MQL prompt for another command.

group Command

Description

Three administrative objects allow you to identify a set of users (persons) who require the same accesses: groups, roles, and associations. *Groups* are a collection of people who work on a common project, have a common history, or share a set of functional skills.

For conceptual information on this command, see *Controlling Access* in Chapter 3.

User Level

Business Administrator

Syntax

The syntax and clauses for defining groups and roles are almost identical. For infomation on how to work with groups, see *role Command*.

history Command

Description

The history command provides a *history* for each business object, detailing every activity that has taken place since the object was created.

For conceptual information on this command, see *History* in Chapter 7.

User Level

Business Administrator

Syntax

```
modify [bus OBJECTID| connection ID|connection bus OBJECTID
from|to OBJECTID relationship NAME] [add|delete] history
[ITEM {ITEMS}];
```

You can add a customized history record through MQL to track certain events, either manually or programmatically. Two parts of the entry are definable: the Tag, and the Comment. The Tag appears at the beginning before the hyphen. Then the user/time/date/current stamp is automatically made, followed by the comment defined by the implementer. The MQL syntax for defining a history entry is:

```
modify bus OBJECTID add history VALUE [comment VALUE];
modify connection ID add history VALUE [comment VALUE];
modify connection bus OBJECTID from|to OBJECTID relationship NAME add
history VALUE [comment VALUE];
```

For example, the following command could be added to an MQL/Tcl program which backs up the database:

```
modify bus $OBJECT add history Backup comment Backup was completed to
tape #12345;
```

The history entry for the above would look something like:

```
(Backup) - user: billy time: Mon Mar 30, 1998 11:28:15 AM Eastern
Standard Time state: planned comment: Backup was completed to tape
#12345.
```

All custom history event entries are enclosed in parentheses in order to distinguish them as such.

Custom history entries do not respond to the `history off` *or* `set system history off` *commands.*

Enable and Disable History

History can be disabled for a session, or until re-enabled. This improves performance when creating or importing many objects. This is also useful when bulk loading business objects, or if a different history logging mechanism is implemented.

System Administrators can execute the following MQL command.

```
history off;
```

After this command is executed, events that occur on the local machine do not cause a history record to be logged in the database. This command affects only the local machine; concurrent user's sessions are not affected. In addition, subsequent sessions on the local machine will have history enabled by default.

When history is set to off, modification dates will NOT be updated even if the business object is modified. This is by design. When history records are not needed, modification dates are not updated.

To enable history recording for the session again, a System Administrator should use:

```
history on;
```

MQL also allows history to be enabled or disabled with the use of a toggle command. Depending on the current setting, history can be enabled/disabled using the same command.

To enable/disable history as a toggle, a System Administrator should use:

```
history;
```

When the recording of history is turned off in a program object, it is important that it gets turned back on. While this may be done at the bottom of the program object's code section, history recording is re-enabled automatically when a top-level program ends its execution even if the code is exited before reaching the command that turns it back on (either successfully or in error).

When used with the Studio Customization Toolkit MQLCommand class, an MQL session lasts for only the duration of one command and not throughout the Studio Customization Toolkit program's session. This means that the `history off` command has no effect, since the MQL session ends and so history is turned back on. This is the design intent, to ensure that history is not inadvertently turned off for longer than intended.

If you would like to turn history off throughout the Studio Customization Toolkit program's session, you can add a flag to the Context.start() method to turn history off for the duration of a transaction rather than only for the duration of one command. When you add this flag, all events that occur within the duration of a transaction will not record history. After the transaction commits, history is turned back on for the next transaction and you will need to explicitly turn history off again with or without the flag.

Since logging history affects performance as well as the size of the database, in some implementations it may be desirable to turn history off permanently. Refer to *Controlling System-wide Settings* in Chapter 9 for more information.

The system history setting should not be issued within program objects, since it affects all users. In these cases, the temporary MQL history off command should be used instead.

Selecte History Entries

History records of either business objects or connections can be selected with the `Select` clause of the following MQL commands:

`print businessobject`	`expand businessobject`
`print set`	`expand set`
`print connection`	

In addition, when using the above commands to print all information about a business object, set or connection, history records can be excluded.

!History Clause

When all information about a business object or connection is printed with the `print` commands listed above, everything about the object(s) or connection, including its history, is listed.

The `!history` clause allows you to exclude the history of a business object or connection, which can be quite lengthy, when printing all the other information. For example, the following command:

```
print bus Assembly RB45621 A !history;
```

prints out all information about the business object *except* its history.

Select History

History can be selected from either business objects or connections, in a manner similar to selecting the owner or current state of an object. For more information, see the *Configuration Guide : Appendix: Selectables* in the online documentation.

To select history from a business object use the following syntax:

```
print bus OBJECT select history.ITEM [history.ITEM];
```

- OBJECT is the Type, Name, and Revision of the business object or its object ID.
- ITEM can be one of the following:

between
`EVENT`
`time`
`user`
`state`

To select history from a connection, use one of the following:

```
print connection ID select history.ITEM;
Or
expand bus OBJECT select relationship history.ITEM;
```

Where ITEM can be:

EVENT
time
user

- ID is the identification of the connection as returned with the select connection ID print command.
- The second command listed will actually return the history of all connections on the specified OBJECT.

Each ITEM clause is described in the sections that follow.

History.Between Clause

The History.between clause can be used on a business object. It returns a subset of that business object's history log when dates are included, or the entire history log when dates are not included.

```
history.between[FROMDATE|TODATE]
```

- The dates included must adhere to the date format of the system.
- If both FROMDATE and TODATE are provided, the list includes events that occurred between the 2 dates, inclusive of FROMDATE but exclusive of TODATE.
- If only FROMDATE is included, the list includes events that occurred between the date specified and the present date, inclusive of both FROMDATE and the present date.
- If only TODATE is specified, the list includes events that occurred before the date specified, exclusive of TODATE.

History.Event Clause

The History.Event clause can also be used on a business object or connection. It returns a list of history records of the EVENT event type. For business objects, EVENT can be one of the following:

approve	custom	moveto
changename	delegate	override
changeowner	demote	promote
changepolicy	disable	purge
changetype	enable	reject
changevault	ignore	removedoid
checkin	lock	removefile

checkout	modify	revise
connect	modifyattribute	schedule
create	movedoid	undelegate
	movefrom	unlock

For a connection, EVENT can be:

changetype	freezethaw	modifyattribute
create	modify	purge
custom	...	...

Example 1

For example, if a user enters:

```
print bus Assembly PR6792 A select history.modify;
```

the following is output:

```
modify - user: patrick time: Mon Aug 6, 1998 5:50:11 PM state: Assigned
description:  test
modify - user: sam time: Mon Aug 6, 1998 6:50:11 PM state: Assigned
description:  test1
modify - user: diane time: Mon Aug 6, 1998 7:50:11 PM state: Assigned
description:  test2
modify - user: ted time: Mon Aug 6, 1998 8:50:11 PM state: Assigned
description:  test3
```

Example 2

You may want to find the date when a signature has been satisfied. Use a command similar to the following:

```
print bus ECR 000122 ""  select history.approve;
```

This command returns:

```
business  object  ECR  000122            history.approve = approve -
user: Cole  time: Sun Sep 5, 1999  1:24:50 PM CDT  state: Approvals
signature: Scrap Approver 1   comment:
```

If you have multiple signatures on specific states, you will have to parse the data for a particular date and signature.

Example 3

To return a list of customized history entries created with the `add history VALUE [comment VALUE]` clause of the modify bus command, use the `custom` event. For example:

```
print bus 2340988 select history.custom;
```

might return:

```
(Backup) - user: billy time: Mon Mar 30, 1998 11:28:15 AM Eastern
Standard Time state: planned comment: Backup was completed to tape
#12345.
```

Note that `freezethaw` is one event that can be selected and will return both freeze and thaw entries.

`History.Time` Clause

The `History.Time` clause can be used on a business object or connection to return a list of the timestamps of every history record. For example:

```
print connection 1234568 select history.time;
```

returns a list of the different times that the connection was updated in the database:

```
time: Fri, Jan 2, 1998  1;48:41 PM
time: Thurs, Feb 6, 1998 2:20:13 PM
time:  Wed, April 8 1998 10:15:12 AM
```

`History.User` Clause

The `History.User` clause can be used on a business object or connection. It essentially gives a list of all users who have operated on the object or connection. For example, the following:

```
print bus Manual NewBook 1 select history.user;
```

may output:

```
user: sue
user: tim
user: jerry
```

`History.State` Clause

The `History.State` clause can be used only on a business object or set. It returns a list of all states the object was in when operations were performed on it. For example:

```
print bus 2340988 select history.state;
```

gives a list of all states the business object was in after every change to the business object:

```
state: Proposed
state: Assigned
state: Described
```

Delete History

System administrators only can purge the history records of a business object or connection via MQL. History records can be deleted based on:

- the type of event (for example, checkout, checkin);
- the user who performed the event (for example, angie);
- the date the operation took place (for example, on, before, or after a specified date).

In addition, *all* history records of a business object or connection can be deleted with one command. Users can optionally write the purged entries to a file. The purge history event itself is recorded in history.

While the various forms of the command provide flexibility and control over exactly which history records are purged, very complicated variations are not supported. This is to ensure that accidental

deletions of important historical events do not occur. In other words, as the criteria for deletion becomes more complex, more delete history commands will be required.

In the History of an object, other objects are sometimes referred to. This is an issue if “Show” access has been denied for a particular user or object in the database. The performance impact of determining whether the current user has access to see the Type, Name and Revision of such objects would be significant and unavoidable. Individual history records can be deleted using the `delete history` clause of the `modify businessobject` or `modify connection` command. This can be used in action triggers to remove such records.

Delete History Clause

This clause can be used alone, or with other refining ITEM clauses. If there are no ITEMS specified, then *all* history records associated with the business object or connection are deleted. The syntax is:

```
modify businessobject OBJECT delete history [ITEM {ITEMS}];
```

OR

```
modify connection ID delete history [ITEM {ITEMS}];
```

Where:

`OBJECT` is the Type, Name, and Revision of the business object or its object ID.

`ID` is the identification of the connection as returned with the `select connection ID` print command.

`ITEM` can be from the list below:

```
event EVENT
by USER
| on     |
| before | DATE
| after  |
| keep | last |
output FILENAME
```

Notice that the keywords “keep” and “last” are mutually exclusive. Also, only one form of the DATE item is allowed in a single command.

For example:

```
modify bus Document PO567932 1 delete history;
```

deletes all history records and adds a history entry similar to the following:

```
History Records Purged by ted on 11/9/98 12:02:03 PM.!!!
```

Each ITEM that can be used with the `history delete` clause is described in the sections that follow.

Event Clause

All history entries that log a specific event can be deleted from a business object or connection using the `event EVENT` item. EVENT is a database event that is logged in history. For business objects, EVENT can be from the following list:

approve	custom	moveto
changename	delegate	override
changeowner	demote	promote
changepolicy	disable	purge
changetype	enable	reject
changevault	ignore	removedoid
checkin	lock	removefile
checkout	modify	minorrevise
connect	modifyattribute	schedule
create	movedoid	undelegate
	movefrom	unlock

EVENT for connections can be from the following list:

changetype	custom	modify
create	freezethaw	purge

Notice that the `purge` history event record itself can be deleted; however, doing so will be generate a new purge history record.

A list of events can be specified, separated by a space and a comma. For example:

```
modify bus Document PO567932 1 delete history event changetype, changename;
```

deletes all entries of these types. And:

```
modify connection 35894008 delete history event freezethaw;
```

removes all freeze and thaw history entries on the connection. Note that freezethaw is one selectable EVENT.

To purge customized history entries created with the `add history VALUE [comment VALUE]` clause of the `modify bus` command, use the `custom` event. For example:

```
modify bus 2340988 delete history custom;
```

By User Clause

All history entries that log events performed by a particular user can be deleted from a business object or connection using the `by USER` item.

`USER` is the person listed in the history entry as the user who performed the operation, and so is, or was, a valid person.

A list of persons can be specified, separated with a space and a comma. Also, the `by` USER clause can be used with or without the keyword `by` and in conjunction with other ITEMS. Here are a few examples:

```
modify bus Document PO567932 1 delete history by chris, tom;
```

This example deletes the records of all events performed by either chris or tom from the object's history.

```
modify bus Document PO567932 1 delete history event changetype by chris, tom;
```

This example deletes all `changetype` history events performed by either christie or tom.

Date Clause

History entries can be deleted from a business object or connection based on when the event occurred using one of the following `DATE` items.

`on DATE`
`before DATE`
`after DATE`

- DATE is the timestamp in the history entry, and may include the time of day, or just the date of the event. If the time of day is not included, then the input is considered a date. Otherwise the timestamp is taken into account.

For example:

```
modify bus Document PO567932 1 delete history date 11/04/
98;
```

This example deletes the record of all events that occurred on November 4, 1998. On the other hand:

```
modify bus Document PO567932 1 delete history before "11/04/98 12:00:00 PM EST";
```

This example deletes all entries for events that occurred before November 4, 1998 at 12:00:00 PM EST. The `after` keyword can be used in the same manner.

Quotes must be used when including the time of day in the DATE.

Dates and times can be entered in the command as specified in the initialization file. For more information about date and time formats, see the *Administration Guide* : *Configuring Date and Time Formats*.

Date items can be used with or without the other ITEMS listed, but only one date item is allowed in a single command. For example:

```
mod bus Document PO567932 1 delete history event checkin by sue after 3/17/01;
```

Only one date ITEM is allowed in a single command.

Keep Item

The `keep` item clause is used in conjunction with the other ITEMS and provides a way of reversing what is specified. The ITEMS that follow `keep`, are NOT deleted, but all other records are. The one exception is that any purge history records are always kept.

In addition to what is specified with the `keep` *clause of the* `delete history` *command, purge history entries are always kept, unless ALL history is deleted.*

`Keep` can be used with any of the other `ITEMS` except `last`, but it must be specified immediately following the `delete history` clause. For example:

```
modify bus Document PO567932 1 delete history keep event
create;
```

This example deletes all records except the creation (originated) entry.

```
modify bus Document PO34675 1 delete history keep event checkin by bill on
November 5, 1998;
```

This example deletes all records except those for checkin events performed by bill on the date specified.

When used, `keep` *must be specified immediately following the* `delete history` *clause, and can NOT be used in conjunction with* `last`.

Last Item

The `last` item clause is used in conjunction with the other ITEMS (except for `keep`) to purge the most recent history records that meet the criteria. When used, `last` must be specified immediately following the `delete history` clause. For example:

```
modify bus Document PO34675 1 delete history last event checkin by bill;
```

This example deletes only one entry - for the last checkin event performed by bill.

When used, `last` *must be specified immediately following the* `delete history` *clause, and can NOT be used in conjunction with* `keep`.

output FILENAME item

The purged history entries can be written to a text file using the `output FILENAME` item. In this manner, history can be archived.

FILENAME is the name of a file to create with the purged history records. Optionally, a directory path can be included.

For example:

```
modify bus Document PO34675 delete history event checkin output history.txt;
```

deletes all “checkin” records and stores them in the file “history.txt.” The file will be saved in ENOVIA_INSTALL unless a path is specified. For example, on a PC, the following could be used:

```
modify bus Document PO34675 delete history event checkin output
“d:\archive\history.txt”;
```

On UNIX it might be:

```
modify bus Document PO34675 delete history event checkin output “/home/archive/
history.txt”;
```

Usage Notes

- Every time a purge is performed for either business objects or connections, a history record is added which says:

  `history = purge - user: USER time: TIMESTAMP 'History Records Purged'`

  These purge entries are deleted only when all history is deleted, or when explicitly deleting them, with something like the following:

  ```
  modify bus Document PO34675 1 delete history event purge;
  ```

  However, if you use the `keep` clause to save all “modify” records, for example, all “modify” and “purge” records will actually be kept.
- If NO ITEMS are specified, then ALL history records associated with a business object or connection are purged, including any purge history entries.
- Once history records are purged, they are completely deleted from the database. So a system administrator should take extra care before purging records. If the output clause is used, the text file could be checked into an object, but entries cannot be imported back into the history log.
- History deletion operations cannot be strung together into one single command. For example, THE FOLLOWING IS NOT SUPPORTED:

  `modify bus TYPE NAME REVISION delete history event lock keep event lock by Jo;`

  The delete history clause has been kept as simple as possible, to ensure that unwanted deletions do not occur. To actually do something like the above, (delete only lock records but keep lock records performed by one user), a different selection criteria must be formulated and accomplished using more than one command.

import Command

Description

Administrative definitions, business object metadata and workflows, as well as checked-in files, can be exported from one root database and *imported* into another. Exporting and importing can be used across schemas of the same version level, allowing definitions and structures to be created and "fine tuned" on a test system before integrating them into a production database.

For conceptual information on this command, see *Working with Import and Export* in Chapter 10.

User Level

System Administrator

Syntax

Use the Import command to import administrative objects from an export file to a database. The export file must be in Exchange Format or in an XML format that follows the Matrix.dtd specification.

```
import [list] [property] ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]...];
```

`ADMIN_TYPE` is the administrative definition type to be imported. It can be any of the following:

`admin`	`group`	`person`	`server`	`vault`
`association`	`index`	`policy`	`site`	`wizard`
`attribute`	`inquiry`	`program`	`store`	
`command`	`location`	`relationship`	`table`	
`form`	`menu`	`role`	`type`	
`format`	`page`	`rule`	`user`	

The `ADMIN_TYPE` may include a `TYPE_PATTERN` which filters the definitions to be imported.

`OPTION_ITEM` is an import clause that further defines the requirements of the import to be performed. It can be any of the following:

`!icon`	!overwrite	continue
commit N	skip N	pause N

`OPTION_ITEM`s can be used in any order before the `from file` clause. The sections below describe these options.

`FILENAME` is the path and name of the existing `.mix` file from which to import the information.

FILE_TYPE can be used to specify files to be used to control the import and log files and exceptions. FILE_TYPE can be any of the following:

```
map             exclude         log             exception
```

Note that the use *keyword must be used with any files that are specified: map files, exclude files, log files, or exception files. If more than one file type is to be used, the* use *keyword should only be stated once. (The* use *keyword in not used at all in the export command).*

FILE is the name of the file to be used with FILE_TYPE. For example, an exclude FILE would be the name of the existing file that lists any objects which should not be included in the import.

If triggers are attached to an object being imported, they may be executed at the time of import. Therefore, the MQL command:

```
trigger off;
```

must be run before importing objects into a database.

List Clause

The List clause of the Import command is used to show on the screen what would be done on import. It enables you to perform a practice run, ensuring that the files to be imported are read correctly. For example:

```
import list admin * from file c:\admin.mix;
```

This might output:

```
vault Standards
vault ...
store Drawings
store ...
attribute Sheet Count
attribute ...
program Cadra
program ...
type Drawing
type ...
relationship Report
relationship ...
format Cadra
format ...
role Manager
role ...
group Sales
group ...
person Dave
person ...
policy Reports
policy ...
```

Overwrite Clause

The `overwrite` clause is used if objects were incrementally exported. If an object already exists, the overwrite clause tells the import command to modify any objects that are found to already exist. Without the overwrite clause, if an administrative object exists, the import will fail, since it will try to create a new object with the same name as an existing object.

When `!overwrite` is used, the objects that don't import because they already exist are written to the exception file (if specified).

When importing, the default for administrative objects is `!overwrite`. *For business objects, the default is* `overwrite`.

Overwrite will not work on administrative objects that are referenced by business objects. This eliminates the possibility editing an administrative object in a development environment, testing, and then importing into a production environment.

Skip N Clause

`Skip N` provides a way not to import the beginning `N` number of objects in an export file. It is helpful if a previous attempt to import a file was unsuccessful and aborted part way through the process.

Commit N Clause

The `commit` clause gives you the abililty to specify `N` number of objects to enclose in transaction boundaries. In this way, some progress could be made on the import even if some transactions are aborted. The default is 100.

Pause N Clause

`Pause` allocates the amount of time in seconds to wait between import transactions. It is used with the `commit` clause. A pause is beneficial when import is running in the background for an extended period of time. It provides a larger window of opportunity for you to access the database machine's resources. Import transactions run continuously when this clause is not included.

Continue Clause

The `continue` clause proceeds with additional imports even if an error is generated. When `continue` is used, it is helpful to use the `log` and/or `exception` clauses as well, so that diagnostics can be performed, and the data that caused the error is trapped.

Map Clause

The Map clause of the Import command indicates a map file which lists any new locations or names for objects. The map file must use the following format:

```
ADMIN_TYPE OLDNAME NEWNAME
ADMIN_TYPE OLDNAME NEWNAME
```

- ADMIN_TYPE can be any of the administrative object types: vault, store, location, server, attribute, etc.
- OLDNAME is the name of the instance that will be found in the import file.
- NEWNAME is the name that should be substituted for OLDNAME when imported into the new database.

As indicated, each definition to be changed must be delimited by a carriage return.

On Windows, the map file requires a carriage return after the last line in order for the last line to be read. On UNIX, there is no such requirement.

Exclude Clause

Use the Exclude clause of the Import command to indicate a file that lists any objects to be excluded. For example:

```
import admin TEST* from file teststuf.mix use exclude
nogood.obj;
```

The exclude file for administrative objects must use the following format:

```
ADMIN_TYPE NAME
ADMIN_TYPE NAME
```

- ADMIN_TYPE can be any of the administrative object types: vault, store, location, server, attribute, program, type, relationship, format, role, group, person, workspace, policy, form, association, or rule.
- NAME is the name of the definition instance that should be excluded in the import. Wildcard patterns are allowed.
- Each definition to be excluded must be delimited by a carriage return.

Log FILE Clause

Apply the Log File clause to specify a file to contain error messages and details for the import process. The output is similar to using the verbose flag, but includes more details.

Exception FILE Clause

Use the Exception File clause to provide a file location for objects to be written to if they fail to import. If a transaction aborts, all objects from the beginning of that transaction up to and including the “bad” object will be written to the exception file.

Import Bus Command

Use the Import Businessobject command to import business objects from an export file to a database. The export file must be in Exchange Format or in an XML format that follows the DTD specification.

```
import [list] bus[inessobject] BUSID [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]...];
```

- BUSID is the Type, Name, and Revision of the business object. Wildcard patterns are allowed. (You cannot use OIDs with import bus).
- OPTION_ITEM further defines the requirements of the import to be performed.

It can be any of the following:

[!]attribute	continue	[!] overwrite	pause N
[!]basic	[!]file	[!]relationship	skip N
[!]captured	[!]history	!fromrelationship	[!]preserve
commit N	[!]icon	!torelationship	[!]state
from vault VAULT_NAME to vault VAULT_NAME			

- OPTION_ITEMs can be used in any order before the from FILE clause. The sections below describe these options.
- FILENAME is the name of the file from which to get the exported ASCII data.
- FILE_TYPE can be used to specify files to be used to control the import and log files and exceptions. FILE_TYPE can be any of the following:

map	exclude	log	exception

Note that the use *keyword must be used with any files that are specified: map files, exclude files, log files, or exception files. If more than one file type is to be used, the* use *keyword should only be stated once. (The* use *keyword in not used at all in the* export *command).*

- FILE is the name of the file to be used with FILE_TYPE. For example, an exclude FILE is the name of the existing file that lists any objects which are not included in the import.

For a detailed description of items in the import business object command, refer back to *import Command*.

From Vault and To Vault Clauses

One way to redirect the import of business objects into a new location (vault) is to use the from vault clause with the To vault clause. For example, to place all business objects that were in vault Test into vault Prod in the new database, use:

```
import businessobject * * * from vault Test to vault Prod
from file c:\revb.mix;
```

Another alternative for redirecting business objects during import is to use a map file as discussed in the section *Map Clause*. In fact, if you want to import business objects that have revisions and put them into a different vault, you *must* use a map file or errors will occur.

Excluding Information

When importing business objects, the default is to include everything about the object. However, you may specify that some parts of the .mix file should not be imported.

Any information that was omitted during the export cannot be included during import, regardless of the use of this clause.

The table below shows the options which can be used when importing business objects to exclude information:

Clause	Used to:
`!attribute`	Exclude attribute values. Generally used with the overwrite option, so that even though other parts of the object will be overwritten, attribute values will not be.
`!basic`	Exclude basic information. Generally used with the `overwrite` option, so that even though other parts of the object will be overwritten, basic information will not be.
`!file`	Exclude file metadata and content of captured store files. See *Excluding Files* for more information.
`!captured`	Exclude file content but include metadata of captured store files. See *Excluding Files* for more information.
`!history`	Exclude history entries
`!icon`	Exclude icons.
`!relationship`	Exclude all relationships.
`!torelationship`	Exclude “to” relationships.
`!fromrelationship`	Exclude “from” relationships.
`!state`	Exclude state information. Generally used with the `overwrite` option, so that even though other parts of the object will be overwritten, current state and signature information will not be. This can also be used to put objects back to the first state in their lifecycle.

For example, to import all objects from file mystuff.mix and exclude all checked in files, use:

```
import businessobject * * * !file from file mystuff.mix;
```

For captured files, file metadata can be included, without the actual file content by using the `!captured` option. See the discussion *Excluding Files* for more information.

To import a single object without including history, use:

```
import businessobject Assembly “ABC 123” A !history from
file obj.mix;
```

To import a single object without any of its relationships, use:

```
import businessobject Assembly "ABC 123"A !relationship
from file obj.mix;
```

When importing objects, several options are available with regard to relationship information:

- Include all relationship information (default)
- Exclude all relationship information (`!relationship`)

- Exclude *to* relationships (`!torelationship`)
- Exclude *from* relationships (`!fromrelationship`).

To import all objects and reset the current state to the beginning of the lifecycle, use:

```
import businessobject * * * !state from file obj.mix;
```

Preserve Clause

The Preserve clause is used when importing business objects. It specifies not to change the object's modification date to the date of the import.

Extracting from Export Files

Sometimes export files contain more information than you want imported. When this is the case, the extract command can be used to create a new file containing only the specified information of the original file.

```
extract |bus OBJECTID     | [OPTION_ITEM [OPTION_ITEM]]from file FILENAME |into|
file NEW;
       |ADMIN ADMIN_NAME|                                                  |onto|
```

- `OBJECTID` is the `OID` or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search. If a pattern is listed, the first match is extracted.
- `ADMIN` is any of the administrative types to be extracted.
- `ADMIN_NAME` is the name of the administrative object to be extracted.
- `OPTION_ITEM` can be any of the following:

`remaining`	`skip N`	`exclude FILE`

- `FILENAME` is an existing export file from which to extract information.
- `NEW` is the file that extract creates or appends with the requested information.

Remaining Clause

The `Remaining` clause extractss from the specified object to the end of the file.

Examples

Import Business Object

The following command:

- imports business objects from export.mix
- commits the import transaction every 5 business objects
- writes the audit trail to import.log

- writes exception objects to error.mix

```
import bus * * * commit 5 continue from file export.mix use
log import.log exception error.mix;
```

If an error occurs, you can look at the entries in the import.log to see what went wrong. Once the problem is resolved, the error.mix file could be used to import the objects that were not imported successfully the first time.

When object imports fail, the transaction aborts and rolls back the import of any objects that precede it in the transaction boundary. These are the objects written to the exception file. A new transaction is started with the next object. Using the example above, suppose the third object attempted caused a problem; the transaction is aborted. The first three objects are written to error.mix and a new transaction begins with the fourth object. After the eighth object was imported, the transaction is committed and the new database has five new objects.

Extract from Export File

For example, if all administrative objects were extracted into one file called admin.mix, you can extract all policies into a separate file as follows:

```
extract policy * from file admin.mix into file policy.mix;
```

To extract all business object from the fifth entry until the end of the file use:

```
extract bus * * * skip 4 remaining from file objects.mix into file newobjs.mix;
```

index Command

Description

An *index* is a collection of attributes and/or basics, that when enabled, causes a new table to be created for each vault in which the items in the index are represented by the columns.

For conceptual information on this command, see *Working with Indices* in Chapter 9.

User Level

System Administrator

Syntax

```
[add|modify|enable|disable|validate|delete]role NAME {CLAUSE};
```

- NAME is the name you assign to the role or group. This name must be unique and cannot be shared with any other type of user (groups, roles, persons, associations). Assign a name that has meaning to both you and users. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the index.

Add Index

An index is created with the add index command:

```
add index NAME [unique] [ADD_ITEM {ADD_ITEM}];
```

unique is a keyword that indicates that a unique constraint is to be placed on the index table in the database schema. This allows a very efficient mechanism for implementations to require unique combinations of values on business objects.

Note that the unique setting only applies to objects within a single vault.

ADD_ITEM provides additional information about the index:

description VALUE
attribute NAME{, NAME}
field FIELD_VALUE{ FIELD_VALUE} [size SIZE]
[!|not]unique
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING

Attribute Clause

This clause assigns attributes to the index. These attributes must be previously defined with the add attribute command, and may not include multiline attributes. Use the Attribute clause if only attributes will be included in the index. If you will include basic properties, use the *field FIELD_VALUE* syntax, described below.

Multi-line attributes may not be added to an index.

A maximum of 25 items may be placed in a single index. However, a typical index will have between 2 and 5 items. For example:

```
add index CostAndWeight
    description "Actual Costs and Weight"
    attribute "Actual Cost","Actual Weight";
```

Notice that in this syntax, attribute names are listed separated by a comma but no space, with the keyword "attribute" used only once. Quotes are needed for names that include spaces.

Generally, all attributes placed in an index should exist in the same type definitions. For example, if an index is created with the attributes Cost, Weight, and Quantity, then all relationships or types that reference Cost, Weight, or Quantity should reference all three of them. If there is a type or relationship that references only some of the attributes in an index, a warning is generated. You can proceed with the index creation, but it will take longer to create an index where this completeness test fails.

For expands, Live Collaboration looks for and uses an index that is associated with relationships; that is, an index that has a relationship attribute as the first item in it.

There are special rules that apply when including a long string attribute in an index. Since a string attribute can be any length, and the index tables are constructed using fixed length columns, string attributes are truncated to 251 characters when written to an index table. This means that only the first 251 characters are searchable on the index. As mentioned below, the same applies to descriptions, but these fields are truncated (and therefore searchable) on the first 2040 characters.

Only the first 251 characters of string attributes and 2040 characters of descriptions are indexed.

This should not really be a concern, since string attributes designed to hold this much data are generally defined as "multi-line" and multi-line attributes may not be included in an index.

field FIELD_VALUE

Field FIELD_VALUE is used when you want to include basic properties in an index. FIELD_VALUE may be any of the following:

```
attribute[NAME]

type

name

revision
```

```
description

policy

current

owner

locker

modified

originated
```

For example:

```
add index "Shipping Details" unique field attribute[Cost]
attribute[Weight] attribute[Quantity] current owner;
```

Notice that in this syntax, attributes names are in square brackets and include the keyword attribute with each one. Also, there is a space but no comma between fields.

You can assign any combination of basic properties (type, name, revision, description, owner, locker, policy, creation date, modification date, and current state) and defined attributes to the index, except for multiline attributes. A typical index will have between 2 and 5 items. While it is possible to have more, most indices should have no more than 5 items.

The first item in the index determines which objects go into the index table. Also, only queries and expands that include the first item will ever use it. If the first item in an index is an attribute, then all business objects and relationships that have that attribute will be added to the index table. If the first item in an index is a basic property, then *all* business objects (since all business objects have basics!) and no relationships are added. Care must be taken when adding an index that has a basic property as its first field. If there are many such indices, performance will suffer, and storage requirements may exceed expectations.

Indices defined with a basic property first include all business objects and require maximum storage facilities. Therefore, adding basic properties first in an index should be limited to those that include only basic properties.

Generally, all attributes placed in an index should exist in the same type definitions. For example, if an index is created with the attributes Cost, Weight, and Quantity, then all relationships or types that reference Cost, Weight, or Quantity should reference all three of them. If there is a type or relationship that references only some of the attributes in an index, a warning is generated. You can proceed with the index creation, but it may take longer to create an index where this attribute completeness test fails.

You cannot include Description in an index in DB2 or SQL Server environments.

The use of Description in an index should be avoided; the advanced search option indexes descriptions much more efficiently.

Size Clause

You can define a unique size restriction up to 2KB for each string based `field FIELD_VALUE` in an index. The size must be explicitly attached to the field it is restricting. For example:

```
add index CostAndWeight field description size 10;
```

Unique Clause

An index may be defined as “unique.” When this is specified, a unique constraint is placed on the index table in the database schema. This allows a very efficient mechanism for implementations that want to require unique combinations of values on business objects.

Note that the unique setting only applies to objects within a single vault.

History Clause

The `history` keyword adds a history record marked “custom” to the index that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Modify Index

After an index is defined, you can change the definition with the modify index command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify index NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the index you want to modify.
- `MOD_ITEM` is the modification you want to make.
- You can make the following modifications. Each is specified in a Modify Index clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Index Clause	Specifies that…
`name NEW_NAME`	The current index name changes to that of the new name entered.
`description VALUE`	The current description, if any, changes to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
`add attribute NAME`	The named attribute is added to the index’s list of items.
`remove attribute NAME`	The named attribute is removed from the index’s list of items.
`add field NAME`	The named field is added to the index’s list of items. This syntax is only valid if you are adding a NEW field. If the index already has this field defined, an error is retuned and in order to modify this field, you must use this syntax: `modify index NAME modify field NAME;`
`remove field NAME`	The named field is removed from the index’s list of items.
`hidden`	The hidden option is changed to specify that the object is hidden.

Modify Index Clause	Specifies that…
`[!|not]hidden`	The hidden option is changed to specify that the object is not hidden.
`unique`	A unique constraint is added to the index's table.
`[!|not]unique`	The unique constraint is removed from the index's table.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked "custom" to the index that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Enable Index

To enable an index, use the following command:

```
enable index NAME;
```

As soon as you enable the index, a table is created in every vault. (The tables are named in the form ixXXXXXX_XXXXX, created by concatenating a prefix associated with the index and a suffix associated with the vault. The prefix can be obtained from print index command and the suffix from the print vault command.) If you then disable the index, or add or remove items, these tables are dropped. Also, if you add an attribute to a type or relationship that is part of an index, the index is disabled. To use the index, it must be re-enabled.

For example, to enable the index named "Index for Finance queries," enter the following MQL command:

```
enable index "Index for Finance queries";
```

Creating and enabling indices are not appropriate actions to be performed within explicit transaction boundaries, particularly if additional operations are also attempted before a commit.

Disable Index

You should disable indices when performing bulk loading or bulk updating of data.

To disable a defined index, use:

```
disable index NAME;
```

Validate Index

Since up-to-date database statistics are vital for optimal query performance, after enabling an index you should generate and add statistics to the new database tables. Use the following command to do so:

```
validate level 4 index NAME [output FILENAME]
```

- The output FILENAME clause is used for DB2 only.
- This is assuming statistics are already up-to-date for all other tables.

Delete Index

If an index is no longer required, you can delete it with the Delete index command:

```
delete index NAME:
```

- `NAME` is the name of the index to be deleted.
- Searches the list of indices. If the name is not found, an error message is displayed. If the name is found, the index is deleted.

For example, to delete the index named “Index for Finance queries,” enter the following MQL command:

```
delete index “Index for Finance queries”;
```

Example

When you create an index, the mxindex table is updated. The ix tables are not created until the index is enabled. For example:

```
add index CostAndWeight
   description “Actual Costs and Weight”
   field attribute[Actual
   Cost],attribute[ActualWeight],owner;
```

inquiry Command

Description

Inquiries can be evaluated to produce a list of objects to be loaded into a table in a JSP application. In general, the idea is to produce a list of business object ids, since they are the fastest way of identifying objects for loading into browsers. Inquiries include code, which is generally defined as an MQL `temp query` or `expand bus` command, as well as information on how to parse the returned results into a list of OIDs.

For conceptual information on this command, see *Inquiries* in Chapter 6.

User level

Business Administrators can create new inquiry objects if they have the Inquiry administrative access.

:

```
[add|copy|modify|evaluate|delete]inquiry NAME {CLAUSE};
```

- `NAME` is the name of the inquiry you are defining. nquiry names cannot include asterisks. You must specify a unique name for each inquiry that you create. The name you choose is the name that will be referenced to evaluate this inquiry within a JSP.
- CLAUSEs provide additional information about the inquiry.

Add Inquiry

To define an inquiry from within MQL use the Add Inquiry command:

```
add inquiry NAME [ADD_ITEM {ADD_ITEM}];
```

- `ADD_ITEM` is an Add Inquiry clause that provides additional information about the inquiry.

The Add Inquiry clauses are:

`description STRING_VALUE`
`pattern VALUE`
`format VALUE`
`code VALUE`
`file FILENAME`
`augument NAME [STRING]`
`property NAME on ADMIN [to ADMIN] [value STRING]`
`history STRING`

Pattern Clause

JPOs are another option in addition to inquiries that can be used for building table IDs. In some cases a JPO should be used in place of an inquiry because of inquiry limitations. Multi-value selectables or relationship based selectables should not be used with inquiries since they may not produce a consistent pattern. JPOs should be used in this case because they do not parse and don't require you to match an explicit pattern. Only attributes and basics should be used with inquiries.

This clause indicates the expected pattern of the results of the evaluated code, and shows how the output should be parsed. It sets the desired field to an RPE variable or macro. Since inquiries are designed to produce a list of business objects, generally the macro that is set is OID.

When you execute a temp query in MQL, the business objects found are returned in a list that includes the type, name and revision, as well as any selectable information specified. For example, the following code:

```
MQL< >temp query bus Part * * select id dump;
```

would return a list like:

```
Part,PT-6170-01,1,21762.30027.65182.63525
Part,PT-6180-01,1,21762.30027.50161.30295
Part,PT-6190-01,1,21762.30027.56625.19298
Part,PT-6200-01,1,21762.30027.37094.65388
```

To indicate that there are four fields that will be returned, delimited with a comma, and the last field is the OID, you would use the following pattern:

```
*,*,*,${OID}
```

For an `expand bus` command, even more information is output before the select fields:

```
MQL< >expand bus Person "Test Buyer" - from relationship "Assigned
Buyer" select businessobject id dump |;
1|Assigned Buyer|to|Buyer Desk|Buy 001|-|37819.19807.45300.63521
```

To parse this output, you need to indicate that the first six fields, delimited by "|", should be ignored, and the seventh field is the OID. You would use:

```
*|*|*|*|*|*|${OID}
```

Format Clause

This clause defines what part of the output results should be saved in the inquiry's list. It references variables or macros specified in the Pattern, and can include delimiters.

The syntax is:

```
format VALUE;
```

- `VALUE` is the part of the output results that should be saved in the inquiry's list.

For example:

```
format ${OID};
```

Code Clause

This clause Code clause of the Add Inquiry command is used to provide the code to be evaluated to produce a list of one or more business objects.

The syntax is:

```
code VALUE;
```

- VALUE is the code commands and commands.

The code provided is generally an MQL `temp query` or `expand bus` command that selects the found objects' ids. It can contain complicated `where` clauses as needed. For example:

```
temp query bus "Package" * *
    where
        "('Project'==to[Vaulted Documents
Rev2].businessobject.to[Workspace Vaults].businessobject.type)
       &&  ('${USER}'==to[Vaulted Documents
Rev2].businessobject.to[Workspace Vaults].businessobject.from[Project
Members].businessobject.to[Project Membership].businessobject.name)"
        select id dump |;
```

When macros are included in the code (${USER} in example above), they should be surrounded by single or double quotes, in case the substitution contains a space. Quotes around both the macro in the code and the Argument when it contains a space ensures that the macro substitution is handled correctly.

File Clause

This clause does not need to be included in the Add Inquiry command itself. It can be written in an external editor.

The syntax is:

```
file FILENAME;
```

FILENAME is the name of the file that contains the code for the inquiry.

Argument Clause

This clause is used to provide any input arguments that the inquiry may need. Arguments are name/value pairs that can be added to an Inquiry as necessary to be used by the inquiry code. Depending upon how you write the code in both the Inquiry and the JSP, you may or may not use arguments.

The syntax is:

```
argument NAME [STRING];
```

- NAME is the name of the argument.
- STRING is the input argument to be added. Include quotes if the value contains a space.

Quotes around both the macro in the code and the Argument when it contains a space ensures that the macro substitution is handled correctly.

History Clause

The `history` keyword adds a history record marked “custom” to the inquiry that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Inquiry

After an inquiry is defined, you can clone the definition with the Copy Inquiry command. Cloning a inquiry definition requires Business Administrator privileges, except that you can copy a inquiry definition to your own context from a group, role or association in which you are defined.

This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy inquiry SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the inquiry definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the inquiry below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the inquiry that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Inquiry

The List Inquiry command displays a list of all inquiries that are currently defined. It is useful in confirming the existence or exact name of an inquiry that you want to modify, since it is case-sensitive.

```
list inquiry [modified after DATE] NAME_PATTERN [select FIELD_NAME {FIELD_NAME}]
[DUMP [RECORDSEP]] [tcl] [output FILENAME];
```

- For details on the List command, see *list admintype Command*.
- Use the list of all the existing inquiries along with the `Print` command to determine the search criteria you want to change.

Use the Modify Inquiry command to add or remove defining clauses and change the value of clause arguments:

```
modify inquiry NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the inquiry you want to modify.

- `MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Inquiry clause, as listed in the following inquiry. Note that you need specify only the fields to be modified.

Modify Inquiry Clause	Specifies that…
`name NEW_NAME`	The current inquiry name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
`pattern VALUE`	The pattern is changed to the new value specified.
`format VALUE`	The format is changed to the new output results specified by value.
`code VALUE`	The code associated with the inquiry is replaced by the new code specified.
`file FILENAME`	The file that contains the inquiry code is changed to the file specified.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the inquiry that is being modified. The STRING argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the arguments that define the inquiry. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying an inquiry, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire command.
- If you are working from a script, group the changes together in a single Modify Inquiry command.

Evaluate Inquiry

You can evaluate the Inquiry to determine if it will parse the output as you have designed the JSP to expect to receive it.

Use the `evaluate query` command to execute the inquiry’s code, parse it, and display the generated list.

To override any specified Arguments, or include input that the inquiry may otherwise receive from the JSP, enter name/value pairs. Include only a space between multiple inputs, using quotes around values that contain spaces.

The syntax is:

```
evaluate inquiry NAME [NAME VALUE [NAME VALUE [...]]];
```

Delete Inquiry

If an inquiry is no longer required, you can delete it using the Delete Inquiry commands

```
delete inquiry NAME:
```

- NAME is the name of the inquiry to be deleted.
- Searches the list of defined inquiries. If the name is found, that inquiry is deleted. If the name is not found, an error message is displayed. For example, to delete the inquiry named “sub-assembly parts,” enter the following:

```
delete inquiry “sub-assembly parts”;
```

After this command is processed, the inquiry is deleted and you receive an MQL prompt for another command.

interface Command

Description

An *Interface* is a group of attributes that can be added to business objects as well as connections to provide additional classification capabilities. When an Interface is created, it is linked to (previously-defined) attributes that logically go together. Interfaces are added to business object or relationship instances.

An Interface can be *derived* from other Interfaces, similar to how Types can be derived. Derived Interfaces include the attributes of their parents, as well as any other attributes associated with it directly. The types or relationships associated with the parents are also associated with the child.

The primary reason to add an interface to a business object or a connection is to add the attributes to the instance that were not defined on the type.

For conceptual information on this command, see *Interfaces* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete] interface NAME {CLAUSE};
```

- NAME is the name you assign to the interface. Interface names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the interface.

Add Interface

An object interface is created with the Add Interface command:

```
add interface NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides additional information about the interface:

description VALUE
attribute NAME{,NAME}
derived INTERFACE_NAME{,INTERFACE_NAME}
abstract [true|false]
type all | TYPE_NAME{,TYPE_NAME}
relationship all | REL_NAME{,REL_NAME}

```
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
```

All these clauses are optional. You can define an interface by simply assigning a name to it. If you do, Live Collaboration assumes that the interface is non-abstract, does not contain any explicit attributes, and does not inherit from any other interfaces. If you do not want these default values, add clauses as necessary. You will learn more about each Add Interface clause in the sections that follow.

Abstract Clause

Interfaces can inherit properties from other interfaces. Such as the following:

- *Abstract interfaces* act as categories for other interfaces.

  Abstract interfaces are not used to add attributes to a business object directly. They are useful only in defining characteristics that are inherited by other interfaces. For example, you could create an abstract interface called “Metal.” Two non-abstract interfaces, “Aluminum” and “Iron”, could inherit from it.

- *Non-abstract interfaces* are used to add groups of attributes to a business object.

  You can add non-abstract interfaces to business objects. For example, assume that Aluminum is a non-abstract interface with its own set of attributes. You can add this interface to objects that are manufactured from aluminum and therefore need this set of attributes.

Abstract interfaces are helpful because you do not have to reenter attributes that are often reused. If an additional field is required, it needs to be added only once. For example, the “Person Record” object interface might include a person’s name, telephone number, home address, and social security number. While it is a commonly used set of attributes, it is unlikely that this information would appear on its own. Therefore, you might want to define this object interface as an abstract interface.

Use one the following clauses:

```
abstract true
Or:
abstract false
```

If you want a user to be able to add the defined interface to a business object, set the abstract argument to `false`. If not, set the abstract argument to `true`. If you do not use the Abstract clause, false is assumed, allowing users to add instances of the interface to a business object.

For example, in the following definition, you cannot add the interface to a business object:

```
add interface "Metal"
   derived "Material used to Manufacture"
   abstract true;
```

Attribute Clause

This clause assigns attributes to the interface. These attributes must be previously defined with the Add Attribute command. (See *Add Attribute*.) If they are not defined, an error message is displayed.

Adding an attribute to an interface should not be included in a transaction with other extensive operations, especially against a distributed database. This is a "special" administrative edit, in that it needs to update all business objects that use the interface with a default attribute.

For the Matrix Navigator user, when viewing attributes they will appear in the reverse order of the programmed order. Therefore, you should put the attribute you want first last in the MQL script.

An interface can have any combination of attributes associated with it. For example, the following Add Interface command assigns 2 attributes to the Metal interface:

```
add interface "Metal"
   description "Material used to Manufacture"
   attribute "Metal1"
   attribute "Metal2";
```

Derived Clause

Use the Derived clause to identify 1 or more existing interfaces as the parent(s) of the interface you are creating. The parent interface(s) can be abstract or non-abstract. A child interface inherits the following items from the parent(s):

- all attributes
- all types to which it can be added

Assigning a parent interface is an efficient way to define several object interfaces that are similar because you only have to define the common items for one interface, the parent, instead of defining them for each interface. The child interface inherits all the items listed above from the parent but you can also add attributes directly to the child interface. Similarly, you can (and probably will) add attributes to the child interface that the parent does not have. Any changes you make for the parent are also applied to the child interface.

For example, suppose you have an interface named "Manufacturing Process", which includes 2 attributes: "Process" and "Vendor". Now you create two new interfaces named "Rolled" and "Molded:"

```
add interface "Rolled"
   derived "Manufacturing Process";
add interface "Molded"
   derived "Manufacturing Process";
```

Both new interfaces acquire the attributes in the Manufacturing Process interface. Rather than adding each of these attributes to the new interfaces, you can make Manufacturing Process the parent of the new interfaces. The new interfaces then inherit the 2 attributes as well as the allowed Types from the parent.

Interfaces can have more than 1 parent.

Type Clause

This clause defines all of the business object types that can use the interface. An interface may be allowed for use with any number of types, and likewise, a type may be allowed to use any number of interfaces. The Type clause is required for an interface to be usable:

```
type all | TYPE_NAME{,TYPE_NAME}
```

- `all` can be used to allow the interface on any business type.
- `TYPE_NAME` is a previously defined object type.

You can list one type or many types (separated by a comma or carriage return). When specifying the name of an object type, it must be of a type that already exists in the database. If it is not, an error message will display.

For example, the following command is a valid Add interface command:

```
add interface "Metal"
      description "Material used to Manufacture"
     type Part,Component;
```

Relationship Clause

This clause defines the relationships that can use the interface. An interface may be allowed for use with any number of relationships, and likewise, a relationship may be allowed to use any number of interfaces. Specify at least a type or a Relationship for an interface to be usable:

```
relationship all | REL_NAME{,REL_NAME}
```

- `all` can be used to allow the interface on any relationship that exists in the database.
- `REL_NAME` is a previously defined relationship.
- You can list one or many relationships (separated by a comma or carriage return). The relationships you specify must already exist in the database. If it is not, an error message will display.

For example, the following command is a valid Add interface command:

```
add interface "Material"
  description "Material used to Manufacture"
  relationship Color,Texture;
```

Copy Interface

After an interface is defined, you can clone the definition with the Copy Interface command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy interface SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the interface definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Interface

You can change the interface definition with the Modify Interface command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify interface NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the interface you want to modify.
- `MOD_ITEM` is the interface of modification you want to make.

You can make the following modifications. Each is specified in a Modify Interface clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Interface Clause	Specifies that…
`name NEW_NAME`	The current interface name changes to that of the new name entered.
`description VALUE`	The current description, if any, changes to the value entered.
`icon FILENAME`	The image is changed to the new image in the field specified.
`add attribute NAME`	The named attribute is added to the interface's list of explicit attributes.
`remove attribute NAME`	The named attribute is removed from the interface's list of explicit attributes.
`derived INTERFACE_NAME{,INTERFACE_NAME}`	The interface being modified inherits attributes and allowed types of the interface(s) named. Overwrites any previously defined parents.
`remove derived`	Removes all parents.
`abstract true`	Business object instances of this interface cannot be created.
`abstract false`	Business object instances of this interface can be created.
`add type NAME`	The named type is added to the interface's list of allowed types.
`remove type NAME`	The named type is removed from the interface's list of allowed types.
`add relationship REL_NAME to [minorrevision] [none|float\replicate]`	The named relationship is added to the interface's list of allowed relationships.
`remove relationship REL_NAME`	The named relationship is removed from the interface's list of allowed relationships.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMININTERFACE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMININTERFACE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMININTERFACE NAME] [value STRING]`	The named property is removed.

Each modification clause is related to the arguments that define the interface.

Adding and removing attributes to an interface have the same effect as adding and removing them from a part. If many objects use the interface, many database rows are affected by the change, and so the transaction has the potential to be very time-consuming.

Deleting an Interface

If an interface is no longer required, you can delete it with the Delete Interface command:

```
delete interface NAME:
```

- `NAME` is the name of the interface to be deleted.
- Searches the list of interfaces. If the name is not found, an error message is displayed. If the name is found and the interface does not derive another interface, and there are no business objects that use the interface, the interface is deleted. If it is a parent interface, you must first delete the interfaces that are derived from it.

For example, to delete the interface named “Metal,” enter the following MQL command:

```
delete interface Metal;
```

Deletes the interface if:

- There are no business objects that use this interface.
- There are no interfaces derived from this interface.

Add/Remove Interface Clause

You can add (or remove) the collection of interface attributes to a business object with the add interface (or remove interface) clause of the modify business object command:

```
add interface NAME
```

```
remove interface NAME
```

For example:

```
mod bus Item 1000 A add interface Aluminum add interface Milled;
mod bus Item 2000 A add interface Iron add interface Rolled
mod bus Item 3000 A add interface Iron remove interface Rolled;
```

Remember to include the keyword add or remove before you specify each interface name.

When modifying a business object’s type, if the new type allows the interface(s), the interface(s) will be preserved. If not, the interface(s) will be removed.

local pathtype Command

Description

When defining a pathtype, you can associate a pathtype with a specific Type or Relationship to create a local pathtype. For more information about the Pathtype Command, see pathtype Command.

A local pathtype can only be used as part of a specific admin type. The following rules apply to local pathtypes:

- All pathtypes (local and global) can coexist in one database.
- Business types and Relationships can have local pathtypes.
- Local pathtypes cannot have . (dot) in the file name.
- You can reference local pathtypes using AdminTypeName and PathType name separated by a . (dot). This form of local pathtype reference is called a qualified name, for example:
  - T1.cost
  - R1.cost

    By definition, the qualified name for global pathtypes is in the form .pathtypeName. For example, the qualified name for a global pathtype cost will be .cost.

    For unique identification of an admin type by qualified name, qualified names must be unique across Types, Relationships, and global pathtypes with . (dot) in the name.
- Two pathtypes, local or global, with the same name cannot live together in the same hierarchy. For example, type T1 can not have a global pathtype of cost added.
- When adding a local pathtype to a Type/Relationship, the system checks that the qualified name does not conflict with existing local pathtype names in eligible Administrative definitions and global pathtypes.
- Local and global pathtypes with the same name can be of different types.
- Deleting an Administration definition with local pathtypes deletes all local pathtypes belonging to this definition as well.
- In the context of Admin Object Definition or Business Object it is not necessary to use qualified names.
- If there is ambiguity in the pathtypes name interpretation, use qualified names.
- Local PathTypes do not support aliases.

User Level

System Administrator

Add Local Pathtype

Use the Add Pathtype command to define a local pathtype.

```
add pathtype NAME type TYPE […] owner  type TYPE_NAME;
```

```
add pathtype NAME type TYPE […] owner  relationship
RELATIONSHIP_NAME interface INTERFACE_NAME;
```

```
add pathtype NAME type TYPE […] owner  type TYPE_NAME;
```

The add/mod type NAME pathtype NAME {, NAME} creates an error if you use it to assign a local pathtype to a type. (The same behavior occurs in a relationship.)

It may be more logical to define a local pathtype through the Admin Definitions they belong to, but it is more costly to program this.

Delete Local Pathtype

The Delete Local Pathtype command use the pathtypes qualified names, for example T1.cost for the local pathtype and cost for the global pathtype. For example:

```
delete pathtype T1.cost;
```

List Local Pathtype

The List Local Pathtype command returns global and local pathtypes when the pathtype is referred to only by name. For example:

```
list pathtype cost;
   will return
cost (for global pathtype)
  I1.cost
  R1.cost
  T1.cost
T2.cost
```

The global pathtype cost is not displayed with qualified name as .cost.

To list only global pathtypes with name cost, use the where clause.

```
list pathtype cost where owner=="";
cost
```

To list local pathtypes owned by an administration object, use the where clause.

```
List pathtype * where owner==t2;
        T2.cost
```

To list only local pathtypes with name cost:

```
List pathtype * where owner!="";
I1.cost
  R1.cost
  T1.cost
T2.cost
```

Where Clauses

Where clauses expand the pathtype name to include local pathtypes as well. There are two ways of using pathtype values inside where clauses:

- Where clause contains `pathtype[NAME] <op> <value-pattern>`. This expression can be converted into efficient SQL. The following conditions are checked:
  - If pathtypes are of different types, the system displays a warning message and the query is not evaluated.
  - If pathtype types are real or integer and the <value-pattern> is a string, which contains dimension and some pathtypes have dimension, while others do not, the system displays a warning message and the query is not evaluated.
  - If pathtype types are real or integer and <value-pattern> is a string, which contains dimension, and pathtype types have different dimensions, a warning message appears and query not evaluated.
- Where clause contains pathtype[NAME].value <op> <value-pattern>. This expression is not converted into efficient SQL. The <value-pattern> from the where clause is evaluated on each object in the result set, and therefore no checking on expanded pathtype type list is done.

For example, when type t1 has local pathtype T1.cost and there is a global pathtype cost with type integer, include the evaluation of the local pathtype for type T1 and global pathtype. For example:

```
temp query * * * where [pathtype[cost]>15]
```

To evaluate the query for local or global pathtype only, use the qualified name of the pathtype. For example, to evaluate the query for global pathtype:

```
temp query * * * where [pathtype[.cost]>15]
```

Export of Local Pathtypes

Local pathtypes are exported as part of the owner's export.

Modify Local Pathtype

After a local pathtype is defined, you can change the definition with the Modify Local Pathtype command. The Modify Local Pathtype command uses the pathtype qualified names, for example, T1.cost for the local pathtype and cost for the global pathtype.

```
modify pathtype T1.cost default 0;
```

The owner modification on the local pathtype is not allowed. You can modify a global pathtype into a local pathtype if it is in use by no more than a single Administration definition. However, you cannot modify a local pathtype to become a global pathtype.

Pathtype modifications allow only a name or qualified name for the local pathtype. For example:

```
mod bus T N R cost 10;
mod bus TNR T.cost 10
```

Print Local Pathtype

In the context of data instance or admin definition, pathtypes display only the pathtype name. For example:

```
 print bus T N R select pathtype;
    pathtype = cost
print type T1 select pathtype;
    pathtype = cost
print connection ID select pathtype;
    pathtype = cost
print relationship R1 select pathtype;
    pathtype = cost
```

In the following examples, pathtype is referred to by name and by qualified name. The following examples return the same value of the local pathtype, cost:

```
print bus T N R select pathtype[cost];
         pathtype[cost] = 1
print bus T N R select pathtype[T.cost];
         pathtype[T.cost] = 1
```

location Command

Description

Locations contain alternate host and path information for a captured store. The host and path in the store definition is considered to be the *default* location for the store, while any associated location objects identify *alternate* file servers.

Think of a location as another store—it is defined as file "location." The same rules apply as in specifying a store.

For conceptual information on this command, see *Locations and Sites* in Chapter 2.

User Level

System Administrator

Syntax

```
[add|modify|synchronize|delete|purge|print]location NAME
[CLAUSEs];
```

- NAME is the name you assign to the location. Location names must be unique. This means a location and store cannot have the same name. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the location.

Add Location

Use the `add location` command to defined a location:

```
add location NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM is an Add Location clause that provides more information about the location you are creating.

The Add Location clauses are:

[!|not]hidden
description VALUE
fcs FCS_URL
host HOST_NAME
password PASSWORD
path PATH_NAME
prefix PREFIX

```
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]
port PORT_NUMBER
property NAME [to ADMINTYPE NAME] [value STRING]
protocol PROTOCOL_NAME
url VALUE
user USER_NAME
deletetrigger DELETE_TRIGGER_IMPLEMENTATION
```

Host Clause

This clause identifies the system that is to contain the location being defined. If a host is not specified, the current host is assumed and assigned.

If the location is to be physically located on a PC and accessed through a network drive, the host name must be set to `localhost`. For example:

```
add location host localhost
```

Protocol and Port Clauses

When creating a location object for captured stores, you can include the parameters `protocol` and `port`. Protocol and port can be defined when creating a location object for captured stores. The protocol defined for a location determines the file access given by the controlling FCS.

```
protocol PROTOCOL_NAME
```

- `PROTOCOL_NAME` is file for captured store locations and HTTP or HTTPS for DesignSync locations.

This protocol is not related to the protocol used for file transfers between FCSs during synchronization operations. The HTTP/S protocol is always used for these operations.

Each protocol has a default port that is used if not specified in the location definition. Include the Port clause to specify a port other than the default.

```
port PORT_NUMBER
```

If a location does not have a protocol specified, Live Collaboration looks at other object attributes to determine which protocol was likely intended.

- If the host is *not* 'localhost' (or empty), and there is no username/password, the protocol used will be 'file' (local file system).
- If it is a designsync store, the default is http.

Note these checks eliminate the need to add protocol/port parameters to store/locations added prior to version 9.

Password Clause

The password provides access to the DesignSync account.

The Password clause uses the following syntax:

```
password PASSWORD
```

- `PASSWORD` is the DesignSync password.

Path Clause

The path identifies where the location is to be placed on the host. A location should NOT point to the same directory as a store since this may cause unnecessary file copying. When a location is defined, a directory for the captured files is created.

It is important to note that if an invalid directory is specified for the PATH parameter during location or store creation, only a warning is issued. The transaction will NOT abort and the location or store is created with the invalid directory. Any checkin or copy operation using this new location or store will fail.

For example, you could use the following clause:

```
path Drawings
```

An absolute path can also be entered; however, the parent directory must already exist. For example, in the following command, if the `/stores` directory does not exist on the target host, the location will not be created.:

```
path stores/store1
```

Prefix Clause

A prefix can be added to the path defined for a captured store or location to enhance file management operations. See the *Prefix Clause* for the store command for details.

Permission Clause

File permissions are no longer supported for captured stores, and have no effect on files and folders that are managed by the FCS.

Warning: Donot directly modify or change folders and files managed by the FCS in any way (etc.. rights, size, etc.).

User Clause

When moving files to/from a store location, the DesignSync username defines the DesignSync account used.

The User clause uses the following syntax:

```
user USER_NAME
```

- USER_NAME is the DesignSync username.

FCS Clause

If using FCS for file checkins and checkouts with a captured location, you must specify the URL for the location's server.

Include the Web application name. The syntax is:

```
fcs http://host:port/WEBAPPNAME
```

For example:

```
fcs http://host:port/ematrix
```

If using Single Sign On (SSO) with FCS, the FCS URL should have the fully qualified domain name for the host. For example:

```
fcs http://HOSTNAME.MatrixOne.net:PORT#/ematrix
```

You can also specify a JPO that will return a URL. For example:

```
fcs ${CLASS:prog} [-method methodName] [ args0 … argN]
```

For information about FCS processing and configuration, see the *File Collaboration Server Guide*.

DeleteTrigger Clause

The DeleteTrigger clause allows you to implement alternate behavior for when files are being deleted from a store or location. To add a no-delete policy on a location, you can use the out of the box implementation in com.matrixone.fcs.backend.NoFileDelete. For example:

```
add location MyLocation type captured deletetrigger
com.matrixone.fcs.backend.NoFileDelete;
```

You can also write your own Java class that implements the com.matrixone.fcs.mcs.DeleteTrigger interface to insert any customized behavior, such as altering the file delete list. It can be either a JPO or a regular Java class that can be found in the classpath.

The DeleteTrigger that is set for a store automatically applies to all locations that are associated with that store, unless it is overridden at a particular location.

Modify Location

After a location is defined, you can change the definition with the Modify Location command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify location NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the location to modify.

- `MOD_ITEM` is the type of modification to make. Each is specified in a Modify Location clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Location Clause	Specifies that…
`description VALUE`	The description is changed to the new value specified.
`host HOST_NAME`	The host name is changed to the value specified.
`icon FILENAME`	The image is changed to the new image in the file specified.
`protocol PROTOCOL_NAME`	The protocol is changed to the value specified.
`port PORT_NUMBER`	The port is changed to the number specified.
`name NAME`	The location name is changed to the new name.
`password PASSWORD`	The DesignSync password is changed to the value entered.
`path PATH_NAME`	The current comment, if any, is changed to the value entered.
`prefix PREFIX`	The prefix is changed to the value entered.
`permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]`	File permissions are no longer supported for captured stores, and have no effect on files and folders that are managed by the FCS. **Warning:** Do not directly modify or change folders and files managed by the FCS in any way (e.g., rights, size, etc.).
`user USER`	The DesignSync username is changed to the name specified.
`url VALUE`	The current URL is changed to the new one entered.
`fcs FCS_URL`	The current FCS URL is changed to the new one entered.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`deletetrigger DELETE_TRIGGER_IMPLEMENTATION`	Captured file delete behavior is modified. DELETE_TRIGGER_IMPLEMENTATION implements the com.matrixone.fcs.mcs.DeleteTrigger interface. It can be either a JPO or a regular Java class that can be found in the classpath. See also *DeleteTrigger Clause*.

As you can see, each modification clause is related to the clauses and arguments that define the location.

When you modify a location, you first name the location and then list the modifications. For example, the following command changes the permissions of the NY-Documents location:

```
modify location "NY-Documents"
   permission rw, r, r;
```

When this command is executed, the permissions of the location become read/write for the owner and read-only for group and world.

Print Location

The Print Location command prints the location definition to the screen allowing you to view it. When a Print command is entered, MQL displays the various clauses that make up the definition.

```
print location LOCATION select FIELD;
```

- `LOCATION` is the name of the specific location instance.
- `FIELD` lets you specify data to present about the location being printed.

Synchronize Location

Synchronization ensures that users access the latest versions of files when checking out or viewing files in a replicated environment. After a new location is added for a store, you can synchronize the location with an existing location defined for that store.

Before you can synchronize two locations, a copy of the files checked into the source location must be available to the FCS of the destination location and the exact file system checked into the source location must be copied to the destination location via HTTP or other means.

After the file system is copied, a System Administrator must verify that it has been copied exactly to the destination location, retaining all directories and physical files. The sync command will not attempt to verify this. If any files are not correctly copied to the destination location, errors may result in subsequent commands that attempt to access those absent files.

Once the file system is copied correctly, you can then synchronize the two locations with the `sync location` command. A database record is created and any business object with files in the source location is now aware of the exact copies of those files in the destination location.

It is highly recommended that the destination location be empty with no records referring to it before the source location directory is copied. Errors may occur if the new location is not empty.

To synchronize two locations defined for a store, use the Sync Location command:

```
sync location NAME_SOURCE to NAME_DEST;
```

- `NAME_SOURCE` is the name of the source location. This location should contain the data you would like to replicate in the new location.
- `NAME_DEST` is the name of the destination location. This location is the newly created location for the store that you would like to syncronize with the source location.

Delete Location

If a location becomes obsolete, you can delete that location by using the Delete Location command:

```
delete location NAME;
```

- `NAME` is the name of the location to be deleted.

Searches the list of defined locations. If the name is not found, an error message is displayed. If the name is found, the location is deleted.

Purge Location

The Purge Location command removes all files from the location. For example, if you want to purge all files in the location called Shelton, enter the following MQL command:

```
purge location Shelton;
```

For more flexibility, you can add a date restriction to the Purge Location command to remove only files that were last checked in prior to the date specified in the command. For example, if you want to purge only files last checked in prior to February 28, 2008 at 9:57:32 AM EST in the location called Shelton, you would enter the following MQL command:

```
purge location Shelton before "Thu Feb 28, 2008 9:57:32 AM
EST";
```

The date format used in this command must be the same format configured in the variable MX_NORMAL_DATETIME_FORMAT for dates in the 3DEXPERIENCE Platform. The time zone in the date restriction must be the MCS time zone and should not be the location time zone.

mail Command

Description

IconMail is a legacy internal communication interface and should no longer be used. It is highly recommended that you use an external mail utility and that all users are defined with a non-3DEXPERIENCE Platform email address.

IconMail is an internal mail system that enables users to easily exchange business objects and text messages. This mail utility is similar to other electronic mail systems. However, as its name suggests, the icon of the discussed object is sent with the message, allowing the recipient to view or edit the object from within the mailbox. You can also send mail without sending a business object at all, or send more than 1 object in 1 mail.

While the IconMail utility is most often used from within the 3DEXPERIENCE Platform, you can access it through MQL. This enables the Business or System Administrator to send messages while performing other MQL commands. For example, the Administrator can use MQL to load external files for a group's use or assign a person to a role. Once the action is completed, in MQL the Administrator could send a mail message to notify the appropriate group that the job is done.

Do not modify or delete the persons creator, guest, or Test Everything using MQL. Modifying or deleting these objects could cause triggers, programs or other application functions not to work.

Every user has access to IconMail unless it has been disabled in the user's person definition by the Business Administrator. For example, a user may create business objects and could use IconMail to notify the appropriate people about the existence and states of the objects.

The sender of mail does not know how it is received. It could be received as IconMail, email, or both. With email, the type, name, and revision of sent objects are added; but the email recipient has no direct access to objects from the mail message.

User level

Business Administrator

Syntax

```
[send|print|delete]mail {CLAUSE};
```

- CLAUSEs provide additional information about the mail.

Send Mail

You use the Send Mail command to send mail to another 3DEXPERIENCE Platform user. You can send just text, or you can include 1 or multiple objects.

```
send mail [businessobject BO_NAME {businessobject BO_NAME}]
[in VAULT] to USER_NAME[{,USER_NAME}][{ITEM}];
```

- `BO_NAME` is the type name and revision of the business object.

- VAULT is the vault where the business object is held.
- ITEM is any of these optional Send Mail command clauses:

```
cc USER_NAME {,USER_NAME}
bcc USER_NAME {,USER_NAME}
subject VALUE
text VALUE
```

Each clause and the arguments they use are discussed in the sections that follow.

To Clause

This clause identifies who should receive the message you are sending. It can contain a list of users:

```
to USER_NAME {,USER_NAME}
```

- USER_NAME is the name of a person, group, role, or association defined within the database. Depending on your system setup, user's names may be case sensitive. If the name is not found, an error message is displayed. The fact that you can insert a role name or group name means that you not only have existing mailing lists, but also can send a message to a person whose name you do not know but whose function it might be to process the information you have.

For example, you may want to send an announcement to everyone working on your project to inform them that certain materials are available for use:

```
send mail to "Vehicle Manufacturing"
   subject "Materials Now Available"
   text "The materials for the V34 Solar Vehicle are now available.
   For more information contact Mike Zimmerman at ext 511.";
```

When this message is sent, it will go to every user defined in the Vehicle Manufacturing group. If additional groups are needed, you can list them also by separating the names with a comma.

CC Clause

This clause circulates the mail to additional people. While the message may be intended for a single individual, this clause lets you notify others that the correspondence has taken place. Use the following syntax:

```
cc USER_NAME {, USER_NAME}
```

- USER_NAME is the name of a person, role or group within the database. Depending on your system setup, user's names may be case sensitive. If the name is not found, an error message is displayed.

BCC Clause

This clause circulates the mail to additional people in such a way that the mail each person receives does not contain the complete list of recipients who also received the mail. Each person can see all

the user names specified in the To: and Cc: fields but cannot see the user names (other than their own) specified in the Bcc: field. Use the following syntax:

```
bcc USER_NAME {, USER_NAME}
```

- USER_NAME is the name of a person, role or group within the database. Depending on your system setup, user's names may be case sensitive. If the name is not found, an error message is displayed.

Subject Clause

This clause places a header on the mail message. This header is usually a short synopsis of the message's content. Use this syntax:

```
subject VALUE
```

- VALUE is any string of characters following the MQL syntax rules. VALUE is limited to 255 bytes.

By examining the subject header, the message reader should be able to identify the content or purpose of the mail message. For example, the following Subject clauses clearly indicate the content of the mail message:

```
subject "Testing of Building Fire Alarms this Weekend"
subject "Company Christmas Party December 19"
subject "Safety while using the new test equipment"
```

Depending on the purpose of the message you are sending, you may not include any text at all.

Text Clause

This clause contains the bulk of your mail message:

```
text VALUE
```

- VALUE is any length character string you want to enter. The message can be short or quite lengthy.

When writing the text of your mail message, you must enclose the entire content within either single quotes (' ') or double quotes (" "). If your message includes apostrophes, enclose the message in double quotes. If your message includes double quotes, enclose the message in single quotes.

Print Mail

IconMail is always sent to the server to which you're connected. However, in a distributed environment, you must point to the server from which you want to read mail.

You can read mail from MQL using the Print Mail command:

```
print mail [server SERVER_NAME] [# | all];
```

- SERVER_NAME is the name of the server from which you want to read mail messages.

is the message number to print. The message number equals the mail message OID (Object ID).

Use `all` to print all messages from the specified server, or from the default server if no server is specified.

The `print mail` command without arguments prints all messages in the current user's mailbox.

Delete Mail

After you are through with a mail message, you can delete it with the Delete Mail command:

```
delete mail [server SERVER_NAME] [# | all];
```

- `SERVER_NAME` is the name of the server from which you want to delete mail messages.

is the number of the specific message to be deleted. The message number equals the mail message OID (Object ID).

Use `all` to print all messages from the specified server, or from the default server if no server is specified.

Searches the list of existing mail messages. If the number is found, the mail message associated with that number is deleted along with any mail references to any business objects. If the number is not found, an error message is displayed.

If you want to delete all your IconMail, use the Delete Mail All command:

```
delete mail all;
```

When this command is processed, all your IconMail messages and the references to any business objects associated with them are deleted.

Only the mail reference to a business object is deleted—the business object remains untouched. Do not worry about inadvertently deleting an object when deleting an IconMail message.

menu Command

Description

Menus can be used in custom Java applications. Menus can be designed to be toolbars, action bars, or drop-down lists of commands.

Before creating a menu, you must define the commands that it will contain, since commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it.

For conceptual information on this command, see *Menus* in Chapter 8.

User Level

Business Administrator with Menu admistrative access

Syntax

```
[add|copy|modify|delete]menu NAME {CLAUSE};
```

- NAME is the name you assign to the menu. Menu names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the menu.

Add Menu

To define a menu from within MQL use the Add Menu command:

```
add menu NAME [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the menu you are defining. Menu names cannot include asterisks.

You cannot have both a command and a menu with the same name.

- ADD_ITEM provides additional information about the menu. The Add Menu clauses are:

description VALUE
label VALUE
href VALUE
alt VALUE
menu NAME {,NAME}
setting NAME [STRING]
[!|not]hidden

```
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

Each clause and the arguments they use are discussed in the sections that follow.

Label Clause

This clause specifies the label to appear in the application in which the menu is assigned. For example, many desktop applications have a File menu.

Href Clause

This clause is used to provide link data to the JSP. The Href link is evaluated to bring up another page. Many menus will not have an Href value at all. However, menus designed for the “tree” menus require an Href because the root node of the tree causes a new page to be displayed when clicked. The Href string generally includes a fully-qualified JSP filename and parameters, which can contain embedded macros and expressions for mapping to database schema. Refer to *Using Macros and Expressions in Configurable Components* in Chapter 8 for more details.

The syntax is:

```
href VALUE;
```

`VALUE` is the link data.

Alt Clause

This clause is used to define text that is displayed until any image associated with the menu is displayed and also as “mouse over text.”

The syntax is:

```
alt VALUE;
```

`VALUE` is the text that is displayed.

For example, you could use the following for a Tools menu:

```
alt “Tools”;
```

Menu Clause

This clause is used to specify existing menus to be added to the menu you are creating. The menus will be displayed in the order in which they are added. Separate items with a comma.

The syntax is:

```
menu NAME {,NAME};
```

`NAME` is the name of the menu you are adding.

Command Clause

This clause is used to specify existing commands to be added to the menu you are creating. The commands will be displayed in the order in which they are added. Separate items with a comma.

The syntax is:

```
command NAME {,NAME};
```

NAME is the name of the command you are adding.

For details on creating commands, see *Add Command*.

Setting Clause

This clause is used to provide any name/value pairs that the menu may need. They can be used by JSP code, but not by hrefs on the Link tab. Refer to *Using Macros and Expressions in Configurable Components* in Chapter 8 for more details.

The syntax is:

```
setting NAME [STRING];
```

For example, an image setting with the image name can be specified to display when the menu is used in a toolbar:

```
setting Image iconSmallMechanicalPart.gif;
```

History Clause

The history keyword adds a history record marked “custom” to the menu that is being added. The STRING argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Menu

After a menu is defined, you can clone the definition with the Copy Menu command. Cloning a menu definition requires Business Administrator privileges, except that you can copy a menu definition to your own context from a group, role or association in which you are defined.

This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy menu SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- SRC_NAME is the name of the menu definition (source) to be copied.
- DST_NAME is the name of the new definition (destination).
- MOD_ITEMs are modifications that you can make to the new definition. Refer to the menu below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the menu that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Menu

The List Menu command is used to display a list of all menus that are currently defined. It is useful in confirming the existence or exact name of a menu that you want to modify, since the Live Collaboration is case-sensitive.

```
list menu [modified after DATE] NAME_PATTERN [select FIELD_NAME {FIELD_NAME}]
[DUMP [RECORDSEP]] [tcl] [output FILENAME];
```

For details on the List command, see *list admintype Command*.

Use the list of all the existing menus along with the `Print` command to determine the search criteria you want to change.

Use the Modify Menu command to add or remove defining clauses and change the value of clause arguments:

```
modify menu NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the menu you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Menu clause, as listed in the following menu. Note that you need specify only the fields to be modified.

Modify Menu Clause	Specifies that…
name NEW_NAME	The current menu name is changed to the new name entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is chanfed to the new image in the file specified.
label VALUE	The label is changed to the new value specified.
href VALUE	The link data information is changed to the new value specified.
alt VALUE	The alternate text is changed to the new value specified.
order menu NAME NUMBER	The order of the named menu item included in the menu is changed to the NUMBER specified. See *Modifying the order of items in a Menu*.
order command NAME NUMBER	The order of the named command item included in the menu is changed to the NUMBER specified. See *Modifying the order of items in a Menu*.
add menu NAME	The named menu is added to the menu.
add command NAME	The named command is added to the menu.
add setting NAME [STRING]	The named setting and STRING are added to the menu.
remove menu NAME	The named menu is removed from the menu.
remove command NAME	The named command is removed from the menu.
remove setting NAME [STRING]	The named setting and STRING are removed from the menu.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.
history STRING	Adds a history record marked “custom” to the menu that is being modified. The STRING argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the arguments that define the menu. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a menu, you can make the changes from a script or while working interactively with MQL.

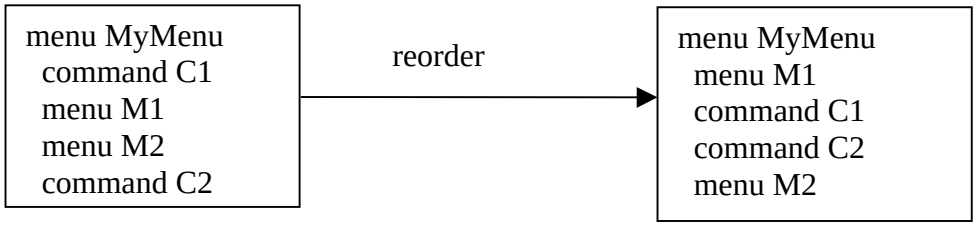
- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire command.
- If you are working from a script, group the changes together in a single Modify Menu commandcommand.

Modifying the order of items in a Menu

The `Order` clause on the `modify menu` command is used to re-order menu items. For example, to place a new command (C3) in the third spot of an existing menu (MyMenu) that already has five items, you would issue the command:

```
modify menu MyMenu add command C3 order command C3 3;
```

Another example is given below showing the before and after effect of ordering.



In order to make this change you would have to issue the following command in MQL:

```
modify menu MyMenu order command C1 2 order command C2 3;
```

Delete Menu

If a menu is no longer required, you can delete it using the Delete Menu command

```
delete menu NAME:
```

`NAME` is the name of the menu to be deleted.

Searches the list of defined menus. If the name is found, that menu is deleted. If the name is not found, an error message is displayed. For example, to delete the menu named “Toolbar” enter the following:

```
delete menu "Toolbar";
```

After this command is processed, the menu is deleted and you receive an MQL prompt for another command.

monitor Command

Description

The *monitor* command is used to provide details about server memory use, active database sessions, and active execution threads.

User Level

System Administrators

Syntax

```
monitor[memory|context|server];
```

- For syntax details, click the links above.

Monitor Memory

The `monitor memory` command issues memory statistics for the Collaboration server:

```
monitor memory;
```

This command is available only on Windows and HP platforms.

Here is an example monitor memory output for a server on Windows with Java 1.4:

```
Used heap 6667360 bytes, free heap 1182488 bytes.
JVM total memory: 2031616
JVM free memory: 916072
JVM memory in use: 1115544
```

For MQL processes, the JVM memory statistics are included in the “monitor memory” output only if the JVM has been loaded (that is, if a JPO invokes it directly or via a trigger).

While the `monitor memory` *command is available on all platforms, general heap information obtained from the OS is included only on HP and Windows. Memory manager and JVM memory information is output on all platforms.*

Example output for an RMI server on HP with Java 1.5. The segment in **bold** is HP-specific. The JVM statistics in *italics* are specific to Java 1.5.

```
free memory is 343907 pages, 1408643072 bytes
This process is using 778313728 bytes of RAM.
This process is using 1035005952 bytes of VM.
This process is using 16707584 bytes of data.
Memory Manager:
1591903 bytes of memory allocated in 177 blocks, highwater= 1611703
bytes
16384 bytes of memory reserved in 1 blocks
```

```
JVM total memory: 643170304
JVM max memory: 643170304
JVM free memory: 637580544
JVM memory in use: 5589760
JVM available processors: 2
```

Only users with System Administrators privileges can execute the `monitor memory` *command.*

JVM Memory Details

JVM memory statistics are part of the output of monitor memory and monitor context commands. Output varies depending on platform and Java version. For example, additional fields with Java 1.5 might look like:

```
JVM total memory: 661782528
JVM max memory: 661782528
JVM free memory: 649235280
JVM memory in use: 12547248
JVM available processors: 4
```

Following is sample output for the `monitor context` command, taken using a Studio Customization Toolkit program that mimics Studio Modeling Platform MQL functionality. Note that the string named after the session name `(current)` indicates the context corresponding to the current user session.

```
mql>monitor context
Pooled session cache: 0 bytes

4 context objects

Session PUF93l21k9lAjAH0hy0O2WOZCYOhggu2dvWd0owsfT9DDHzH1I5P|
-263669613110616712
0/167839130/6/7001/7001/7002/7002/7001/-1
    User:   'Test Everything' logged in
    Vault:  'eService Sample'
    Last:   t@2208, select.bosBusinessObject
    Last recorded cache size: 0
    idle:   14 minutes 52 seconds
    0 active sessions

Session mx1027692483622676970837 (current)
    User:   'creator' logged in
    Vault:  'ADMINISTRATION'
    Last:   t@2104, executeCmd.bosMQLCommand
    Last recorded cache size: 0
    Idle: 0 seconds
    1 active session
      session 0
        transaction active,readonly,wait
        0 cached entries, 0 bytes
```

```
Session mx10277030735012117155733
    User:   'creator' logged in
    Vault:  'ADMINISTRATION'
    Last:   t@1940, executeCmd.bosMQLCommand
    Last recorded cache size: 505579
    idle:   5 seconds
    2 active sessions
      session 0
        transaction active,update,wait
          savepoint save1
        3 cached entries, 3192 bytes
      session 1
        transaction active,update,wait
        3 cached entries, 502387 bytes

Session mx1027703338082-1990050644
    User:   'creator' logged out
    Vault:  'ADMINISTRATION'
    Last:   t@2284, executeCmd.bosMQLCommand
    Last recorded cache size: 0
    idle:   11 minutes 39 seconds
    0 active sessions

Total cache size: 505579 bytes
```

Only users with System Administrator privileges can execute the `monitor context` *command.*

Notes

- If a context is currently executing at the time another context issues the `monitor context` command, output for the active session will resemble the following:

```
Session
PUGpAOiBm3OcnMK5Bk27QcQYjcm2iq2UwMQzxh9KGjTTddp52xkr|-2636696131106167
12
0/167839130/6/7001/7001/7002/7002/7001/-1
    User:   'Test Everything' logged in
    Vault:  'eService Sample'
    Last:   t@1956, executeCmd.bosMQLCommand
    Active: 2 seconds
    Last recorded cache size: 0 (update requested; reissue monitor
context command)
*** Cannot report session stats - session is active ***
```

  The above sample shows the “Active” time for the session, and a warning message states “…session is active.”

In the interest of thread safety, monitor context processing reports only what is safe to report. Access to cache and transaction information during monitor context processing is done in a thread-safe fashion so it does not impact system performance and stability of other sessions and the system as a whole.

Monitor Context

The `monitor context` command is used to count and list currently registered context objects and give statistics.

The `monitor context` command provides the following information per session:

- number of contexts
- idle vs. active session
- total cached bytes
- session ID
- context set/logged in status
- username
- timestamp for and name of last Studio Customization Toolkit call and thread on which it executed
- transaction status (for example: active, mode, savepoints, etc.)
- estimate of context-specific cache size in use, if possible, for each active session

The command displays the following environment information for all sessions:

- Total number of sessions
- Total session cache size
- Pooled session cache size
- JVM memory statistics

```
monitor context [SESSION-ID] [set|!set] [terse];
```

`SESSION-ID` is used to limit the display of session information to the context object for the specified session ID. If not used, all sessions are reported.

The `[set|!set]` option limits the display of session information to contexts that are marked as "set" or "not set" respectively. (The Studio Customization Toolkit `context.shutdown()` method will mark a context "not set.") Use this option only when the session ID is not specified.

`terse` displays only cumulative statistics and statistics, not individual sessions.

Monitor Server

The monitor server command can be issued against a running server through the emxRunMQL admin tool, or by connecting directly to the server from a separate standalone Java program or custom JSP. Any user can execute the `monitor server` command to print details of all active threads (though not Java stack trace information as it is included with kill -QUIT command).

```
monitor server [age SECONDS] [port] [filename FILENAME
[append]];
```

Include `age SECONDS` to limit the list of Studio Customization Toolkit calls found in the core to those running for longer than the specified number of seconds.

Include `port` to report the MX_DEBUG_PORT number on which the server is listening, or -1 if not listening.

Port and age options are mutually exclusive.

Include `filename FILENAME` to dump the output to the file specified, to be created in the directory pointed to by the MX_TRACE_FILE_PATH variable on the server. If the file already exists, you can specify append, or else the original file will be renamed with the date and timestamp.

There is no special user privilege required to execute the monitor server command.

mql Command

Description

MQL is the Matrix Query Language. Similar to SQL, MQL consists of a set of commands that help the administrator build and test a database quickly and efficiently. You can also use MQL to add information to and extract information from the existing database.

User Level

Business Administrator

Syntax

Enter the MQL command using the following syntax:

```
mql [options] [-] [file...]
```

Clauses (Params)

The mql clauses are:

-b Option	By default, Live Collaboration reads matrix-r as the bootstrap file. The -b modifier indicates that an alternate bootstrap file should be used. The bootfile is expected to be in your ENOVIA_INSTALL directory. For example, suppose you have two databases that you frequently access, a test system (Mx_qtest) and a production system (Mx_production). You could create two shortcuts on your Windows Start menu. The shortcut for the test database could be called QAmatrix and contain the following: c:\enoviaV6R2011\studio\bin\winnt\mql.exe -b Mx_qtest The shortcut for the production database could be called PRDmatrix and contain the following: c:\enoviaV6R2011\studio\bin\winnt\mql.exe -b Mx_production Generally, there is no need to duplicate or move Matrix .ini files. However, if multiple files will be opened for view or edit from a database that is not using the standard matrix-r file, errors will occur.

-c Option	This option enables you to enter MQL commands from the system command line. This option specifies that the MQL commands enclosed within the double quotes are processed. For example, the following command assigns new telephone and facsimile numbers to a defined user (Joe) from the system command line: mql -c "modify person Joe phone 598-4354 FAX 598-4355;" You can include more than one MQL command in the command string. In the syntax COMMAND; {COMMAND;} you can include as many commands as you want, providing each command ends with a semicolon and space and all the MQL commands are enclosed within the quotes.
-d Option	This option suppresses MQL window, but not the title information. See also -t Option.

-install option	Use this option to create a bootfile to connect to the database (also may be referred to as a connection file or bootstrap file). It needs to include the following arguments:
	-install -user DBUSER -password DBPASSWORD -host "CONNECTSTRING "-driver DRIVER [-bootfile FILENAME] [-dataspace DATA_TABLESPACE] [-indexspace INDEX_TABLESPACE]
	Note that the host value must be enclosed in double quotes. Use of single quotes or no quotes will fail.
	When creating a connection file you can optionally include the bootfile parameter (-bootfile). Include this parameter if you want to name the connection file differently than the default (MATRIX-R).
	You can also optionally include the data and index tablespace parameters (-dataspace and -indexspace, respectively). These tablespaces will be the default for all vaults and, more importantly, the data and index tablespaces used for the non-vault tables; that is, the MX tables that are created when the V6 database is initialized.
	On Windows, the -driver value must not be put in single quotes as this leads to a crash. Using no quotes or double quotes works fine. For example, this is OK:
	`C:\ENOVIA_HOME\win_b64\code\bin>mql -install -user DBUSER -password DBPASSWORD -host "CONNECTSTRING" -driver MSSQL/ODBC`
	Whereas, this will lead to a crash:
	`C:\ENOVIA_HOME\win_b64\code\bin>mql -install -user DBUSER -password DBPASSWORD -host "CONNECTSTRING" -driver 'MSSQL/ODBC'`

-k Option	An abort-on-error is the default when MQL commands are read from a script or file. An abort happens when you encounter an error and you are returned to the operating system. However, you may not want a script to terminate when an error is found during processing. The -k option specifies that the 3DEXPERIENCE Platform continues on to the next MQL command if an error is detected in an MQL command. In this case, an error message is printed when the error is encountered and the script continues to run.

-t Option	This option suppresses the printing of the opening MQL title. This title identifies the product name, copyright information, and version number. When you use MQL frequently, the -t option saves time since the information is not displayed. This also suppresses the display of the MQL window. See also -d Option.
-v Option	This option specifies verbose mode. In this mode, the 3DEXPERIENCE Platform prints all messages generated during startup and the processing of commands. If you do not need this level of detail, you still receive error messages and other important MQL messages without the -v option.

Where:

- FILENAME is the name of the file to be created.
- DBUSER is the name used to connect to the database. This is the database user that is created to hold the 3DEXPERIENCE Platform database. It is established by the database administrator.
- DBPASSWORD is the security password associated with the Username. This is established by the database administrator. The user password is encrypted as well as encoded.
- DRIVER is one of:
  - Oracle/OCI80
  - DB2/CLI
  - MSSQL/ODBC

depending on which database you use.

Example

For example, to run an mql script with error files you may add something similar to the following to the MQL target in a Windows shortcut property:

```
c:\enoviaV6R2011\studio\BIN\winnt\mql -k -t "-stderr:d:\path with
spaces\a.err" "d:\path with spaces\mql_script.mql"
```

Another example:

```
-install -bootfile matrix-NEW -user Scott -password Tiger -driver
Oracle/OCI80
```

page Command

Description

A *page* is a type of administrative object used to create and manage properties used by apps, or Java applications. For app properties, all properties load the first time the API is called. At this call, the properties page loads from the classpath (SERVERHOME/managed/properties directory), then loads the properties page installed by any other app that may have altered some property values (such as accelerators), then loads a page object with the exact name as the properties file. That page object only contains properties that the customer specifically changed.

For applets or Java apps, Live Collaboration first looks for a property file using the classpath, but if the file is not found, it looks for a page object of the same name.

You can use MQL commands to create, delete, copy, modify, print, and list to manage page objects.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]page NAME {CLAUSE};
```

- `NAME`is the name you assign to the page. Page names must be unique. The name you choose is the name that will be referenced to include this page within a Web page. For more information, see *Administrative Object Names*. For pages that store properties for apps, the name but be the same as the property file you are configuring.
- CLAUSEs provide additional information about the page.

Add Page

As a Business Administrator, you can create new page objects. A page object requires code that defines a page or a part of a page. The are two ways to specify the code for a page:

- Enter the code as value for the `content` clause.
- Write the code in another editor and save the file, then include the name of the file in the `file` command.

Use the `Add Page` command to define a new page:

```
add page NAME [ADD_ITEM {ADD_ITEM}];
```

- `NAME` is the name you assign to the page.
- `ADD_ITEM` is an `Add Page` clause that provides additional information about the page you are defining.

The Add Page clauses are:

```
content VALUE
description VALUE
file FILENAME
mime VALUE
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

All clauses are optional for defining a page, since the page can later be modified. But to be used in a Web page, you must include either content or a file. Each clause and the arguments they use are discussed in the sections that follow.

File Clause

This clause specifies a file that contains the code that constitutes the page. This is provided as an alternative to the *Content Clause*. With the File clause, you can type and save the code in any editor of your choice.

```
file FILENAME;
```

For example, you might create a file that contains the frameset that is used to display your Web pages. You could create a page object named InitialFrame to store this data.

```
add page InitialFrame
   file InitialFrame.jsp;
```

Content Clause

The Content clause of the Add Page command is used to add code that defines the Web page. Content can consist of embedded tags and text.

```
   content VALUE;
```

- VALUE can be any combination of code and text that is displayable in a Web browser, including HTML, XHTML, XML, JavaScript, tags, CSS, etc. If the code contains embedded double quotes, use single quotes to define the start and end of the content. For properties pages, use the same format as in the properties file you are configuring.

For example, you can define the following page, which might be included as a footer on every page within an application:

```
add page IncludeFooter
  content '<a href="http://www.XYZCorp.com">XYZ Corporation</a><br>
  Voice: (555) 123-4567<br>
  Fax: (555) 123-4568<br>
  <a href="mailto:support@XYZCorp.com">support@XYZCorp.com</a><br>
  <a href="mailto:sales@XYZCorp.com">sales@XYZCorp.com</a><br>'
```

As an alternative, you can write the code for the page in a separate file and use the *File Clause* to include the page content in the page definition.

Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, do not use other characters, such as those created with the ESC key, for ANY field, including business and administrative object names, description fields, program object code, or page object content.

Mime Clause

This clause associates a MIME (Multi-Purpose Internet Mail Extension) type with a page. It defines the content type of the file named in the *File Clause*.

```
mime VALUE;
```

- VALUE is the content type of the file. The format of value is a type and subtype separated by a slash. For example, text/plain or text/jsp.
- The major MIME types are application, audio, image, text, and video. There are a variety of formats that use the application type. For example, application/x-pdf refers to Adobe Acrobat Portable Document Format files. For information on specific MIME types (which are more appropriately called “media” types) refer the Internet Assigned Numbers Authority Website at http://www.isi.edu/in-notes/iana/assignments/media-types/. The IANA is the repository for assigned IP addresses, domain names, protocol numbers, and has also become the registry for a number of Web-related resources including media types.

History Clause

The `history` keyword adds a history record marked “custom” to the page that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Page

After a page is defined, you can clone the definition with the `Copy Page` command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy page SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the page definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the page that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Page

Use the `Modify Page` command to change the definition of an existing page object. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify page NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the page you want to modify.
- `MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Page clause, as listed in the following table.

Note that you need to specify only fields to be modified.

Modify Page Clause	Specifies that…
`name NEW_NAME`	The current page name changes to the new name entered.
`content VALUE`	The current page content is replaced with new content.
`description VALUE`	The current description value, if any, is changed to the value entered.
`icon FILENAME`	The image is changed to the new image in the file specified.
`file FILENAME`	The page file is changed to the new file specified.
`mime MIMETYPE`	The MIME type for the page is changed to the new value specified.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the page that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

As you can see, each modification clause is related to the clauses and arguments that define the page. For example, you would use the Name clause of the Modify Page command to change the name for the page file.

Delete Page

If a page is no longer required, you can delete it by using the `Delete Page` command:

```
delete page NAME;
```

- `NAME` is the name of the page to be deleted.
- Searches the list of defined pages. If the name is found, that page is deleted. If the name is not found, an error message is displayed.

For example, to delete the AddStateFrame page, enter the following command:

```
delete page "AddStateFrame";
```

After this command is processed, the page is deleted and you receive an MQL prompt for another command.

Usage Notes

For global database access, pages generally need to be provided in multiple languages. And with the wide use of cell phones and other hand-held devices in accessing Web pages, you may also need to support the page's display on a small LCD in wireless markup language (wml). When this is the case, you can use the following Studio Customization Toolkit call to open a page with a language and format argument, following the syntax:

```
open(BASE_PAGE_NAME, LANG, MIMETYPE)
```

For example:

```
open(login.jsp, fr, wml)
```

When evaluating this code, the system first looks for the file named `login_fr_wml.jsp`. If this page is not found, it then attempts to find `login_wml.jsp`. As a last resort, it searches for the page `login.jsp`. Of course, you could call `login_fr_wml.jsp` directly, but the addition of arguments gives you much more flexibility when writing the code.

In this case, you would first create `login.jsp`. Next, if you wanted to support wml, you would then create the wireless version of the page and name it `login_wml.jsp`. Then for each language you want to support, you would translate the text portions of the page(s) and save as `login_LANG.jsp` and `login_LANG_wml.jsp`. For example, to support multiple languages you might have pages with the following names in the database:

```
login.jsp
login_ch-tw.jsp
login_ch-gb.jsp
login_it.jsp
```

To then add support for wml for these languages, you might add the following pages:

```
login_wml.jsp
login_ch-tw_wml.jsp
login_ch-gb_wml.jsp
login_it_wml.jsp
```

Note that the base page does not have to be in English. Also, the LANG argument could be more than two characters, such as en-us, en-uk, or ch-tw.

password Command

Description

Before defining users, you should consider what your company's password policies are and set system-wide password settings to enforce them.

User Level

System Administrator

Syntax

The following command allows you to set or change system-wide password settings.

```
set password PASSWORD_ITEM {PASSWORD_ITEM};
```

- `PASSWORD_ITEM` is a Set Password clause that provides more information about password settings. You must include at least one clause, and you can include several. The Set Password clauses are:

`minsize NUMBER`
`maxsize NUMBER`
`lockout NUMBER_OF_TRIES`
`expires NUMBER_OF_DAYS`
`[!|not]allowusername`
`[!|not]`allowreuse
`[!|not] mixedalphanumeric`
`[!|not]minsize`
`[!|not]maxsize`
`[!|not]lockout`
`[!|not]expires`
`cipher CIPHER_NAME`

Only Business Administrators with Person access are allowed to set system-wide password settings.

Minsize Clause

This clause requires that all passwords be at least a certain number of characters. To remove a minimum size password setting, use the keywords `!minsize` or `notminsize`.

Defining a minimum password size of at least 1 ensures that users actually create a password when changing their password. If there is no minimum password size, a user could leave the new password boxes blank when changing passwords, resulting in the user having no password.

Maxsize Clause

This clause sets an upper limit on the number of characters a password can contain. To remove a maximum size password setting, use the keywords `!maxsize` or `notmaxsize`.

For example, to require that users' passwords are least 6 characters and not more than 15, use:

```
set password minsize 6 maxsize 15;
```

By default, passwords are limited to 8 significant characters, in which case a password of 12345678xxxx is the same as password 12345678. The number of significant characters can, however, be controlled using the Cipher Clause of the `set password` *command.*

Lockout Clause

The Lockout clause of the Set Password command prevents a user from logging in after s/he has entered an incorrect password *n* number of times during a session.

After being locked out, the user's person definition is changed to "inactive." The only way for the user to log in again is to contact the Business Administrator to have the setting changed.

In the event that all Business Administrators are locked out, it is possible to resort to the use of SQL to access the database.

To remove a lockout setting, use the keywords `!lockout` or `notlockout`.

For example, the following command allows the user three tries to provide the correct password:

```
set password lockout 3;
```

Expires Clause

This clause requires that users create a new password every *n* number of days. After the specified number of days has elapsed, the system requires users to create a new password in order to log in. To remove the setting, use the keywords `!expires` or `notexpires`.

For example, use the following command if you want users to provide a new password every month:

```
set password expires 30;
```

When you turn on password expiration, passwords that were created prior to version 8 will expire the next time users attempt to log in.

If an implementation has the need for wide-spread expiring passwords but also uses "secret agents" that perform work programmatically, you can remove the necessity for updating these kinds of programs for expiring passwords by making the user agent's password never expire.

Allowusername Clause

This clause allows users to create a password that is the same as their username. This is the default. To prevent users from having the same username and password, use the following:

```
set password notallowusername;
```

Allowreuse Clause

This clause allows users to enter the same password as their old password. This is the default. To prevent users from keeping the same password, use the following:

```
set password notallowreuse;
```

`Mixedalphanumeric Clause`

This clause requires that passwords contain at least one number and at least one letter. To remove the setting, use the keyword `!mixedalphanumeric` or `notmixedalphanumeric`.

Cipher Clause

This clause specifies the algorithm used to encrypt passwords.

```
set password cipher CIPHER_NAME;
```

- `CIPHER_NAME` is the cipher to be used. It must be one of the LDAP supported ciphers: `crypt`, `md5`, `sha`, `smd5`, `ssha`. The default is `crypt`, which uses only the first eight characters for encryption and comparison.

Setting a new cipher for password encryption does not affect existing passwords. That is, only passwords created or changed after the cipher is specified with the above command will be stored using the new encryption algorithm. To make use of the new cipher, existing users must change their password. Business Administrators can include the *Expires Clause* when setting the cipher to ensure that all users redefine their password. For example:

```
set password cipher ssha expires 1;
```

After the above command is issued, existing user passwords will expire in one day, forcing users to enter a new password. Newly defined passwords will be encrypted using the ssha cipher.

Business Administrators can determine which cipher is in use (as well as other system-wide settings) using the MQL `print password` command.

Refer to http://www.openldap.org/faq/data/cache/346.html and ftp://www.ietf.org/rfc/rfc2307.txt for more information on ciphers.

pathtype Command

Description

A path is a kind of relationship. It is a way for an object to point to another using a virtual path; elements in the path hold the physicalID and localIDs of items in the path. Paths can be attached to BOs or relationships (dynamic or non-dynamic) exclusively. Their path elements can point to anything that has a physicalid+logicalid+majorid+updatestamp, that is any BO or rel.

A path is a mutable object. Once created, you can modify its attribute values, as well as the ProxyStamp values and its path elements. However, you cannot mutate the length of the path recorded. That is, you cannot shorten a path nor lengthen it. To do this requires destruction and recreation of the given path.

Up to 100 paths are supported.

When defining a pathtype, you can associate a pathtype with a specific Type or Relationship to create a local pathtype. For more information, see local pathtype Command.

Deleting or modifying the BOs or rels pointed to by a path does not affect the path in any way. Correspondingly, one can get a path from a BO or rel even if the pointed objects do not exist in the database or are not visible to the current user.

User Level

System Administrator

Syntax

```
[add | delete | list | modify | print]pathtype NAME
[ADD_ITEM{Add_ITEM}];
```

- NAME is the name of the pathtype.

Add Pathtype

Use the Add Pathtype command to define a pathtype.

```
add pathtype NAME [ADD_ITEM{Add_ITEM}];
```

ADD_ITEM provides more information about the pathtype that you are adding. Where ADD_ITEM is:

```
| attribute NAME {,NAME} |
| description VALUE |
| [!|not]hidden |
```

```
| from ADD_SUB_ITEM {,ADD_SUB_ITEM}                  |
| from cardinality | 1    |                          |
                   | one  |
                   | many |
                   | n    |
| to ADD_SUB_ITEM {,ADD_SUB_ITEM}                    |
| property NAME [value STRING]                       |
| property NAME [to ADMINTYPE NAME] [value STRING] |
```

Where ADD_SUB_ITEM is:

```
| type | mod TYPE_NAME {,TYPE_NAME} ||
```

```
       | all |
|relationship |TYPE_NAME [,TYPE_NAME] |
       | all|
```

Where ADMIN is:

```
| TYPE NAME |
```

You can define a pathtype for use in both business objects and relationships.

Delete Pathtype

The Delete Pathtype command is:

```
delete pathtype NAME;
```

List Pathtype

The List Pathtype command is:

```
list pathtype [modified after DATE] NAME_PATTERN [SELECT [DUMP
[RECORDSEP]]] [tcl] [output FILENAME];
```

Where RECORDSEP is:

```
recordseparator [SEPARATOR_STRING]
```

Modify Pathtype

After a pathtype is defined, you can change the definition with the Modify Pathtype command.

```
modify pathtype NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the pathtype to modify.
- MOD_ITEM is the type of modification to make. Each is specified in a Modify Pathtype clause, as listed in the following table.

Where MOD_ITEM is:

```
name NAME
```

```
description VALUE
```

```
[!|not]hidden
```

```
add | attribute NAME | property NAME to ADMIN [value STRING] | property NAME [value STRING]
```

```
remove  | attribute NAME | property NAME to ADMIN  | property NAME |
```

```
property NAME [to ADMINTYPE NAME] [value STRING]
```

```
property NAME [value STRING]
```

Print Pathtype

You can use the Print Pathtype to print pathtypes:

```
print pathtype NAME [SELECT][DUMP][tcl][output FILENAME];
```

Where SELECT is:

```
select FIELD_NAME {FIELD_NAME}
```

Where FIELD_NAME is:

```
SUB_FIELD[.SUB_FIELD{.SUB_FIELD}]
```

Where SUB_FIELD is:

```
string | string [string]
```

Where DUMP is:

```
dump [SEPARATOR_STRING]
```

Where RECORDSEP is:

```
recordseparator [SEPARATOR_STRING]
```

The support of adding properties to pathtypes or otehr admin objects is mandatory because we might want to attach UUID values to these admin objects.

Path Commands for PathTypes

You can also manipulate instance data, that is actual paths, but not pathtypes. The following commands are the Path commands:

```
add path PATHTYPE owner OBJ_NAME  [order N] ADD_ITEM {ADD_ITEM};
```

Where OBJ_NAME is:

```
businessobject ID | businessobject TYPE NAME REV | connection ID
```

Where ADD_ITEM is:

```
element [kind businessobject|connection] type TYPE_NAME physicalid
ID logicalid ID majorid ID proxystamp STAMP_VALUE  relevant
BOOLEAN_VALUE
```

The TYPE_NAME is resolved to determine if it is a businesstype, relationshiptype or dynamic relationshiptype. The following cases are errors:

- If the TYPE_NAME is not resolved.
- If the TYPE_NAME resolves to both a businesstype and a relationshiptype.

```
delete path PATH;
```

Where PATH is:

```
OBJ_NAME pathtype PATHTYPE order N
```

```
modify path PATH [MOD_ITEM {MOD_ITEM}];
```

Where MOD_ITEM is:

```
element N [kind businessobject|connection] [type TYPE_NAME]
[physicalid ID] [majorid ID] [logicalid ID] [proxystamp
STAMP_VALUE] | ATTRIBUTE_NAME VALUE
```

ATTRIBUTE_NAME and VALUE must be last.

```
print path PATH;
```

```
print path PATH [SELECT] [DUMP] [tcl] [output FILENAME];
```

```
print path selectable;
```

```
query path [type PATTERN] [vault PATTERN] [starting with SUBPATH |
ending with SUBPATH | containing SUBPATH] [where EXPR] [orderby
FIELD_NAME] [limit N] [SELECT] [DUMP] [RECORDSEP] [tcl] [output
FILENAME]
```

person Command

Description

A *person* is someone who uses *any* app, not only Matrix Navigator, Business Modeler, and MQL, but also all 3DEXPERIENCE apps, as well as custom applications. The system uses the persons you define to control access, notification, ownership, licensing, and history.

Do not modify or delete the persons creator, guest, or Test Everything using MQL. Modifying or deleting these objects could cause triggers, programs or other application functions not to work.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]person NAME [CLAUSEs];
```

- NAME is the name of the person you are creating. All persons must have a unique person name assigned. This name cannot be shared with any other user types (groups, roles, persons, associations). This name will appear in all windows where persons are listed. You could use the system login, if available. In this way you can link the user's context and vault which are associated with the system login. Then, when the person starts a session, s/he will automatically begin in his/her context with the specified vault. For additional information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the attribute.

Add Person

Define users with the Add Person command:

```
add person NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides more information about the person you are defining. While clauses are not required to make the person usable, they define the person's relationship to existing groups and roles and provide useful information about the person.

The Add Person clauses are:

Reserve controls the modification of businessobjects or connects to change their reserved status.

access ACCESS_MASK {,ACCESS_MASK}
admin ADMIN_ACCESS_MASK {,ADMIN_ACCESS_MASK}
address VALUE
assign [group GROUP_NAME] [role ROLE_NAME]

comment VALUE
email VALUE
enable email
disable email
[!|not]hidden
enable iconmail
disable iconmail
VALUE
vault VAULT_NAME
site SITE_NAME
password VALUE
no password
disable password
[!|not]passwordexpired
[!|not]neverexpire
phone VALUE
type TYPE_ITEM {, TYPE_ITEM}
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING

Each clause and the arguments they use are discussed in the sections that follow.

Access Clause

This clause specifies the maximum amount of access a person will be allowed. When this clause is used, you can select from many different forms of access. Each access is used to control some aspect of a business object's use. With each access, other than `all` and `none`, you can either *grant* the access or explicitly *deny* the access. You deny an access by entering `not` (or `!`) in front of the access in the command.

Business objects are governed by a policy. The policy defines who has access and when. Depending on the business object's state and the person involved, a user may have full, limited, or no access. Generally, the policy that governs the object restricts the access by role or group name. In some cases, that may mean that more access is granted than you might desire for a particular person.

If you want to prevent a person from ever having a form of access, you may do so by denying that access in the person's definition. For example, assume you have a user who continually overrides the signature requirements for business objects. This can be done with an access clause such as:

```
add person George
   access all, notoverride;
```

Even if George is granted override access via a policy, the access will be denied.

For more information on user access, see *Controlling Access* in Chapter 3. For more information on how access is controlled, see *Policies* in Chapter 4.

Access can be assigned or denied using one of two methods:

- **all**—The person has access to *all* functions except those listed.
- **none—**The person has access to *only* the functions listed.

The method you use will depend on whether most access will be permitted or denied. The default is None.

Using All:

```
access all, notchangeowner, notcheckout, notdisconnect, notdelete,
   notdemote, notdisable, notenable, notoverride, notschedule
```

Using None:

```
access none, checkin, connect, create, promote, lock, unlock, modify, read
```

If the amount of access being permitted or denied is about the same, the method does not matter. However, when the amount of access being permitted or denied is heavily weighted toward one or the other, the method you choose can save you time (and typing!).

If the access clause is not included when defining a user "none" is assumed.

A note about program access

In native apps (MQL or Matrix), you cannot see the code in a hidden program unless you have "admin program" access. However, in a server environment this restriction does not apply .The reasoning is that you must be able to restrict end-users of rich clients from seeing "secret" programs, including those in which passwords are hidden. But in a server environment:

- app-level code needs to be able to use such secret information and;
- ordinary end-user are not supposed to have such direct access to MQL commands and so it should remain secure.

Admin Clause

Users defined as Business and System Administrators can be assigned access to the definitions for which they are responsible. For example, one Business Administrator may be responsible for adding and modifying users; another for the business definitions of attributes, types, policies, etc.; and a third for the programs, wizards, and forms. All administrators will be able to view all definitions, but create and modify access is controlled by the settings in the Administration Access of the person.

The Admin Clause of the Add Person command specifies the access that Business and System Administrators will have to business objects. Any of the following accesses can be specified.

Administration Accesses			
association	attribute	form	format
group	inquiry	location	menu
person	policy	portal	program
property	relationship	role	rule
site	store	table	type
vault	wizard		
You can type not or ! at the beginning of an access to explicitly deny the access; for example, !policy or notserver.			

Administrative access can be assigned or denied using all or none in the same manner as when User access is assigned. For example:

```
add person George type business
   access all, notoverride admin type attribute
   policy;
```

If a person's type is business or system, and the Admin clause is not included all is assumed.

Address Clause

This clause specifies an address for the person. This address could be a mail stop, a street address, or an electronic address. Although this clause is not required, it is helpful to reach the person. An address is limited to 255 characters.

For example, you could assign an address to a user named Wolfgang using either of these commands:

```
add person wolfgang
   address "43 Hill Brook Ave, Shelton CT 06484";
Or:
add person wolfgang
   address "Mail Stop ES3-A5";
```

Although each example shows a different address, you can include only one Address clause in a person's definition. Remember, though, that the value can be any length. You *could* include all associated addresses in the value of the Address clause. If you do this, remember to place the most important addresses first and to keep it to a minimum to improve readability.

Assign Clause

This clause associates the person you are defining with one or more roles or groups. Although it is not necessary to assign a person to a group or role, it is commonly done to provide easy access to business objects and access privileges.

The Assign clause uses the following syntax:

```
assign [group GROUP_NAME] [role ROLE_NAME]
```

- GROUP_NAME is the name of a previously defined group.
- ROLE_NAME is the name of a previously defined role.
- If either name is not found, an error message is displayed.

When using the Assign clause, you have the option of assigning only a group, a role, or a combination of a role within a group. You can *not* include more than one group or role within a single Assign clause. You can, however, use multiple Assign clauses. For example, each of the following commands include a valid Assign clause:

```
add person myung
   assign group "Trade Show Support";
```

```
add person jenine
   assign role "Product Demonstrator";
```

```
add person jamar
   assign group "Trade Show Support" role "Trade Show
Coordinator";
```

Assigning a Role

To assign a role, use the Assign Role variation of the Assign clause. The key to determining whether a person should be assigned to a role is the job s/he performs. The role identifies the person's need to access particular business objects, the amount of access required to do the job, and when the access is needed.

Access to business objects and the files they contain is governed by a policy. The policy can define the groups, roles, and persons that do or do not have access to the business objects (see *Policies* in Chapter 4). The access and amount of access for these classifications can change at various stages of the project life cycle. In many cases, it is easier to specify the roles or groups that should have access in a policy rather than list individual users. If personnel changes during some stage of the project, you do not have to edit every policy to change person names.

To assign a person to a role in the Add Person command, you use the Assign Role clause:

```
assign role ROLE_NAME
```

ROLE_NAME is the name of a previously defined role. If that role was not previously defined, an error message is displayed.

You can assign a user to a role in two ways, depending on how you are building your database:

- In the person definition, as described here.
- With the Assign Person clause in the Add Role command described in *role Command*.

If you choose to define the persons first, you can assign them to the role later. If you choose to define persons last, you can make the role assignment in the Add Person command. Regardless of where you define it, the link between the role and person will appear when you later view either the role definition or the person definition.

Assume you are adding a person named Antonio Pelani with the following command:

```
add person "antonio pelani"
   comment "Directs the allocation and assignment of ER technical staff"
   assign role "ER Doctor"
   assign role "Lead ER Doctor";
```

In this definition, the defined person is assigned two roles. While the roles are similar, there are distinctions between the Lead ER Doctor and other doctors. Therefore both role assignments are made. Remember that a single person may play many roles in a project. If the role is associated with a policy, it is easier to move a person from one role to another than to edit the policy for each person's name.

Assigning a Group

Persons can be assigned to groups by using the Assign Group variation of the Assign clause. As with assigning roles to a person, assigning a group is not required. However, assigning a group to a person is often the simplest way to assign access privileges to a new user.

A group identifies a set of users who should share access to selected business objects. In an engineering environment, it might be everyone who is working on a particular project. They would need to have access to drawings and documentation at different stages of the project in order to perform their jobs.

You can specify which groups have access to business objects and when they have access by using the group name in the policy definition.

As stated in the previous section, access to business objects is governed by a policy. When a group is listed as a valid user in the policy, every person associated with that group has access to all business objects governed by the policy. If you know that a person will work with a group of people in the same function or project, it is easier to assign the person to the group than to edit the policy to add the person's name as a valid user.

To assign a person to a group in the Add Person command, you use the Assign Group clause:

```
assign group GROUP_NAME
```

`GROUP_NAME` is the name of a previously defined group. If this name was not previously defined, an error message is displayed.

You can assign a user to a group in two ways, depending on how you are building your database:

- In the person definition, as described here.
- With the Assign Person clause in the Add Group command described in *group Command*.

If you choose to define the persons first, you can assign them to the group later. If you choose to define persons last, you can make the group assignment in the Add Person command. Regardless of where you define it, the link between the group and person will appear when you later view either the group definition or the person definition.

Assume you entered the following Add Person command:

```
add person sheila
   comment "Assesses Training Needs and Implements Classes"
   assign role "Training Coordinator"
   assign group "Corporate Training"
   assign group "Customer Service";
```

In this definition, the person is associated with two groups and one role. The person needs access to the customer service objects in order to work with customers who require training. The person also

needs access to the business objects used by the training group. Each of these categories implies different access capabilities.

Since the Training Coordinator role is related to the Corporate Training group, this definition could be simplified by assigning the role with the related group. The following example rewrites the definition using the role and group combination:

```
add person sheila
   comment "Accesses Training Needs and Implements Classes"
   assign group "Corporate Training" role "Training
Coordinator"
   assign group "Customer Service";
```

Remember to determine which objects a person will need access to and when that access is required. The answer to those questions can guide you when assigning groups and roles to a person.

When a person is assigned a role within a group and then the assignments are printed, the output indicates this. For example:

```
print group "Corporate Training" select assignment;
group Corporate Training
   assignment = Corporate Training rob
   assignment = Corporate Training sheila Training Coordinator
```

Comment Clause

This clause provides general information about the person's function and the required privileges. You can have only one Comment clause in any person definition.

There is no limit to the number of characters you can include in the comment. However, keep in mind that the comment is displayed when the mouse pointer stops over the person in the User chooser. Although you are not required to provide a comment, this information is helpful when a choice is requested.

There may be subtle differences in the access privileges required by different users. You can use the Comment clause as a reminder of why this person is defined a certain way. If a person is assigned the wrong group or role, s/he may not be able to fully access the types of business object at the proper times in the object life cycle or may have inappropriate access to an object. Therefore, it is important to completely distinguish all persons.

For example you could have two quality control persons. One person performs testing of component assemblies and the other performs testing of the final machine. While they may belong to the same group (Quality Control) and there are similarities in their roles (Quality Testing), they are unique persons.

To distinguish between persons, you should include any descriptive comments meaningful to you to determine the person to contact when services are required.

For example, in the commands that follow, you can clearly identify the person you would call if you were interested in upgrading your personal computer.

```
add person sandy
   comment "Provides PCs sales support";
add person amed
   comment "Provides Apple product sales support";
add person miquel
   comment "Provides workstations and mainframes sales
support";
```

As you can see, each person provides sales support. However, the types of products they serve are different. These clauses enable another user to distinguish the persons. The clauses enable you to determine if a person needs access to business objects.

Mail Options

When mail is sent, the sender does not know how it is being delivered. The recipient's person definition establishes how it is received—as IconMail, email, both IconMail and email, or neither IconMail nor email.

Email Clause

This clause specifies an external electronic mail address. This is an address that is used by the user's non-3DEXPERIENCE Platform electronic mail utility to connect him/her to other users in the system. Since this address is highly dependent upon the external mail utility, there can be a wide variation in the style and content of the email address. This clause is required if email is enabled (as described below). For more information, see *Administrative Object Names*.

The following is an example of an email clause:

```
add person harriet
email comments@harriet.com;
```

This email address does not affect the internal mail system (IconMail). The internal mail utility uses the person's defined name, role, or group as the address.

Enable Email (Distribution of Mail) Clause

This clause enables an external electronic mail address for email. The email address specified in the person definition (using the email clause described above) delivers mail outside of 3DEXPERIENCE Platform. This address is used by the system (not the 3DEXPERIENCE Platform mail utility, IconMail) to connect the user to other users in the system.

You can specify that outgoing messages are sent to email or IconMail (as described for the *Enable Iconmail Clause*) or both.

The following is an example of an `enable email` clause:

```
add person harriet
email comments@harriet.com;
enable email
```

Since this address is dependent upon the system's mail utility, there can be a wide variation in the style and content of the email address. This address does not affect the internal mail system (IconMail). IconMail uses the person's defined name, role, or group as the address.

When a Person is cloned or created with email enabled but no email address has been specified, the following warning is received:

```
Warning: #1900296: Person 'NAME' has email enabled, but no email
address. Use of a fully qualified email address is recommended.
```

The warning is also shown when a person's email setting is enabled during modification. If an existing Person has email enabled with no address specified, no warning is given if only other settings are modified. Warnings are also not given when Person's are created during import.

Disable Email Clause

This clause disables an external electronic mail address for email. The email address specified in the person definition (using the Email clause described above) does not deliver mail outside of Live Collaboration.

Enable Iconmail Clause

IconMail is a legacy internal communication interface and should no longer be used. It is highly recommended that external mail utility be used and that all users are defined with an email non-3DEXPERIENCE Platform email address.

This clause specifies that outgoing messages are sent to IconMail. You can send messages by using email, IconMail or both.

The following is an example of an Enable Iconmail clause:

```
add person harriet
 email comments@harriet.com
 enable iconmail;
```

IconMail uses the person’s defined name, role, or group as the address.

Disable Iconmail Clause

This clause disables IconMail so that messages are not received via IconMail.

Fax Clause

This clause associates a fax number with the person. You can include additional information with the fax number. The Fax clause is limited to 64 characters. For example, the following are valid uses of the Fax clause:

```
add person shandrika
   fax "(203) 987-5584";
```

```
add person "A-1 Consulting Agency"
   fax "(617) 535-9857 please call before faxing";
```

Fullname Clause

This clause specifies the full name of the person you are defining. This name could be the name of the person as it appears in personnel records or the person’s signature name. Often when defining a person, the name is abbreviated or shortened. You can include the person’s full name as well with the Fullname clause. For more information, see *Administrative Object Names*.

All persons must have a named assigned. When you create a person, you should assign a name that has meaning to both you and the user. If the number of users is small, you may want to use only the first or last name as the person name. If there are larger numbers of users, you may want to use the full name or a name and initial to help distinguish users. For example, each of the following is valid person name:

```
Pat M.

Patty

Patricia L. Melrose

P. Melrose

Pat the Manager
```

Password Options

The use of passwords is similar to the way that passwords work on most systems. Passwords are an effective means of preventing unauthorized access to business objects. Without passwords, any user could set their context to that of any other user. This would essentially make policies and states meaningless since you could easily bypass access restrictions by pretending to be someone else. Therefore, a password should be required in order to ensure that the person who is logging in is indeed that person. In particular you should thoroughly consider the option of passwords for Business and Systems Administrators.

There are three clauses that use passwords to restrict access to a person's context:

- Password clause assigns a password to the person you are defining (as described below).
- Disable Password clause restricts access to the user whose system ID or login matches the person ID (as described in *Disable Password Clause*).
- No Password clause specifies that no password is required to set context. (as described in *No Password Clause*).

Only one password-related clause (Password, Disable Password, or No Password) should be given.

```
add person dave
   no password
   access checkin, create, delete, read, modify, checkout,
connect;
```

Each user can redefine his/her own password by using the Preferences (password) option in Matrix Navigator.

Password Clause

The Password clause assigns a password to the person you are defining. The password will be required whenever someone wants to access this person's business objects. When a password is assigned, anyone who knows the password can set their context to this person.

For example, the following command defines a person with a password value:

```
add person chris
   password SaturnV;
```

If you wanted to set your context to Chris, you would need to know the password SaturnV.

The defined password is not visible to anyone. As a Business Administrator, you can change a password without seeing or knowing it. For more information, see *Administrative Object Names*.

By default, passwords are limited to 8 significant characters, in which case a password of 12345678xxxx is the same as password 12345678. The number of significant characters can, however, be controlled using the Cipher Clause of the `set password` *command.*

Disable Password Clause

The Disable Password clause lets you use the security for logging into the operating system as the security for setting context. When a user whose password is disabled attempts to set context, the system compares the user name used to log into the operating system with the list of persons defined. If there is a match, the user can set context without a password. (The context dialog puts the system user name in as default, so the user can just hit enter.) If they do not match, the system denies access.

When Disable Password is chosen for an existing person, this clause modifies the password so that others cannot access the account. This means that the user with the disabled password can only log into the 3DEXPERIENCE Platform from a system where the O/S ID matches the 3DEXPERIENCE Platform ID. This is similar to the way automatic SSO-based user creation is handled. To re-enable a password for such a person, create a new password for the person as you normally would.

No Password Clause

The No Password clause specifies that no password is required to access the person's business objects. When No Password is specified, other users can set their context to that of the person you are defining. Once the context is set to the person, the user can act as that person with all applicable access privileges.

The No Password clause generally is not recommended since removing the password requirement permits other users to set context as that user. However, if a person leaves the company, for example, or if you are setting up a guest account, you may want to remove the password requirement.

Passwordexpired Clause

When the Passwordexpired clause is included in a person's definition, the system requires the person to define a new password the next time s/he attempts to log in. The system will not allow the person to log in without entering a new password. After the user defines a new password, the clause is removed from the person definition.

For example, you could use the following command to ensure that Jordan defines a new password on next login:

```
add person jordan passwordexpired;
```

This option allows you to require users to establish a password without requiring you to assign them one now. Having users establish their own passwords helps them remember their password. It also prevents you from having to enter an initial password for every new user, which you would then have to communicate to users. You can also use this clause if you suspect that a person's password has been compromised.

Neverexpire Clause

You can allow a user to be exempt from password expiration with the `neverexpire` clause. This may be used when the system is setup to use expiring passwords, but the implementation has the need for “secret agents” or automatons that generally perform work programmatically. To remove the necessity for updating these kinds of programs for expiring passwords, the user agent’s password can be set to `neverexpire` as follows:

```
add person “User Agent” neverexpire;
```

This command unsets the passwordexpired option on the person if it was set. If a business administrator attempts to set the passwordexpired option on a person that is set to `neverexpire`, an error will occur.

Phone Clause

The Phone clause of the Add Person command associates a telephone number with the person you are defining. This number could be an internal extension number, business phone number, car phone number, or home phone number. The Phone clause is restricted to 64 characters.

For example, you could assign a phone number as follows:

```
add person andrea
   phone “business: x433, home: (203) 987-6543”;
```

Notice that you can include additional information about the number to guide those who might reference it.

Site Clause

A site is a set of locations. It is used in a distributed environment to indicate the file store locations that are closest to the group. The Site clause specifies a default site for the person you are defining. Consult your System Administrator for more information.

To write a Site clause, you use the name of an existing site. If you are unsure of the site name or want to see a listing of the available sites, use the MQL List Site command. This command produces a list of available sites from which you can select. (Refer to *Locations and Sites* in Chapter 2.)

Sites can be set on persons, groups and roles, as well as on the Live Collaboration Server (with MX_SITE_PREFERENCE). The system looks for and uses the settings in the following order:

- if using a Collaboration Server, the MX_SITE_PREFERENCE is used.
- if not using a Collaboration Server, or the MX_SITE_PREFERENCE is not set on the server, if there is a site associated with the person directly, it is used.
- if no site is found on the person, it looks at all groups to which the user belongs. If any of those groups have a site associated with it, the first one found is used.
- if no sites are found on the person or their groups, it looks at all roles the person is assigned. If any of those roles have a site associated with it, the first one found is used.

Add the `MX_SITE_PREFERENCE` *variable to the Live Collaboration Server's initialization file (enovia.ini). This adjustment overrides the setting in the person, group, or role definition for the site preference, and should be set to the site that is local to the server. This ensures optimum performance of file checkin and checkout for Web clients.*

Type Clause

The Type clause specifies the type of user you are defining. There are four basic types of users. Each type has a different level of access:

`Application User`	A licensed user that can access the database only through the Server or MQL. All other active users have Application User privileges automatically.
`Full User`	A licensed user with normal user access.
`Business Administrator`	A licensed user with access to the Business Administrator functions. When a person is defined as a Business Administrator, they may have the privilege of being able to set context to any defined person without a password depending on the system setting for `privilegedbusinessadmin`. Refer to *Privileged Business Administrators* in Chapter 9 for more information.
`System Administrator`	A licensed user with access to the System Administrator functions.
`Inactive`	A defined user who does not currently have access.
`Trusted`	A licensed user for whom read access is not checked.

Business and System Administrators are also assigned as Full Users.

All persons have a type. If the type is not explicitly assigned, the person will be automatically assigned to the types `application` and `full`. To assign a type, write a Type clause using the following syntax:

```
type TYPE_ITEM {, TYPE_ITEM}
```

- `TYPE_ITEM` assigns or denies the privileges associated with each of the five user types. There are twelve `TYPE_ITEM`s. Six of the items assign the user type and six remove a user type assignment:

TYPE_ITEMs	The user has…
`application`	Access only through the Server or MQL
`notapplication`	No access
`full`	Normal access
`notfull`	No normal access
`business`	Access to the Business Administrator functions
`notbusiness`	No access to Business Administrator functions
`system`	Access to the Business and System Administrator functions
`notsystem`	No access to the System Administrator functions
`notinactive`	Current access
`inactive`	No current access
`trusted`	Read access that is not checked
`nottrusted`	Read access that is checked

When you define a person, a type of application User and Full User is automatically assigned to the person. This means that the user is defined as:

```
type full, application
```

If you want a type that is different from this, you need to write a Type clause. For example, the following command defines a person who is both a Full User and a Business Administrator:

```
add person rodolph
   type full, business;
```

If you want the person's type to be something other than Full and Application, be sure to use the 'not' prefix on any types defined by default. For example, to define a consultant named Feng that is a full user but not and application user, use the following:

```
add person Feng
   type notapplication;
```

If a person is not allowed access, why have that person defined within the database? One answer has to do with business object creation and ownership. Let's assume that you have an employee who worked for some time within the system. After that person left the company, you still have many business objects that were created and controlled by the person. Rather than change object ownership, you may want to maintain the old ownership so that you have a record of the original creator. By making the person *inactive*, you remove that person's ability to access the system while maintaining the status of all business objects the person created. Although access for other users remains as defined, if an owner is inactive the Business Administrator may want to reassign ownership to another person.

Vault Clause

The vault clause specifies a default vault for the person you are defining. It is similar to setting a default directory in your operating system. When a person starts the 3DEXPERIENCE Platform, the context dialog fills in this vault as the default. The vault should be the one most commonly used by the person. Although the person can change to a different vault once the 3DEXPERIENCE Platform starts, it is helpful to set the vault that the person starts in.

To write a Vault clause, you use the name of an existing vault. If you are unsure of the vault name or want to see a listing of the available vaults, use the MQL List Vault command. This command produces a list of available vaults from which you can select.

To reduce network loads, vaults are often created locally, serving selected groups of users. When you are assigning the vault, you should determine if there is a vault local to the person you are defining. If there is more than one local vault, determine which is better suited for the person based on the type of work or objects used.

For example, the following person definition assigns the engineer to a vault related to the types of projects she will work on:

```
add person mcgivern
   fullname "Jenna C. McGivern"
   assign role Engineer
   assign group "Building Construction"
   vault "High Rise Apartments";
```

Assignment selectable on Person

The select keyword “assignment” provides the selectables for groups and roles from the person. For example you might have the following boolean expression filter:

```
organization match context.user.assignment.parent
```

This evaluates to true for data allowed for a user’s Organization. See also *Add Business Object*.

History Clause

The `history` keyword adds a history record marked “custom” to the person that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Person

After a person is defined, you can clone the definition with the Copy Person command. This command lets you duplicate person definitions with the option to change the value of clause arguments:

```
copy person SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the person definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the person that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Person

After a person is defined, you can change the definition with the Modify Person command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify person NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the person to modify.
- `MOD_ITEM` is the type of modification to make. Each is specified in a Modify Person clause, as listed in the following table.

- Note that you need to specify only fields to be modified.

Modify Person Clause	Specifies that…
`access ACCESS {,ACCESS}`	The person is restricted to the listed access. Values for ACCESS can be found in the table in *Accesses* in Chapter 3.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`address VALUE`	The address to associate with the person is changed to the value entered. This address could be the person's street address.
`admin ADMIN_ACCESS {,ADMIN_ACCESS}`	The Business or System Administrator is restricted to the listed access. Values for ADMIN_ACCESS can be found in the *Admin Clause* section.
`application` APPLICATION_NAME	The person is assigned the named application. In case the person already has an owning application, the old application is replaced by the named application.
`assign [group GROUP_NAME] [role ROLE_NAME]`	The person is associated with the listed group or role.
`assign all`	The person is assigned to all groups and roles.
`comment VALUE`	The current comment, if any, is changed to the value entered.
`disable email`	Incoming messages are not sent to the e-mail address.
`disable iconmail`	Incoming messages are not sent to IconMail.
`disable password`	Access can be set to the person only from a system where the O/S user is the same as the 3DEXPERIENCE Platform user.
`email VALUE`	A valid electronic mail address is set for the person. This address must be in a form understood by the user's e-mail utility.
`enable email`	Incoming messages are sent to the e-mail address specified with the Email clause.
`enable iconmail`	Incoming messages are sent to IconMail.
`fax VALUE`	The person's fax number is changed or is set to the value entered.
`fullname VALUE`	The full name of the person is changed or is set to the value entered.
`hidden`	The hidden option is changed to specify that the object is hidden.
`icon FILENAME`	The image is changed tp the new image in the field specified.
`name VALUE`	The current person name is changed to the new name
`no password`	When setting context, there is no password required to access this person's context.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`password VALUE`	When setting context, the password is changed to the value entered.
`passwordexpired`	When setting context, a person must define a new password.
`neverexpire`	The person's password does not expire regardless of the password expires system setting.

Modify Person Clause	Specifies that…
`!neverexpire`	The neverexpire setting is removed and the person uses the system setting.
`phone VALUE`	The person's phone number is changed or is set to the value entered.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`remove assign all`	The person is removed from all groups and roles. Any links the person has with any groups or roles is dissolved.
`remove assign [group GROUP_NAME] [role ROLE_NAME]`	The person is removed from the specified group or role.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`type TYPE_ITEM {, TYPE_ITEM}`	The person is assigned or denied the privileges associated with the listed user types. Values for TYPE_ITEM can be found in the *Type Clause* section.
`vault vault_NAME`	The current vault is changed to the new vault.
`history STRING`	Adds a history record marked "custom" to the person that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

As you can see, each modification clause is related to the clauses and arguments that define the person. For example, assume you want to alter the address and phone number of a person who moved. You could do so by writing a Modify Person command similar to the following:

```
modify person roosevelt
  address "White House, Pennsylvania Ave, Washington"
  phone "Unlisted";
```

This command changes the address to that of the White House and designates the current phone number as "Unlisted."

When modifying a person:

- The roles or groups that you assign to the person must already be defined within the database. If they are not, an error will display when you try to assign them.
- Note how access privileges are shared between roles and groups. Although a person is restricted access because of the group assignment, the user may have access via his/her role assignment. To restrict a user that has access or increase a person's access, you will have to determine the best method for giving that access.

  Remember that altering the group or role access affects everyone associated with that group or role. If it is a singular case of special access, you may want to assign that person to the policy directly or define a role that is exclusively used by the person in question.

Delete Person

If a person leaves the company or changes jobs so that they no longer need to use the database, you can delete that person by using the Delete Person command:

```
delete person NAME;
```

NAME is the name of the person to be deleted.

Searches the list of defined persons. If the name is not found, an error message is displayed. If the name is found, the person is deleted only if there are no business objects that are owned by the person. If the name is still attached to any objects, the person cannot be deleted.

If the person has created an extensive number of objects or has been heavily involved in the history of many objects (such as a manager signing off), you may want to make the person inactive rather than delete them. By assigning an Inactive type to the person, you have a point of reference when that user name comes up in history records or elsewhere.

In addition, IconMail references to the deleted person cause the mail to be unreadable. This includes messages sent by the person, or cc'd to the person. If the person used IconMail extensively, it is better to make the person Inactive than to delete it.

When a person is deleted, any linkages to that person are dissolved. That means the person is automatically removed from any role or group that was included in the person's definition. The person is removed from any signatures in all policies, and if they were the only user referenced in the signature, it is removed as a requirement.

For example, if you wanted to delete the person named jones, you would enter the following MQL command:

```
delete person jones;
```

After this command is processed, the person is deleted and you receive an MQL prompt for another command.

policy Command

Description

A *policy* controls a business object. It specifies the rules that govern access, approvals, lifecycle, revisioning, and more. If there is any question as to what you can do with a business object, it is most likely answered by looking at the object's policy.

For conceptual information on this command, see *Policies* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify|print|delete] policy NAME [CLAUSEs];
```

- NAME is the name you assign to the policy. Policy names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the policy.

Add Policy

Policies are defined using the Add Policy command:

```
add policy NAME [ITEM {ITEM}];
```

- ITEM defines information such as the types of objects governed by the policy, the types of formats permitted by the policy, the labeling sequence for revisions, the storage location for files governed by the policy, and the states and conditions that make up an object's lifecycle.

The Add Policy clauses are:

```
description VALUE
```

```
type | TYPE_NAME {,TYPE_NAME} |
     | all                     |
```

```
format | FORMAT_NAME {,FORMAT_NAME} |
       | all                         |
```

```
defaultformat FORMAT_NAME
```

```
[not]enforce
```

```
minorsequence REVISION_SEQUENCE
```

```
majorsequence REVISION_SEQUENCE
```

```
delimiter DELIMITER
```

allstate [ALLSTATE_ITEM {,ALLSTATE_ITEM}]
state STATE_NAME [STATE_ITEM {,STATE_ITEM}]
store STORE_NAME
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING

Where ALLSTATE_ITEM is:

ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]

Where STATE_ITEM is:

action COMMAND
check COMMAND
icon FILENAME
ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]
notify | USER_NAME {,USER_NAME} message VALUE | | signer message VALUE |
promote | [true] | | false |
minorrevision | [true] | | false |
majorrevision | [true] | | false |
checkouthistory | [true] | | false |
published | [true] | | false |
route USER_NAME message VALUE
signature SIGN_NAME [SIGNATURE_ITEM {,SIGNATURE_ITEM}]
version | [true] | | false |

```
stateproperty NAME [to ADMIN] [value STRING]
TRIGGER PROG_NAME [input ARG_STRING]
```

Where SIGNATURE_ITEM is:

```
approve USER_NAME {,USER_NAME}
ignore USER_NAME {,USER_NAME}
reject USER_NAME {,USER_NAME}
branch STATE_NAME
filter EXPR
```

Where TRIGGER is:

```
trigger EVENT_TYPE | action   |
                   | check    |
                   | override |
```

Where EVENT_TYPE is:

```
approve
demote
disable
enable
ignore
override
promote
reject
schedule
unsign
```

Some of the clauses are required and some are optional, as described in the sections that follow.

Where ACCESS_USER is:

```
   | [revoke] [login] public [key STRING]          |
   | [revoke] [login] owner [key STRING]           |
   | [revoke] [login] user USER_NAME [key STRING]  |
```

Reserve controls the modification of businessobjects or connects to change their reserved status. Where ACCESS_ITEM is:

```
| all                  |
| none                 |
| [not]changename      |
| [not]changeowner     |
| [not]changepolicy    |
| [not]changetype      |
| [not]changevault     |
| [not]checkin         |
| [not]checkout        |
| [not]create          |
| [not]grant           |
| [not]delete          |
| [not]demote          |
| [not]disable         |
| [not]enable          |
| [not]execute         |
| [not]freeze          |
| [not]fromconnect     |
| [not]fromdisconnect  |
| [not]lock            |
| [not]majorrevise     |
| [not]modify          |
| [not]modifyform      |
| [not]override        |
| [not]promote         |
| [not]read            |
| [not]reserve         |
| [not]unreserve       |
| [not]revoke          |
| [not]revise          |
| [not]schedule        |
| [not]show            |
| [not]thaw            |
| [not]toconnect       |
| [not]toconnect       |
```

```
   | [not]unlock          |
   | [not]viewform        |
```

Where USER_ITEM is:

```
[any | single | ancestor | descendant] organization
[any | single | ancestor | descendant] project
[any | context] owner
[any | no | context | inclusive] reserve
[any | no | public | protected | private |notprivate|ppp] maturity
[any|oem|goldpartner|partner|supplier|customer|contractor]category
[filter | localfilter] EXPR
```

Type Clause

This clause defines all of the business object types governed by the policy. Just as a policy may govern many different object types, each object type may have many different policies that govern it. (However, an object instance can only have one policy associated at any time.)

For example, assume you have an object type named Drawing. This type may be governed by two policies named “Engineering Drawing Process” and “Documentation Drawing Process”. When an object of type Drawing is created, you must decide which policy will govern the object instance. A drawing meant for documentation will have a different review and lifecycle than a drawing of a component to an engineering assembly. By associating one policy with the created object, you control the types of files that can be checked in, who will use the object, and when it is used.

The Type clause is required for a policy to be usable:

```
type TYPE_NAME{,TYPE_NAME}
Or
type all
```

- `TYPE_NAME` is a previously defined object type.

You can list one type or many types (separated by a comma or carriage return). When specifying the name of an object type, it must be of a type that already exists in the database. If it is not, an error message will display.

For example, the following two commands are valid Add Policy commands that identify object types associated with the policy:

```
add policy "Engineering Revision Process"
   description "Quality Control Process for Engineering Revisions"
   type Drawing,"Multipage Drawings",Schematics,Manual;
add policy "Documentation Revision Process"
   description "Quality Control Process for Documentation Revisions"
   type Manual,"Release Notes";
```

In the first command, the user can associate the “Engineering Revision Process” policy with object instances of four different object types: Drawing, Multi-page Drawings, Schematics, and Manual. In the second command, the user can associate the “Documentation Revision Process” policy with only two object types: Manual and Release Notes. If the user created an object named “MQL User Guide” of type Manual, could he assign a policy of “Engineering Revision Process?” The answer is yes, although the “Documentation Revision Process” policy might be more appropriate.

A variation of the Type clause uses the keyword 'all' in place of one or more type names (Type All). When this keyword is used, all existing and future business object types defined within 3DEXPERIENCE Platform are allowed by the policy-the policy governs objects of all types. The 'all' keyword is more like an option that can be turned on for the policy rather than a list of types. If a list of types is already defined for the policy and the 'all' keyword is used, the system will clear the list and turn the 'type all' option on for the policy.

Use caution when including this clause. While all of the types currently defined may apply to the policy, what happens if a new one is created that should not apply?

If you have a policy that uses most of the defined objects, you may want to assign all of the types rather than list them. Then you can use the Modify Policy command to remove the unwanted types.

Format Clause

This clause defines the formats permitted under the policy for checked in files. Depending on the policy and the object created, certain files are appropriate or inappropriate. The Format clause restricts the types of files that can be associated with a business object.

The Format clause is required if you want to check in files under the policy:

```
format FORMAT_NAME{,FORMAT_NAME}
Or
format all
```

- `FORMAT_NAME` is the name of a previously defined format. If the format was not previously defined, an error message results.

For example, you could have a policy that governs photos. This policy would need formats for processing files that contain photographic images. In this case, there may be two file formats allowed: GIF and JPG. These formats specify the software commands required to view, edit, and print files of type Photo. You might write this policy definition as:

```
add policy Photo
   description "Photo Process"
   type Photo
   format GIF,JPG
   defaultformat GIF
   store Photo
   sequence "1,2,3,..."
   state base
     public all
     owner all;
```

Like the Type clause, the Format clause has a variation that uses the keyword 'all.' When this clause is used, all existing and future formats defined within the 3DEXPERIENCE Platform are allowed under this policy-the policy governs objects of all formats. The 'all' keyword is more like an option that can be turned on for the policy rather than a list of formats. If a list of formats is already

defined for the policy and the 'all' keyword is used, the system will clear the list and turn the 'format all' option on for the policy.

Use caution with the Format All clause. Are there any formats that you do not want to permit under this policy? If there are, you will need to either list all of the desired formats or edit the policy after assigning all formats.

Defaultformat Clause

This clause is required in order to check any files in unless the Checkin clause specifies the format. If only one format is specified in the Format clause, it is automatically the default.

The Defaultformat clause has the syntax:

```
default format FORMAT_NAME
```

- `FORMAT_NAME` can be any previously defined format that is listed in the Format clause of the Add Policy command. The Format clause identifies all formats permitted by the policy.

For example, the following Add Policy command defines the default format as a text file that uses BestWord to process it:

```
add policy "Proposals"
   description "Process for generating, reviewing, and releasing proposals"
   type Proposal, Plan
   format ASCII, "BestWord", Drawing
   defaultformat "BestWord";
```

If an object does not have any files checked into its default format, execution of a View, Edit, or Print command will check its other formats and open files found there.

Sequence Clause

This clause defines a scheme for labeling revisions. With this clause, you can specify the pattern to use when an existing object is revised. This pattern can include letters, numbers, or enumerated values. For example, you could have revisions labeled “1st Rev,” A, or 1.

To define a scheme for labeling revisions, you must build a revision sequence. This sequence specifies how objects should be labeled, the type of label to be used, and the number of revisions allowed. When you create a revision sequence, use the following syntax rules:

Rule	Example
Hyphens denote range.	A-Z signifies that all letters from A through Z inclusive are to be used.
Commas separate enumerated types.	Rev1,Rev2,Rev3,Rev4 is a sequence with four revision labels. Rev1 will be assigned before Rev2, which will be assigned before Rev3, and so on.
Square brackets are used for repeating alphabetic sequences.	[A-Z] signifies that the sequence will repeat after Z is reached. When it repeats, it returns to the front of the label list and doubles the labels so that the next sequence is AA, AB, AC, and so on.
Rounded brackets are used for repeating numeric sequences.	(0-9) signifies a regular counting sequence. (When 9 is reached it will repeat and add a 1 before the symbol).
A trailing ellipses (…) means a continuing sequence.	A,B,C,… signifies the same thing as A-Z. 0,1,2,… signifies the same thing as (0-9)

These rules offer flexibility in defining the revision labeling sequence. Although you cannot have two repeating sequences in a single definition, you can include combinations of enumerated values and ranges within a repeating sequence. For example, the following revision sequence definition specifies that the first object should be labeled with a hyphen and the first revision should be labeled I, the second II, the third III, etc. After the fifth revision, all revisions will have numeric sequencing.

```
-,I,II,III,IV,V,(0-9)
```

If your location requires a numeric value, for example, for pre-released revisions, and then an alphanumeric scheme after that, the approach should be to change the policy at the point when the revision scheme should change. A separate policy is created and applied to a new revision, providing different states, signatures, etc. as well as a different revision scheme. For example, if a revision sequence is defined as:

```
0,1,2,...,-,[A,B,C,D,E,F,G,H,J,K,L,M,N,P,R,T,U,V,W,Y]
```

the automatic sequencing will never get beyond the number counting, so the entries after that are ignored. Two policies should be established for the object type with revision sequences defined as follows:

```
0,1,2,...
```

and

```
-,[A,B,C,D,E,F,G,H,J,K,L,M,N,P,R,T,U,V,W,Y]
```

If you want to exclude certain letters (such as I, O, Q, S, X, and Z in above), you must indicate only those you want to include as above. Use of a sequence such as [A-H,J-N] skips the letter I when automatically entering the revision during object creation, but does not prevent manually entering it during object creation or object modification.

If you enter blank spaces within the definition of a revision sequence, the Live Collaboration uses the blank spaces literally. (In general, you should NOT use blank spaces within a revision sequence.) Consider the following examples:

Enter the revision sequence as:	3DEXPERIENCE Platform recognizes this as:
A, B, C	"A" " B" " C"
A,B,C,	"A" "B" "C"
1st Rev, 2nd Rev, 3rd Rev	"1st Rev" " 2nd Rev" " 3rd Rev"
1st Rev,2nd Rev,3rd Rev	"1st Rev" "2nd Rev" "3rd Rev"

After you define your revision sequence, simply insert it into the Sequence clause using the following syntax:.

```
sequence REVISION_SEQUENCE
```

REVISION_SEQUENCE must follow the syntax rules given above.

For example, the following policy definition uses an enumerated revision sequence:

```
add policy "Engineering Proposal Process"
   type Proposal
   format Text
   sequence Unrevised,1st Rev,2nd Rev,3rd Rev,4th Rev;
```

In this command, when a file named “Proposed Solar Vehicle” is checked into an object of type Proposal, it is named in the window as Proposal, “Proposed Solar Vehicle”, Unrevised. After the first revision, the object is given a revision label of “1st Rev” (in place of Unrevised). As it is further defined, Live Collaboration progresses through the enumerated sequence values until it reaches “4th Rev.” Since this sequence is non-repeating, no further revisions are allowed.

Enforce Clause

This clause is optional and can be used to prevent one user from overwriting changes to a file made by another user. When an object is governed by a policy that has enforce locking turned on, the only time a user can check in files *that replace existing files* is when:

- the object is locked

and

- the user performing the checkin is the locker

To ensure that the person who locked the object is the person who checked out the file, enforce locking disables the manual lock function (`lock businessobject OBJECTID;`). The only way to lock an object that is governed by a policy that enforces locking is by locking it when checking out the file (for example: `checkout bus OBJECTID lock;`).

When checking in files *that do not replace existing files* (for example, if you check files into a format that contains no files or you append files), as long as the object is unlocked, you can check in new files. When an object is locked, no files can be checked into the object until the lock is released, even if the file does not replace the checked-out file that initiated the lock. This means that attempts to open for editing, as well as checkin, will fail. Files can be checked out of a locked object and also opened for viewing.

Enforce locking ensures that when a user checks out a file and locks the object, signifying that the user intends to edit the file, no other user can check in a file and overwrite the files the original user is working on. When the original user checks the file back in, the user should unlock the object.

Be aware that the manual unlock command (`unlock businessobject OBJECTID;`) is available for users who have unlock access, but users should avoid using the command for objects that have enforce locking. For example, suppose Janet checks out a file and locks the object with the intention of editing the file and checking it back in. Steve, who has unlock access, decides he needs to check in an additional file for the object so he unlocks the object manually. When Janet attempts to check in her edited file, replacing the original with her updated file, the system won’t allow her to because the object isn’t locked. In order to check in the file, Janet has several options:

- she can check in the edited file in such a way that it won’t replace existing files; for example, change the name of the edited file and append it or delete the original file and check in the edited file
- she can check out the original file again and lock the object, taking care not to replace the edited file on her hard drive with the older file she is checking out, and then check in the edited file

For more information on unlock access, see *Accesses* in Chapter 3.

A user would end up in a situation similar to the one described above if the user forgets to lock the object when checking out a file for editing. When the user attempts to check in the edited file, the system won't allow the checkin because the object is unlocked (or possibly locked by another user who checked out the file after the first user).

For example, to enforce locking on the Proposals policy:

```
add policy "Proposals"
   description "Process for generating, reviewing, and releasing proposals"
   type Proposal, Plan
   format ASCII, "BestWord", Drawing
   defaultformat "BestWord"
   enforce;
```

The `not enforce` clause is available when modifying policies to turn the feature off.

State Clause

This clause defines all information related to a policy state including: who can access a business object, what type of access a user can have, whether new revisions are allowed, and the conditions for changing from one state to another. The State clause uses the following syntax:

```
state STATE_NAME [STATE_ITEM {,STATE_ITEM}]
```

- `STATE_NAME` is the name of the state you are defining. All states must have a named assigned. This name must be unique within the policy and should have meaning for both you and the user. For additional information, refer to *Administrative Object Names*.

For example, assume you have a process for performing and evaluating lab tests. The first state might involve receiving the initial test request, gathering information on the item or person to be tested, and getting approval for the test. This state could be called "Initial Test Processing" or 'Test Request." Once the testing is approved, the test object might enter a second state where the test is actually performed. This state could be called the "Testing," "Actual Test Processing," or "Lab Work" state. After the test is completed, the object might then be available for evaluation and review. This final state could be called the "Test Results," "Test Evaluation," or "Test Review" state. In each example, the names provide some indication of what is happening to the test object in each state.

When defining a policy, you must have at least one state defined. Within that state, you must define some type of object access. All other information is optional. The sections that follow describe these clauses and the arguments they use.

Action, Check, and Notify Subclauses

The best practice for actions, checks, and notifications is to use triggers instead of these keywords to implement the functionality.

The Action, and Check subclauses are no longer supported.

The Action subclause associates a program with the promotion of this state. Once an object is promoted to this state, the program specified by the clause is executed. (Refer to the example on the next page.)

Although the Action subclause is optional, it is useful for executing procedures that might notify non-3DEXPERIENCE Platform users, generate reports, or place orders for equipment or services:

```
action PROGRAM
```

- PROGRAM is the name of a program object, or method, that has been or will be defined by the Business Administrator.

The Check subclause associates a verification procedure with the promotion of the object out of the state. The procedure is specified as a program which is executed when a person tries to promote the object. When executed, the procedure returns a true or false value. If the value is true, the object is promoted. If the value is false, the promotion is denied. Refer also to *Programs* in Chapter 7.

Use the following syntax to write a Check subclause:

```
check PROGRAM
```

- PROGRAM is the name of a program object, or method, that has been or will be defined by the Business Administrator.

The Notify subclause sends a message to selected users once a business object has entered the state. The message might provide special instructions or notify users that an object is ready for a particular action. The notification message is limited to 255 characters.

Use the following syntax to write a Notify subclause:

```
notify USER_NAME {,USER_NAME} message VALUE
```

Or:

```
notify signer message VALUE
```

- USER_NAME is the name of a person, group, role, or association to notify.
- Signer refers to the users who are included in the signature requirements for the next state. This allows one notify message to notify all signers automatically.
- VALUE is the text of the message to be sent to the user(s).

For example, assume you have a user manual that is being written. In its beginning state, only the author and the author's manager might access the document. However, once the manual is ready for review, it would most likely be promoted to a state where it is available to other users for comments. When this occurs, users must be notified that the manual is available and that review comments are required. The following subclause notifies two groups that a manual is ready for review:

```
notify Engineering, Training
   message "The User Guide is now ready for review. Please have your review
comments
    completed in two weeks."
```

Example of Action, Check, and Notify

The Action, and Check subclauses are no longer supported.

Assume that the states of the policy governing a type "Solid Model" are:

Planned —> Started —> Ready for Detail —> Released

Also assume that the states of the policy for Drawing objects are:

Planned —> Submit for Check —> Checked —> Released

When a solid model reaches the “Ready for Detail” state, the drafter begins work on the drawing. An action could be declared at this state to execute a program to create a Drawing business object of the same name as the solid model object. The program also could connect the two objects with the “Solid-Drawing” relationship. When arriving in this state, IconMail is sent (using “notify” in the state definition) to the drafting manager explaining that the solid model is ready for detail and that drawing files are checked into the attached Drawing object.

Before the drawing is released, a check should be performed to be sure the solid model was released. The check would be declared in the transition arrow before the “Released” state of the Drawing policy. The check would execute a program that would expand the object connected by the “Solid-Drawing” relationship and check their current state. If the current state is “Released,” promotion of the Drawing to the “Released” state would be allowed; otherwise, promotion would fail.

Route Subclause

The Route subclause automatically reassigns ownership when an object enters a business state. Since ownership implies greater access privileges and responsibility, changing ownership can be an effective means of controlling an object. The Route subclause is also used to notify the new owner of the reassignments. The route message is limited to 255 characters.

For example, assume you have a manual that was just completed by a writer. The manual is promoted into a state called “Formatting and Editing” in which an editor takes charge of the manual. The editor prepares the document for review and oversees any changes required to prepare it for publication. Since the writer is no longer involved, you may want to assign the manual’s ownership to the editor. While the editor might be among the users notified that the manual is finished, the editor should receive special notification that the manual now “belongs” to him/her. After the manual is published, you might again change the ownership to that of the company librarian. When changes in ownership occur, the new owner should be notified.

The Route subclause is an optional subclause that is very similar to the Notify subclause (described in *Action, Check, and Notify Subclauses* .

```
route USER_NAME {,USER_NAME} message VALUE
```

- `USER_NAME` is the name of a person, group, or role who will receive ownership of the business object.
- `VALUE` is the text of the message to be sent to the new owner(s)

For example, the following subclause notifies the editor that ownership of a manual was transferred to him/her:

```
route Editor
   message "Ownership of this manual has been transferred to you."
```

ACCESS_USER Subclauses

Use the following syntax to define access rules: .

```
ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]
```

where ACCESS_USER is:

```
    | [revoke] [login] public [key STRING]          |
    | [revoke] [login] owner [key STRING]           |
    | [revoke] [login] user USER_NAME [key STRING] |
```

Use the public, owner, and user keywords to specify that this rule applies to everyone (public), the object owner (owner), or to a named role (user USERNAME).

With revoke, all accesses specified as ACCESS_ITEMs are rescinded. If all conditions are true, then accesses are rescinded for the current user.

Revoke supersedes all other rules. If a rule revokes access for a user, no other rule can grant access to that user.

Login specifies that the rule only applies if the role it specifies matches a role associated with the user's current login context. This is more restrictive than the default, where the system looks at all user assignments to see if a rule applies to any of them.

The key STRING is a label that differentiates multiple rules defined for the same user. For example, if you have two rules defined for public or for user Designer, you can use the key STRING to need to differentiate them.

ACCESS_ITEM specifies the kinds of access this rule governs, either granting or revoking access. The ACCESS_ITEMs are described in *Accesses* in Chapter 3. With each ACCESS_ITEM, other than all and none, you can either grant the access or explicitly deny the access. You deny an access by entering not (or !) in front of the ACCESS_ITEM in the command. For example, !todisconnect or notchangeowner.

where USER_ITEM is:

[any | single | ancestor | descendant] organization
[any | single | ancestor | descendant] project
[any | context] owner
[any | no | context | inclusive] reserve
[any | no | public | protected | private |notprivate|ppp] maturity
[any|oem|goldpartner|partner|supplier|customer|contractor]category
[filter | localfilter] EXPR

The USER_ITEM specifies a variety of ways to define matching options between the current context's membership in various organization and policy against the project or organization ownership of the object being checked.

The flags have the following meanings:

- `any|single|ancestor|descendant]organization`—Specifies that the object's organization owner must be checked against the context's member organization.
- `any|single|ancestor|descendant] project`—Specifies that the object's organization owner must be checked against the context's member project.
- `any owner`—No check is performed on owner (default).
- `context owner`—Checks that object is owned by context user.
- `any reserve`—No check is performed on whether the object is reserved (default).
- `no reserve`—Checks that the object is not reserved by anyone.
- `context reserve`—Checks that the object is reserved by the current context user.
- `inclusive reserve`—Checks that the object is either reserved by the current context user or by no one.
- `any maturity`—No check is performed on the maturity of the project owning the object (default).
- `public maturity`—Checks that the object is owned by a project that is public.
- `protected maturity`—Checks that the object is owned by a project that is public.
- `private maturity`—Checks that the object is owned by a project that is private.
- `notprivate maturity`—Checks that the object is owned by a protected or public project.
- `ppp maturity`—Member of a private, protected, or public project, that is any project with the maturity property.
- `any|oem|goldpartner|partner|supplier|customer|contractor]category`—Checks that the current context or current login security context is flagged with the corresponding property.
- `[filter | localfilter] EXPR`—A filter expression defined for the user access. Use localfilter instead of filter in policies and access rules to return only results to which the current user definitely has access. When you use localfilter, the expression is not evaluated by full-text search.
  Expression access filters can be any expression that is valid in the system. Expressions are supported in various modules of the system, for example, query where clauses, expand, filters defined on workspace objects such as cues, tips and filters, etc. See *Working With Expression Access Filters* in Chapter 3 for details about how to use access filters.

Signature Subclause

This subclause specifies who can control the promotion or rejection of a business object. When an object is promoted, it moves to the next defined state and is subject to the access rules associated with that state. When an object is rejected, it remains in the current state until it meets the criteria for promotion.

The Signature subclause has the syntax:

```
signature SIGN_NAME [SIGNATURE_ITEM {,SIGNATURE_ITEM}
```

- `SIGN_NAME` identifies the type of signature. Try to use a name that identifies what the signature represents. For example, the signature might represent initial acceptance, completion, or final sign-off. Each of these terms could be used for a SIGN_NAME. For additional information, refer to *Administrative Object Names*.

- SIGNATURE_ITEM identifies the type of state change that will occur when a group, role, or person signs off on an object. Use any of the following:

approve USER_NAME {,USER_NAME}	Specifies that the object can be promoted to the next state. This clause is deprecated. Use the Approve access right for a user.
reject USER_NAME {,USER_NAME}	Specifies that the object must remain in the current state until it meets with the approval of the user. This clause is deprecated. Use the Reject access right for a user.
ignore USER_NAME {,USER_NAME}	Enables you to override the approval or rejection of the object. When ignore is specified, the named user can sign in the place of others. This might be useful to allow a senior manager the ability to sign off for a lower manager. This clause is deprecated. Use the Ignore access right for a user.
branch STATE_NAME	Specifies what the next state will be after a signature is applied.
filter EXPRESSION	Enables filtering to ensure that the promotion of an object meets certain criteria.

STATE_NAME is any previously-defined name of a state included in the current policy. By specifying a branch, you can decide which signatures are required to transition to a given state during a promote operation. When promotion is initiated, the system will choose the state for which all signatures are satisfied. If more than one branch is enabled, an error is generated.

EXPRESSION is a command that evaluates to either true or false. If a signature requirement *filter* evaluates to true, then the signature requirement is fulfilled. This is useful for adding required signatures that are conditional, dependent on some characteristic of a business object. The default rule is:

```
current.signature[NAME].satisfied
```

which is a select field that means the signature has been approved, ignored, or overridden. When you specify a filter, this default rule is replaced.

Approval can become dependent on any selectable field of the business object. This includes attributes as well as states and other signatures. The real power of filters comes from the use of combinatorial logic using and's and or's between state information and business object information.

For help formatting the expressions that can be entered into the Filter area, refer to the *Configuration Guide : About Selectables* in the online documentation.

USER_NAME is any previously defined name of an association, group, role, or person. If the name you give is not defined, an error message will result. If you are unsure of a user name, remember that you can obtain a complete listing of all of the user names by entering the MQL `List User` command.

For example, assume you have a state where objects are started and worked on prior to general review. While the object is in this state, you may want two types of signatures: one to indicate that the project is complete and another to indicate acceptance. This state definition might appear as:

```
state started
   revision false
   public all, notenable, notdisable, notoverride
   owner all, notenable, notdisable, notoverride
   user Manager override
   signature Complete
     approve Writer
     reject Writer
     ignore Manager
   signature Accepted
     approve Manager
     reject Manager
     ignore "Senior Manager"
```

When including an Ignore Signature item in a state definition, you should not confuse this with the override privilege. The Override privilege allows you to promote an object without any signatures at all. The Ignore Signature item assumes that a signature is required for object promotion. It simply allows the specified person to provide that required signature.

When a signature has been approved, it can subsequently be rejected. But when a signature has been rejected, `ignore` cannot subsequently be used to satisfy the signature.

Stateproperty Subclause

Integrators can assign ad hoc attributes, called Properties, to a state as well as to the policy as a whole. Properties allow associations to exist between administrative definitions that aren't already associated, and may be helpful to programmers implementing additional functionality. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, only the name joined with the object reference must be unique for any object that has properties.

```
add policy NAME
   state NAME stateproperty NAME [to ADMINTYPE NAME] [value
STRING];
```

In order to use the Stateproperty clause you must be a business administrator with admin property access. For additional information on properties, see *Administrative Properties* in Chapter 7. Refer to the Properties clause of the policy for details on adding properites to a policy.

Trigger Subclause

Event Triggers allow the execution of a Program object to be associated with the occurrence of an event. The following lifecycle events support Triggers:

approve	demote	disable
enable	ignore	override
promote	reject	schedule
unsign		

For example, when a state was scheduled, each successive state could be scheduled automatically by a specified offset value. These transactions are written into program objects which are then called by the trigger.

State Triggers use the following syntax:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input
ARG_STRING];
```

- `EVENT_TYPE` is any of the valid events for Policies: approve, demote, disable, enable, ignore, override, promote, reject, schedule, or unsign.
- `TRIGGER_TYPE` is Check, Override, or Action. Refer to the *Configuration Guide : Triggers* in the online documentation.
- `PROG_NAME` is the name of the Program object that will execute when the event occurs.
- `ARG_STRING` is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.
- Environment variable “0” always holds the program name and is set automatically by the system.
- Arguments following the program name are set in environment variables “1”, “2”,… etc.

For example:

```
state started
   revision false
   public all, notenable, notdisable, notoverride
   owner all, notenable, notdisable, notoverride
   user Manager override
   signature Complete
     approve Writer
     reject Writer
     ignore Manager
   signature Accepted
     approve Manager
     reject Manager
     ignore "Senior Manager"
   trigger schedule action "Schedule Offsets";
```

For more information, see the *Configuration Guide : Triggers* in the online documentation.

Minorrevision Subclause

This subclause specifies whether or not revisions of the object are allowed while the object is in this state.

This subclause uses two arguments: true and false.

```
minorrevision [true|false]
```

When the revision argument is set to `true`, revisions of the object are allowed. If the revision argument is set to `false`, no revisions are allowed.

For example, the following Minorrevision subclause prohibits the creation of a new revision while the object is in this state:.

```
state "Tax Return Completed"
   minorrevision false
   public none, read
   owner none, read, demote
   user Manager none, read, promote
```

In this example, you have a state for completed tax forms. In this state, you do not want anyone to revise the objects containing the finished tax returns. If further modification is required, a change in state must occur. The owner can demote the object to its former state to make changes and the manager can promote the object into the audit state.

The Minorrevision subclause is optional. Live Collaboration assumes that revisions are allowed if no subclause or argument is used. Use the `false` keyword to turn off the ability to create revisions.

Version Subclause

This subclause indicates whether or not any file can be checked in while the object is in the state.

Like the Minorrevision subclause described above, this subclause uses two arguments: true and false.

```
version [true|false]
```

When the version argument is set to `true`, files can be checked in. If the argument is set to `false`, no new files are allowed.

In the following state definition, the Customer or the builder/owner can check in files (that might contain room layouts, exterior views, or electrical plans, for example).

```
state “House Design Phase"
   revision false
   version true
   public none, read
   owner all
   user Customer none, read, checkin, modify
```

The Version subclause is optional. The Live Collaboration assumes that files can be checked into the object while the object is in the state if no subclause or argument is used. Use the `false` keyword to turn off the ability to check in files.

Promote Subclause

This subclause specifies whether or not the Live Collaboration tests the business object for promotion when a signature is modified, and promotes it automatically if all requirements are met.

This subclause uses two arguments: true and false.

```
promote [true|false]
```

If the keyword `true` is used, when a signature is approved or ignored, the Live Collaboration tries to promote the business object. If all signature and check requirements are satisfied, the business object are promoted automatically. If there are no signatures or check requirements on a state, this setting has no meaning.

The Promote subclause is optional. If the Promote subclause is not used, promotions are not automatic. Use the `false` keyword to turn auto-promotion off.

If there are no requirements on a State, the promote subclause has no meaning.

Checkouthistory Subclause

The generation of history information on the checkout event is optional. The need to disable checkout history stems from the implementation of distributed databases and the advanced search partial index process. In distributed databases, creating history records requires that a distributed transaction be run across multiple servers. If any server is unavailable, the transaction will fail. This means that all servers must be available in order to checkout/view files. If checkout history is disabled, only the local server needs to be accessible in order for the transaction to run to completion.

During a partial index process, business objects are indexed according to their modification date. If checkout history enabled, the history record and the modification date of the business object is updated whenever a file is checked out. Since the modification date is changed, during a partial index process, the business object and its associated files will be indexed again even if the business object has not changed except for the file check out. Disabling checkout may beneficial in preventing a lot of unnecessary indexing, especially if large files are often checked out. This subclause uses two arguments: true and false.

```
checkouthistory [true|false]
```

Store Clause

This clause identifies where files checked in under the policy are stored by default. All files must be stored as captured files. (For more information on file stores, see *Add Store*.) If you intend to associate files with business objects that are governed by the policy, you must include a Store clause in the policy definition:

```
store STORE_NAME
```

- `STORE_NAME` is the name of a previously defined file store. If the name you provide is not defined, an error message will result.

When using a Business Process Services app to check in a file, the person or company default store is used regardless of the store set by the policy.

For example, assume you have a policy for proposing and presenting drawings for review. These drawings may be of various types and formats. However, all information about the drawings can be contained in one file store. This file store identifies how the drawing files are managed and where they are stored. If you were to examine this policy, it might appear similar to the following:

```
add policy "Proposed Drawings"
   description "Policy for Drawing Proposal and
Presentation"
   type Drawing, Layout, Schematic, Sheet
   format Cadra-III, Rosetta-preView, CCITT-IV
   defaultformat Cadra-III
   sequence "A,B,C,..."
   store Drawings
   state planned
      public all
     owner all
   state started
     public all, notenable, notdisable, notoverride
     owner all, notenable, notdisable, notoverride
     user Employee enable, disable
     user Manager override
     signature Complete
        approve Manager
        reject Employee
        ignore Designer
     signature Accepted
        approve Manager
        reject Manager
        ignore Manager
   state presented
     public all, notdelete
     owner all;
```

In this example, the policy governs four types of business objects and uses three different file formats. However, when a file is checked into a business object governed by this policy, that file is managed and stored according to the definition of the Drawings file store.

MQL users and programmers can override the store specified in the policy by including the store in the `checkin` *command.*

For information on changing the store for a policy, see *Modify Policy*.

Allstate Clause

This clause allows you to define who can access a business object and what type of access they have across every state in the policy. Rather than specifing this for each state, the access you grant or deny will be applied to all states and is in addition to any access defined for a particular state.

The ability to deny or revoke access is another option for controlling access. Revoking access in an allstate access definition is different from the other ways access is revoked in that it is evaluated by the system as denying access rather than related to granting access.

The system first evaluates if access has been revoked prior to evaluating if access has been granted. No access is revoked by default. Revoke access must be explicitly defined. To define allstate access rules for a policy:

```
allstate [ALLSTATE_ITEM {,ALLSTATE_ITEM}]
Or
allstate revoke [ALLSTATE_ITEM {,ALLSTATE_ITEM}]
```

- ALLSTATE_ITEM is an allstate subclause which provides additional information about the allstate access rule you are defining. The allstate definition subclauses are:

For a description of these subclauses, see *ACCESS_USER Subclauses*.

Examples

To define a policy Software Maintenance with only read access to public on all states, use the following command:

```
add policy "Software Maintenance" allstate public read;
```

In this example, public has read and show access but does NOT have modify access if attribute[attribute1] != value1:

```
policy policy1
    description
    state allstate
      revoke public modify
         filter attribute[attribute1] == value1
      public read,show
         filter attribute[attribute1] == value1
      owner none
```

Additional examples of usage:

```
add policy simple-u type all sequence "0-9,A-Z"
    format f
    store captured-u
    allstate public checkin
        user Designer modify filter "organization == US"
    state one
       owner all
       public read,show
       user Designer read,show
    state final
       owner all
       public read,show;
print policy simple-u select allstate allstate.*;
   allstate = TRUE
   allstate.publicaccess = checkin
   allstate.owneraccess = all        // default if not specified
   allstate.access[Designer] = modify
   allstate.filter[Designer] = organization == US
modify policy simple-u allstate remove user Designer all;
print policy simple-u select allstate allstate.*;
   allstate = TRUE
   allstate.publicaccess = checkin
allstate.owneraccess = all
```

Filter expressions of the form `current.access[ACCESS_TYPE] == TRUE` *accept minorrevise as an ACCESS_TYPE.*

History Clause

The `history` keyword adds a history record marked “custom” to the policy that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Majorsequence and Minorsequence Clauses

The Majorsequence and Minorsequence clauses allow you to define major and minor revision sequences for a policy.

```
add policy NAME majorsequence A,B,C,... minorsequence 1,2,3
  delimiter '-';
```

The `delimiter` can be only ASCII non-alphanumeric characters. The following characters are not permitted as delimiters: `? * ' " , [ ] ( )`

Both major and minor sequences are validated using the following rules:

1. Commas are used to separate enumerated types.
2. A trailing ellipsis (…) means a continuing sequence. A leading ellipsis means a sequence consisting of three dots.
3. In the absence of a trailing ellipsis, hyphens denote a range. For example:
   - A-Z signifies that all letters from A through Z, inclusive, are to be used.
   - A-1,... The trailing ellipsis used here signifies that A-1 is a revision string, not a range.
4. Only the beginning of a range is checked for the presence of a delimiter.

Strings generated or passed in major or minor revisions are checked for the presence of the delimiter. The same applies to full revision strings. In addition, full revision strings are checked for the presence of more than one instance of the delimiter.

See *Published States* in Chapter 4 for more information.

Copy Policy

After a policy is defined, you can clone the definition with the Copy Policy command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy policy SRC_NAME DST_NAME [MOD_ITEM] {,MOD_ITEM};
```

- `SRC_NAME` is the name of the policy definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the policy that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Policy

After a policy is defined, you can change the definition with the Modify Policy command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify policy NAME [MOD_ITEM] {MOD_ITEM};
```

- `NAME` is the name of the policy you want to modify.
- `MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Policy clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Policy Clause	Specifies that…
`defaultformat FORMAT_NAME`	The default is set to the format named.
`description VALUE`	The current description, if any, is changed to the values entered.
icon FILENAME	The image is changed tp the new image in the field specified.
`add type TYPE_NAME {,TYPE_NAME}`	The named types are added to the list of object types that can have this policy.
`add type all`	All existing and future object types are permitted under this policy. For details on the ‘all’ keyword, see *Type Clause*.
`remove type TYPE_NAME {,TYPE_NAME}`	The named object types are removed from the list of objects that can have this policy.
`remove type all`	Removes the 'type all' option from the policy. When the 'type all' option is removed or turned off, no types will be defined for the policy and no objects can be created under the policy until a type defined. If the ‘type all’ option was never used for the policy, 'remove type all' does nothing.
`add format FORMAT_NAME {,FORMAT_NAME}`	The named formats are added to the list of formats permitted by this policy.
`add format all`	All existing and future format types are permitted with this policy. For details on the ‘all’ keyword, see *Format Clause*.
`remove format FORMAT_NAME {,FORMAT_NAME}`	The named formats are removed from the list of formats permitted by this policy.
`remove format all`	Removes the ‘format all' option from the policy. When the ‘format all' option is removed or turned off, no formats will be defined for the policy. If the ‘format all’ option was never used for the policy, 'remove format all' does nothing.

Modify Policy Clause	Specifies that…
`add state STATE_NAME [before STATE_NAME] [STATE_ITEM {,STATE_ITEM}]`	The named state is added to the policy with the state definitions listed. If you do not want the new state added after the existing states, you must specify which existing state the new state should precede. If a state is added to an existing policy which already governs objects, all object instances will be affected. If an object is in a state that precedes the new state, a state is added, as desired, in the object's lifecycle. However, if the object's current state is beyond where the new state is added, the object will never reach that state except through demotion. In some cases, this is not a concern; but, states should be added to existing policies with care.
`remove state STATE_NAME`	The named state is removed from the policy if there is at least one state remaining after the removal. Removing a state from a policy that is governing objects is not recommended. An alternate approach is to clone the policy and then remove the state from the clone. There is the notion that "from this point on" the policy will control these types of objects. New objects should use the new policy and older objects can change to the new policy, if desired. When a state is removed from a policy, all signatures to and from the state are removed. If the policy is in use, all signature approvals, comments, etc. are deleted.
`add allstate USER [ACCESS_LIST] [filter | localfilter EXPR]`	The named allstate access is added to the policy. You can also add allstate access that revokes access. You can only revoke access for Public and Owner. For example: `modify policy "Product Requirement" add allstate revoke public read,modify,checkin;`
`remove allstate USER [ACCESS_LIST] [filter | localfilter EXPR]`	The named allstate USER or just the specified ACCESS_LIST for that user that was previously defined for the policy is removed. You can also remove allstate access that revoked access.
`state STATE_NAME remove stateproperty NAME`	The stateproperty is removed from the named state.
`name VALUE`	The current state name is changed to that of the new name.
`sequence REVISION_SEQUENCE`	The revision sequence is changed to the sequence entered.
`state STATE_NAME [STATE_MOD_ITEM {STATE_MOD_ITEM}]`	The named state is changed according to the state modification clauses entered.

Modify Policy Clause	Specifies that…
`store STORE_NAME`	The file store is changed to use the file store named. Keep in mind that if you change the store for a policy, files that are already checked into objects governed by the policy will still reside in the old store. If these objects are revised or cloned, the new revision/ clone references the original file in its storage location and thus the clone or revision will be placed in the old storage location. When the time comes for the file reference to become an actual file (as when the file list changes between the 2 objects) the file copy is made in the same store the original file is located in. However any new files that are checked in will be placed in the new store. For more details, see the *Implications of Changing Stores* in Chapter 2.
`checkouthistory true`	The generation of history information on the checkout event is enabled.
`checkouthistory false`	The generation of history information on the checkout event is disabled.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the policy that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the clauses and arguments that define the policy. For example, assume you want to modify the policy for proposing, presenting, and releasing drawings. It has been decided to use the policy for only CAD drawings while a new policy is defined for handling non-CAD drawings. To customize the existing policy for CAD use, you might begin modifying the name and the policy’s description clause with the following Modify Policy command:

```
modify policy Drawings
   name "CAD Drawings"
   description "Policy for CAD Drawing Proposal, Review and Release";
```

In the following example, the policy Product Requirement is modified to add access for Sam on all states in the policy:

```
modify policy “Product Requirement” add allstate
user Sam read,modify,checkin;
```

These changes leave only the states to be modified. But how is that done? Just as the Modify Policy clauses resemble the Add Policy clauses, the subclauses that modify states resemble those that define them. These are described in the following sections.

Modifying Policy States

The following subclauses are available to modify the existing states in a policy:

Modify Policy State Subclause	Specifies that…
`action COMMAND`	The action defined by the command is taken when the object is promoted to this state.
`check COMMAND`	The verification test to be executed when the object is promoted to this state changes as specified by the command. This test must return a true or false value.
icon FILENAME	The image is changed tp the new image in the field specified.
`name VALUE`	The name of the current state is changed to the new name entered.
`add notify USER_NAME {,USER_NAME} [message VALUE]`	The user(s) listed is/are added to those notified when the object is promoted to this state. If a message value is entered, the message changes to the new value.
`remove notify USER_NAME {,USER_NAME}`	The user(s) listed is/are removed from the list of users who are notified when the object is promoted to this state.
`add notify signer [message VALUE]`	The user(s) included in the signature requirements for the next state is/are notified when the object is promoted to the state. This allows one notify message to notify all signers automatically.
`remove notify signer`	The user(s) included in the signature requirements for the next state are removed from the list of users who are notified when the object is promoted to the state.
`promote false`	Promote to the state will not occur automatically even if all conditions are met.
`add owner ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]`	One or more access items are added to the owner access list. This list specifies the access privileges the owner has when the governed object is in this state. See *Accesses* in Chapter 3 for more information. For a description of these subclauses, see *ACCESS_USER Subclauses*.
`remove owner ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]`	The listed access item is removed from the owner access list. See *Accesses* in Chapter 3 for more information. For a description of these subclauses, see *ACCESS_USER Subclauses*.
`add public ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]`	One or more access items are added to the public access list. This list specifies the access privileges the public has when the governed object is in this state. See *Accesses* in Chapter 3 for more information. For a description of these subclauses, see *ACCESS_USER Subclauses*.
`remove public ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]`	The listed access item is removed from the public access list. See *Accesses* in Chapter 3 for more information. For a description of these subclauses, see *ACCESS_USER Subclauses*.
`minorrevision true|false`	If true, revisions are allowed in this state. Otherwise, revisions are not allowed in this state.
`add route USER_NAME {,USER_NAME} [message VALUE]`	The users listed are added to those who will receive ownership of the business object when the object is promoted to this state. If a message value is entered, the message changes to the new value.

Modify Policy State Subclause	Specifies that…
`remove route USER_NAME {,USER_NAME}`	The users listed are removed from the list of users who receive ownership of the object when it is promoted to this state.
`route message VALUE`	The message sent to users who receive ownership of the business object when the object is promoted to this state changes to the value entered.
`add signature SIGN_NAME [SIGNATURE_ITEM {,SIGNATURE_ITEM}]`	The names listed can promote, reject, or ignore the business object.
`remove signature SIGN_NAME`	The entered signature is removed from the list of signatures required to alter the object's state.
`signature SIGN_NAME [SIGNATURE_MOD_ITEM {SIGNATURE_MOD_ITEM}]`	The named signature is modified according to the modification clause(s) listed.
`add trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME`	The specified trigger is added or modified for the listed event.
`remove trigger EVENT_TYPE TRIGGER_TYPE`	The specified trigger type is removed from the listed event.
`add user ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]`	One or more access items are added to the user access list. This list specifies the access privileges the named user has when the governed object is in this state. See *Accesses* in Chapter 3 for more information. For a description of these subclauses, see *ACCESS_USER Subclauses*.
`remove user ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]`	The listed access items are removed from the named user's access list. See *Accesses* in Chapter 3 for more information. For a description of these subclauses, see *ACCESS_USER Subclauses*.
`version true`	New versions are allowed in this state.
`version false`	New versions are not allowed in this state.
`majorrevision TRUE|FALSE`	It is allowable to create a new major revision of an object in this state, in support of major or minor revisioning. See *Published States* in Chapter 4 for details.
`minorrevision TRUE|FALSE`	It is allowable to create a new minor revision of an object in this state, in support of major or minor revisioning. See *Published States* in Chapter 4 for details.
`published TRUE|FALSE`	The state can be marked as "published," in support of major or minor revisioning. See *Published States* in Chapter 4 for details.

When modifying the states of a policy, the state modification clauses are similar to those used to define states. For example, assume you had the following state definition:

```
state "In Progress"
   revision false
   version true
   public all
   owner all
```

Now you want to have a Manager oversee the work in progress. Rather than let the owner or public promote the object into the next state, you want to give that privilege to the Manager only. To make these changes, you might write a command such as:

```
modify policy "Manual Release"
   state "In Progress"
     add public notenable, notdisable, notoverride, notpromote
     add owner notenable, notdisable, notoverride, notpromote
     add user Manager promote, enable, disable, override;
```

When this command is processed, the state definition is modified to appear as:

```
state In Progress
   revision false
   version true
   public notenable, notdisable, notoverride, notpromote
   owner notenable, notdisable, notoverride, notpromote
   user Manager promote, enable, disable, override;
```

Modifying Signature Requirements

If you want to alter the signature requirements within a state, you must use a Signature subclause within the State subclause:

```
signature SIGN_NAME [SIGNATURE_MOD_ITEM
{SIGNATURE_MOD_ITEM}]
```

- `SIGN_NAME` is the name of the signature item to modify.
- `SIGNATURE_MOD_ITEM` identifies the type of modification.

You can make the following types of modifications to a signature clause. These types of modifications are related to the subclauses you saw in defining the signature requirements:

Modify Signature Subclause	Specifies that…
`[add] approve USER_NAME {,USER_NAME}`	The users are added to the list of people who can approve of the object. The add keyword is optional and behavior is the same with or without it.
`remove approve USER_NAME {,USER_NAME}`	The users are removed from the list of people who can approve of the object.
`[add] ignore USER_NAME {,USER_NAME}`	The users are added to the list of people who can sign in place of an approver or rejecter.
`remove ignore USER_NAME {,USER_NAME}`	The users are removed from the list of people who can sign in place of an approver rejecter.
`[add] reject USER_NAME {,USER_NAME}`	The users are added to the list of people who can reject the object. The add keyword is optional and behavior is the same with or without it.
`remove reject USER_NAME {,USER_NAME}`	The users are removed from the list of people who can reject the object.

Modify Signature Subclause	Specifies that…
`add branch STATE_NAME`	The branch is added to the signature.
`remove branch STATE_NAME`	The branch is removed from the signature.
`add filter EXPR`	The filter is added for the signature.
`remove filter EXPR`	The filter is removed from the signature.

The following state definition creates user documentation:

```
state "In Progress"
   revision false
   version true
   public all, notenable, notdisable, notoverride
   owner all, notenable, notdisable, notoverride
   user Manager override
   signature Complete
     approve Writer
     reject Writer
     ignore Manager
```

Now you want to allow the editor to approve or reject the object. You also want to add a second signature that shows the manual has been accepted by the editor. To make these changes, you could write a Modify Policy command similar to the following:

```
modify policy "Manual Release" state "In Progress"
   signature Complete
     add approve Editor
     add reject Editor
   signature Accepted
     add approve Editor
     add reject Editor
     add ignore Manager;
```

After this command is processed, the state definition appears as:

```
state In Progress
   revision false
   version true
   public all, notenable, notdisable, notoverride
   owner all, notenable, notdisable, notoverride
   user Manager override
   signature Complete
     approve Writer, Editor
     reject Writer, Editor
     ignore Manager
   signature Accepted
     approve Editor
     reject Editor
     ignore Manager
```

Print Policy

You can view the definition of a policy using the Print Policy command.

For example, the following command uses the boolean selectable *state.published* to report whether or not the `published` flag is set for the policy in support of major/minor revisioning:

```
print policy NAME select state.published;
```

The following command uses the boolean selectables majorsequence, minorsequence, and delimiter to report the majorsequence pattern, the minorsequence pattern, and the delimiter, respectively.

```
print policy NAME select majorsequence minorsequence delimiter;
```

The following command reports whether a minor revision can be created on an object in a particular state:

```
print policy NAME select state.minorrevisionable;
```

See *Published States* in Chapter 4 for details.

Delete Policy

If you decide that a policy is no longer required, you can delete it by using the Delete Policy command:

```
delete policy NAME;
```

`NAME` is the name of the policy to be deleted. If the name is not found, an error message will result.

Searches the list of policies. If the name is found, that policy is deleted IF there are no objects that use the policy. If there are objects that use the policy, they must be reassigned or deleted before the policy can be removed from the policy list.

For example, you might enter this command to delete the policy named “Performance and Salary Review:”

```
delete policy "Performance and Salary Review";
```

portal Command

Description

A *portal* is a collection of *channel*s, as well as the information needed to place them on a Web page. Some portals are installed with the Framework and used in apps to display PowerView pages, but they may also be created for use in custom Java applications.

For conceptual information on this command, see *Portals* in Chapter 8.

User level

Business Administrator with portal administrative access.

Syntax

```
[add|copy|modify|delete] portal NAME [CLAUSEs];
```

- NAME is the name you assign to the portal. Portal names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the portal.

Add Portal

To define a portal from within MQL use the add portal command:

```
add portal NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

user USER_NAME can be included if you are a business administrator with person/group/role access defining a channel for another user. This user is the item's "owner." If the user clause is not included, the portal is a system item.

ADD_ITEM is an add portal clause that provides additional information about the portal. The add portal clauses are:

description VALUE
label VALUE
href VALUE
alt VALUE
channel CHANNEL_ID{,CHANNEL_ID} [{channel CHANNEL_ID{,CHANNEL_ID}}];
setting NAME [STRING]
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

```
visible USER_NAME{,USER_NAME};
history STRING
```

Label Clause

This clause specifies the label to appear in the application in which the portal is assigned.

Href Clause

This clause is used to provide link data to the JSP. The href link is evaluated to bring up another page. Many portals will not have an href value at all. The href string generally includes a fully-qualified JSP filename and parameters, which can contain embedded macros and expressions for mapping to database schema.

The syntax is:

```
href VALUE;
```

- `VALUE` is the link data.

Alt Clause

This clause is used to define text that is displayed until any image associated with the portal is displayed and also as “mouse over text.”

The syntax is:

```
alt VALUE;
```

- `VALUE` is the text that is displayed.

For example, you could use the following for a Program Manager portal:

```
alt "Program Manager";
```

Channel Clause

This clause is used to specify existing channels to be added to a single row in the portal you are creating. You can add channels that are owned by you (in your workspace), or system channels. Channels will be displayed in the order in which they are added. Separate items with a comma. To indicate a new row in the portal, use the channel clause again.

You cannot add channels from other user’s workspaces.

The syntax is:

```
channel CHANNEL_ID{,CHANNEL_ID} [{channel
CHANNEL_ID{,CHANNEL_ID}}];
```

CHANNEL_ID is the name of the channel you are adding, plus an optional keyword system. Specify 'system' if you are creating a workspace portal and the channel is a system channel. If the portal you are creating is a system portal, the channels added are also assumed to be system.

For example, the following portal has 3 rows of channels:

```
add portal MyPortal channel ABC system,DEF channel EFG
system channel XYZ;
```

For details on creating channels, see *Add Channel*.

Setting Clause

This clause is used to provide any name/value pairs that the portal may need. They can be used by JSP code, but not by hrefs on the Link tab. Refer to *Using Macros and Expressions in Configurable Components* in Chapter 8 of this guide for more details.

The syntax is:

```
setting NAME [STRING];
```

For example, an image setting with the image name can be specified to display when the portal is used in a toolbar:

```
setting Image iconSmallMechanicalPart.gif;
```

History Clause

The `history` keyword adds a history record marked "custom" to the portal that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Portal

After a portal is defined, you can clone the definition with the Copy portal command.

If you are a Business Administrator with portal access, you can copy system portals. If you are a Business Administrator with person access, you can copy portals in any person's workspace (likewise for groups and roles). Other users can copy visible workspace portals to their own workspaces.

This command lets you duplicate portal definitions with the option to change the value of clause arguments:

```
copy portal SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

`SRC_NAME` is the name of the portal definition (source) to be copied.

`DST_NAME` is the name of the new definition (destination).

`COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any portal of the same name belonging to the user specified in the `touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications. Note that you need specify only the fields to be modified.

Clone/Modify Portal Clause	Specifies that…
`name NEW_NAME`	The current portal name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed tp the new image in the field specified.
`label VALUE`	The label is changed to the new value specified.
`href VALUE`	The link data information is changed to the new value specified.
`alt VALUE`	The alternate text is changed to the new value specified.
`height VALUE`	The height is changed to the new value specified.
`place CHANNEL1 [system] [newrow] before CHANNEL2 [system]`	The named CHANNEL1 is moved or added before CHANNEL2. Use `newrow` to indicate that the channels are not next to each other but in separate rows. If CHANNEL2 is an empty string, CHANNEL1 is placed before all channels in the portal.
`place CHANNEL1 [system] [newrow] after CHANNEL2 [system]`	The named CHANNEL1 is moved or added after CHANNEL2. Use `newrow` to indicate that the channels are not next to each other but in separate rows. If CHANNEL2 is an empty string, CHANNEL1 is placed after all channels in the portal.
`add setting NAME [STRING]`	The named setting and `STRING` are added to the portal.
`remove channel NAME [system]`	The named channel is removed from the portal. Specify system if it is a channel that is not owned by the same user as owns the portal. If modifying a system portal, the channel is assumed to be a system channel.
`remove setting NAME [STRING]`	The named setting and `STRING` are removed from the portal.
`visible USER_NAME{,USER_NAME};`	The named user(s) have visibility to the portal.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the portal that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Portal

If you are a Business Administrator with portal access, you can modify system portals. If you are a Business Administrator with person access, you can modify portals in any person's workspace (likewise for groups and roles). Other users can modify only their own workspace portals.

You must be a business administrator with group or role access to modify a portal owned by a group or role.

Use the modify portal command to add or remove defining clauses or change the value of clause arguments:

```
modify portal NAME [user USER_NAME] [MOD_ITEM {MOD_ITEM}];
```

`NAME` is the name of the portal you want to modify. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.

`MOD_ITEM` is any of the items in the table above.

Each modification clause is related to the arguments that define the portal. To change the value of one of the defining clauses or add a new one, use the modify clause that corresponds to the desired change.

When modifying a portal, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire command.
- If you are working from a script, group the changes together in a single modify portal command.

History Clause

The `history` keyword adds a history record marked "custom" to the portal that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Delete Portal

If you are a Business Administrator with portal access, you can delete system portals. If you are a Business Administrator with person access, you can delete portals in any person's workspace (likewise for groups and roles). Other users can delete only their own workspace portals.

You must be a business administrator with group or role access to delete a channel owned by a group or role.

If a portal is no longer required, you can delete it using the delete portal command

```
delete portal NAME [user USER_NAME];
```

`NAME` is the name of the portal to be deleted. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.

Searches the list of defined portals. If the name is found, that portal is deleted. If the name is not found, an error message is displayed. For example, to delete the portal named “Program Manager” enter the following:

```
delete portal "Program Manager";
```

After this command is processed, the portal is deleted and you receive an MQL prompt for another command.

product Command

Description

The *product* command is used to define the names of products that are sold and licensed as dynamic product configurations and support product named user licensing. For more information about user licensing, see the *Installation Guide : Obtaining and Installing Your User Licenses*. The product command allows you to use MQL in addition to the user interface provided in Business Process Services to manage user licenses. For information about using the user interface, see the *Business Process Services Common User Guide* : *Working with User Licenses*.

A dynamic product configuration consists of a base product configuration plus a set of add-on products. For example, 3DLive (LIV) + CATIA Mechanical Design (MDE) + VPM Central (VPM). Base product configurations are pre-defined products and installed with the software. They can viewed with the “list product * where 'dynamic==false'” command.

User level

Organization Manager

Syntax

```
[add|modify|list|print|delete]product NAME {CLAUSE};
```

- `NAME` is the name of the product.
- CLAUSEs provide additional information about the product.

The Add Product and Delete Product commands are only valid for dynamic product configurations.

Add Product

The Add Product command is used to define a dynamic product configuration:

```
add product NAME [ADD_ITEM];
```

- `NAME` is the name of the dynamic product configuration you are defining. The name must use the naming convention for dynamic product configurations: `CCC-XXXYYYZZZ`. A name including only the `CCC` portion of this naming convention is not valid.

  `CCC` is the trigram of the base product configuration. This trigram must be a valid base product configuration.

  `XXX`, `YYY` and `ZZZ` are the trigrams of any add-on products that were purchased. Each add-on product must be a trigram. If a trigram for a non-existent add-on product is used, the non-existent trigram will be ignored and a warning will be issued.You must alphabetize this list of trigrams.

  A dynamic product configuration must be a rich client and no trigram in the name can be for a web client. If a trigram for a web client is in the name, then the dynamic product configuration definition will fail.

- ADD_ITEM provides additional information about the dynamic product configuration. The Add Product clauses are:

```
description VALUE
```

```
title TITLE
```

Title Clause

This clause specifies the full name of the dynamic product configuration.

List Product

The List Product command is used to confirm the existence or exact name of any product. Additional clauses allow you to limit the objects listed, include selectable data about each object listed, and/or provide definition for the output format. To list a product:

```
list product NAME_PATTERN [where WHERE_CLAUSE] [SELECT
[DUMP [RECORDSEP]]] [tcl] [output FILENAME];
```

- NAME_PATTERN is the name of the product.

For additional information on using the list command, see *list admintype Command*.

The SELECT fields for a product include:

```
title
description
dependency
derivative
addon
richclient
webclient
technical
dynamic
person
hidden
property
modified
originated
```

Print Product

The Print Product command prints a product definition to the screen allowing you to view it. When a Print command is entered, MQL displays the various clauses that make up the definition.

```
print product NAME [SELECT] [[DUMP] RECORDSEP] [tcl] [output
FILENAME];
```

- NAME is the name of the product.
- SELECT lets you specify data to present about the item being printed. See the list of selectable fields for a product above.

For additional information on using the print command, see *Item Commands*.

Modify Product

The Modify Product command is used to add or remove defining clauses and change the value of clause arguments for a product definition:

```
modify product NAME [MOD_ITEM];
```

- `NAME` is the name of the product you want to modify.
- `MOD_ITEM` is the type of modification you want to make.

You can make the following modifications. Note that you need to specify only fields to be modified.

Modify Product Clause	Specifies that…
`title TITLE`	The title or full name of the custom configuration changes to that of the new title entered. This clause only applies to custom configurations.
`description VALUE`	The description is changed to the VALUE given. This clause only applies to custom configurations.
`add person NAME`	Adds a person to the product definition. See *Assigning and Removing User Licenses*.
`remove person NAME`	Removes a person from the product definition. See *Assigning and Removing User Licenses*.
`[!|not]hidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMININTERFACE NAME] [value STRING]`	The named property is modified.
`onlineinstance NAME`	Online instance ID. See *Assigning Casual Licenses*.
`casualhour HOUR`	Prepaid hours of a casual license. See *Assigning Casual Licenses*.

Assigning and Removing User Licenses

When you add a user to a product definition, a license is assigned to that user. It is important to note that this license is not automatically reserved for that user. Licenses are reserved for a user ONLY at the time they log in. This means if a user is assigned a license for a product after they have logged in, the new license will not be reserved until that user logs in again. Once a user logs in, a license is consumed in each product that contains a role assigned to that user.

Users assigned to roles in different products require licenses to be available in all those products. If a license is not available for every product the user requires, the user will not be able to login. All of the licenses associated with a user must be available. Additionally, even if a user is unable to login because all of the licenses associated are not available, a CPF license will still be reserved for that user.

When you remove a user from a product definition via MQL, that user's license is not immediately released and made available. To release the user license immediately, you must remove the user via the user interface in BPS.

Assigning Casual Licenses

Casual licensing allows a customer to order a license with a maximum number of prepaid usage hours per month.

The MQL command to add, remove, or update the number of casual license hours is as follows:

```
modify product NAME [MOD_ITEM];
```

where MOD_ITEM is:

```
| add [onlineinstance NAME] [casualhour HOUR] person NAME [person
NAME]
| remove [onlineinstance NAME] person [all|NAME] [person NAME]
| update [onlineinstance NAME] [casualhour HOUR] person NAME [person
NAME]
```

The following is a typical casual licensing scenario:

1. The **V6 Customer** orders casual licenses from DSX-ECO and enrolls the license keys into the Dassault Systèmes License Server (DSLS):
   - **a)** Order casual licenses (prepaid hours per month) from DSX-ECO.
   - **b)** Enroll casual license keys into the DSLS.
2. The **V6 Administrator** assigns products to persons. Each product-person assignment can be either a full product assignment or a casual product assignment. A full product assignment means that the kernel requests a full license when the user logs in, whereas a casual product assignment means that the kernel requests a casual license instead of a full license. A person can have several products assigned, including some full products and some casual products, but cannot have a product assigned more than once. For example, a person could have two products assigned, such as CPF (full) and ENG (casual, 40 hours), but not the same product assigned twice, such as CPF (full) and CPF (casual, 40 hours).
   - **a)** Retrieve casual license info (inclusing prepaid hours) from the DSLS.
   - **b)** Assign full and casual products (with prepaid hours) to persons. The number of prepaid hours must be specified when doing the casual product assignment because a customer might order, for example, two casual CPF licenses (one for 30 hours per month and another for 40 hours per month). The kernel needs to know which casual license is assigned to the person, CPF 30 hours or CPF 40 hours.
   - **c)** Save product-person assignments to the database.
3. The **V6 webtop user** can log into the Server, once all required products are assigned to him, and start to work. After the person's username and password have been entered at login time, the kernel retrieves all assigned products (full and casual) from the database and reserves the corresponding licenses from the DSLS.

   The total number of hours that a casual license has been used per month is calculated by continually sending a User Still Alive event to the DSLS at a defined interval (the default is 15 minutes). The DSLS calculates the total hours that a casual license has been used in a month based on the login time stamp and these User Still Alive events.
   - **a)** Log into the Server.
   - **b)** Retrieve the assigned products (full and casual) from the database.
   - **c)** Reserve licenses by passing the full and casual (with prepaid hours) assigned product list to the DSLS.
   - **d)** Runtime end user interaction.

e) Send User Still Alive events to the DSLS (for casual license users only, unless MX_NUL_FULL_USAGE_REPORT is set to true).

f) Log out.

g) Send User End Session event to tell the DSLS to stop recording usage (for casual license users only, unless the variable MX_NUL_FULL_USAGE_REPORT is set to true).

For casual users to be able to check the User Still Alive and User End Session events, the MX_NUL_TRACE variable must be enabled.

The selectables `casualhour` and `onlineinstance`, and their subselectables, can be applied to types Person and Product to retrieve information on their casual licenses.

Below are two examples of the MQL code that assigns the product ENG to two persons, one with a full license and another with a casual license for 30 hours per month. The first example shows the case of a non-online instance, the second one shows a case using an online instance:

```
// assign product to person (non-onlineinstance case)
MQL<1>modify product ENG add person p1;
MQL<2>modify product ENG add casualhour 30 person p2;
MQL<3>print product ENG select person;
product   ENG
   person = p1
MQL<4>print product ENG select casualhour;
product   ENG
   casualhour = 30
MQL<5>print product ENG select casualhour[30].person;
product   ENG
   casualhour[30].person = p2
MQL<6>print person p1 select product;
person    p1
   product = ENG
MQL<7>print person p2 select casualhour;
person    p2
   casualhour = 30
MQL<8>print person p2 select casualhour[30].product;
person    p2
   casualhour[30].product = ENG

// assign product to person (onlineinstance case)
MQL<1>modify product ENG add onlineinstance oi1 person p3;
MQL<2>modify product ENG add onlineinstance oi1 casualhour 40 person
p4;
MQL<3>print product ENG select onlineinstance[oi1].person;
product   ENG
   onlineinstance[oi1].person = p3
MQL<4>print product ENG select onlineinstance[oi1].casualhour;
product   ENG
   onlineinstance[oi1].casualhour = 40
MQL<5>print product ENG select
onlineinstance[oi1].casualhour[40].person;
product   ENG
```

```
   onlineinstance[oi1].casualhour[40].person = p4
MQL<6>print person p3 select onlineinstance[oi1].product;
product   ENG
   onlineinstance[oi1].product = ENG
MQL<7>print person p4 select onlineinstance[oi1].casualhour;
person    p4
   onlineinstance[oi1].casualhour = 40
MQL<8>print person p4 select
onlineinstance[oi1].casualhour[40].product;
person    p4
   onlineinstance[oi1].casualhour[40].product = ENG
```

Delete Product

The Delete product command allows you to delete a dynamic product configuration that is no longer needed. Because deleting certain items can affect other elements, use it carefully.

```
delete product NAME;
```

- `NAME` is the name of the dynamic product configuration you are deleting.

program Command

Description

A *program* is an object created by a Business Administrator to execute specific commands.

For conceptual information on this command, see *Programs* in Chapter 7.

User Level

Business Administrator

Syntax

```
[add|copy|modify]program NAME {CLAUSE};
```

- NAME is the name you assign to the program. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the attribute.

Add Program

A program is created with the Add Program command:

```
add program NAME [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name you assign to the program.
- ADD_ITEM is an Add Program clause which provides additional information about the program. The Add Program clauses are:

code CODE
description VALUE
java|external|mql
file FILENAME
execute immediate|deferred
execute user USER_NAME
[!|not]needsbusinessobject
[!|not]downloadable
[!|not]pipe
[!|not]pooled
[!|not]hidden

```
rule NAME
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

All of these clauses are optional. You can define a program by simply assigning a name to it. You will learn more about each Add Program clause in the sections that follow.

Code Clause

This clause defines the commands for the program. Below are examples of program code that could be written to provide various functionality.

Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as those created with the ESC key, should not be used for ANY field, including business and administrative object names, description fields, program object code, or page object content.

Format Definition Example Program

To be used in a format definition, a program object definition must include these characteristics:

- The needsbusinessobject clause must be true.
- The code clause must contain the command needed to execute the program and the syntax for the command must be appropriate for the operating system.
- The code clause should end with the $FILENAME macro so the program opens any file. Enclose the macro in quotes to ensure that files with spaces in their names are opened correctly.

To launch Microsoft Word 97 or 2000 on Windows and open a document file for viewing and editing, the following program code might be used in the external program named `MSWORD`:

```
${PROG} /w "${FILENAME}"
```

The `${PROG}` macro returns the command needed to execute the program defined by the file association mechanism of windows. The /w is needed for Word 2000, but is ignored for other versions.

To open multiple PDF files you can create an external Program with the following code:

${PROG} /n ${FILENAME}

The /n is needed for multiple files to be opened without errors.

For a simple generic format that uses file associations, do not define any programs for the edit, view and print clauses of the format. For a more complex and flexible generic Windows format that

will open any file for view, edit, or print based on its file association, include a condition exception for Word. For example:

```
tcl;
eval {
... parsing code to take out quotes / args / etc. from the
registry
 if [string match *winword* ${PROG}] {
    exec [${PROG} /w "${FILENAME}"]
 } else {
    exec [${PROG} "${FILENAME}"]
 }
}
```

On a UNIX system, you might use the following for a text format:

```
edit textedit "${FILENAME}"
```

Note that the quotes allow the file name to contain spaces.

Some examples follow.

Action Program Example

You might define a program to be used as an action on a State as follows:

```
**
 ** Note that the macros (EVENT,OBJECTID) required by the program must be explicitly
 ** specified when the program is configured on the policy.  The macros are passed in
 ** the 'args' array when the trigger is run.
 **
 ** This trigger will increment the 'docInt' attribute on an object when it is promoted
 ** from state Created to state Working, and will decrement the attribute when an
 ** object is demoted from state Working to state Created.
 **
 ** To configure this as a promote and demote action on a policy:
 ** mod policy docPolicy state Created add trigger promote action
 **            docPromoteAction input "-method mxMain ${EVENT} ${OBJECTID}";
 ** mod policy docPolicy state Working add trigger demote action
 **            docPromoteAction input "-method mxMain ${EVENT} ${OBJECTID}" ;

/** docPromoteAction: example java trigger program
  */
import matrix.db.*;
import matrix.util.*;

public class ${CLASSNAME}
{
    public ${CLASSNAME}(Context ctx,String[] args)
    {
    }
    public int mxMain(Context ctx,String[] args)
    {
```

```
        try
        {
            // Declarations
            String event = new String();
            String objId = new String();
            String attrName = new String("docInt");
            MQLCommand mql = new MQLCommand();
            // Get and check arguments arguments
            if (args.length > 0)
                event = args[0];
            if (args.length > 1)
                objId = args[1];
            // Generate notices for bad arguments
            if (event == null || event.equals("")) {
                mql.executeCommand(ctx, "notice 'No event for docPromoteAction'");
            }
            else if (objId == null || objId.equals("")) {
                mql.executeCommand(ctx, "notice 'No objId for docPromoteAction'");
            }

            // Make sure it is a promote/demote event
           if (!event.equalsIgnoreCase("promote") && !event.equalsIgnoreCase("demote"))
                return 0;

            // Construct and open the businessobject
            BusinessObject obj = new BusinessObject(objId);
            obj.open(ctx);

            // Get the current attribute value
            Attribute attrInt = obj.getAttributeValues(ctx, attrName);
            int val = Integer.parseInt(attrInt.getValue());
            if (event.equalsIgnoreCase("promote"))
                val++;
            else
                val--;

            // Update the attribute, and the business object
            attrInt.setValue(String.valueOf(val));
            AttributeList attrList = new AttributeList();
            attrList.addElement(attrInt);
            obj.setAttributes(ctx, attrList);
            obj.close(ctx);
        }
     catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return 0;
    }
}
```

It is recommended that Actions and Checks be configured as promote Triggers, and not as Lifecycle Checks and Actions. Refer to the *Configuration Guide : Triggers* in the online documentation.

Check Program Example

```
/*
 **
 ** docLockCheck: example java trigger program
 **
 ** Note that the macros (EVENT,OBJECTID) required by the program must be explicitly
 ** specified when the program is configured on the policy. The macros are passed in
 ** the 'args' array when the trigger is run.
 **
 ** This trigger reads the docString attribute on the object and compare it to the
 ** current user's name. It will block the event (return 1) if they do not match.
 **
 ** To configure this as a promote and demote action on a policy:
 ** mod type docType add trigger lock check
 **          docLockCheck input "-method mxMain ${EVENT} ${OBJECTID}";
 **
 **
 */
import matrix.db.*;
import matrix.util.*;

public class ${CLASSNAME}
{


    public ${CLASSNAME}(Context ctx,String[] args)
    {
    }
    public int mxMain(Context ctx,String[] args)  throws Exception
    {

        // Initialize return value to "ok"
        int retval = 0;

        try
        {
            // Declarations
            String event = new String();
            String objId = new String();
            String attrName = new String("docString");
            MQLCommand mql = new MQLCommand();

            // Get and check arguments
            if (args.length > 0)
                event = args[0];
            if (args.length > 1)
```

```
            objId = args[1];
        // Generate notices for bad arguments
        if (event == null || event.equals("")) {
            mql.executeCommand(ctx, "notice 'No event for docLockCheck'");
        }
        else if (objId == null || objId.equals("")) {
            mql.executeCommand(ctx, "notice 'No objId for docLockCheck'");
        }

        // Make sure it is a lock event
        if (!event.equalsIgnoreCase("lock"))
            return 0;

        // Construct and open the businessobject
        BusinessObject obj = new BusinessObject(objId);
        obj.open(ctx);

        // Get the current attribute value and compare to current user
        Attribute attrString = obj.getAttributeValues(ctx, attrName);
        String attrUser = attrString.getValue();
        String currentUser = ctx.getUser();
        // If no match, block event
        if (!attrUser.equals(currentUser)) {
            retval = 1;
            String msg = "You are not user " + attrUser;
            System.out.println(msg);
            mql.executeCommand(ctx, "error '" + msg + "'");
        }
        // Close the object
        obj.close(ctx);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        retval = 1;
    }
    return retval;
  }
}
```

It is recommended that Actions and Checks be configured as promote Triggers, and not as Lifecycle Checks and Actions. Refer to the *Configuration Guide : Triggers* in the online documentation.

Creating a Program for Execution as Needed

```
/*
 **
 ** docTableProgram: example java program for use in table or cue
 **
 ** This program returns the number of objects of type "docType" connected to the
 ** current object by the relationship "docRel"
```

```
 **
 ** Note that the macro OBJECTID must be explicitly specified when the program is
 ** configured in a table/cue definition so it can be loaded into the 'args' array
 ** program is executed.
 **
 ** To configure this as a table column:
 ** add table docTable column label Rels
 **             businessobject program[docTableProgram -method mxMain ${OBJECTID}]
 **
 ** To configer this as a cue:
 ** add cue docCue appliesto businessobject
 **                color red
 **                where 'program[docTableProgram -method mxMain ${OBJECTID}] > 0';
 **
 */
import matrix.db.*;
import matrix.util.*;

public class ${CLASSNAME}
{


    public ${CLASSNAME}(Context ctx,String[] args)
    {
    }
    public String mxMain(Context ctx,String[] args)  throws Exception
    {

        // Initialize number of connected objects
        int retval = 0;

        try
        {
            // Declarations
            String objId = new String();
            MQLCommand mql = new MQLCommand();

            // Get and check arguments
            if (args.length > 0)
                objId = args[0];
            if (objId == null || objId.equals("")) {
                mql.executeCommand(ctx, "notice 'No objId for docTableProgram'");
            }

            // Construct and open the businessobject
            BusinessObject obj = new BusinessObject(objId);
            obj.open(ctx);

            // Get the connected objects
            String relType = "docRel";
```

```
            String objType = "docType";
            StringList objSelect = new StringList();
            StringList relSelect = new StringList();
            ExpansionWithSelect exp = obj.expandSelect(ctx,
                                                       relType,   // relationship pattern
                                                         objType,   // type pattern
                                                        objSelect, // selects on objectes
                                                         relSelect, // selects on rels
                                                         true,      // getTo direction
                                                         true,      // getFrom direction
                                                         (short)1);  // levels

            // And return the count of related objects
            RelationshipWithSelectList relList = exp.getRelationships();
            retval = relList.size();

            // Close the object
            obj.close(ctx);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        return String.valueOf(retval);
    }
}
```

Java or External or MQL Clause

You can specify the type of program as `java` or `external` or `MQL`.

A Java Program Object (JPO) is just another type of Program object that contains code written in the Java language. Generally, anywhere a Program object can be used, a JPO can be used. See *Java Program Objects* in Chapter 7 for details.

An external program consists of commands that are evaluated by the command line syntax of the machines from which they will be run. When creating external programs, remember that the commands that you enter will be evaluated at each user's workstation as if they were being typed at the operating system's command prompt. Be sure that the users have the appropriate application files available from their workstation. External program objects can also be defined as "piped," providing a built-in MQL command line service to handle standard input and output. Refer to *Piped Clause* for more information.

An MQL program can be run from any machine with the 3DEXPERIENCE Platform installed; it is not necessary for MQL to be available on the machine. If not specified, `mql` is the default.

```
add program NAME mql;
```

Since MQL can be launched from a command line, MQL code could be specified in an external program. This would spawn a separate MQL session that would run in the background. In this case, MQL would have to be installed on every machine that will run the program.

```
add program NAME external;
```

A Java program has code written in the Java language. A Java program can be run anywhere a program object can be used.

```
add program NAME java;
```

Refer to *Java Program Objects* in Chapter 7 for details.

Execute Clause

Use the `Execute` clause to specify when the program should be executed. If the `Execute` clause is not used, `immediate` is assumed.

`Immediate` execution means that the program runs within the current transaction, and therefore can influence the success or failure of the transaction, and that all the program's database updates are subject to the outcome of the transaction.

`Deferred` execution means that the program is cued up to begin execution only after the outer-most transaction is successfully committed. A deferred program will not execute at all if the outer transaction is aborted. A deferred program failure affects only the new isolated transaction in which it is run (the original transaction from which the program was launched will have already been successfully committed).

For example, to defer the execution of the "Roll up Cost" program:

```
modify program "Roll up Cost" execute deferred;
```

However, there are a number of cases where deferring execution of a program does not make sense (like when it is used as a trigger check, for example). In these cases the system will execute the program immediately, rather than deferring it until the transaction is committed.

There are four cases where a program's execution can be deferred:

- Stand-alone program
- Method
- Trigger action
- State action

There are six cases where deferred execution will be ignored:

- Trigger check
- Trigger override
- State check
- Range program
- Wizard frame prologue/epilogue
- Wizard widget load/validate

There is one case where a program's execution is always deferred:

- Format edit/view/print

A program downloaded to the Web client for local execution (see *Downloadable Clause*) can be run only in a deferred mode. Therefore, if you use the `downloadable` option, program execution is automatically deferred.

Note that the `Usesexternalinterface` clause continues to be supported for historical reasons. Scripts and programs that use this clause result in Program objects which have their `execute` flag set to `deferred` and their `downloadable` flag set.

Execute User Clause

You can assign a user to a program object so when other users execute the program, they get the accesses available to the user specified in the program definition. This access inheritance includes both business and system administrative access, as well as any business object access the context user didn't already have that the program user does.

For example:

```
add program DocActionTrigger execute user bill;
```

Only one user is ever associated with a program. For nested programs, the user's access from the first program is inherited only if the called program has no associated user of its own. If the called program does have a user, then that user's accesses are made available instead. Once the called program returns to the calling program, the latter's user is restored as the person whose accesses are added to the current context. Thus, only one person's accesses are ever added to the current context (not including access granted on a business object).

JPOs and Program User Access

JPO execution has special rules. Consider this sample code for JPO B:

```
public class ${CLASSNAME} extends ${CLASS:A} implements ${CLASS:C}
{
    public int mxMain(Context ctx,String[] args)
    {
      ${CLASS:D} dObject = new ${CLASS:D}(ctx);
      dObject.methodOfD();

      methodOfA(ctx);

      retVJPO.invoke(context, "D", null, "mxMain", null);

_mql = new MQLCommand();
_mql.executeCommand(ctx, "execute program D");
    }
}
```

The program above can be run in any of the following manners:

1. JPO B extends JPO A and a method of A is run on an object of type B as shown by methodOfA.
2. JPO B implements JPO C. References to C objects really execute a different program object.
3. JPO B runs a method of JPO D without using JPO.INVOKE as shown with dObject.
4. JPO B uses JPO.INVOKE to run JPO D.
5. JPO B uses MQLCommand to run "execute program <program name>" as shown with _mql.

In the first 3 cases, execution is handled solely by the JVM, so that the Live Collaboration is never aware of when the methods of another JPO get invoked and returned. In these cases, the user of program B will remain in effect and the users, if any, of programs A, C and D, are ignored. In cases 4 and 5, execution goes through the kernel code and the programs are invoked as program objects, not just Java code by the JVM, and so the usual rules apply.

Needs Business Object Clause

You can specify that the program must function with a business object. For example, you would select this option if the program promotes a business object. If, however, the program creates a business object, the program is independent of an existing object and this option would not apply.

```
add program NAME needsbusinessobject;
```

The selected object is the starting point for any program specified as "needs business object." If a method does not use a business object, the selected object is not affected.

If not set, some macros, including Business Object Identification Macros, are not available.

The `doesneedcontext` selectable is available on programs to determine this setting in an existing program.

The following indicates that a business object is not needed:

```
add program NAME !needsbusinessobject;
```

When defining a type or format, you can specify program information:

- When defining a type, you can indicate any defined programs. Even programs that do not require a business object could be associated with a type in order to make them available to users.
- When defining a format, only the programs defined as “needs business object” are appropriate for the view, edit, and print procedures since the Live Collaboration passes a file from a business object to the program.

Downloadable Clause

If the program includes code for operations that are not supported on the Web product (for example, Tk dialogs or reads/writes to a local file) you can include the `Downloadable` clause. If this is included, this program is downloaded to the Web client for execution (as opposed to running on the Collaboration Server). For programs not run on the Web product, this flag has no meaning.

```
add program NAME downloadable;
```

If the Downloadable clause is not used, `notdownloadable` is assumed.

Due to the restriction that downloaded programs must execute in a deferred mode, there are several cases that need to be addressed by system logic. If just the `Downloadable` clause is given, then `deferred` is assumed (see *Execute Clause*). If the `Downloadable` clause is given, and the `Execute` clause is `immediate`, an error will be generated. Likewise, if on program modification command a mismatch occurs, an error will be generated that reads:

```
A program that is downloaded cannot execute immediately.
```

Note that the `Usesexternalinterface` clause continues to be supported for historical reasons. Scripts and programs that use this clause result in Program objects which have their `execute` flag set to `deferred` and their `downloadable` flag set.

Java Program Objects cannot be downloadable.

Piped Clause

Not used for Java Program Objects.

You can specify that external program objects use the “piped” service. Piped programs can use a built-in MQL command line service to handle standard input and output. When piped is specified, External is assumed. The piped service is not available to MQL or Java program objects. Execution can be immediate or deferred. Piped programs cannot be downloadable.

Pooled Clause

Not used for Java Program Objects.

Each time an MQL program object runs Tcl code and then exits out of Tcl mode, a Tcl interpreter is initialized, allocated, and then closed. During a session, you may execute several programs, and one program may call other programs, all of which require a Tcl interpreter and therefore the overhead of its use. In an effort to optimize multiple Tcl program execution, Tcl program objects may be specified as “pooled.” When such a program is first executed in a session, a pool of Tcl interpreters is initialized, one of which is allocated for the executing code. When the code is completed, the interpreter is freed up. Subsequent Tcl code that is executed in a pooled program during the session will use an interpreter from the already initialized pool.

When programs are created, the default is that they are not pooled. To define or modify an MQL type program to use the pool of interpreters, use the following syntax:

```
mql< > add|modify program PROG_NAME [!]pooled;
```

The “!” can be used to turn off the `pooled` setting of a program.

The number of interpreters available in a session is controlled by the `MX_PROGRAM_POOL_SIZE` setting in the initialization file (enovia.ini). `MX_PROGRAM_POOL_SIZE` sets the initial size of the Tcl interpreter pool. This setting is also used to extend the pool size when all the interpreters in the pool are allocated and another is requested. The default is 10.

Usage

Enabling the Tcl interpreter benefits MQL/Tcl programs that are nested or run in a loop, such as the trigger manager. Also, while wizards do not support the pooled setting, the Tcl programs that make up a wizard (load, validate, etc.) should make use of an interpreter pool to optimize performance.

Unexpected results may occur if the pooled setting is turned on in a program without first reviewing and validating its code. Good programming techniques must be adhered to ensure proper results. When using an interpreter pool, Tcl variables are not cleared before freeing up an interpreter. This means that programs must explicitly set variables before using them, in case a previously executed program made use of the same variable name.

External, downloadable and Java programs do not use the Tcl interpreter pool, regardless of the setting in the program definition. In addition, MQL/Tcl programs that use the TK toolkit will not benefit from using the pooled setting, since user interaction is required. In fact, unexpected results, such as leaving a TK dialog displayed, may occur due to the use of variables as described above.

Before modifying existing programs to use the pooled setting, the code should be reviewed and validated. Only programs that include tcl code but no TK code should use the pooled setting.

Rule Clause

Rules are administrative objects that define specific privileges for various users. The Rule clause enables you to specify an access rule to be used for the program.

```
add program NAME rule RULENAME;
```

History Clause

The `history` keyword adds a history record marked “custom” to the program that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Program

After a program is defined, you can clone the definition with the Copy Program command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy program SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the program definition (source) to copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the program that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Program

After a program is defined, you can change the definition with the Modify Program command. When modifying a program that is used to launch an application, however, consider upward and downward compatibility between software versions.

The following command lets you add or remove defining clauses and change the value of clause arguments:

```
modify program NAME [MOD_ITEM] {MOD_ITEM};
```

- `NAME` is the name of the program you want to modify.
- `MOD_ITEM` is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Program clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Program Clause	Specifies that…
`code CODE`	The current code definition changes to that of the new code entered.
`description VALUE`	The current description, if any, changes to the value entered.
`external|mql|java`	The specification of the program type (external or MQL or Java) changes as entered.
`file FILENAME`	The contents of file are added to the code section of the program.
icon FILENAME	The image is changed tp the new image in the field specified.
`add rule NAME`	The named rule is added.
`remove rule NAME`	The named rule is removed.
`name NAME`	The current name of the program changes to the name entered. Note: If you rename a program, it may become available within certain features. For example, if you rename a program that is part of a toolset, the program will need to be added to the toolset again.
`[!]needsbusinessobject`	The status of the need for a business object changes as indicated here: `needsbusinessobject` is used when a business object is needed. `!needsbusinessobject` (or `notneedsbusinessobject`) is used when a business object is not needed.
`[!|not]downloadable`	The status of downloadable changes as indicated here: `downloadable` is specified when the program includes code for operations not supported on the Web product (for example, Tk dialogs or reads/writes to a local file). `!downloadable` (or `notdownloadable`) is specified when the program does not include code for operations not supported on the Web product.
`[!|not]pipe`	The external program uses the “piped” service or not.
`[!|not]pooled`	The program uses pool interpreters or not.
`execute immediate`	The status of program execution changes so the program runs within the current transaction.
`execute deferred`	The status of program execution changes so the program runs only after the outermost transaction is successfully committed.
`execute user USERNAME`	Assigns a user to the program object such that when other users execute the program, they get the accesses available to the user specified.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.

Modify Program Clause	Specifies that…
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the program that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the arguments that define the program.

property Command

Description

Ad hoc attributes, called *Properties*, can be assigned to an administrative object by business administrators with Property access. Properties allow links to exist between administrative definitions that aren't already associated.

For conceptual information on this command, see *Administrative Properties* in Chapter 7.

User Level

Business Administrator

Syntax

```
[add|list|print|modify|delete]property NAME {CLAUSE};
```

- NAME is the name you assign to the property.
- CLAUSEs provide additional information about the property.

Add Property

Properties can be created and attached to an object at the same time using the `add property` command. A property must have a name and be "on" an object. It can, optionally, define a link to another administrative object using the "to" clause. This command, therefore, takes two forms, with and without the "to" clause. The length of property names are a maximum of 255 characters.

```
add property NAME on ADMIN_TYPE ADMIN_NAME [system] [to
ADMIN_TYPE ADMIN_NAME [system]] [value VALUE];
```

- NAME is the name of the new property.
- ADMIN_TYPE is the keyword for an administrative or workspace object:

association	group	policy	set	vault
attribute	index	program	site	view
command	inquiry	query	store	wizard
cue	location	relationship	table	
filter	menu	role	tip	
form	page	rule	toolset	
format	person	server	type	

ADMIN_NAME is the name of the administrative object instance.

The `to ADMIN_TYPE ADMIN_NAME` is optional.

`system` is used only when adding properties on/to system tables.

`VALUE` is a string value of the property. The “value” clause is optional. The value string can contain up to 2gb of data.

For creation and subsequent identification (modification, deletion, etc.) purposes, a property with a “to” clause is identified by the following arguments:

- the property NAME
- the “on” ADMIN_TYPE ADMIN_NAME
- the “to” ADMIN_TYPE ADMIN_NAME

While a property without a “to” clause is identified by only:

- the property NAME
- the “on” `ADMIN_TYPE` `ADMIN_NAME`

For example, Programs are associated to Formats inherently, since they make up part of the Format definition. But let’s say we want to add a Format property to a Program definition to indicate the type of environment required to execute it. We could add a property to a program as follows:

```
add property Format on Program Perlscript to Format Perl value
yes;
```

A Format property could be added to other Programs as well. And other perl programs would be added “to” the Perl Format. However, the properties are unique in that the “on” object would differ.

Adding Properties to Administrative Definitions

A property can be added to an administrative object in three ways:

- It can be added using the property command, as shown above.
- In addition, a property can be added when an administrative object or workspace object is created, using the property clause with the add command.
- A third way is within the modify command for administrative/workspace objects.

```
add ADMIN_TYPE ADMIN_NAME add property NAME [to ADMIN_TYPE ADMIN_NAME] [value
VALUE];
```

```
modify ADMIN_TYPE ADMIN_NAME add property NAME [to ADMIN_TYPE ADMIN_NAME] [value
VALUE];
```

For example, the following are equivalent to the command given above:

```
add program Perlscript add property Format to Format Perl value yes;
```

```
modify program Perlscript add property Format to Format Perl value yes;
```

Adding Properties to User Workspace Items

Properties can be added to the logged on user’s personal workspace objects. For example, the following adds the date property to the set MYSET:

```
add property Date to set Myset value 4/19/99;
```

Workspace objects include filters, cues, queries, tips, tables, sets, toolsets, and views.

List Property

Properties can be listed with the `list property` command, which takes the following forms:

```
list property [system] [on ADMIN_TYPE ADMIN_NAME];
```

```
list property [system] [user person USER_NAME];
```

```
list property [system] [user all];
```

The `System` option is used to list the system-defined properties. Without it, only user-defined properties are listed.

The user option is for workspace properties. If `all` is indicated as the user, all properties on all Persons' workspace items will be displayed. USER_NAME is the name of the person.

The following should be noted:

- All properties on an administrative object or on a user's workspace objects can be listed.
- Currently only person users have workspace objects.
- The On and User clauses cannot be used together.
- The `all` keyword goes with the User clause—either a Person or `all` is specified.

To list all user properties

```
list property;
```

This will list all user properties on all non-workspace objects.

To list both system and user properties

```
list property system;
```

To list user properties on an administrative object

```
list property on ADMIN_TYPE ADMIN_NAME;
```

`ADMIN_TYPE ADMIN_NAME` here could be a workspace object (table, set, tip, etc.) of the current user, or a non-workspace object.

To list all properties of a Person's workspace objects

```
list property system user person USER_NAME;
```

Print Property

The properties of administrative objects are selectable, using the following syntax:

```
print ADMIN_TYPE ADMIN_NAME select property;
```

The above will list all user properties associated with the specified administrative object, including their name, their “to” object, and their values. To further refine the list you can also select the following:

```
property.name
property.value
property.to
```

For example:

```
MQL<12>print program Perlscript select property;
program   Perlscript
    property = Format to Format Perl value 4
MQL<13>print program Perlscript select property.name;
program   Perlscript
    property[Format].name = Format
MQL<14>print program Perlscript select property.value;
program   Perlscript
    property[Format].value = 4
MQL<15>print program Perlscript select property.to;
program   Perlscript
    property[Format].to = Format Perl
```

Modify Property

The value of a property can be modified using either the `modify property` or `modify ADMIN` commands. The length of property names are a maximum of 255 characters.

```
modify property NAME on ADMIN_TYPE ADMIN_NAME [to ADMIN_TYPE ADMIN_NAME] [value
VALUE];
```

```
modify ADMIN_TYPE ADMIN_NAME property NAME [to ADMIN_TYPE ADMIN_NAME] [value
VALUE];
```

This command will create the property if it does not exist or will modify its value if it does. If other changes are required (for example, changing any ADMIN values) the property should be deleted and redefined.

Delete Property

Properties can be deleted with either of the following commands:

```
delete property NAME on ADMIN_TYPE ADMIN_NAME [to ADMIN_TYPE ADMIN_NAME];
```

```
modify ADMIN_TYPE ADMIN_NAME remove property NAME [to ADMIN_TYPE ADMIN_NAME];
```

query Command

Description

A *query* is a search on the database for objects that meet the specified criteria. The query is formulated by an individual and, in MQL, it must be saved for subsequent evaluation. A user has access only to queries created during a session under her or his own context. It is then run or *evaluated* and the Live Collaboration finds the objects that fit the query specification. The found objects appear on the screen or are listed in MQL. If the found objects are often needed as a group, they can be saved in a set which can be loaded at any time in the session or MQL session under the same context.

For conceptual information on this command, see *Queries* in Chapter 6.

Queries that might hit foreign vaults should be enclosed in transactions (start trans; ...; end trans;) to prevent locking problems.

User Level

System Administrator

Syntax

```
[add|temporary|evaluate|copy|modify|delete] query NAME {CLAUSE};
```

- NAME is the name of the query you are defining. The query name cannot include asterisks.
- CLAUSEs provide additional information about the query.

In a query, the select expression value is used to qualify the search criteria (in a Where clause) by comparing it with another (given) value. See *Modify Index*.

The backslash (\) character can be used in MQL commands or expressions to escape any other character in a variety of contexts. In particular, MQL users can create attribute values that contain both single and double quotes, and search for objects with an attribute value of this sort. See *Using the Escape Character*.

Add Query

To define a saved query from within MQL, use the Add Query command:

```
add query NAME [user USER_NAME] {ITEM};
```

NAME is the name of the query you are defining. The query name cannot include asterisks.

USER_NAME can be included with the user keyword if you are a business administrator with person access defining a query for another user. If not specified, the query is part of the current user's workspace.

When assigning a name to the saved query, you cannot have the same name for two queries. If you use the name again, an error message will result. However, several different users could use the same name for different queries, since queries are local to the context of individual users.

For example, several users could each have a query called Current Project. But the contents of each Current Project query may differ from user to user depending on the individual needs of each person. As the user adds business objects regarding Current Project, contents may change also. If you change context from one user to another, evaluating the Current Project query will most likely produce different results.

{Item} defines what you are searching for. You can include any or all of the following:

businessobject TYPE_PATTERN PATTERN REVISION_PATTERN
[!|not]hidden
owner PATTERN
vault PATTERN
[!|not] expandtype
visible USER_NAME{,USER_NAME};
where QUERY_EXPR
property NAME [to ADMINTYPE NAME] [value STRING]

None of these clauses is required. The default is an asterisk (*), indicating that the query will find all business objects in the database.

Most of these clauses use PATTERN rather than NAME values. This offers greater flexibility in specifying possible name values. Patterns enable you to use wildcard characters and to list multiple values when specifying a name.

The most commonly used wildcard character in queries is the asterisk (*). If only an asterisk is used for the PATTERN, all definitions for that field will be searched. When an asterisk is inserted into a name, it acts as a substitute for a group of letters. This group may contain many letters or none. For example, if you specify a business object name as Ca*, you might find objects named Catalogue93, CadDrawingA49, CaseLog, Ca668, etc. All objects that begin with the letters “Ca” are searched for and any that are found appear in a list. If you enter a value of dr*t, you will find all objects whose names begin with “dr” and end in “t.”

In this sample query definition below, all objects that start with the letters A, B, and C are searched for. Each of the other clauses uses only an asterisk for a value. The search includes all vaults and all owners. To restrict the search further, you could inclsude specific values in those clauses.

```
add query “Name Search”
   businessobject * A*,B*,C* *
   vault *
   owner *;
```

The first and last asterisks (*) in the Businesobject clause indicate that all types and revisions should be included.

The following sections explain how each Add Query clause is used to filter out and locate desired business objects.

Businessobject Clause

This clause searches for business objects with a particular or similar name.

```
businessobject TYPE_PATTERN PATTERN REVISION_PATTERN
```

- `TYPE_PATTERN` is a value that translates into one or more business object types.
- `PATTERN` is a value that represents a business object name.
- `REVISION_PATTERN` is a value that translates into one or more revision designators.

For example, the following is a valid Businessobject clause:

```
businessobject Customer Tre*,How* *
```

The first value is an actual object type name: Customer. The second value uses wildcard characters, multiple values, and character strings.

A user might search for a customer named Howard Trevor. But the user is unsure of the name spelling and does not know if the customer is stored as Howard or Trevor. By specifying the object name with this pattern, all object names that begin with the letters “Tre” or “How” are searched for. If any are found that are of type Customer, they are listed. The final asterisk indicates that all revisions are allowed.

When listing multiple values as part of a pattern, you cannot have spaces within the pattern unless the spaces are enclosed within quotation marks. If you accidentally include spaces, the search may read the value as the next part of the object specification. For example, the following clause would produce false values:

```
businessobject Customer Tre*, How* *
```

When this clause is processed, Customer is again used for the object type. However, rather than searching for names that begin with “Tre” or “How,” The search is only for objects whose names begin with “Tre”. How* is interpreted as the revision pattern, not part of the name pattern.

For a complete listing of all defined business objects regardless of their exact specification, you can simply insert an asterisk into each pattern of the Businessobject clauses:

```
businessobject * * *
```

Since this clause could produce a large number of objects, it is usually desirable to restrict the query searches in some way. Rather than use an asterisk for each pattern, you may want to use an asterisk in only one or two of the required patterns.

The goal is to provide enough information to filter out unwanted objects. However, you do not want to make your search too narrow or you might miss an important object.

Vault Clause

This clause searches for business objects that are in a particular or similar vault:

```
vault PATTERN
```

- `PATTERN` is the vault(s) you are searching for.

The Vault clause uses wildcard characters in a way similar to the Businessobject clause.

For example, the following query definition requests all business objects that reside in the “Vehicle Project” vault:

```
add query “Vault Search”
   businessobject * * *
   vault “Vehicle Project”;
```

Using an asterisk in the Vault clause searches only local and Foreign (Adaplet) vaults.

Expandtype Clause

This clause is used to find all the specified type hierarchy of types. This is the default.

For example, a type of Frame Assembly may have derived types of Handle and Guard. To limit the search to objects with a type of Frame only, use `!expandtype`:

```
add query “Frame Assembly”
   businessobject Frame * *
   !expandtype;
```

Temporary Query

You can perform a temporary query that is not first named or saved within the MQL database. In other words, a query can be evaluated without first adding it to the database as an object. The syntax is as follows:

```
temporary query businessobject TYPE NAME REV
[!expandtype]
[vault VAULTNAME]
[owner USERNAME]
[limit VALUE]
[querytrigger]
[orderby FIELD_NAME]
[where QUERY_EXPR]
[select]
[size N]
[dump "SEPARATOR_STR"]
[recordseparator "SEPARATOR_STR"]
[tcl]
[output FILENAME];
```

For example, to find all business objects in a small database use the following:

```
temporary query bus * * *;
```

To find all Assemblies and Assembly subtype objects, use:

```
temporary query businessobject Assembly * * expandtype;
Or
temporary query businessobject * * * expandtype where type==Assembly;
```

The `Owner`, `Vault`, and `Where` clauses can be used with or without `businessobject`. For example, you could find all business objects owned by user cslewis, or all business objects of type Part owned by user cslewis:

```
temporary query owner cslewis;
temporary query bus Part * * owner cslewis;
```

Use the `Limit` clause to control the number of items returned in the search. The system will stop searching after it has reached the specified number of items.

Use the `tcl` clause after the `Dump` clause, if used, and before the `Output` clause to return the results in Tcl list format. This facilitates the parsing of output from MQL `select` commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see *Tcl Clause*.

Using a size of 0 is the same as not using the Size clause; the query is streamed using the value of MX_QUERY_PAGE_SIZE, if set. For information about query page size, refer to the *Installation Guide : Optional Variables*.

Include the `Querytrigger` clause when you want a program named ValidateQuery to be executed. Even with triggers turned off, when `querytrigger` is included in a query command, the ValidateQuery program gets run. Refer to the *Configuration Guide : Triggers* in the online documentation.

Orderby Clause

Use the Orderby clause to specify a selectable for sorting. You can prefix the field name with a "+" to sort in ascending order or "-" to sort in descending order. If no prefix is specified, the results are sorted in ascending order.

Sorting is generally done inside the database, but some sorting uses memory. Thus, when sorting query results, it is necessary to have a larger amount of memory swap space since a large number of business objects and relationships might be brought into memory at the same time.

The results of saved queries cannot be sorted.

When using the Orderby clause, it must precede any Select clause criteria in the query. You can use a maximum of three Orderby clauses in a temporary query or in a query connection command to specify more than 1 sort criteria.

If using the Orderby clause it must precede any Select clause criteria in the query.

For example:

```
 temp query bus Part Meta* * orderby -type orderby name
orderby +Color select attribute[Color];
```

String attributes that may contain values of more than 255 characters must be defined as multi-line attributes, or sorting on such an attribute will fail.

Size Clause

Use the Size clause in a temp query or expand bus command to enable streaming in the query or expand and return the specified number of objects per page. When you enable streaming, data is returned when the first page is available, resulting in improvement in performance and memory consumption.

Only results from local vaults are streamed.

When streaming is used, no data is held in cache. Therefore repeating the same operation in a transaction can result in the system issuing the same sql database requests each time.

For example, to find all business objects in a small database using the streaming capability, use the following:

```
temporary query bus * * * size 100;
```

This will return all business objects in the database with one hundred entries listed in a page.

Streaming can be enabled for all temporary queries (and expands) through the MX_QUERY_PAGE_SIZE environment variable, through classes in the Studio Customization Toolkit, or at runtime by using the size clause in MQL temp query or expand bus commands.

Using a size of 0 is the same as not using the Size clause and the expand is streamed using the value of MX_QUERY_PAGE_SIZE or the classes in the Studio Customization Toolkit, if set. For information on MX_QUERY_PAGE_SIZE, see the *Installation Guide : Optional Variables*. The Size clause overrides any other settings.

Enabling streaming through the Java classes usually provides the greatest improvement in performance and memory consumption.

When processing selectables in a streaming query, memory consumption is not necessarily reduced versus not using streaming. Although paging is used when the selectables are processed, the entire result set is still buffered in memory. For example:

```
temp query bus T N R size 1000 select to.from;
```

This retrieves business objects 1000 at a time, and only 1000 will ever be held in memory at one time. However, the evaluation of to.from for ALL of the found objects will be held in memory until the query has completed.

`temp query select` output is the same as `print bus` or `set select`, unless the `dump` keyword is used. Note the following examples:

Example 1. Temp query without dump keyword:

```
MQL<7>temp query bus Note *e* * select owner;
businessobject Note BingTest 1
    owner = creator
businessobject Note CapTest 1
    owner = creator
businessobject Note attrtest 1
    owner = creator
businessobject Note attrtest1 1
    owner = creator
businessobject Note attrtest2 1
    owner = creator
```

Example 2. Temp query with dump keyword:

```
MQL<8>temp query bus Note *e* * select owner dump;
Note,BingTest,1,creator
Note,CapTest,1,creator
Note,attrtest,1,creator
Note,attrtest1,1,creator
Note,attrtest2,1,creator
```

Example 3. Print set select without dump keyword:

```
MQL<11>print set seltestset select owner;
set seltestset
  member businessobject Note BingTest 1
    owner = creator
  member businessobject Note CapTest 1
    owner = creator
  member businessobject Note attrtest 1
    owner = creator
  member businessobject Note attrtest1 1
    owner = creator
  member businessobject Note attrtest2 1
    owner = creator
```

Example 4. Print set select with dump keyword:

```
MQL<12>print set seltestset select owner dump;
creator
creator
creator
creator
creator
```

Evaluate Query

Once a query is defined, you need to evaluate it, using the Evaluate Query command, to find the information.

```
evaluate query NAME;
```

When a query is evaluated, all business objects that meet the search criteria are displayed in the window or listed on your screen. If you want to save this collection of objects, you can assign a set name to it and reference the set name when you want to view the collection. (Refer also to *Sets* in Chapter 6.)

Saving query results as a set can be useful when you have a changing environment. Even when you use the same query, it is possible that you would get different results if the objects are undergoing change. Therefore, to save query results for a later time, you should place them in a set.

Use the Evaluate Query command to process the query and optionally save the results of a query in a set. This command contains the following optional clauses:

```
into set SET_NAME
onto set SET_NAME
over set SET_NAME into|onto
querytrigger
```

QUERY_NAME is the name of the query to be used.

SET_NAME is the name to be assigned to the collection of objects created by the query.

INTO form:	If the named set exists, the set is cleared and the results of the current query is placed into the set. If the named set does not exist, the set is created and the results are placed into it.
ONTO form:	If the named set exists, the results of the current query are added to the set contents. If the named set does not exist, the set is created and the results are placed into it.
OVER form:	Performs a find on an existing set.
querytrigger	Executes the program named ValidateQuery, even if triggers are turned off. See the *Configuration Guide : Validate Query Trigger* for more information.

If the INTO or ONTO form of the Evaluate Query command is used, the found set is not listed on the screen. To view them, you must print the set. If you only evaluate the query and do not save onto or into a set, the found values are listed on the screen. For example:

```
add query sarah where 'type==drawing';
print query sarah;
query sarah
   businessobject * * *
   vault *
   owner *
   where 'type==Drawing'
eval query sarah;
Drawing test A
Drawing ttest A
Drawing 726602 A
Drawing 726601 A
Drawing 726600 A
Drawing 726596 B
Drawing 726595 A
Drawing 726594 A
Drawing 726593 A
Drawing 726592 A
Drawing 726591 C
Drawing 726590 B
Drawing 50234 F
Drawing 50225 D
Drawing 50461 B
Drawing 50023 F
Drawing 50403 B
eval query sarah into set sarah;
print set sarah;
set sarah
```

```
member businessobject Drawing test A
member businessobject Drawing ttest A
member businessobject Drawing 726602 A
member businessobject Drawing 726601 A
member businessobject Drawing 726600 A
member businessobject Drawing 726596 B
member businessobject Drawing 726595 A
member businessobject Drawing 726594 A
member businessobject Drawing 726593 A
member businessobject Drawing 726592 A
member businessobject Drawing 726591 C
member businessobject Drawing 726590 B
member businessobject Drawing 50234 F
member businessobject Drawing 50225 D
member businessobject Drawing 50461 B
member businessobject Drawing 50023 F
member businessobject Drawing 50403 B
```

Copy Query

You can modify any query that you own, and copy any query to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, business administrators can change their workspace to that of another user to work with queries that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

After a query is defined, you can clone the definition with the copy query command.

If you are a Business Administrator with person access, you can copy queries to and from any person's workspace (likewise for groups and roles). Other users can copy visible queries to their own workspaces.

This command lets you duplicate query definitions with the option to change the value of clause arguments:

```
copy query SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the query definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any query of the same name belonging to the user specified in the `Touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Query

You change the search criteria for an existing query by using the Modify Query command:

```
modify query NAME [user USER_NAME] {ITEM};
```

NAME is the name of the query you want to modify. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.

ITEM is the type of modification you want to make. With the Modify Query command, you can use these modification clauses to change a query:

businessobject TYPE_PATTERN PATTERN REVISION_PATTERN
owner PATTERN
vault PATTERN
[!|not]expandtype
visible USER_NAME{,USER_NAME}
where PATTERN

These clauses are the same clauses that are used to define the initial query. When making modifications, you simply substitute new values for the old.

Although the Modify Query command allows you to use any combination of search criteria, no other modifications can be made. To change the query name or remove the query entirely, you must use the Delete Query command (see *Delete Query*) and/or create a new query.

For example, assume you have a query named "Product Comparison" with the following definition:

```
query "Product Comparison"
   businessobject *
   revision *
   type Perfume
   vault "Perfume Formulas"
   owner channel, taylor;
```

To this query, you want to add another owner for the search criteria. To make the change, you would write a Modify Query command similar to the following:

```
modify query "Product Comparison"
   owner channel, taylor, cody;
```

This alters the query so that it now appears as:

```
query "Product Comparison"
   businessobject *
   revision *
   vault "Perfume Formulas"
   owner channel, taylor, cody
```

Using Select Clauses in Queries

The purpose of select expressions is to obtain or use information related to a particular business object. In a query, the select expression value is used to qualify the search criteria (in a Where clause) by comparing it with another (given) value.

Obtainable information includes not only attribute values and other business object data, but also administrative object information, such as the governing policy, vault, and so on. The key property of a select expression is that it can access information *related* to an object.

In all cases, the expression is processed from the context of a starting object. In a query, the starting points are business objects that meet other selection criteria (vault, type, and so on). The phrase *starting point* is used because the select mechanism actually uses the same concept of navigation from one object to another that makes the rest of the system so flexible. This is possible because most information is actually represented internally by a small object and not by a text string or numeric value as it appears to the user.

These internal objects are all linked in much the same way business objects are connected by relationships. The links can be traversed to travel from one object to another (navigation). The presence of these links is indicated in the select expression notation by a period.

The following sections provide examples and information about select expressions for queries in MQL.

Definitions

- A Select clause is single-valued if `print bus T N R select <clause> dump` outputs exactly one line of text.
- A Select clause is multi-valued if `print bus T N R select <clause> dump` outputs multiple lines of text.
- A Select clause is NULL if `print bus T N R select <clause> dump` outputs zero lines of text.

Using the Format File Dump Clause

Assume you have the following four objects:

- `Assembly A` has a file checked into format Assembly
- `Assembly W` has file checked into format Word
- `Assembly AW` has a file checked into each format: Assembly and Word
- `Assembly NONE` has no files checked in.

Single-valued—Each single-valued Select clause produces a single field of output. Select clauses that do NOT allow square brackets are all single valued. Selecting attributes are a special case; they allow 0 or 1 value to be output according to whether the attribute is present on the business object. In keeping with the desire to have the number of outputs equal the number of Select clauses, attribute selects output an empty field in the case where the attribute does not exist on the business object.

However, other clauses that allow square brackets are format, state, relationship, to, from, revisions, history, and method. These are used in two ways:

To check for existence. For example:

```
relationship[BOM];
```

```
format[ASCII];
```

```
format[ASCII].hasfile.
```

In these cases, the number of outputs is not ambiguous, and a true or false value is always returned.

To get subfields. For example:

```
relationship[BOM].to.name;
```

```
from[].to.name format[ASCII].file.
```

These may represent zero, one or many pieces of data, depending on the number of relationships, formats, files, etc. that are possessed by the business object, so it is not possible to guarantee that the number of outputs will equal the number of Select clauses. Therefore, selecting subfields will produce output fields only for data that is actually present. They do NOT output empty fields to represent the absence of data.

If you enter the single-valued Print Businessobject command for Assembly A, as follows:

```
print bus Assembly A '' select format[ ].file dump;
```

the results indicate a Word document is included in the object Assembly A:

```
andersen:D:\test\Monitor FSP.doc:
```

If you enter the single-valued Print Businessobject command for Assembly W

```
print bus Assembly W '' select format[ ].file dump;
```

the results indicate a Word document is included in the object Assembly W:

```
andersen:d:\test\Monitor FSP.doc:
```

Multi-valued—If you enter the multi-valued Print Businessobject command for Assembly `AW`:

```
print bus Assembly AW '' select format[ ].file dump;
```

the results indicate two Word documents are included in the object Assembly AW, separated by a comma:

```
andersen:d:\test\select.txt,
andersen:d:\test\Monitor FSP.doc:
```

NULL-valued—If you enter the NULL valued Print Businessobject command for Assembly NONE:

```
print bus Assembly NONE '' select format[ ].file dump;
```

the results are NULL.

Using the Format Hasfile Dump Clause

If you use the `format[].hasfile Dump` clause instead of the `format[].file Dump` clause, the value of `TRUE` is returned for each document included in the object.

If you enter the Print Businessobject command with the `Hasfile` clause for Assembly A

```
print bus Assembly A '' select format[ ].hasfile dump;
```

the TRUE response indicates that there is a document included in the object Assembly A, without its file details.

If you enter the Print Businessobject command with the Hasfile clause for Assembly W

```
print bus Assembly W '' select format[ ].hasfile dump;
```

the TRUE response indicates that there is a document included in the object Assembly W, without its file details.

If you enter the Print Businessobject command with the Hasfile clause for Assembly AW

```
print bus Assembly AW '' select format[ ].hasfile dump;
```

the TRUE,TRUE response indicates there are two documents included in the object Assembly AW, without their file details. One TRUE is displayed for each document.

If you enter the Print Businessobject command with the Hasfile clause for Assembly NONE

```
print bus Assembly NONE '' select format[ ].hasfile dump;
```

the NULL response indicates that there are no documents included in the object Assembly NONE.

NULL Clauses

Select clauses that are found to be NULL have special handling for the *equal* and *not equal* logical operators: ==, ~~, ~=, !=, !~~, and !~=.

- A NULL Select clause is NEVER equal (==, ~~, ~=) to anything
- A NULL Select clause is ALWAYS not equal (!=, !~~, !~=) to everything.

So, for our example, we have:

```
temp query bus Assembly  * '' where 'format[Assembly].hasfile==TRUE';
```

results in:

```
Assembly A
Assembly AW
```

and then:

```
temp query bus Assembly  * '' where 'format[Assembly].hasfile != TRUE';
```

results in:

```
Assembly NONE
Assembly W
```

Multi_valued Select Clauses

Multi-valued Select clauses are handled as a string of OR's. That is, each of the multiple values is used separately to evaluate the boolean expression. If any one of these single-valued comparisons are TRUE, the whole multi-valued comparison is considered TRUE.

For example, consider these objects:

- Assembly A contains an ASCII file:

```
format.file = d:\doc\select.txt
```

- Assembly AW contains an ASCII file and a Word document
  ```
  format.file = d:\doc\select.txt
  format.file = d:\doc\specification.doc
  ```
- Assembly W contains a Word document
  ```
  format.file = d:\doc\specification.doc
  ```
- Assembly DELETED contains no files.
- Assembly NONE contains no files.

If you enter:

```
temp query bus Assembly * * where ' "format.file" MATCH "*.doc" ';
```

it results in:

`Assembly AW` (multi-valued, and 2nd one is a match)
`Assembly W`

If you enter:

```
temp query bus Assembly * * where ' "format.file" MATCH "*.txt" ';
```

it results in:

`Assembly A`
`Assembly AW` (multi-valued, and 1st one is a match)

Note that using NMATCH will also pick up the objects with no files. If you enter:

```
temp query bus Assembly * * where ' "format.file" NMATCH "*.txt" ';
```

it results in:

`Assembly AW` (multi-valued, and NMATCH is TRUE for the 2nd one)
`Assembly W` (singlevalued, and NMATCH is TRUE)
`Assembly DELETED` (NULL, so NMATCH is always TRUE)
`Assembly NONE` (NULL, so NMATCH is always TRUE)

Using Fromset and Toset Selectables

Two selectables are available for business objects, `fromset[]` and `toset[]`, that make it possible to obtain information about the relationships from or to a given object if they have an object from a specified set at the other end. In particular, they can be used in `Where` clauses of queries as a way to specify that an object be returned only if it is at the “to” or “from” end of a specific relationship having an object of a given set at the other end. The use of these keywords in this manner solves problems of functionality and performance that are difficult, if even possible, to solve any other way.

Suppose the following query is run:

```
temp query bus Part * * where ((current == Approved) &&
  (attribute[Material Category] ~~ 'Plastic') &&
  (to[Component Substitution].from.name ~~ '*Clutch*') &&
    (to[Component Substitution].from.name ~~ '*Transmission*'));
```

The way the system runs such a query is that it would first find all objects that matched this query:

```
temp query bus Part * * where ((current == Approved) &&
   (attribute[Material Category] ~~ 'Plastic'));
```

Then the system would test the truth of the remaining clauses against each object found from the first query. This method has the potential of being extremely slow. There may be a large number of objects returned that have to be checked against the remaining clauses, and few of them might test true. Furthermore, the work needed to test the remaining clauses can be very intensive. All the to-relationships of an object must be obtained and then the name of the object at the other end must be tested until a match is found. Since the objects at the other ends can live in different vaults, a single join cannot accomplish this goal, so multiple SQL commands are needed.

These performance issues can be avoided by using the `toset[]` selectable. Two MQL commands are required instead of one, so you must perform the query in a program. The two commands would be:

```
temp query bus * "*Clutch*,*Transmission*" * into set t1;
temp query bus Part * * where ((current == Approved) &&
   (attribute[Material Category]~~'Plastic') &&
   (toset[t1,Component Substitution] == True);
```

The first command finds all objects that satisfy the conditions that the “objects at the other end” needed to satisfy. In this example, these conditions have been replaced by a clause that uses `toset[]`. These selectables, `toset[]` (and `fromset[]`), must include brackets that contain a set name followed, optionally, by a relationship type name. Multiple relationship names can be given, each separated by a comma from the previous one. In the above example, `toset[]` returns True if there is a relationship from an object in the set `t1` of type Component Substitution to the object being tested. If no relationship type is specified, the query returns True as long as some relationship exists between such objects, regardless of type. The `fromset[]` selectable works the same as `toset[]` except that the ends are reversed—the relationship must be *to* an object in `t1` and *from* the object being tested.

With `toset` and `fromset` selectables, the values “True” and “False” are case sensitive and always appear in title case (initial capital letter followed by lower case letters). In queries with these selectables, you must type “True” or “False” using this case to get a valid result.

What distinguishes this query from the single-command query is that the `toset` condition can be included with the other conditions when objects are gathered so that only objects that satisfy all the conditions of the query are displayed. As long as the first query does not put too many objects into set t1, this query should have much better performance.

The `fromset` and `toset` selectables can be followed by additional selectables, similar to the way that the selectables `from` and `to` work. In such cases, the selectable that follows is evaluated against each relationship that satisfies the condition indicated by the `fromset` or `toset` selectable. For example, `fromset[t1,assembly,component].name` returns a list of the names of the relationships of type `assembly` or `component` from a given object to an object in set `t1`.

Additionally, these selectables can be used to express conditions that cannot be expressed in a single query. Using the example above, the following conditions:

```
(to[Component Substitution].from.name ~~ '*Clutch*')
```

and

```
(to[Component Substitution].from.name ~~ '*Transmission*')
```

each express that a given object be the to-object in a relationship of type Component Substitution where the object at the other end satisfies a given condition. These two conditions can be true even if no one object at the other end satisfies both conditions. But it may be intended that the two conditions be satisfied by the same object. This query does not express that, and no single query can. However, `toset[]` can be used to express this condition by making sure that the set in question contains only objects that satisfy both conditions. This goal is accomplished by replacing the command that created the set by this command:

```
temp query bus * "*Clutch*" * where "name match '*Transmission*'" into
set t1;
```

Sets are workspace objects and are shared by anyone using the same login name. If two people (or applications) are logged in with the same user name, they may overwrite each other's sets if the same names (t1, t2, etc.) are used. To avoid this, developers should wrap the creation of the set and the final query inside a transaction boundary. This will keep the set from becoming visible to other sessions until it is no longer needed.

`kindof` selectable for types

When a very large/deep type hierarchy exists in the schema, using the type field generally causes problems generating SQL on the Oracle side. In this case, the `kindof` selectable can be used to force Live Collaboration to do the work instead of Oracle. It is not an SQL convertible expression.

`kindof` *may not be suitable for use in all schemas. It is designed for use in large and deep type hierarchies, where specifying TYPE in a query is inefficient.*

`type.kindof[TYPE_IN_HIERARCHY]` can do the following:

- Get all children of the specified parent type
- Get the parent of the specified child type
- Resolve a parent/child relationship to TRUE or FALSE

For example, a Document might be a parent type having children HRForm, Purchase Request, and Proposal, (all derived from Document). You can search for the children types of Document using the following query syntax:

```
MQL<> temp query bus * * * where "type.kindof[Document]";
```

Or

```
MQL<> temp query bus * * * where "type.kindof[Document]==TRUE";
```

Both queries above result in a listing of all HRForm, Purchase Request, and Proposal objects. Any subtypes of these, such as EmploymentHistory form will also be returned.

You can also use “kindof” with print commands. When the parent name is specified in brackets, the field evaluates to true or false, depending on whether the examined type is a subtype of the type in brackets. The example below resolves the child/parent relationship of Proposal/Document to TRUE:

```
MQL<> print type Proposal select kindof[Document];
business type   Proposal
    kindof[Document] = TRUE
```

If not passed a name in brackets, “kindof” evaluates to the name of the type’s base class, as shown below:

```
MQL<> print type Proposal select kindof;
business type   Proposal
    kindof = Document
```

Below is the syntax for using “print bus” and specifying type, name, and revision:

```
print bus Proposal WebDesign 1.0 select type.kindof[Document];
business object Proposal WebDesign 1.0
    type.kindof[Document] = TRUE
```

and

```
print bus Proposal WebDesign 1.0 select type.kindof;
```

```
business object Proposal WebDesign 1.0
    type.kindof = Document
```

Using the Escape Character

The backslash (\) character can be used in MQL commands or expressions to escape any other character in a variety of contexts. In particular, MQL users can create attribute values that contain both single and double quotes, and search for objects with an attribute value of this sort.

When escaping is enabled, the character that follows a backslash loses any special meaning it might have in the particular context.

You can specify that a backslash (\) should be treated as an escape character for any other character when used in:

- MQL commands
- expressions, such as used in the following:
  - Where clauses (including queries, expand businessobject command, cues, tips, and filters)
  - object tip definitions
  - table column definitions
  - evaluate expression command
  - expression access
- output of a range program
- program arguments, such as appear in these contexts:
  - execute program command
  - execute businessobject command
  - triggers
  - various places in wizard definitions
- combo and list boxes in wizards

This can be specified globally or on a case-by-case basis (except for wizard components).

To Enable Escaping

To enable the use of the backslash as an escape character globally, Business Administrators can use the following MQL command:

```
set escape on;
```

Other commands available are:

```
set escape off;
```

```
print escape;
```

- When `escape` is set to `on`, a backslash will always act as an escape character.
- Use `print escape` to check whether it is enabled or not.
- Use `set escape off` to disable it.

The escape status is determined the first time an application is started. When setting escape, all newly started applications will use the setting, but any applications that were already started will not, including the session that made the setting. Business Administrators should decide which way to set escape and set it once.

If you want to avoid enabling the escape character globally, you can enable it on an as needed basis, prefixing each relevant string to be parsed with the keyword `escape` plus a space. When processed on the command line, these extra characters are first stripped off and the remainder will be processed in the usual way.

When more than one expression is specified in a command or definition, (for example, in tip definitions and `expand bus where` commands) you can escape one and not the other by including the Escape clause in only the expression that needs it.

Regardless of whether escaping is enabled or disabled, expressions using the `match` *keyword always treat '*' and '?' characters as wildcards and those using the is equal to "==" or does not equal "!=" keyword always treat the '*' and '?' characters as literals. In other words, escaping does not allow an override of this behavior.*

How It Works

When an MQL command is processed by the system, the system first breaks the command into parts that are called tokens. When escape processing is on for the command (whether by the global or ad-hoc setting), it affects the breakdown of the command into tokens. An expression embedded in a command is treated as one token, from the point of view of the command parser. Prefixing a command with the word "escape" affects only the processing of the command into tokens. For example the following would not work as required:

```
temp query bus * * * where "attribute[height]=="5'"";
```

Without using the escape mechanism and backslashes, this `Where` clause would be processed as:

```
attribute[height]==
```

The text after the equal sign would be ignored, since the second quotation mark (before the 5) seems to indicate the end of the `Where` clause (`Where` clauses must be enclosed in single or double quotes). To correctly search for objects that have a height of five feet, you would use:

```
escape temp query bus * * * where "attribute[height]==\"5'\"";
```

With the backslashes and escape processing turned on, the `Where` clause of this command is correctly processed as:

```
attribute[height]=="5'"
```

In some cases, however, escape processing is needed for both parsing the command into tokens and also for parsing the `Where` clause for evaluation. The `Where` clause parsing is done after the command is parsed as a second pass. The rules for this parsing are somewhat different, as expressions have their own syntax and special keywords. For example, to find all objects that have a height of 5'6" is tricky, since the value itself contains both single and double quotes and must also be surrounded by quotes. In this case you would need to escape process the `Where` clause (and include the escape character) in order to make it work as a `Where` clause, as follows:

```
escape attribute[height]=='5\'6"'
```

To perform the query, you would then need to use escape processing at the command level as well as at the expression level. You could use the following:

```
escape temp query bus * * * where "escape
attribute[height]=='5\\'6\"'";
```

Notice that you must escape the backslash so that one would be in place after the initial escape processing of the command, for the second pass that escape processes the `Where` clause. A triple backslash is required if the character used to enclose the expression is included in the expression itself, such as:

```
escape temp query bus * * * where 'escape
attribute[height]==\'5\\\'6"\'';
```

In summary, escapes are needed when you want to place a value within a string that will be parsed; in order to have the string parsed as one token, you need to place quotes around it. Without escape processing, this only works if the string does not itself contain the quote character. With escape processing, you can use such quotes as a delimiter safely if you modify the string by following these simple rules in this order:

1. Escape each escape character in the string.
2. Escape each quote character (of the same type — single or double) in the string.

Expressions and program arguments that start with “escape” will be treated similarly, as will the output of a range program.

Using Escaping With Tcl

When using Tcl you can enable escaping (either globally or within the command) to put any kind of string into the database or pass such a string as an argument to an MQL program regardless of which characters it contains and without the need for the user to worry about backslashes at all—except for those necessitated in order to set Tcl variables. In general, by using Tcl you can avoid the need to add quotes around tokens for the purpose of making the MQL command processor treat them correctly. That then removes the need to use an escape character with such quotes. When an MQL command is executed in Tcl, it is Tcl that initially breaks the command into tokens. When Live Collaboration receives these tokens from Tcl, it attempts, as best as possible, to put them together into a command string so that the MQL command processor will treat each of the tokens received from Tcl as a single token. So if the token contains a space or a quote character or any of a number of other characters, internally Live Collaboration puts quotes around it and then put it together with the other tokens to create an MQL command string that is parsed as any normal MQL command is parsed. When escape processing is on, Live Collaboration also escapes the contents of the string properly so that it will be interpreted as one token by the MQL command processor.

So, in Tcl programs, if you put values into a variable and then use the variable to set attribute values, you should enable escape (either globally or within the command), and Live Collaboration adds the necessary backslashes to the processed commands. For example:

```
tcl;
set myvar { my string with many weird characters such as ';#" - but no
squiggly brace }
mql escape modify businessobject type name rev MyAttr $myvar
```

If you needed to use a squiggly brace inside the squiggly braces of the `set` *command, you would have to backslash it — to satisfy Tcl’s demands, but not MQL’s.*

To get Tcl special characters passed successfully to Tcl with a minimum of fuss and worry is to create MQL commands as a Tcl list, with the list elements representing the distinct tokens as you want MQL to see them. If you want a TCL special character to be ignored by Tcl and passed through as part of a string to MQL, you need to use \.

In short, when constructing commands in Tcl, you must realize that tcl will consume backslashes because it always considers backslash as an escape character. And you have to provide additional

backslashes to escape any other characters that are meaningful to Tcl (", =, ...). With complicated Tcl/MQL commands, it is more manageable to use the following programming technique:

1. Create Where clauses as a single tcl string.
2. Create commands as a tcl list of strings.
3. Execute the tcl list using eval.

For example:

```
# Tcl parsing will turn \\ into \ ==> MQL gets TEST\'4
 set sWhere "escape name == TEST\\'4"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 # Tcl parsing will turn \\\" into \" ==> MQL gets TEST\"4
 set sWhere "escape name == TEST\\\"4"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 # Tcl parsing will turn \\ into \ ==> MQL gets TEST\*4
 set sWhere "escape name == TEST\\*4"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 # Tcl parsing will turn \\\\ into \\ ==> MQL gets TEST\\4
 set sWhere "escape name == TEST\\\\4"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 # Tcl parsing will turn \\\= into \= ==> MQL gets TEST\=4
 set sWhere "escape name == TEST\\\=4"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 # Same logic for attributes.
 set sWhere "escape attribute\[string-u\] == A\\'B"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 set sWhere "escape attribute\[string-u\] == A\\\"B"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 set sWhere "escape attribute\[string-u\] == A\\*B"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd

 set sWhere "escape attribute\[string-u\] == A\\\\B"
 set lCmd [list mql temp query bus * * * where $sWhere]
 eval $lCmd
```

```
  set sWhere "escape attribute\[string-u\] == A\\\=B"
  set lCmd [list mql temp query bus * * * where $sWhere]
  eval $lCmd
```

More examples, followed by the output they generate:

```
tcl;
eval {

 # Example 1: getting double-quotes passed through tcl to mql for
inclusion in an attribute value
  set lCmd [list mql modify bus T1 N1 0 "Multiline String"  "String
with \"double-quotes\" and more"]
  puts "\nlCmd=$lCmd"
  set mqlret [catch {eval $lCmd} sOut]
  if {$mqlret != 0} {
      puts "Error: $sOut"
  }

 # Example two: getting square brackets through - frequent in dealing
with various selects.
  set lCmd [list mql print bus T1 N1 0 select "attribute\[Multiline
String\]" "state\[one\].actual" dump]
  puts "\nlCmd=$lCmd"
  set mqlret [catch {eval $lCmd} sOut]
  if {$mqlret != 0} {
      puts "Error: $sOut"
  } else {
     puts "Result: $sOut"
  }

 # Example 3: getting single-quotes passed through tcl to mql for
inclusion in an attribute value
 # Easy - single quotes are not special to tcl
  set lCmd [list mql modify bus T1 N1 0 "Multiline String"  "String
with 'single-quotes' and more"]
  puts "\nlCmd=$lCmd"
  set mqlret [catch {eval $lCmd} sOut]
  if {$mqlret != 0} {
      puts "Error: $sOut"
  }
  # Verify:
  set lCmd [list mql print bus T1 N1 0 select "attribute\[Multiline
String\]" dump]
  set mqlret [catch {eval $lCmd} sOut]
  if {$mqlret != 0} {
      puts "Error: $sOut"
  } else {
     puts "Result: $sOut"
  }
}
```

Output from the above looks like this:

```
lCmd=mql modify bus T1 N1 0 {Multiline String} {String with
"double-quotes" and more}

lCmd=mql print bus T1 N1 0 select {attribute[Multiline String]}
{state[one].actual} dump
Result: String with "double-quotes" and more,Fri Aug 20, 2004 8:58:10
AM EDT

lCmd=mql modify bus T1 N1 0 {Multiline String} {String with
'single-quotes' and more}
Result: String with 'single-quotes' and more
```

Exceptions

One exception to escape processing is that it cannot be used for characters that are part of a keyword in an expression. Keywords are such things as `match`, `attribute`, `ge`, `==`, and `AND`. They have a special meaning within expressions and backslashing their characters is neither necessary nor will it work properly.

Use of special characters in administrative object names must be avoided. Even escaping special characters in this case does not work as shown in the example below (the name of the attribute is “My]Attr”):

```
escape print bus MyType MyName MyRev select attribute[My\]Attr];
```

There is also one instance where a backslash does not change the function of the following character. In an MQL command, if a line with a comment ends with a backslash, the comment does not continue to the next line. Refer to the *Configuration Guide : Macros* in the online documentation.

You can modify any query that you own, and copy any query to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, business administrators can change their workspace to that of another user to work with queries that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

Delete Query

If a query is no longer needed, you can delete it using the Delete Query command:

```
delete query NAME [user USER_NAME];
```

`NAME` is the name of the query to be deleted. If you are a business administrator with person access, you can include the User clause to indicate another user’s workspace object.

Searches the local list of existing queries. If the name is found, that query is deleted. If the name is not found, an error message results.

For example, assume you have a query named “Overdue Invoices” that you no longer need. To delete this query from your area, you would enter the following MQL command:

```
delete query “Overdue Invoices”;
```

After this command is processed, the query is deleted and you receive the MQL prompt for the next command.

When a query is deleted, there is no effect on the business objects or on queries performed by other users. Queries are local only to the user's context and are not visible to other users.

Usage Notes

The problem with queries is simple: poorly structured queries cause memory and performance problems. For example, the result of a query may be so large that the computer system depletes resources in order to handle the result. Other queries may consume more processing time than they should.

This section contains practices and guidelines for improving query execution and performance.

Where Clause Guidelines

This section contains tips on how to include where clauses within a query.

Using Select Fields in well-structured Queries

Queries that perform well always include at least 1 field that can be converted to SQL to limit the results returned from the database server. Criteria that cannot be converted to SQL are evaluated against this candidate result set in memory and the final result set is displayed. The smaller the candidate result set, the better. Well-structured queries use several criteria that are "SQL convertible" so that the work performed in memory is kept to a minimum.

SQL convertible fields were formerly referred to as "indexed" fields, before the advent of indexed attributes.

The query optimizer processes a query in the following manner:

1. The where clause is first factored into the "OR" form: A || B || C || D. This is done using the boolean logical equivalences. For example:

   ```
   (A || B) && C
   ```

   becomes

   ```
   (A && C) || (B && C) and !(A && B) == !A || !B
   ```

   Each of the OR'd terms will be processed independently.

2. Then the query is parsed to identify all fields that are SQL convertible (that is, which terms can be converted into SQL and be included in the select commands issued to the database server).

3. It is then determined whether the set of SQL convertible terms are primarily related to business objects or relationships to decide if the initial selects are requested against an lxBO table (businessobjects) or lxRO table (relationships).

4. The commands are then issued to the database server to select rows from lxBO or lxRO including these SQL convertible terms. This will return a candidate result set which satisfies the SQL convertible terms (associated to either objects or relationships).

5. The remaining criteria (non-SQL convertible terms) is then processed against the candidate objects (or relationships). This requires additional db selects to get the data specified by those terms, and evaluating the expressions in memory within the process, and producing a final result set.

6. Then any selects associated with the query are then processed, which may involve further db selects to get additional data about the final result set.

Steps 4-6 are repeated for each vault identified by the Vault field of the query.

The most optimal queries use sufficiently specific SQL convertible terms to result in the smallest possible candidate result set. This strategy provides the following advantages:

- using SQL fully leverages the database server's optimization and indexing
- minimizing the candidate result set means there are less objects to postprocess to get a final result set, which requires both less time and less in-process memory.

The following selectables are SQL convertible. All queries should include at least one of these criteria:

SQL Convertible Field	Meaning
`Name`	object name
`Type`	object type
`Owner`	object owner
`Policy`	governing policy of object
`Format`	format of files checked into object
`Originated`	date object was created
`Modified`	date object was last modified
`Current`	current state of object
`attribute[]`	value of attribute on object
`to[].attribute[]`	value of attribute on relationship connected to object
`from[].attribute[]`	value of attribute on relationship connected from object
`relationship[].attribute[]`	value of attribute on relationship connected to or from object
`format.file.store`	store containing files checked into object
`format.file.location`	location containing files checked into object
`search[]`	result of full text search
`reserved`	Boolean indicating reserved status of business object or connection.
`reservedby`	a non-empty string or context-user
`Queries that match unreserved objects and connections are unindexable. Indexable cases therefore do not include reservedby matching an empty string, or reserved being false.`	
`reservedstart`	date and timestamp of when the object is reserved
`to[NAME] == TRUE`	to find objects with relationships of given NAME pointing to them
`from[NAME] == TRUE`	to find objects with relationships of given NAME pointing from them
`revision == first`	to find objects which are the first revision of their revision sequence
`revision == last`	to find objects which are the last revision of their revision sequence

You can use SQL convertible fields with any relational operator (==, !~=, <> etc.) applied to a constant value that may include wildcards. For example,

```
name ~= 'A*B'
```

returns all objects whose name begins with “A” and ends with “B”. Note that use of select fields in where clauses that are not in the list above will likely result in non-optimal queries since non-SQL convertible fields have to be evaluated on the client.

It is best not to use any word that can be a selectable for a business object or connection as a value in the Where clause of a query because it will be evaluated as a select clause rather than being taken literally. If it is unavoidable, refer to *Using `const` for reserved words*.

Clauses that are not SQL convertible

Any selectables not on the above list are not SQL convertible when used in a where clause. For example, the following are not SQL convertible:

```
description
grantor (and grantee, granteesignature)
state[]
revisions[]
previous (and next)
type.kindof
```

Note that “revision” is not SQL convertible when included in the where clause, but is SQL convertible when included in the business object specification part of the query.

Selectables that are not SQL convertible cannot be built into the SQL commands that get objects and connections from the database server, so these clauses are not used to limit the number of objects that are retrieved from the server. Qualifying these clauses is very sensitive to the number of objects retrieved, since for each such object, additional SQL calls have to be constructed to retrieve the values of these fields, and data has to be stored in the clients memory. Given these realities:

NEVER execute a query that has only fields that are not SQL convertible.

ALWAYS make sure that a query containing fields that are not SQL convertible also has some SQL convertible fields to limit the number of objects retrieved.

Here are a few examples of common queries that are bound to be slow. Note that in all of them, the type (PART) is specified, but a production database can have huge numbers of PARTs, so the type specification is not very limiting:

```
PART * * where 'description ~~ "*a word*"'
PART * * where 'state[StateName].satisfied == TRUE'
PART * * where 'revision == last'
```

In the above queries, it would be best to include an attribute that substantially limits the number of PARTs that the system would have to examine.

The number of objects retrieved from the database server (that is, the size of the candidate result set) by the SQL convertible fields is the biggest factor in query performance and client memory requirements. The length (in characters) of the where clause or number of expressions within the where clause are immaterial. Actually, a longer where clause is desirable if much of it is comprised of SQL convertible fields that limit the number of objects retrieved:

```
PART * * where ' (description ~~ "*a word*") AND (attribute[Keyword] ~~
"abc*") AND (attribute[Keyword] ~~ "123*") '
```

Criteria Based on Evaluating Values Against the Execution of a Program Object

Kernel where clause evaluation includes the possibility to evaluate a value or set of values against the execution of a JPO.

Example:

MQL> temp query bus Person * * where "VALUE SMATCHLIST program[emxPerson -method getStrRolesSymbolicAssigned ${OBJECTID} '${TYPE}'] '|'"

When utilizing this capability there are several factors to consider:

- If the JPO returns multiple values, then the JPO will return elements with a delimiter that matches the last parameter of the SMATCHLIST statement. (In the example a '|')
- If the value you compare against contains multiple clauses, these values are separated into distinct where clause elements.

  For example:

  temp query bus Person * *
  where "VALUE1 SMATCHLIST program[emxPerson -method getStrRolesSymbolicAssigned ${OBJECTID} '${TYPE}'] '|' OR VALUE2 SMATCHLIST program[emxPerson -method getStrRolesSymbolicAssigned ${OBJECTID} '${TYPE}'] '|' "
- VALUE contains wildcard characters. (Kernel IR IR237357)
- As always, performance must be profiled for any use case utilizing this technique because it depends on the efficiency of the JPO.

Indexed Attributes and Basics

A Business Administrator can create an “Index” that includes attributes (and optionally basic properties) to improve query performance. The Index, once enabled, is used to access the specified group of selectables together, resulting in improved performance of queries that use those items as criteria. If your queries consistently include the same set of selectable criteria, ask your business administrator about creating an Index. Query performance can be drastically improved. For details, refer to *Working with Indices* in Chapter 9.

For more information about building well structured queries, see *Modeling Considerations*.

Processing Relational Expressions

The processing of a where clause is driven by the left-hand side of relational expressions in several different ways:

- Creating efficient SQL for a relational expression depends on the left-hand side being a select keyword that can be mapped to SQL AND the right hand side being a constant. The only exception to the rule that the right-hand-side must be a constant are the specific expressions 'revision == first' and 'revision == last'.
- The left hand side (if it is a select keyword) also drives the datatype of the expression. That is, if the left-hand side refers to an integer attribute, Live Collaboration attempts to interpret the right-hand side as an integer; if the left-hand side is a date, it tries to interpret the right-hand side as a date.
- When the left-hand side represents a string in the database that is SQL convertible (such as “attribute[Synopsis]”), SQL is constructed to request the database server to return objects which match the right-hand side. In doing so, the server will treat the database values as literal strings with no wildcards. Wildcard matching only applies to the right-hand side.

- However, when the left-hand side represents a string in the database that is not SQL convertible (such as “description”, or “state[S].signature[SIG].signer”), the system must read the value into memory for each object and do a match. In this case, * and ? within the read values will be treated as wildcards. This is also true in the case where “.value” is appended (such as “attribute[Synopsis].value”), since that makes the clause not SQL convertible.
- Only the right-hand side is interpreted to include wildcards (unless you use == or !==).
- When used with the Equal and Not Equal operators (==, !=) in the where clause, the system treats the wildcard characters “?” and “*” as literal characters on *both* sides of the expression. Do not use these characters when querying using the Equal or Not Equal operators. On the other hand, MQL interprets “?” and “*” as wildcards when used with the four Match operators and when used in the Type, Name, or Revision fields of the query. For example, if you type the following query:

  ```
  temp query bus * A*B *
  ```

  MQL returns all objects that start with A and end with B. However, if you use the equality operator in the where clause, as follows:

  ```
  temp query bus * * * where name == A*B
  ```

  MQL looks for the literal A*B as the entire object name rather than treating the * as a wildcard.

Searching based on lengthy string fields

The database stores most string attribute values in the lxStringTable for the object’s vault. However, lxStringTable cannot hold more than 254 bytes of data. When a string attribute’s value is larger than this limit, the data is stored in the descriptions table (lxDescriptionTable), and a pointer to this table is placed in the lxStringTable.

When performing an “includes” search (using match operators: match, match case, not match, not match case) on string attribute values, Live Collaboration searches on both lxDescription and lxString tables only when the attribute involved is of type “multiline.” Also, if you use the equal operators (==, !=) and give a string of more than 254 bytes to be equal to, Live Collaboration checks the values in the lxDescriptionTable only.

To search on description or other string attribute values for given text, and to force the search of both tables, you can use the “long” match operators. These operators can be used in any expression (including the where clause entry screen) but are not offered in the query dialog in either the desktop or web version. Refer to *Relational Operators (RELATIONAL_OP)* for more information.

Alternatively, you can include the `.value` syntax in the where clause, as shown below:

```
attribute[LongString].value ~~ "matchstring"
```

For more information about `.value` keyword, see the section below.

Using .value

If a query has a clause that is known not to be a good search candidate, .value will make that clause the last thing evaluated in the query. For example, consider the two following queries and the SQL they generate.

The following examples are not valid queries per se, but examples to show how .value is used.

Query 1:

```
rev = "*", type = "A", name = "*", and attribute[A] = 'this'
```

SQL generated:

```
(type=="A") && (revision == "*") && (name == "*") && (attribute[A] ==
'this')
```

Query 2:

```
rev = "*", type = "A", name = "*", and attribute[A].value = 'this'
```

SQL generated:

```
(type=="A") && (minorrevision == "*") && (name == "*") )
```

The difference is the processing order of the query. In query 1, all the clauses are evaluated in order. In query 2, the type, name, and minorrevision clauses are evaluated first, then the result is evaluated with the attribute[A].value clause. It is important to note that .value takes an attribute that is considered SQL convertible and makes it not SQL convertible. Generally it is done because the final part of the query does not help the search become more efficient due to a schema problem. The work in this case is not done by the database, but by Live Collaboration.

When using .value, you should include as much criteria as possible — several ANDs and/or ORs.

Query Syntax

- You must include a **space** before and after the operator in all where expressions.
- To find objects where a particular property is blank, use a **double quote** (no space). For example, if you want find objects that do not contain a description, use (description ~~ “”). For Boolean attributes, searching on “” results in finding objects where the value for the attribute is False, even if the default is True. (Boolean attributes, are either True or False, as opposed to Boolean expressions, which can be True, False or Unknown.)
- To find objects where a particular property is non-blank, use a **double asterisk**. For example, if you want be sure that all objects in the result of your query contain a description, use (description ~~ “**”).
- In complicated expressions, particularly those that use `!` or `not`, use **parentheses** to clearly state your intent. For example, in the following, the `!` is applied to the entire clause:

  ```
  !relationship[Categorize] == True || relationship[Categorize].from.id
  =="43482.46832.5291.38424
  ```

  To apply the `!` to only the portion before the OR (`||`), change it to:

  ```
  (!relationship[Categorize] == True) ||
  relationship[Categorize].from.id == 4448.10921.47699.5768
  ```

- The datatype of relational expressions (>,<,==) in where clauses is determined by the **left operand**. In the following example, the left operand is a hard coded string:

  ```
  temp query bus "Engineering Drawing" * * where ' "Jul 30, 2001 00:00:00
  AM EDT" > originated ';
  ```

  What appears to be a date (Jul 30...) is seen as a string, and so it does not find June originated dates because June alphabetically comes after July. For the inequality to do a date comparison, put originated on the left to establish the data type for the expression:

  ```
  where 'originated > "Jul 30, 2001 00:00:00 EDT"
  ```

- Always enclose the entirety of the `where` clause in SINGLE quotes. If the expression is not enclosed, MQL will not read it as a single value. Most query expressions contain multiple values and spaces. Therefore quotes are necessary to determine the boundaries of the expression.
- Strings in square brackets can have spaces—the square brackets delimit them, but any other strings with spaces should be enclosed in DOUBLE quotes.
- When using `where` clauses with `expand bus`, you must always insert `select bus` or `select rel` before a `where` clause. See *SELECT_ BO and SELECT_REL Clauses* for details.

The syntax of a query command affects the execution of the query. Occasionally, there are core changes that handle commands slightly differently, or maybe add a few new options to a command,

for additional functionality. For example, the change from version 9521 to version 9601 resulted in the handling of the `relationship` keyword differently within a `where` clause. (The `relationship` keyword can still be used in a `select` clause without adversely affecting performance.) This caused queries that ran fast under the older version to run much more slowly. The following query is an example:

```
temp query bus myType * * where
'(relationship[myRelationship].attribute[myAttribute] ...)';
```

In the latest version of Live Collaboration, the solution is to get rid of the `relationship` keyword and instead create separate clauses with "from" and "to" in them, as shown here:

```
temp query bus myType * * where
'(from[myRelationship].attribute[myAttribute] ...)
```

Or

```
(to[myRelationship].attribute[myAttribute] ...)';
```

Query Parsing

The order of the query created becomes important when trying to structure a query for performance. For example, consider a query:

```
where "A && (B || C)"
```

As the kernel parses the query, the expression passed to the database becomes:

```
(A && B) || (A && C)
```

The kernel uses ORs at the top level to separate the query into parts, then to accumulate the results. Generally, the processing of a query is done left to right. Knowing this, a user can structure queries so that the indexed attributes are used first. Front loading the query in this manner restricts the sets of objects searched with non-indexed attributes.

Selects and Macros in Where Clauses

If a where clause makes use of a select, the product evaluates the select for each object found by the query. However, when using a macro, the macro is evaluated one time only. This can result in a drastic performance difference.

In the sample queries below, the first query uses a select, while the second query uses a macro. The first query will return all the objects that match it, then apply the where clause. The second query will evaluate the macro and use the result as part of the query. As a result, it will return a smaller subset that already matches the owner, giving far better performance.

Examples:

```
temp query bus <TYPE> * * where owner = context.user
temp query bus <TYPE> * * where owner = $USER
```

Searching on Date/Time

In order to include Date/Time in your query, you must use the format defined for your system initialization file. Live Collaboration provides six different formats for the display and entry of dates and times. Each can be modified by adding lines to the .ini file or the startup scripts. See the *Administration Guide* : *Configuring Date and Time Formats*.

If your Date/Time query does not conform to the expected format, you will receive an error message similar to the following:

```
Invalid date/time format ‘July 22. ‘02’.
Allowed formats are:
[day] mon dom, yr4 h12:min[:sec] [mer] [tz]
[day] mon dom, yr4
moy/dom[/yr2] h12:min[:sec] [mer]
```

```
moy/dom[/yr2]
```

When entering a date as a value in the Where clause box, 3DEXPERIENCE platform assumes the time is 12:00 AM unless a specific time is specified as part of the query.

The following table explains the tokens used in Date formats:

Token	Meaning
`day`	day of the week (mon, tue, wed,...)
`DAY`	day of the week, not abbreviated
`mon`	month (jan, feb, mar,...)
`MON`	month, not abbreviated
`tz`	time zone (edt, cdt, pdt,...)
`TZ`	time zone, not abbreviated
`mer`	time meridian (am, pm, or blank for 24 hour time)
`sec`	seconds, 0 - 59
`min`	minutes, 0 - 59
`h12`	hour in 12 hour format, “mer” will be non-blank
`h24`	hour in 24 hour format, “mer” will be blank
`yr2`	abbreviated year (96, 97, 98,...)
`yr4`	full year (1996, 1997, 1998..)
`dom`	day of month (1, 2, 3,..., 31)
`doy`	day of year (1, 2, 3,..., 365)
`moy`	month of year (1, 2, 3,..., 12)

Creating sets to search

If you are trying to search based on information about a relationship certain objects may have, you should first query on the kinds of objects you are interested in, save them in a set, and then use the toset or fromset selectables in a query to help qualify your search. Refer to *Using Fromset and Toset Selectables* for more details.

Using `const` for reserved words

Reserved words such as keywords and select expressions must be afforded special consideration in exact equal (==) Live Collaboration expressions. For example, the following command might be written to find business objects where the value of the attribute ‘Regression’ is ‘first’.

```
temp query bus * * *
where 'attribute[Regression]==first';
```

But because `first` is a select keyword that returns the first revision of a business object and is evaluated as such, the result of the evaluation — rather than the literal word ‘first’— is compared with the attribute.

For this type of situation, use `const` to indicate that whatever follows should not be evaluated. For example:

```
temp query bus * * *
where 'attribute[Regression]==const"first"';
```

`Const` has three possible forms: all uppercase, all lowercase, and initial letter capitalized followed by all lowercase. No space can appear after the word `const`. It must be followed by a quote (double or single, depending on the syntax of the rest of the command). Almost any character can appear within the quotes, with the exception of backslash and pound sign. The characters between the initial and closing quotes remain unevaluated.

If your implementations are using JSP/Tcl to compose a `where` clause dynamically (that is, using a variable to construct the `where` clause), the `const` syntax must be used because the value you pass in could be a keyword. If this happens, the `where` clause will not return an error, but you will get unexpected results.

Here are some examples of queries that will NOT return correctly without using `const`:

```
attribute[Some Attribute]==‘first’
attribute[Some Attribute]==‘FIRST’
attribute[Some Attribute]=="Current"
attribute[Some Attribute]=="owner"
description=="Current"
description=="policy"
```

However, the following will return correctly:

```
attribute[Some Attribute]==‘my first order’
attribute[Some Attribute]~=‘*first*’
```

Invalid Where clauses

All queries that include at least some searchable criteria are evaluated, even if some parts of the where clause is invalid or incomplete. Queries that include some invalid items in a where clauses will issue warnings when:

- The where clause has a relational term in which both sides are constants.
- A conjunct or disjunct or unary negation has only a constant term.
- The entire where clause is just a constant.

(A constant term is one that is not a select clause, thus does not vary from object to object.)

A message is also output to warn about these situations in those cases where the expression might be indexed (made part of the SQL query to get candidate objects):

- The keywords `fromset` and `toset` used without a set name.
- The keywords `from`, `to`, and `relationship` used with a name that should be the name of a relationship type, but it is not.
- The keyword `attribute` used with a name that should be the name of an attribute type, but it is not.

If you receive such a warning, you should cancel the query evaluation and correct the problem.

Modeling Considerations

A big key to query performance is the data model that is used. It is much more efficient to build queries that use attributes rather than to build queries using fields that are not SQL convertible. Careful and intelligent use of attributes and triggers can limit the queries created. Certain queries, while looking simple and returning a very small subset, may actually cause extreme performance problems.

Below is a typical query, targeting objects of type Manufacturer, Facility, Group, and Project:

```
temp query bus Manufacturer,Facility,Group,Project * * where
'from[Group Assignment].to.type == Person' select id dump ¿;
```

This query resulted in a small set of 12 objects, but it caused the generation of nearly four hundred select commands and excessive memory growth. A better solution would be to create an attribute that would make the query true or false. To create a SQL convertible query, first add an attribute (for example, “Assigned to Person”) to each of the four target types. For the purposes of this example, the values of this attribute are “YES” and “NO”. When a business object of these particular types is created, set the value of attribute “Assigned to Person” to “NO”. Next, create triggers to modify the “Assigned to Person” attribute, either setting the value or unsetting it. As users update objects of the desired type(s), the assignedToPerson trigger checks the relationships of the object. The work the trigger will do can be summarized as “if the object meets the criteria specified in the original non-SQL convertible query, then modify the 'Assigned to Person' attribute to show that it does meet the criteria.” The trigger will check to see whether the business object has a relationship to a person from its group assignment. If it does, set the “Assigned to Person” attribute to indicate “YES”. If the user removes that relationship, set the attribute value to “NO”.

The new query would look something like this:

```
temp query bus Manufacturer,Facility,Group,Project * * where
'attribute[Assigned to Person] == YES' select id dump ¿;
```

It is also important to include some consideration of query performance when designing your data model—in particular, when defining a new type.

For example, here are three modeling alternatives that minimize the need for queries on descriptions. All three can easily be maintained using the ModifyDescription trigger on the type:

- Add an attribute Synopsis, which holds the first 100 characters of the description field.
- Add an attribute Keywords, and require values to be specified at create time.
- Write the description to a text file, check it into the object, and replace uses of 'description ==' by 'search[] ==’.

In addition to the data model, knowing the application is another key. Really understanding how the application works enables users to create better queries. The end user knows what search parameters return lots of objects and which ones return a limited object collection.

Keep in mind that Live Collaboration is best suited to expand/navigate operations. Queries are not the most efficient way to access the object-oriented data that the application manages. The best way to improve the performance of queries is to eliminate queries. A data model focused on the expand/navigation of relationships helps tremendously. However, one caveat concerning the use of type hierarchies and the use of `!expandtype`: Performance problems may occur when using 1000s of subtypes and then querying against the name of a base class. Traversing giant type trees can also cause problems. Refer to *`kindof` selectable for types* for a possible solution if large and deep type hierarchies are unavoidable.

Avoiding Unbounded Queries

Unbounded queries can pose problems, especially with large databases. For example, a command similar to "temp query bus * * *" will search every single item in the database. If the database is small, this is not an issue. As the database size increases, this becomes a tremendous performance problem. To avoid this problem you can restrict the query by vault or type, impose a find limit on the query, or create a query trigger.

Querying Vaults

Restricting the query to objects within the same vault is an obvious way to restrict a query. Limiting the search to objects that are all known to be stored a certain way cuts down on processing because all SQL commands related to the query have to be duplicated for each vault involved in the search.

Avoiding Large Expands

Similar to a large query, a large expand also causes performance problems. For example, the command “expand bus T N R recurse all;” attempts to expand all items connected to the specified object, expand all of those objects, and on, and on. Once again, a small database can hide the potential problem with this query because its results do not manifest themselves until the database size gets very large. A suggested solution is to limit the number of levels to expand (do not use recurse all, use recurse to 2).

Object Existence

Generally, you should not use temp queries to check for the existence of objects.

Rather than using “mql temp query T N R”, it is better to use “mql print bus T N R select exists”, which will return true if the object exists, false otherwise.

Query Instantiation

In instantiating queries, do not use a null string, use an empty string. Declaring a query as “Query q = new Query()” is not as good as “Query q = new Query q("")”. The empty string query constructor could cause stability problems if the same users modify the same query at the same time. Also, in this case the temp query opens the query .finder, updates it with the new type, name, revision and vault information and then runs the query. If .finder happens to have a where clause specified, it is not cleared. As a result, the query might not return the MQL equivalent of "temp query bus type name rev vault". Using the empty string also results in better performance since the query finder will not have to be written to the database.

Nested Queries

An additional problem with queries and/or transactions results from nesting them inside one another. For example, the first set shows a transaction nested within another; the second set shows two separate transactions.

Invalid sequence:

Start trans1

Start trans2

Commit trans2

Commit trans1

Valid sequence:

Start trans1

Commit trans1

Start trans2

Commit trans2

Live Collaboration does not allow nested transactions. However, savepoints can be used to divide large transactions into smaller parts. If the start() method of the context class is called a second time before the commit or abort methods are called for the first transaction, the system throws a MatrixException. Use the isTransactionActive() method to determine if a transaction is still alive before starting a transaction. If the result is false, it is safe to begin a transaction; if true, the current transaction must be aborted or committed before starting a new one. In addition, all exceptions within a transaction must be handled, else the transaction could get stranded. If the transaction is stranded, it cannot be recovered. Using a transaction timeout (to abort/commit the transaction once the threshold has been exceeded) can help to solve the problem of stranded transactions.

relationship Command

Description

A *relationship* definition is used along with the policy to implement business practices. Therefore, they are relatively complex definitions, usually requiring planning.

For conceptual information on this command, see *Relationships* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete]relationship NAME {CLAUSE};
```

- NAME is the name you assign to the relationship. Relationship names must be unique. For additional information, refer to *Administrative Object Names*.
- CLAUSEs provide additional information about the relationship.

Add Relationship

A relationship between two business objects is defined with the Add Relationship command:

```
add relationship NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides more information about the relationship you are creating. The Add Relationship clauses are:

attribute ATTRIBUTE_NAME {, ATTRIBUTE_NAME}
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]
description VALUE
from ADD_SUB_ITEM {, ADD_SUB_ITEM }
to ADD_SUB_ITEM {, ADD_SUB_ITEM }
[!|not]preventduplicates
derived RELATIONSHIP_NAME
abstract [true|false]
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

```
history STRING
dynamic
```

Either the Derived clause or both the From and To clauses are required to make a relationship usable. However, each clause is beneficial when defining relationships. In the sections that follow, the clauses and the arguments they use are discussed.

Attribute Clause

This clause associates one or more attributes with a relationship. To assign an attribute to a relationship, it must be previously defined. Attributes are useful in providing information specific to the connection you are making. In some cases the information is included in the business objects; in other cases, it is more appropriate to associate the information with the body of the connection. When making a connection, the user will fill in the attribute values.

For example, many different Assembly objects might use a Component object. The cost for installing the component into each assembly may differ considerably. Therefore, you may want to define an attribute called “Installation Cost” and associate it with the component relationship. Whenever a connection is made between a Component object and an Assembly object, the user can insert the cost associated with that connection.

While you may define an attribute for use in a relationship, it can also be used in other relationship definitions or type definitions. Therefore, it is possible for the attribute to be contained within the object itself.

As another example, a Quantity attribute is assigned to an assembly object to track the number of components used. You also can assign the Quantity attribute to a relationship to track the number of times the component is required for an Assembly. Since the quantity of the component may differ from assembly to assembly, the relationship records the amount as part of its definition. When a specific Component object is connected to a particular Assembly object, the user automatically has a means of inserting the quantity information.

An Add Relationship command may have many or no Attribute clauses depending on the types of objects being connected and why. For example, the following command associates three attributes with the relationship named “Assembly Relationship”:

```
add relationship “Assembly Relationship”
   description “Identifies component objects used in an assembly object”
   attribute Quantity
   attribute Units
   attribute “Installation Cost”
   from
     type Assembly
     meaning “Is composed of”
     cardinality n
     minorrevision float
     clone none
   to
     type Component
     meaning “Is used by”
     cardinality n
     minorrevision float
     clone none;
```

Once this command is processed and the relationship is defined, you can create connections between instances of Component type objects and instances of Assembly type objects by using the Connect Businessobject command (as described in *Making Connections Between Business Objects* in Chapter 5). These connections have three fields where the user can define the values for the attributes: Quantity, Units, and Installation Cost. Although the user is not required to enter values for the attributes, they are always available.

If you add an attribute that is part of an index, the index is disabled. Refer to Working with Indices in Chapter 9 for more information.

Trigger Clause

Event Triggers provide a way to customize behavior through Program objects. Triggers can contain up to three Programs, (a check, an override, and an action program) which can all work together, or each work alone. The Trigger clause specifies the program name, which event causes the trigger to execute, and which type of trigger program it is. Types support triggers for many events. Refer to the *Configuration Guide : Triggers* in the online documentation.

For example, when a relationship is instantiated (created), a trigger could check that an attribute value is equal to a certain value, and notification of the connection could be sent to an appropriate user. In fact, if the attribute value did not meet a specified set of criteria, a different event could replace the original. These transactions are written into program objects which are then called by the trigger.

The format of the Trigger clause is:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input
ARG_STRING]
```

- `EVENT_TYPE` is any of the valid events for Relationships:

`create`	`delete`	`freeze`
`modify`	`modifyattribute`	`modifyfrom`
`modifyto`	`thaw`	`--`
* The modify EVENT_TYPE only supports action triggers.		

Refer to the *Configuration Guide : Triggers* in the online documentation.

`TRIGGER_TYPE` is Check, Override, or Action.

`PROG_NAME` is the name of the Program object that will execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

For example:

```
add relationship "Assembly Relationship"
   description "Identifies the component objects used in an assembly object"
   attribute Quantity
   attribute Units
   attribute "Installation Cost"
   from
     type Assembly
   to
     type Component
   trigger create check "Quantity Check"
   trigger create override "Use Alternate Relationship"
   trigger create action "Notify John";
```

The modifyfrom and modifyto events provide customization hooks for the following scenarios:

- When objects are cloned or revised, connections are created or modified based on the revision or clone rules of either float or replicate (as set in the relationship definition).
- When the MQL modify connection command is used to replace 1 object with another on either end.
- When Matrix Navigator is used to drag and drop a business object on a relationship to initiate a replace operation.

Refer to the *Configuration Guide : Macros* in the online documentation.

To and From Clauses

Theses clauses define the ends of the connections. These clauses identify:

- The types of business objects that can have this relationship.
- The meaning of each connection end.
- The rules for maintaining the relationship.

All relationships will occur between two business objects. These objects may be of the same type or different types. When looking at a relationship, the objects are connected TO and FROM one another.

In some relationships, you can assign either object to either end. However, the To and From labels help you identify the reason for the connection.

You must define the connection with one To clause and one From clause in your Add Relationship command. If either clause is missing or incomplete, the relationship will not be created.

Both the To and From clauses use the same set of subclauses because they define the same information about each connected business object. The separate values you use to define each connection end will vary according to the type of connection you are making and the types of objects involved.

When you define one end of a connection, only one subclause is required: type or relationship.

`type TYPE_NAME {,TYPE_NAME}`
`type all`
`relationship REL_NAME {,REL_NAME}`

`relationship all`
`meaning VALUE`
`cardinality CARDINAL_ID`
`minorrevision REVISION_RULE`
`clone CLONE_RULE`
`[!|not]propagatemodify`

- `type` defines the types of business objects the user can use for each connection end within the relationship.
- `relationship` defines the types of connections the user can use for each connection end within the relationship.
- `meaning` (optional) helps the user identify the purpose of the objects being connected. For more information, see *Administrative Object Names*.
- `cardinality`, `minorrevision` and `clone` (all optional) deal with the number of relationships that the object can have on each end and how those relationships are maintained when one of the connected objects is revised or cloned. There must be a level of agreement between the clauses in order for both to work properly. `cardinality` defaults to MANY, `minorrevision` and `clone` default to NONE.
- `propagatemodify` specifies how modifications to the relationship instance are reflected in the modified timestamp of the objects on each end. This is helpful for monitoring changes made on a certain day or within a specified time period. With this switch on, objects whose only modifications have been to their relationships can be found by queries searching on modification date values. The default is off, so when adding relationships that don't require this feature, the `notpropagatemodify` clause is not necessary.

These clauses are discussed in the paragraphs that follow.

Type and Relationship Subclauses

The Type and Relationship subclauses specify the types of business objects and/or connections that can be used for each end of the relationship. You must define Types and Relationships to use them. You must specify at least one business object type or relationship at each end in order for the relationship to be valid.

When both ends of the connection involve a single type or relationship, the relationship name can reflect the items being connected. For example, a relationship name of "Model and Drawing" might always refer to a connection between an electronic drawing and a physical model made from the drawing.

A name of "Alternative Component" might always refer to a connection between two component objects used interchangeably in an assembly. In both examples, the name reflects the type of objects connected by the relationship.

An Add Relationship command to define the relationships between components, assemblies, and subassemblies, might appear as:

```
add relationship "Part Usage"
   description "Identifies the objects used in an assembly or subassembly"
   attribute Quantity
   attribute "Installation Time"
   from
     type Assembly, Subassembly
     meaning "Is composed of"
     cardinality n
     minorrevision float
     clone none
   to
     type Component, Subassembly
     meaning "Is used by"
     cardinality n
     minorrevision float
     clone none;
```

When a connection end can be assigned multiple business types, the name of the relationship needs to be more generic. For example, a relationship named "Part Usage" might have one connection end that is either an Assembly or Subassembly object type. The other connection end is a either a Component or Subassembly object type. While you have only one relationship, you actually have four types of connections you can make:

- Component objects and Subassembly objects
- Component objects and Assembly objects
- Subassembly objects and Subassembly objects
- Subassembly objects and Assembly objects

This enables you to relate all components and subassemblies to their larger subassemblies and assemblies without defining a relationship for each connection type.

In the following example, the relationship Feature Test is created and is allowed between objects of Type Feature and connections of Relationship type Build.

```
add relationship "Feature Test" from type Feature to
relationship Build;
```

A variation of the Type and Relationship subclauses use the `all` keyword in place of one or more type names (type all or relationship all). When this keyword is used, all types or relationships defined are allowed with the named relationship. This means that the specified relationship end can have any object or connection type. Be aware that any additional object or relationship types added at a later date will be allowed within the relationship. This can create problems if the new types are incompatible with the relationship ends. Therefore, you should use the `type all` and `relationship all` subclauses with care.

In the sections that follow, the use of the term "object" means both business objects and relationships.

Meaning Subclause

An end's *meaning* is a descriptive phrase that identifies how the connection end relates to the other end (when viewed from the other end). The meaning helps the user identify the purpose of the

objects being connected. Although the Meaning clause is not required, its use is strongly recommended.

In the example command on the previous page, the Meaning clauses identify the arrangement when a Subassembly object is connected to another Subassembly object. The Meaning clauses identify the order.

Even when both objects are equivalent, inserting a Meaning clause is helpful. It tells the user that the order does not matter. For example, assume you wrote a relationship definition to link equivalent components. In the following definition, the Meaning clause states that both objects have the same meaning. While the user might guess the meaning from the relationship name and description, the Meaning clause eliminates doubt.

```
add relationship "Alternative Components"
   description "Identifies interchangeable components"
   from
     type Component
     meaning "Can be used in place of"
     cardinality 1
     minorrevision none
     clone none
   to
     type Component
     meaning "Can be used in place of"
     cardinality 1
     minorrevision none
     clone none;
```

Cardinality Subclause

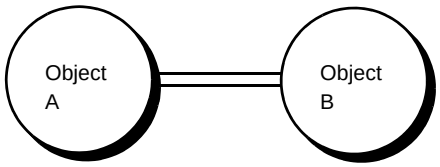
Cardinality refers to the number of relationships that can simultaneously exist between the two instances of objects. The Cardinality subclause defines this number. When you define the cardinality, it can have one of these values:

- **One** or **1—**The object can only have one connection of this relationship type at any time.
- **Many**—The object can have several relationships of this type simultaneously.

Since cardinality is defined for each end of a connection, it is possible to have three cardinal relationships between the ends:

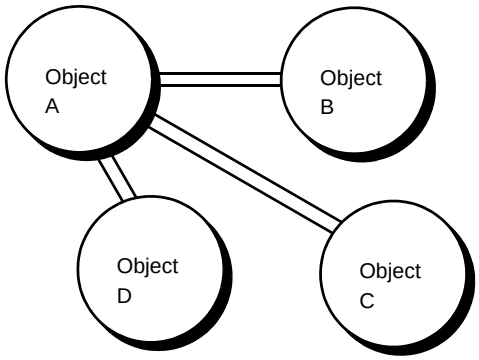
```
1 to 1          Or:          one to one

1 to n          Or:          n to 1

n to n          Or:          many to many
```

ONE-to-ONE



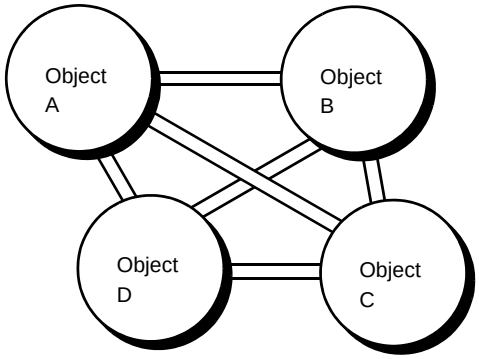
Object A can connect only with Object B.

MANY-to-ONE



Objects B, C, and D can have only one connection. Object A can have many connections.

MANY-to-MANY



All objects can have multiple connections simultaneously.

One-to-One

In a One-to-One relationship, the object on each end can be connected to only one other object with this type of relationship. An example of this type of cardinality might apply with a Change Order object connected to a Drawing object. Only one Change Order can be attached to the Drawing at any time and only one Drawing object can be attached to a Change Order.

In another example, you might have only one Customer object connected to a Flight Reservation object. The same customer cannot be on two flights simultaneously and two paying customers cannot occupy the same seat on a flight. Even a mother with a baby is contained in a single Customer object since she still represents a single paying customer.

One-to-Many Or Many-to-One

In a One-to-Many or Many-to-One relationship, the object on the “many” side can be attached to many other objects with this relationship type while the object on the “one” end cannot. An example of this type of relationship is a training course with multiple course evaluations. In an evaluation relationship, a single Training Course object can have many Course Evaluation objects attached to it. Therefore, the side of the relationship that allows the Training Course type needs a cardinality of One so that each Course Evaluation object can be connected to only one Training Course. On the other hand, the side of the relationship that allows the Course Evaluation type needs a cardinality of Many to allow many of them to use this kind of relationship to attach to the Course object.

Tip: Think of how many objects will typically exist on each end of the relationship. If it is one, the cardinality is ONE. If it is more than one, the cardinality is MANY.

Many-to-Many

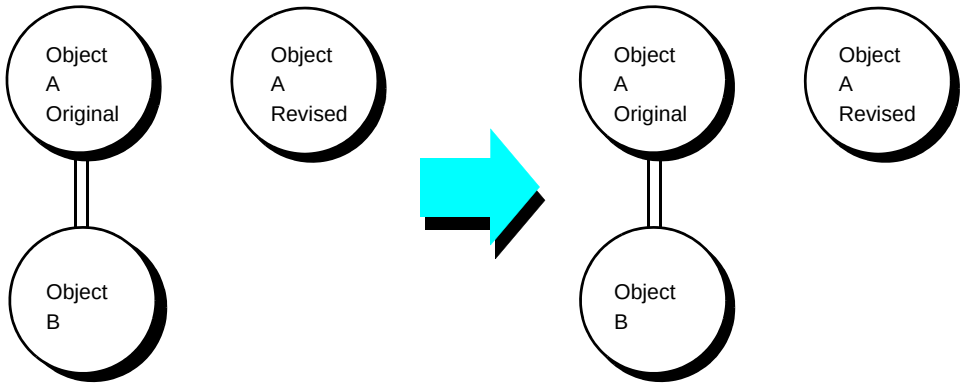
In a Many-to-Many relationship, objects on both ends of the relationship can have multiple simultaneous connections of this relationship type. This type of cardinal relationship is evident in a relationship between Component and Assembly objects. One Component object may be simultaneously connected to many different Assembly objects while one Assembly object may be simultaneously connected to many different Component objects. Both sides of the relationship are defined with a cardinality of Many since both can have more than one connection of this type at any time.

Minorrevision Subclause

When you are defining the cardinal value for a connection end, one factor that you must consider is revision. What will happen to the relationship if one of the connection ends is revised? Will you shift the relationship to the revised object, create a second new relationship with the revised object, or simply maintain the status quo by retaining a relationship with the unrevised object? The answer is specified by the revision rule associated with each connection end, as described in the following paragraphs.

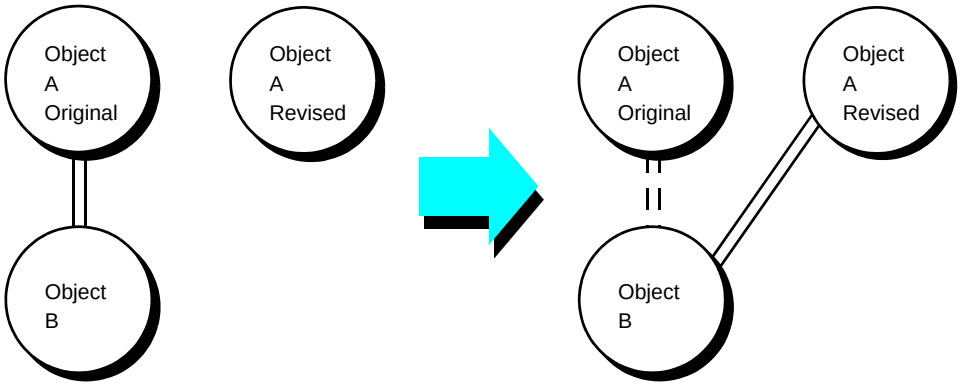
The Minorrevision subclause identifies the rule for handling revisions of the connection object. There are three revision rules for handling revised connection ends: **None**, **Float**, and **Replicate** (as illustrated below).

NONE



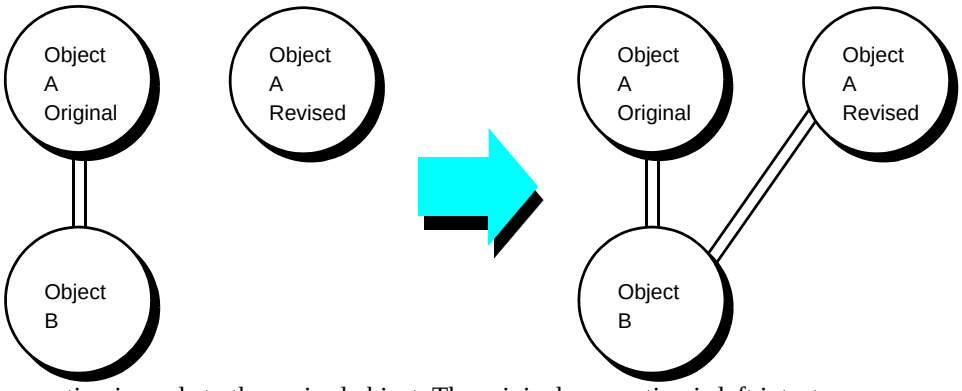
Status quo is maintained. The revised object has no connection.

FLOAT



The connection shifts to the revised object.

REPLICATE



A new connection is made to the revised object. The original connection is left intact.

None

When a connection end uses the None rule, nothing happens to the established connection when the end object is revised. The revised object does not automatically have a relationship attached to it after it is created. If you wanted to connect the revised object to the same object as the unrevised one, you would have to manually connect it with the Connect Business Objects command (see *Making Connections Between Business Objects* in Chapter 36).

Using the None rule is useful when an object revision removes the need for the connection. For example, when you have a connection between a Training Course object and a Course Evaluation

object, the connection may no longer be required or useful if the Training Course object is revised. While you may want to maintain the connection between the old version of the Training Course and the evaluation, the evaluation does not apply to the new version of the Training Course object. Therefore the connection end occupied by the Training Course object would use the None rule. But what of the other connection end?

If the Evaluation object is revised, it is still useful to the Training Course object. The revised object should remain connected to the Training Course object but the unrevised object no longer applies. To handle this situation, you would define the Course Evaluation end with the Float rule.

Float

The Float rule specifies that the relationship should be shifted whenever the object is revised. When the Float rule is used, the unrevised (or older version of the) object loses the connection with its other end. In its place the other end is automatically connected to the revised object. Now the older version of the object will be unattached while the newer version will have the relationship. This floating of the connection ensures that the latest versions of the object(s) are linked together.

But what if you want to maintain the old relationship while still creating a relationship with the new version? This would actually produce two connections: one between the unrevised object and its other end and one between the revised object and its other end. In this case, you would use the Replicate rule.

Replicate

The Replicate rule automatically creates new relationships whenever a connection end is revised. This results in a connection end that may have more than one simultaneous relationship of the same relationship type. For this reason, any connection end that uses the Replicate rule must also use a cardinality of “n,” as described in section *Cardinality Subclause*. If a cardinality value of 1 (one) is used, Replicate can not work.

The Replicate rule is useful when you want to keep track of former relationships. Since the old relationships are maintained while new ones are created, a user can easily see all versions that are related to a connected object. For example, you might have a relationship between a Specification object and a Specification Change object. As the Specification object is revised, you want to maintain the relationship so that you know that this Specification Change applied to this version of the Specification.

However, you also want the relationship to exist between the revised Specification and the Specification Change. This new relationship enables you to trace the history of the changes made and the reasons for them. In this situation, the Specification object should use the Replicate revision rule while the Specification Change object might use the Float or Replicate rule.

An Add Relationship command for the Specification Change might appear as:

```
add relationship "Specification Change"
   description "Associates a change notice with a specification"
   from
     type "Specification Change Notice"
     meaning "Contains changes to be made"
     cardinality 1
     revision float
     clone replicate
   to
     type Specification
     meaning "To be changed"
     cardinality n
     revision replicate
     clone none;
```

Note that the two connection ends have different revision rules and cardinality. Remember that these values should be determined for each connection end based upon the needs and requirements for that object as it relates to the connection being made. Since you may want only the latest version of the Specification Change Notice to be attached to the Specification, Float is the best choice for the revision rule. If you wanted to keep track of all notices that were attached to the Specification, you could change the revision rule to Replicate.

Clone Subclause

Just as you must define what should happen to the relationship if one of the connection ends is revised, you must define what should happen if one of the connection ends is cloned. The same three rules available for revisions are available for clones: **None**, **Float**, and **Replicate**.

Most business rules require a clone to be treated much differently than a revision, so you may often select a different rule for clones than for revisions. For example, the None rule is often useful when a connection end is cloned. Consider a Specification Change Notice object that is connected to a Specification object. If the Specification is cloned for a new product that is very similar, it's unlikely that the Specification Change Notice applies to the cloned Specification. So the original connection between the Specification Change Notice object and Specification object should remain but no new connection is needed for the cloned Specification object.

Propagate Modify Subclause

The `propagatemodify` setting on the `From` or `To` clause controls whether or not changes to a relationship instance affect the modification timestamp of the from/to object(s). When not used, changes to the relationship instance do not affect the modified date of the objects. The `not` (or `!`) form of the subclause can be used when modifying relationships to turn the setting off, if required. The default is off, so when adding relationships that don't require this feature, the `Notpropagatemodify` clause is not necessary.

For example:

```
add relationship BOM
   attribute Quantity
   to
     type Component
   from
     type Assembly
     propagatemodify;
```

Using the relationship above, changes in the quantity of the component in the assembly will be reflected in the modification date of the Assembly.

Propagate Connection Subclause

The `propagateconnection` setting on the `from` or `to` clause controls whether modification timestamps on connected objects are recorded when the objects connect and disconnect. You can turn off modification timestamps on a per relationship basis and control the “from” and “to” end of the relationship.

This is particularly useful for “one to many” relationships where the “many” side potentially may have hundreds of objects, connected by multiple users. Such frequent timestamp logging can slow down performance and cause potential concurrency issues. Also, if you query the database for objects that have been modified, you might not want to include every object that has merely participated in a connect/disconnect operation without having their actual definition changed.

The system will run faster and the chance of deadlocks is greatly reduced when both modification timestamps and history are turned off during connect/disconnect operations, since only the relationship tables (not the business object tables) are affected. (See *Enable and Disable History* more information.)

By default, the system updates modification timestamps on business objects every time they participate in a connect/disconnect operation. Use the `not` (`!`) form of the subclause to turn off these updates.

For example, there might be a Specification relationship defined for Feature objects on the “from” end and Document objects on the “to” end. You might turn off the “to” end, but leave the timestamp active on the “from” end. Whenever a connect/disconnect event is performed between these two objects, the Feature object’s timestamp is updated, but the Document object’s is not.

```
add relationship Specification
   to type Document !propagateconnection
   from type Feature;
```

Preventduplicates Clause

A flag can be set in the relationship definition that will prevent duplicates of the relationship type to exist between the same two objects. The default is that duplicates are allowed.

For example, to prevent duplicates of the relationship Documents, use the following command:

```
add relationship Documents
   preventduplicates
```

`!preventduplicates` would turn this feature off.

The `preventduplicates` *flag does not prevent a second relationship between two objects if it points in the opposite direction. For example, given BusObjA connected ONCE to BusObjB with* `preventduplicates`*, connecting BusObjA with* `preventduplicates` *to BusObjB will fail. Connecting BusObjB with* `preventduplicates` *to BusObjA will succeed.*

Derived Clause

Use the Derived clause to identify an existing relationship as the parent of the relationship you are creating. The parent relationship can be abstract or non-abstract. A child relationship inherits the following items from the parent:

- all attributes
- all methods
- all triggers

  A child relationship can override a trigger defined for its parent for the same event.
- governing rule

  A child relationship can override a rule by defining its own rule. In such a case, the parent rule doesn't govern the child relationship. For example, if a rule lists the parent relationship as a governed relationship, then this rule can also govern the child relationship. Note that in such a case, the child relationship is not listed as a governed relationship in the rule definition.
- allowed from and to end business types

  For example,

```
add relationship parentrel from type T1 to type T2;
add relationship childrel derived parentrel;
```

  If the parent relationship, “parentrel,” allows the type “T1” to be on the FROM end and the type “T2” to be on the TO end, then the child relationship, “childrel,” will allow “T1” to be on the FROM end and the type “T2” to be on the TO end. Inherited from and to business types can be specialized.

From and To Clauses with Derived Clause

A child relationship can define its own from or TO end business types or both. These types must be derived from the types listed by the parent relationship. If not, an error will occur. For example:

```
add relationship parentrel from type T1 to type T2;
add relationship childrel from type T1child to type T2child
derived parentrel;
```

The child relationship, “childrel,” allows the type “T1child” to be on the FROM end. “T1child” must be derived from “T1” since “T1” is defined by the parent relationship, “parentrel.” Only “T1child” will be allowed to be on the FROM end in “childrel.” “T1” will NOT be allowed.

Subclauses defined by the To and From clauses

The subclauses defined by the to and from clauses are also inherited by the child relationship:

- meaning
- cardinality
- revision rule
- clone rule
- propagate modify
- propagate connection

These subclauses are ALWAYS inherited and cannot be overridden (besides the meaning subclause). The meaning subclause can be overridden. The only way to redefine the other subclauses in a child is to redefine them in the parent.

Abstract Clause

An abstract relationship indicates that a user will not be able to create a connection of the relationship. An abstract relationship is helpful because you do not have to reenter groups of

attributes that are often reused. If an additional field is required, it needs to be added only once. Use one the following clauses:

```
abstract true
Or:
abstract false
```

If the user can create a connection of the defined relationship, set the abstract argument to `false`. If not, set the abstract argument to `true`. If you do not use the Abstract clause, false is assumed, allowing users to create instances of the relationship. For example:

```
add relationship parentrel from type T1 to type T2 abstract
true;
add relationship childrel derived parentrel;
```

Since the relationship “parentrel” is abstract, there will never be any actual instances made of the “parentrel” relationship. However, “parentrel” can be inherited by other relationships that are not abstract. The child relationship “childrel” is not abstract and instances of this relationship can be created.

History Clause

The `history` keyword adds a history record marked “custom” to the relationship that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Dynamic Clause

The `dynamic` keyword implements support for minor-revision insensitivity. This keyword is mutually exclusive with any of the Replicate/Float/None options on the TO end, since it defines a built-in dynamic behavior of its *to* pointer.

- There is a `dynamic` subselectable on relationship types for determining whether a named relationship type is of this kind (similar to `abstract` flag on relationship types).
- It is not possible to modify existing relationship types to become dynamic, only to create new relationship types as dynamic. Existing instance data must be migrated.
- Once a relationship type has been set as `dynamic`, relationship types that inherit from it cannot be non-dynamic. Dynamic is a property of the entire inheritance tree.
- There is no support for dynamic relationships to adaplet business objects. Attempting to create such a relationship instance will result in an error.
- Once a relationsip types has been marked dynamic, it starts participating in the new dynamic expansion. When adding a relationship instance (with Connect Bus or Add Connection), the kernel automatically adds an entry in the mxFamily table, if warranted.
- Relationships that are dynamic cannot point to other relationships (i.e., "rel to rel" is not supported for dynamic relationships.

See also *Dynamic Relationships* in Chapter 4.

Copy Relationship

After a relationship is defined, you can clone the definition with the Copy Relationship command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy relationship SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- SRC_NAME is the name of the relationship definition (source) to copied.
- DST_NAME is the name of the new definition (destination).
- MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The history keyword adds a history record marked “custom” to the relationship that is being copied. The STRING argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Relationship

After a business object relationship is defined, you can change the definition with the Modify Relationship command. This command lets you add or remove defining clauses and change the value of clause arguments.

```
modify relationship NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the relationship you want to modify.
- MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Relationship clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Relationship Clause	Specifies that…
name NAME	The current relationship name is changed to that of the new name.
add attribute NAME	The attribute listed here is associated with this relationship.
remove attribute NAME	The attribute listed here is removed from the relationship.
description VALUE	The current description, if any, is changed to the value entered.
add trigger EVENT_TYPE TRIGGER_TYPE PROGNAME	The specified trigger is added or modified for the listed event.
remove trigger EVENT_TYPE TRIGGER_TYPE	The specified trigger type is removed from the listed event.
icon FILENAME	The image is changed tp the new image in the field specified.
from MOD_SUB_ITEM {,MOD_SUB_ITEM}	A modification to the *from* connection end is made. This modification may involve altering the types of objects used by the relationship and/or how the relationship should be maintained if there is a revision to an object instance. Refer to the description of Connection End Modifications below.

Modify Relationship Clause	Specifies that…
`to MOD_SUB_ITEM {,MOD_SUB_ITEM}`	A modification to the *to* connection end is made. This modification may involve altering the types of objects used by the relationship and/or how the relationship should be maintained if there is a revision to an object instance. Refer to the description of Connection End Modifications below.
`add rule NAME`	The specified access rule is added. If you modify the rule for a derived relationship, the new rule overrides the parent's rule. Only one rule can be defined for a derived relationship. If no rule is defined, the parent's rule governs the relationship.
`remove rule NAME`	The specified access rule is removed
`preventduplicates`	The `preventduplicates` flag is changed so that duplicate relationship types are not allowed.
`notpreventduplicates`	The `preventduplicates` flag is changed so that duplicate relationship types are allowed.
`derived RELATIONSHIP_NAME`	The relationship being modified inherits attributes, methods, triggers, governing rules, and allowed from and TO end business types of the relationship named.
`abstract true`	Connection instances of this relationship cannot be created.
`abstract false`	Connection instances of this relationship can be created.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked "custom" to the program that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.
`add ownership [ORGANIZATION] [PROJECT] [for COMMENT] [as ACCESS]`	Adds an ownership to a relationship. See *Adding or Removing Relationship Ownerships*, below. See also *Ownership* in Chapter 4.
`remove ownership [ORGANIZATION] [PROJECT] [for COMMENT] [as ACCESS]`	Removes an ownership from a relationship. See *Adding or Removing Relationship Ownerships*, below. See also *Ownership* in Chapter 4

As you can see, each modification clause is related to the clauses and arguments that define the relationship. When you modify a business object relationship, you first name the relationship to be

changed and then list the modifications. For example, to change the name of the Alternative relationship and add an attribute and a trigger, you might write this command:

```
modify relationship Alternative
   name "Component Alternative"
   add attribute "Cost Comparison"
   add trigger create check "Cost Check"
   add trigger create override "Use Alternate Relationship"
   add trigger create action "Notify John";
```

Connection End Modifications

When you are modifying the connection ends of the relationship, the `MOD_SUB_ITEM` clause is similar to the clauses used to define the end objects.

Connection End Modification Clause	Specifies that…
`add type all`	All defined types are allowed to be associated with this relationship end.
`add type TYPE_NAME {,TYPE_NAME}`	The object type(s) listed here are allowed to be associated with this relationship end.
`add relationsip TYPE_NAME {,TYPE_NAME}`	The relationship(s) listed here are allowed to be associated with this relationship end.
`remove type all`	All object types are removed from this relationship end.
`remove type TYPE_NAME {,TYPE_NAME}`	The object type(s) listed here are removed from this relationship.
`remove relationsip TYPE_NAME {,TYPE_NAME}`	The relationship(s) listed here are removed from this relationship.
`meaning VALUE`	The current meaning, if any, changes to the value entered.
`cardinality CARDINAL_ID`	The value for cardinality is set to the value entered.
`revision REVISION_RULE`	The revision rule changes to the value entered.
`clone CLONE_RULE`	The clone rule changes to the value entered.
`propagatemodify`	The modification timestamp of the object on the specified end will be updated when the relationship's attribute values are modified.
`[!|not]propagatemodify`	The object's modification timestamp on the specified end will not be updated when the relationship's attribute values are modified.
`propagateconnection`	The object's modification timestamp on the specified end will be updated during connect/disconnect operations.
`[!|not]propagateconnection`	The object's modification timestamp on the specified end will not be updated during connect/disconnect operations.

These clauses are used within the To and From clauses as they are used in the Add Relationship command. Assume you have the following definition:

```
add relationship Comment
   description "Associates comment with related object"
   from
     type Comment
     meaning "Provides comment about "
     cardinality n
     minorrevision none
     clone none
     propagatemodify
     !propagateconnection
   to
     type Assembly, Document, Specification, Layout
     meaning "Is commented in "
     cardinality n
     minorrevision none
     clone none
     propagatemodify
     !propagateconnection;
```

To this definition, you might want to add a new object type that can connect to Comment objects, and you want objects on the FROM end to propagate modifications of the relationship to the business object. You also want to save all revised comments and include them in all revisions with the commented object. To make these changes, you might write the following Modify Relationship command:

```
modify relationship Comment
   to
     minorrevision replicate
   from
     add type "Process Plan" propagatemodify;
```

When this command is processed, the definition for the Comment relationship appears as:

```
relationship Comment
   description "Associates comment with related object"
   from
     type Comment
     meaning Provides comment about
     cardinality n
     minorrevision replicate
     clone none
     propagatemodify
   to
     type Assembly, Document, Specification, Layout,
     Process Plan
     meaning Is commented in
     cardinality n
     minorrevision none
     clone none;
```

Adding or Removing Relationship Ownerships

Relationships can have multiple owners. To add an ownership to or remove an ownership from a relationship:

```
modify relationship NAME add|remove ownership
[ORGANIZATION] [PROJECT] [for COMMENT] [as ACCESS];
```

- `ORGANIZATION` is a "role" object that represents an organization.
- `PROJECT` is a "role" object that represents a project.
- `COMMENT` is a short string that is primarily used for annotation.
- `ACCESS` is a comma-separated list of security tokens.

Each of these fields may be specified when providing an ownership. The first three (ORGANIZATION, PROJECT, and COMMENT) together provide a unique identifier for the ownership entry. Tokens following "for" constitute the ownership comment; those following "as" constitute the access mask.

The following are some examples of usage:

```
modify relationship NAME add ownership "MyCompany"
"MyProject" for "ownership”;
modify relationship NAME add ownership "Supplier1"
"MyProject" for "Project Applicability";
modify relationship NAME add ownership "Supplier2"
"MyProject" as show,read,checkout;
modify relationship NAME remove ownership "Supplier1"
"MyProject" for "Project Applicability";
```

Delete Relationship

If a relationship is no longer desired between business objects, you can delete it with the Delete Relationship command:

```
delete relationship NAME;
```

`NAME` is the name of the relationship to be deleted.

When this command is processed, the list of existing business object relationships is searched. If the name is not found, an error message is displayed. If the name is found, the relationship is deleted along with all information about that relationship. A relationship can only be deleted if there are no connections using the relationship or relationships derived from the relationship. In order to delete a parent relationship, you must delete all its child relationships.

For example, to delete the relationship named “Maintenance Relationship,” enter the following MQL command:

```
   delete relationship “Maintenance Relationship”;
```

After this command is processed, the relationship is deleted and you receive an MQL prompt for another command.

resource Command

Description

Resources are legacy administrative objects that should no longer be used.

A *resource* is a administrative object that stores binary files of any type and size. Apps use resources to display output to a standard Web browser or a small LCD device by providing components for Web pages. They can be any resource that you use in a Web application, including:

- GIF
- JPEG
- MPEG
- AVI
- WAV
- JAR
- CAB

For example, you can include in the database a resource that represents the company corporate logo. In a Web application, the company corporate logo may be referred to many times.

The resource editor allows you to name the administrative definition, give it a description, and define a MIME (Multi-Purpose Internet Mail Extension) type that is used to ensure that the browsers know what kind of component this is. For example, the company corporate logo could be an *image/gif* MIME type, indicating that it is a .gif file that should be rendered in the browser.

You can use standard MQL commands such as create, delete, copy, modify, print, and list to manage resources. You can also create, modify, and delete resources using the Business Modeler application. Specialized functions and embedded commands facilitate evaluation, translation, or formatting of objects or output on HTML pages.

Resource objects currently are not supported in a J2EE environment.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]resource NAME {CLAUSE};
```

- `NAME` is the name of the resource you are defining. The name you choose is the name that will be referenced to include this resource within a Web page. Resource names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the resource.

Add Resource

Use the Add Resource command to define a new resource:

```
add resource NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the resource.

ADD_ITEM is an Add Resource clause that provides additional information about the resource you are defining. The Add Resource clauses are:

description VALUE
file FILENAME
mime VALUE
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING

All clauses are optional for defining a resource, since the resource can later be modified. But to be used in a Web page, at least the file and MIME type must be defined. Each clause and the arguments they use are discussed in the sections that follow.

File Clause

This clause defines the binary file that contains the resource. This is often a .gif or other image file, but they can be any resource that you use in a Web application.

```
file FILENAME;
```

For example, to define a resource for the company logo, you could use:

```
add resource Logo
   file Matrixlogo.gif;
```

Mime Clause

This clause associates a MIME (Multi-Purpose Internet Mail Extension) type with a resource. It defines the content type of the file.

```
mime VALUE;
```

Value is the content type of the file. The format of value is a type and subtype separated by a slash. For example, image/gif or video/mpeg.

The major MIME types are application, audio, image, text, and video. There are a variety of formats that use the application type. For example, application/x-pdf refers to Adobe Acrobat Portable Document Format files. For information on specific MIME types (which are more appropriately called “media” types) refer the Internet Assigned Numbers Authority Website at http://www.isi.edu/in-notes/iana/assignments/media-types/. The IANA is the repository for assigned IP addresses, domain names, protocol numbers, and has also become the registry for a number of Web-related resources including media types.

History Clause

The `history` keyword adds a history record marked “custom” to the resource that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Resource

After a resource is defined, you can clone the definition with the Copy Resource command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy resource SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

`SRC_NAME` is the name of the resource definition (source) to copied.

`DST_NAME` is the name of the new definition (destination).

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the resource that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Resource

Use the Modify Resource command to change the definition of an existing resource. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify resource NAME [MOD_ITEM {MOD_ITEM}];
```

`NAME` is the name of the resource you want to modify.

`MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Resource clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Resource Clause	Specifies that…
`name NEW_NAME`	The current resource name changes to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed tp the new image in the field specified.
`file FILENAME`	The resource file is changed to the new file specified.
`mime VALUE`	The mime type for the resource is changed to the new value specified.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.

Modify Resource Clause	Specifies that…
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the resource that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

As you can see, each modification clause is related to the clauses and arguments that define the Resource.

Delete Resource

If a resource is no longer required, you can delete it using the Delete Resource command:

```
delete resource NAME;
```

`NAME` is the name of the resource to be deleted.

Searches the list of defined resources. If the name is found, that resource object is deleted. If the name is not found, an error message is displayed.

For example, to delete the Logo resource, enter the following command:

```
delete resource "Logo";
```

After this command is processed, the Logo resource object is deleted.

Usage Notes

For global database access, resources generally need to be provided in multiple languages. And with the wide use of cell phones and other hand-held devices in accessing Web pages, you may also need to support the resource’s display on a small LCD in wireless markup language (wml). When this is the case, you can use the following Studio Customization Toolkit call to open a resource with a language and format argument, following the syntax:

```
open(BASE_RESOURCE_NAME, LANG, MIMETYPE)
```

For example:

```
open(logo.gif, fr, wml)
```

When evaluating this code, the resource servlet first looks for the file named `logo_fr_wml.gif`. If the servlet does not find this resource object, it attempts to find `logo_wml.gif`. As a last resort, it searches for the resource `logo.gif`. Of course, you could call `login_fr_wml.gif` directly, but the addition of arguments gives you much more flexibility when writing the code.

In this case, you would first create `logo.gif`, and then create the wireless version of the resource (generally a much smaller version) and name it `logo_wml.gif`. Then, for each language that requires a different version of the resource (a translated textual image, a sound byte, or even a pure image that, due to cultural differences, requires a localized version) you would create the different resource files and reference them in a new resource object and save as `logo_LANG.gif` and `logo_LANG_wml.gif`. For example, to support multiple languages, you may have resources with the following names in the database:

```
logo.gif
logo_ch-tw.gif
logo_ch-gb.gif
logo_it.gif
```

To add support for wml, you might add the following resource:

```
logo_wml.gif
```

In this case, all languages would use the same wml resource.

Note that the base resource does not have to be in English. Also, the LANG argument could be more than two characters, such as en-us, en-uk, or ch-tw.

role Command

Description

Three administrative objects allow you to identify a set of users (persons) who require the same accesses: groups, roles, and associations. *Roles* are a collection of people who have a common job type: Engineer, Supervisor, Purchasing Agent, Forms Adjuster, and so on. Additionally, a role can be defined as either a project or organization. The ability to create projects and organizations is important for applications that use access models based on these user categories.

For conceptual information on this command, see *Controlling Access* in Chapter 3.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete|list|print] role NAME {CLAUSE};
```

- NAME is the name you assign to the role or group. This name must be unique and cannot be shared with any other type of user (groups, roles, persons, associations). Assign a name that has meaning to both you and users. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the attribute.

Add Role or Group

The clauses for defining groups and roles are almost identical. While only a name is required, the other parameters can further define the relationships to existing users, as well as provide useful information about the group or role.

Groups and Roles are created and defined with one of following MQL commands:

```
add group NAME [ADD_ITEM {ADD_ITEM}];
```

```
add role NAME [ADD_ITEM {ADD_ITEM}];
```

ADD_ITEM is a clause that provides more information about the group or role you are creating. While none of the clauses are required to create the group or role, they are used to assign specific users and roles to the group or users and groups to the role. Roles can also be defined as either a project or organization. The ADD_ITEM clauses are:

```
assign person PERSON_NAME [group GROUP_NAME]
```

```
child ROLE_NAME {,ROLE_NAME}
```

```
description VALUE
```

```
maturity [none | public | protected | private]
```

```
parent ROLE_NAME
```

site SITE_NAME
[!|not]hidden
| asaproject | | asanorg | | asarole |
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING

If you are creating a group that is used in an app you must register the group with the framework, as described in the Business Process Services - AEF User Guide in the online documentation.

Maturity Clause

The maturity of a project determines the access to objects in that project. The maturity attribute is intended primarily for roles that represent projects, but is not limited to that category. The maturity attribute on roles may be set to `public`, `protected`, `private`, or `none`.

See also *Administration Guide : About Project-Based Access* in the online documentation.

Parent and Child Clauses

These clauses define the relationship of the new group or role to other defined groups. This hierarchy allows one group or role to share access privileges with another, saving time when defining access privileges when a policy is defined. You can have any number of child groups and any number of parent groups. For example:

```
add group "Technical Marketing" parent Marketing;
add role "Engineer" child "Principal Engineer";
add group "Quality Engineering Managers" parent
Engineering,Management;
add role "Engineering Manager" parent Manager,Employee;
```

Of course, the group or role named as the parent or child must be previously defined.

Refer to the *Administration Guide : User Categories* for more information.

Assign clause

The Assign Person clause assigns specific users to the group or role. A group or role can have no users, or they could have many. Groups or Roles with no users may be defined to show a hierarchical relationship between groups or roles. In that case, the defined group or role acts as a parent for other groups or roles.

Do not modify the persons creator, guest, or Test Everything using MQL. Modifying these objects could cause triggers, programs or other application functions not to work.

Most groups and roles will have users assigned to them. When assigning users, the number is limited only by the maximum number defined in the database. Use the Assign Person clause to assign a person to a group or role:

```
assign person PERSON_NAME [role ROLE_NAME]
assign person PERSON_NAME [group GROUP_NAME]
```

- `PERSON_NAME` is the previously defined name of a person.
- `ROLE_NAME` is the previously defined role.
- `GROUP_NAME` is the previously defined role.
- If these names are not previously defined, an error message is displayed.

You can assign users groups and roles in two ways, depending on how you are building your database:

- With the Assign Person clause in the add Group or add Role command, as described here.
- In the Person command described in *Add Person*.

Since previously defined names are required to make the Assign Person clause valid, it is not uncommon to wait before assigning persons to a role or group definition. When building the database, you may want to define only the roles and groups and then handle the assignment of users in the person definitions. But, if you choose to define the persons first, or if you are adding a role or group to an existing database, you can assign users to the group or role with the assign person clause. Regardless of where you define it, an assignment made to a role or group becomes visible from all applicable definitions. This means that the link between the group or role and the person will appear when you later view either the group or role definition or the person definition.

When assigning a person to a group, you can also define that person's role within the group. Likewise when assigning a person to a role, you can also define that person's group. Including a role assignment with the person assignment serves as another means of making a role assignment to a person. Again, once the assignment is made, it can be seen in all person, group, and role definitions when the definitions are viewed. Use the method and commands most convenient for your application and database.

When a person is assigned a role within a group and then the assignments are printed, the output indicates this. For example:

```
print group "Corporate Training" select assignment;
group Corporate Training
   assignment = Corporate Training rob
   assignment = Corporate Training sheila Training Coordinator
```

For example, assume you want to add a role called “Trade Show Support” that associates members of the sales and customer service groups with it. You could write a command similar to the following:

```
add role "Trade Show Support"
   description "Personnel for Trade Show Support"
   child "Trade Show Backup Support"
   parent Marketing
   assign person elsie group "Sales Force"
   assign person mark
   assign person richard
   assign person jenine;
```

When this command is processed, the role of “Trade Show Support” is assigned to four persons. After this command is executed, you can examine the person definitions to see a role assignment included in the definition.

Site Clause

A *site* is a set of locations. It is used in a distributed environment to indicate the file store locations that are closest to the group. The Site clause specifies a default site for the group or role you are defining. Consult your System Administrator for more information.

To write a Site clause, you use the name of an existing site. If you are unsure of the site name or want to see a listing of the available sites, use the MQL List Site command. This command produces a list of available sites from which you can select. (Refer to *Locations and Sites* in Chapter 2.)

Sites may be set on persons, groups and roles, as well as on the Server (with MX_SITE_PREFERENCE). The system looks for and uses the settings in the following order:

- if using a Collaboration Server, the MX_SITE_PREFERENCE is used.
- if not using a Collaboration Server, or the MX_SITE_PREFERENCE is not set on the server, if there is a site associated with the person directly, it is used.
- if no site is found on the person, it looks at all groups to which the user belongs. If any of those groups have a site associated with it, the first one found is used.
- if no sites are found on the person or their groups, it looks at all roles the person is assigned. If any of those roles have a site associated with it, the first one found is used.

Add the `MX_SITE_PREFERENCE` *variable to the Live Collaboration Server's initialization file (enovia.ini). This adjustment overrides the setting in the person, group, or role definition for the site preference, and should be set to the site that is local to the server. This ensures optimum performance of file chicken and checkout for Web clients.*

As a Project Clause

A project is created by defining a role as a project with the Asaproject clause. Roles are often hierarchical and it is important to note that you will not be able to define a role as a project if the parent of that role is not defined as a project.

```
add role "my project" asaproject;
```

In the example above, the project, “my project,” is created. The name of a project must be unique and cannot be used by another role or user category. Projects are still of the type role user category and have the same parameters as roles. The type shown for a project will be a role. The selectable list for a project is exactly the same as for a role.

As an Organization Clause

An organization is created by defining a role as an organization with the Asanorg clause. Roles are often hierarchical and it is important to note that you will not be able to define a role as an organization if the parent of that role is not defined as an organization.

```
add role “my organization” asanorg;
```

In the example above, the organization, “my organization,” is created. The name of an organization must be unique and cannot be used by another role or user category. Organizations are still of the type role user category and have the same parameters as roles. The type shown for an organization will be a role. The selectable list for an organization is exactly the same as for a role.

As a Role Clause

The Asarole clause is used to modify a role defined as a project or organization back into a role. Use this clause with the Modify Role command.

```
mod role "my project" asarole;
```

In the example above, the role previously defined as the project, “my project,” is turned back into a role.

Ancestor selectable on Group and Role

The select keyword “ancestor” is allowed against groups and roles, which returns the entire upward hierarchy of parents, as well as the user against which it is executed. For example:

```
project.ancestor match context.user.assignment.parent
```

This evaluates to true for data allowed for a user’s project. See also *businessobject Command*.

History Clause

The `history` keyword adds a history record marked “custom” to the group or role that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Modify Role or Group

Use the Modify Role or Modify Group command to modify a role or group, respectively:

```
modify role|group NAME [MOD_ITEM {MOD_ITEM}]
```

The MOD_ITEM clauses for a role are:

```
[add] assign person PERSON_NAME [group GROUP_NAME]
name NEW_NAME
description VALUE
parent ROLE_NAME
child ROLE_NAME {,ROLE_NAME}
maturity [none | public | protected | private]
remove assign | person PERSON_NAME [group GROUP_NAME] |
              | all                                    |
remove child  | ROLE_NAME {,ROLE_NAME} |
              | all                    |
remove parent
site SITE_NAME
[!|not]hidden
| asaproject |
| asanorg    |
| asarole    |
add property NAME [to ADMIN] [value STRING]
remove property NAME [to ADMIN]
property NAME [to ADMIN] [value STRING]
```

The MOD_ITEM clauses for a group are:

```
[add] assign person PERSON_NAME [role ROLE_NAME] (for a group)
name NEW_NAME
description VALUE
parent GROUP_NAME
child GROUP_NAME {,GROUP_NAME}
remove assign | person PERSON_NAME [role ROLE_NAME] |
              | all                                 |
remove child  | GROUP_NAME {,GROUP_NAME} |
              | all                      |
remove parent
```

site SITE_NAME
[!|not]hidden
add property NAME [to ADMIN] [value STRING]
remove property NAME [to ADMIN]
property NAME [to ADMIN] [value STRING]

After you establish a group or role definition, you can add or remove defining values. Refer to the description in *Modify Item*. When modifying a group or role, it is important to consider how accesses are shared between parent and children. If you do not want a child to assume the same accesses to business objects as the parent, you will have to modify the policy so the privileges for the child role are specified. If a role is not specifically referenced in a policy, Live Collaboration looks for access hierarchically. For example, if the role has a parent and the parent is named in the policy, the child shares the parent's privileges.

If you remove a parent or child, you might inadvertently remove access privileges from the children. Make sure that you consult the policies you have created when you alter the hierarchy of defined groups and roles.

A role definition can only be modified into a project or organization if the parent of that role is of the same role definition: a role defined as a project or a role defined as an organization respectively. You can modify a role defined as either a project or organization back into a role with the asarole clause.

If an application is already using roles to model project or organization user categories, you can use the Asaproject clause or Asanorg clause to officially define those roles as projects or organizations.

History Clause

The `history` keyword adds a history record marked "custom" to the group or role that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Delete Group or Role

If a group or role is no longer required, you can delete it. Refer to the description in *Delete Item*. When deleting a group or role, the elimination of linkages may affect another group's or role's access to business objects. For example, suppose you have these two roles: Assembly Manager and Component Assembler. Assembly Manager is the parent of Component Assembler.

According to the policy governing the Assembly Manager role, the Assembly Manager role has read access to all business objects of type Assembly. Since a child role inherits the access abilities of its parent role (unless specified otherwise), the Component Assembler role also has read access.

If the Assembly Manager role is deleted, the linkages disappear between Assembly Manager and Component Assembler. Component Assembler becomes a stand-alone role and loses the read access inherited from the Assembly Manager role. If this access was critical, you can modify the role definition for Component Assembler with read access.

rule Command

Description

Rules can be used to limit user access to attributes, forms, programs, and relationships. Unlike policies, rules control access to these administrative objects regardless of the object type or state. .

When you create a rule, you define access using the three general categories used for in policies: public, owner, and user (specific person, group, role, or association).

For conceptual information on this command, see *Rules* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify|delete|list|print|assign] rule NAME {CLAUSE};
```

- NAME is the name you assign to the rule. Rule names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the rule.

Add Rule

Rules are defined using the Add Rule command:

```
add rule NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM defines information about the rule including the description and access masks. The Add Rule clauses are:

description VALUE
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]

Where ACCESS_USER is:

```
    | [revoke] [login] public [key STRING]          |
    | [revoke] [login] owner [key STRING]           |
    | [revoke] [login] user USER_NAME [key STRING]  |
```

Where ACCESS_ITEM is:

```
| all                  |
| none                 |
| [not]changevault     |
| [not]changename      |
| [not]changeowner     |
| [not]changepolicy    |
| [not]changetype      |
| [not]checkin         |
| [not]checkout        |
| [not]create          |
| [not]grant           |
| [not]delete          |
| [not]demote          |
| [not]disable         |
| [not]enable          |
| [not]execute         |
| [not]freeze          |
| [not]fromconnect     |
| [not]fromdisconnect  |
| [not]lock            |
| [not]majorrevise     |
| [not]modify          |
| [not]modifyform      |
| [not]override        |
| [not]promote         |
| [not]read            |
| [not]reserve         |
| [not]unreserve       |
| [not]revoke          |
| [not]revise          |
| [not]schedule        |
| [not]show            |
| [not]thaw            |
| [not]toconnect       |
| [not]todisconnect    |
| [not]ubnlock         |
| [not]viewform        |
```

Where USER_ITEM is:

[any | single | ancestor | descendant] organization
[any | single | ancestor | descendant] project
[any | context] owner
[any | no | context | inclusive] reserve
[any | no | public | protected | private |notprivate|ppp] maturity

```
[any|oem|goldpartner|partner|supplier|customer|contractor]category
[filter | localfilter] EXPR
```

Some of the clauses are required and some are optional, as described in the sections that follow.

Assigning Access

Each access form is used to control some aspect of a business object's use. With each access, other than `all` and `none`, you can either *grant* the access or explicitly *deny* the access. You deny an access by entering `not` (or `!`) in front of the access in the command. For every administrative object (attribute, form, program, relationship) governed by a rule, a user has access only if the rule specifically grants the user access. If the user isn't granted access by the rule, the user won't have access, even if the policy grants access to the user.

For example, suppose you don't want anyone to modify an attribute called Priority unless they belong to the Management role. However, everyone should be able to view the attribute's values. (Assume the person and policy definitions grant the appropriate accesses.) You would create a rule that governs the Priority attribute and define the following accesses:

Owner: read

Public: read

Management role: all

The available accesses are summarized in *Accesses* in Chapter 3.

It is important to keep in mind that while the complete list of access items is available when creating rules, when they are actually used, only the applicable privileges are checked. The table below shows the accesses each administrative type uses:

Accesses Used By:				
Attributes	**Forms**	**Programs**	**Relationships**	
read	viewform	execute	toconnect fromconnect	freeze
modify	modifyform		todisconnect fromdisconnect	thaw
			changetype	modify (attributes)
Owner Access does not apply to Relationships.				

Use the following syntax to define access rules:.

```
ACCESS_USER ACCESS_ITEM {,ACCESS_ITEM} [{USER_ITEM}]
```

where ACCESS_USER is:

```
    | [revoke] [login] public [key STRING]          |
    | [revoke] [login] owner [key STRING]           |
    | [revoke] [login] user USER_NAME [key STRING] |
```

Use the public, owner, and user keywords to specify that this rule applies to everyone (public), the object owner (owner), or to a named role (user USERNAME).

With revoke, all accesses specified as ACCESS_ITEMs are rescinded. If all conditions are true, then accesses are rescinded for the current user.

Revoke supersedes all other rules. If a rule revokes access for a user, no other rule can grant access to that user.

Login specifies that the rule only applies if the role it specifies matches a role associated with the user's current login context. This is more restrictive than the default, where the system looks at all user assignments to see if a rule applies to any of them.

The key STRING is a label that differentiates multiple rules defined for the same user. For example, if you have two rules defined for public or for user Designer, you can use the key STRING to need to differentiate them.

ACCESS_ITEM specifies the kinds of access this rule governs, either granting or revoking access. The ACCESS_ITEMs are described in *Accesses* in Chapter 3. With each ACCESS_ITEM, other than all and none, you can either grant the access or explicitly deny the access. You deny an access by entering not (or !) in front of the ACCESS_ITEM in the command. For example, !todisconnect or notchangeowner.

where USER_ITEM is:

`[any | single | ancestor | descendant] organization`
`[any | single | ancestor | descendant] project`
`[any | context] owner`
`[any | no | context | inclusive] reserve`
`[any | no | public | protected | private |notprivate|ppp] maturity`
`[any|oem|goldpartner|partner|supplier|customer|contractor]category`
`[filter | localfilter] EXPR`

The USER_ITEM specifies a variety of ways to define matching options between the current context's membership in various organization and policy against the project or organization ownership of the object being checked.

The flags have the following meanings:

- `any|single|ancestor|descendant]organization`—Specifies that the object's organization owner must be checked against the context's member organization.
- `any|single|ancestor|descendant] project`—Specifies that the object's organization owner must be checked against the context's member project.
- `any owner`—No check is performed on owner (default).
- `context owner`—Checks that object is owned by context user.
- `any reserve`—No check is performed on whether the object is reserved (default).
- `no reserve`—Checks that the object is not reserved by anyone.
- `context reserve`—Checks that the object is reserved by the current context user.
- `inclusive reserve`—Checks that the object is either reserved by the current context user or by no one.

- `any maturity`—No check is performed on the maturity of the project owning the object (default).
- `public maturity`—Checks that the object is owned by a project that is public.
- `protected maturity`—Checks that the object is owned by a project that is public.
- `private maturity`—Checks that the object is owned by a project that is private.
- `notprivate maturity`—Checks that the object is owned by a protected or public project.
- `ppp maturity`—Member of a private, protected, or public project, that is any project with the maturity property.
- `any|oem|goldpartner|partner|supplier|customer|contractor]category`—Checks that the current context or current login security context is flagged with the corresponding property.
- `[filter | localfilter] EXPR`—A filter expression defined for the user access. Use localfilter instead of filter in policies and access rules to return only results to which the current user definitely has access. When you use localfilter, the expression is not evaluated by full-text search.
  Expression access filters can be any expression that is valid in the system. Expressions are supported in various modules of the system, for example, query where clauses, expand, filters defined on workspace objects such as cues, tips and filters, etc. See *Working With Expression Access Filters* in Chapter 3 for details about how to use access filters.

History Clause

The `history` keyword adds a history record marked “custom” to the rule that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Rule

After a rule is defined, you can clone the definition with the Copy Rule command. This command lets you duplicate rule definitions with the option to change the value of clause arguments:

```
copy rule SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM};
```

- `SRC_NAME` is the name of the rule definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The `history` keyword adds a history record marked “custom” to the rule that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Rule

After a rule is defined, you can change the definition with the Modify Rule command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify rule NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the rule to modify.
- MOD_ITEM is the type of modification to make. Each is specified in a Modify Rule clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Rule Clause	Specifies that…
name NAME	The current name is changed to the new name.
description VALUE	The current description, if any, is changed to the value entered.
icon FILENAME	The image is changed tp the new image in the field specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
add owner ACCESS {,ACCESS}	The specified owner access is added. Values for ACCESS can be found in the *Accesses* in Chapter 3.
remove owner ACCESS {,ACCESS}	The specified owner access is removed. Values for ACCESS can be found in the *Accesses* in Chapter 3.
add public ACCESS {,ACCESS}	The specified public access is added. Values for ACCESS can be found in the *Accesses* in Chapter 3.
remove public ACCESS {,ACCESS}	The specified public access is removed. Values for ACCESS can be found in the *Accesses* in Chapter 3.
add user USER_NAME ACCESS {,ACCESS} [filter EXPRESSION]	The specified user access is added. USER_NAME defines the person, group, role or association for which the rule is being modified. Values for ACCESS can be found in the *Accesses* in Chapter 3.
remove user USER_NAME ACCESS {,ACCESS} [filter EXPRESSION]	The specified user access is removed. USER_NAME defines the person, group, role or association for which the rule is being modified. Values for ACCESS can be found in the *Accesses* in Chapter 3.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.
history STRING	Adds a history record marked “custom” to the rule that is being modified. The STRING argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Assign Rule

Once you have created Rules, you can assign them to existing administrative objects that require them.

Each Attribute, Form, Program, and Relationship definition can refer to one Rule that encompasses owner, public and as many different users as required. The Rule becomes part of the definition of these kinds of objects.

Attributes, Forms, Programs, and Relationships created before version 6.0 have no Rules defined. By default, all existing definitions will still have no Rules, and will remain available to all users as before. However, use the procedures below if you want to add a Rule to an existing definition.

To add a Rules::

```
modify attribute ATTRIBUTENAME add rule RULE_NAME;
```

```
modify form FORMNAME add rule RULE_NAME;
```

```
modify program PROGNAME add rule RULE_NAME;
```

```
modify relationship RELNAME add rule RULE_NAME;
```

Delete Rule

If you decide that a rule is no longer required, you can delete it by using the Delete Rule command:

```
delete rule RULE_NAME;
```

`RULE_NAME` is the name of the rule to be deleted. If the name is not found, an error message will result.

When this command is processed, Live Collaboration searches the list of rules. If the name is found, that rule is deleted IF there are no objects that are governed by the rule. If there are objects that are governed by the rule, they must be reassigned to another rule or the object must be deleted before the rule can be removed from the rule list.

To remove a rule from a specific administrative object, use the Remove Rule command on the object's modify command. For example, to remove a rule from an Attribute, use the `modify attribute` command:

```
modify attribute ATTRIBUTENAME remove rule RULE_NAME;
```

Rules can be removed from attributes, forms, policies, relationships, and programs. Refer to *Assign Rule* for additional information.

searchindex Command

Description

This command controls Advanced Search processes, such as starting or monitoring an indexing process. The file indexing queue is backed up by a persistent, file-based queue, which represents the pending jobs in the queue. This specifically affects the Start, Status, Clear, and Stop Searchindex commands. See the description of each command for details on how the file-based queue affects that command.

User level

System Administrator

Syntax

```
[start|update|stop|status|modify|clear|validate|help|print]
searchindex {CLAUSE};
```

- CLAUSEs provide additional information about the search index.

Start Searchindex

You start an indexing process with the Start Searchindex command:

```
start searchindex mode FULL|PARTIAL [vault PATTERN] [limit N];
```

- `mode` specifies whether a baseline (`FULL`) or incremental (`PARTIAL`) indexing process is to be performed. This clause is mandatory.
- `vault` specifies the vault(s) to be indexed. The `PATTERN` argument allows you to specify a pattern of vault names, including a wildcard '*' character. For example, if there were two vaults named v1 and v2, instead of issuing the command twice, once for each vault, you could simply issue it once using `vault v*`. This clause is optional. See also *Status Searchindex*.
- `limit` specifies the maximum number (`N`) of objects to be indexed. This clause is optional.

Update Searchindex

You update a type in the index with a new attribute using the Update Searchindex command. The syntax for Full-text Search Server with Exalead is as follows:

```
update searchindex [vault PATTERN] type TYPE{,TYPE}[ type
TYPE{,TYPE}] [!|not][file];
```

- `vault` specifies the vault(s) to be indexed. The `PATTERN` argument allows you to specify a pattern of vault names, including a wildcard '*' character. For example, if there were two vaults named v1 and v2, instead of issuing the command twice, once for each vault, you could simply issue it once using `vault v*`. This clause is optional. See also *Status Searchindex*.
- `type` specifies the object type to be updated.
- For Full-text Search Server with Exalead, the command `update searchindex type Part`, for example, is the only option to update the index with that new field.

Stop Searchindex

You stop an indexing process with the Stop Searchindex command. This command stops the persistent-queue fetch process on each FCS server that it can reach. It leaves the persistent queue intact with any pending jobs that are still in the queue unprocessed.

```
stop searchindex [vault PATTERN];
```

- `vault` specifies the vault(s) to be indexed. The `PATTERN` argument allows you to specify a pattern of vault names, including a wildcard '*' character. For example, if there were two vaults named v1 and v2, instead of issuing the command twice, once for each vault, you could simply issue it once using `vault v*`. This clause is optional. See also *Status Searchindex*.

Status Searchindex

You can check the status of an indexing process with the Status Searchindex command. This command polls each FCS Server that it can reach to see whether there is a persistent queue and if so, reports details of that queue including its directory.

The status for the Searchindex commands had previously been stored in the vault property SearchIndexStats. It is now stored in the index as a 'checkpoint.' A checkpoint is much like a property in that it can store an arbitrary string, but it is also associated with the data that has been committed in the index, so it ties the vault status to what has been processed in the index.

```
status searchindex [vault PATTERN];
```

- `vault` specifies the vault(s) to be indexed. The `PATTERN` argument allows you to specify a pattern of vault names, including a wildcard '*' character. For example, if there were two vaults named v1 and v2, instead of issuing the command twice, once for each vault, you could simply issue it once using `vault v*`. This clause is optional.

The output of this command includes the start time and duration of the indexing process, for example:

```
MQL<4>status searchindex;
test: Complete  last indexed: 08/23/2011 20:48:06       number of
objects: 1    start: 9/9/2011 5:33:10 PM      duration: 7 seconds
```

Modify Searchindex

You can change the last_modified time stamp of an indexing process with the Modify Searchindex command. This command is useful, for example, if the last_modified date becomes corrupted or some other issue requires already indexed data to be re-indexed. The syntax of this command is as follows:

```
modify searchindex modified DATE [vault VAULT];
```

- `DATE` specifies the last_modified timestamp of an incremental indexing operation. This clause is required. The format is `MM/dd/yyyy HH:mm:ss`. You will be informed with an error message if the format of the entered data is incorrect.
- `vault` specifies the vault(s) to be indexed. This clause is optional.

Clear Searchindex

You clear an index from the Advanced Search Server with the Clear Searchindex command. This command clears the persistent queue on each FCS Server that it can reach.

```
clear searchindex;
```

The following command clears only the status of the indexing process.

```
clear searchindex statsonly [vault PATTERN];
```

- `vault` specifies the vault(s) to be indexed. The `PATTERN` argument allows you to specify a pattern of vault names, including a wildcard '*' character. For example, if there were two vaults named v1 and v2, instead of issuing the command twice, once for each vault, you could simply issue it once using `vault v*`. This clause is optional. See also *Status Searchindex*.

Validate Searchindex

You can validate the search index with the Validate Searchindex command.

```
validate searchindex;
```

Help Searchindex

You can display help for all of the indexing commands with the Help Searchindex command.

```
help searchindex;
```

Print System Searchindex

You can print the config.xml file currently stored in the database with the Print System Searchindex command.

```
print system searchindex;
```

server Command

Description

Use this command to add a server.

User level

System Administrator

Syntax

```
[add | copy | delete | disable | list | modify | monitor |
print] server NAME {CLAUSE};
```

- NAME is the name you assign to the server. Server names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the server.

Add Server

Use the Add Server command to define a server:

```
add server NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides more information about the server you are creating. Although the clauses are not required to make the server usable, they are used to define a server location other than the current default host and path. In addition, the clauses can help users understand the purpose of the server.

The Add Server clauses are:

description VALUE
user USER_NAME
password PASSWORD
connect CONNECT_STRING
timezone TIMEZONE
[!|not]foreign
[!|not]hidden
property NAME [value STRING]
property NAME to ADMIN [value STRING] where ADMIN is: TYPE NAME

Copy Server

Use the Copy Server command to copy a server.

```
copy server SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

Delete Server

Use the Delete Server command to delete a server.

```
delete server NAME;
```

Disable Server

Use the Disable Server command to disable a server.

```
disable server NAME;
```

List Server

Use the List Server command to list servers.

```
list server NAME_PATTERN [SELECT [DUMP[RECORDSEP]]][tcl]
[output FILENAME];
```

where SELECT is:

select [[! | not] substitute] FIELD_NAME {FIELD_NAME}

where FIELD_NAME is:

SUB_FIELD [.SUB_FIELD{.SUB_FIELD}]

where SUB_FIELD is:

string

string[string]]

where DUMP is:

dump [SEPARATOR_STRING]

where RECORDSEP is:

recordseparator [SEPARATOR_STRING]

Modify Server

Use the Modify Server command to modify a server.

```
modify server NAME[MOD_ITEM{MOD_ITEM}];
```

where MOD_ITEM is:

description VALUE

user USER_NAME

password PASSWORD

connect CONNECT-STRING

timezone TIMEZONE

[!|not]foreign

[!|not]hidden

add | property NAME to ADMIN [value STRING]

| property NAME [value STRING]

remove | property NAME to ADMIN

| property NAME

property NAME to ADMIN [value STRING]

property NAME [value STRING]

Monitor Server

Use the Monitor Server command to monitor a server.

```
monitor server [[port]|[[age SECONDS][xml]][filename
FILENAME[append]];
```

Print Server

Use the Print Server command to print from a server.

```
print server NAME [SELECT][DUMP] [tcl] [output FILENAME];
```

where SELECT is:

select [[! | not] substitute] FIELD_NAME {FIELD_NAME}

where DUMP is:

dump [SEPARATOR_STRING]

Disable Server

Use the Disable Server command to disable a server.

```
disable server NAME;
```

sessions Command

Description

Some of the tasks you must perform as the System Administrator require that you shut down Live Collaboration or system software. Before doing this you need to know which users are currently using the database

User level

System Administrator

View User Session Information

You can use the Sessions command to view a list of all current users.

```
sessions;
```

This command provides output of the form:

```
USERNAME MACHINENAME PROGRAMNAME
USERNAME MACHINENAME PROGRAMNAME
```

`PROGRAMNAME` includes the path, executable name, and any command line options being used. For example:

```
peter PETESMACHINE c:\enoviaV6R2011\studio\bin\winnt\mql.exe -k
```

If the executable was started with a shortcut on Windows, the `PROGRAMNAME` *displayed is limited to the first 64 characters.*

You must be logged in as the System Administrator to use the Sessions command. If you receive the following error message, access to the table that stores session information was not configured when the system was installed:

```
Table or View does not exist.
```

The Oracle Database Administrator must run the following SQL*Plus command:

```
GRANT SELECT ON "SYS"."V_$SESSION" TO "MATRIX";
```

set Command

Description

A *set* is a logical grouping of business objects created by an individual user. Sets are often the result of a query made by a user. While sets can be manually constructed based on the needs of the individual, queries are a fast means of finding related business objects.

The contents of a set can be viewed at any time and objects can be added or deleted from a set easily. However, a user has access only to sets created while in session under his/her context.

For conceptual information on this command, see *Sets* in Chapter 6.

User Level

System Administrator

Syntax

```
[add|copy|modify|expand|print|sort|delete]set NAME {CLAUSE};
```

- NAME is the name of the set you are defining.
- CLAUSEs provide additional information about the set.

Add Set

Sets are often the result of a query. To save the contents of a query in a set, see *Evaluate Query*.

To improve performance, it is recommended that you use a list of object IDs or physicalIDs when you add a new set or modify a set to add new objects, instead of using a list of TNR.

The kernel is lenient towards inclusion of invalid or non-existent IDs in the list when adding or modifying sets. The objects when not present or incorrect are not included in the set and there are no errors for invalid or non-existent IDs during the operation.

For example, the following code works without any errors and the set is created or modified, but without any new objects added to it:

```
add set set_1 member businessobject
00000000000000000000000000000000,11111111111111111111111111111111;

mod set set_1 add businessobject
00000000000000000000000000000000,11111111111111111111111111111111;
```

Use the following command to show any valid objects:

```
print set set_1
```

To manually define a set from within MQL, use the Add Set command:

```
add set NAME [user USER_NAME][ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the set you are defining.

USER_NAME can be included with the user keyword if you are a business administrator with person access defining a set for another user. If not specified, the set is part of the current user's workspace.

All sets must have a unique name assigned within a given context. If you duplicate a name, an error message is displayed. Although a single user cannot duplicate a set name, different users can use the same name. Sets are local to the context of individual users. This means that several users could each have a set called "Current Project." However, as you change context from one user to another, the contents of "Current Project" will most likely vary.

For more information, see *Administrative Object Names*.

ADD_ITEM is an Add Set clause that provides more information about the set you are defining. None of the clauses is required. The Add Set clauses are:

member businessobject OBJECTID [in VAULT]
member businessobject ID SEARCH_CRITERIA
SEARCH_CRITERIA
[!|not]hidden
visible USER_NAME{,USER_NAME}
property NAME [to ADMINTYPE NAME] [value STRING]

- OBJECTID is the OID or Type Name Revision of the business object.

Each clause and the arguments they use are discussed in the sections that follow.

Member Clause

After assigning a set name, specify the business objects to include in the set. Business objects can be referenced by their complete business object name/specification or by their ID:

```
member businessobject TYPE NAME REVISION [in VAULT]
```

Or

```
member businessobject ID
```

When using the complete specification, you *must* include the object type, the name of the object, and the revision designator in this order. Note that some business objects may not have a revision designator. If a revision designator was not assigned to an object, "" (double quotes) are required in MQL.

The following are examples of complete business object names:

```
businessobject Assembly "Keyboard" ""
businessobject Assembly Monitor E
businessobject Assembly "Laptop Computer" AA
```

To group these business objects into a set, you can write a command similar to the following:

```
add set "Computer Set"
   member businessobject Assembly "Keyboard" ""
```

```
member businessobject Assembly Monitor E
member businessobject Assembly "Laptop Computer" AA;
```

When this command is processed, a set called “Computer Set” is created. It contains three business objects. These objects will appear grouped in a window whenever the set name is referenced.

You can also optionally specify the vault in which the business object is held. When the vault is specified, only the named vault needs to be checked to locate the business object. This option can improve performance for very large databases.

When business objects are created they are given an internal ID. As an alternative to `TYPE NAME REVISION`, you can use this ID when indicating the business object to be acted upon. The ID of an object can be obtained by the use of the `print businessobject... selectable` command. Refer to *Print Business Object* for information.

Search Criteria

A `SEARCHCRITERIA` can be added to a set. This allows you to add sets, queries, temporary sets, temporary queries and expansions of business objects to sets. Any of these items can be used in combinations covering multiple levels of complexity using the binary operators `and`, `or` and `less`. For details, see *SEARCHCRITERIA Clause*.

Copy Set

After a set is defined, you can clone the definition with the Copy Set command.

If you are a Business Administrator with person access, you can copy sets in any person’s workspace (likewise for groups and roles). Other users can copy visible sets to their own workspaces.

This command lets you duplicate set definitions with the option to change the value of clause arguments:

```
copy set SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the set definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any set of the same name belonging to the user specified in the `Touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Set

If you are a Business Administrator with person access, you can modify sets in any person's workspace (likewise for groups and roles). Other users can modify only their own sets.

You must be a business administrator with group or role access to modify a set owned by a group or role.

To improve performance, it is recommended that you use a list of object IDs or physicalIDs when you add a new set or modify a set to add new objects, instead of using a list of TNR.

The kernel is lenient towards inclusion of invalid or non-existent IDs in the list when adding or modifying sets. The objects when not present or incorrect are not included in the set and there are no errors for invalid or non-existent IDs during the operation.

For example, the following code works without any errors and the set is created or modified, but without any new objects added to it:

```
add set set_1 member businessobject
00000000000000000000000000000000,11111111111111111111111111111111;

mod set set_1 add businessobject
00000000000000000000000000000000,11111111111111111111111111111111;
```

Use the following command to show any valid objects:

```
print set set_1
```

You can use the Modify Set command to modify any set to add or remove business objects and change set options:

```
modify set NAME [user USER_NAME] [MOD_ITEM {MOD_ITEM}];
```

`NAME` is the name of the set to modify. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.

`MOD_ITEM` is the type of modification to make. Each is specified in a Modify Set clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Set Clause	Specifies that…
`add businessobject TYPE NAME REVISION [in VAULT]`	The named business object is added to the set.
`add businessobject ID`	The business object with the specified ID is added to the set.
`remove businessobject TYPE NAME REVISION [in VAULT]`	The named business object is removed from the set.
`remove businessobject ID`	The business object with the specified ID is removed from the set.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`visible USER_NAME{,USER_NAME};`	The object is made visible to the other users listed.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.

Modify Set Clause	Specifies that…
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

Each modification clause is related to the clauses and arguments that define the set.

For example, assume you have a set named “Product Comparison.” You want to add two new business objects and remove an old one. You could write a command similar to the following:

```
modify set "Product Comparison"
   add businessobject "Scent Formula" "Perfume P39" B;
   add businessobject "Scent Formula" "Perfume P40" A;
   remove businessobject "Scent Formula" "Scent P13" D;
```

To remove a set entirely, use the Delete Set command, as described in *Delete Set*. To change the name of the set, remove the current set and create a new set.

Expand Set

Print Business Object describes how you can use the Expand Businessobject command to display connections made between two objects. The Expand Businessobject command enables you to see other objects connected to the one you have. Depending on which form of this command you use, you can display the objects that are connected to it, connected from it, meet selected criteria, or are of a specific type.

When using the Expand Businessobject command, you are working with a single business object. To expand multiple objects, use the Expand Set command, which expands each of the objects contained within a set. The Expand Set command also lets you search for relationships that meet the search criteria.

```
expand set NAME [user USER_NAME] [EXPAND_ITEM{EXPAND_ITEM}];
```

`NAME` is the full specification of a set name.

Since MQL will use the names of the objects from the set, you do not need to specify an object with this command. MQL will expand each set depending on the keyword (and values) that you specify.

In addition to a list of business objects that meet the specified criteria, the Expand Set command provides information about the connections. The level number of the expansion and the relationship type of the connection are provided along with the business object name, type, and revision.

In addition, you can use the Recurse argument to indicate that you want to expand the business objects connected/related to the initially specified business object.

Refer to *Evaluate Query*, for a description of the Evaluate command to *find on a set*. `EXPAND_ITEM` is one of:

`from`
`to`
`type PATTERN {,PATTERN}`
`relationship PATTERN {,PATTERN}`

fixed points ID [,Id]
withroots
filter PATTERN
activefilters
reversefilters
view NAME
recurse [to | N |]
recurse to |all|
[into|onto] set NAME
SELECT_BO
where WHERE_CLAUSE
SELECT_REL
DUMP

Relationship Expressions

Select expressions are applied directly to relationship instances, enabling selection of its relationship attribute values for use in Indented Tables and Visual Cues.

For example, the following can be used in a table column definition:

```
relationship [Drawing].attribute[Quantity] =
```

Expanding From

When you use the From form of the Expand Set command, you expand away from the starting object. This gives you all the related objects that are defined as the TO ends in a relationship. Consider the starting object as the tail of the arrow and you are looking for all the arrow heads. The From form uses the syntax:

```
expand set NAME from [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]
   [SET_CLAUSE|DUMP CLAUSE]];
```

The From form of the Expand clause returns all objects connected from the starting object (the arrow points out) regardless of the relationship used.

- `NAME` is the full specification of a set name.
- `PATTERN` is the pattern of the relationship or type name.
- `RECURSE_CLAUSE` expands the business object from which the initially specified business object is connected (as described in *Recurse Clause*).
- `SET_CLAUSE` places the output of the expand command into a set. All forms of the Expand Set command use a Set clause (as discussed in *Set Clause*).

- DUMP_CLAUSE specifies a general output format for the expanded information (as discussed in *Dump and Output Clauses*).

For example, assume you have a set of components which are used in larger assemblies. You need a list of all objects that each part goes into. The command would be:

```
expand set Components from;
```

This might produce a list of objects such as a calculator, telephone, VCR, and so on. Since no other criteria is specified, the From form of the Expand Set command gives you ALL objects that occupy the TO end of a relationship definition.

Expanding To

When the To form of the Expand Set command is used, the set's business objects are again used as the starting point. However, now it is assumed that each object is defined as the TO end of the relationship definition. Therefore, you are looking for all objects that lead *to* the named object. These are the objects defined as the FROM ends in a relationship definition.

The To form of the Expand Set command uses the syntax:

```
expand set NAME to [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]
   [SET_CLAUSE|DUMP CLAUSE];
```

The To form of the Expand clause returns all objects connected to the starting object (the arrow points in) regardless of the relationship used.

For a description of the clause, see *Expanding From*.

Using the To form is useful when you want to work backwards. For example, you may want to know what components make up each assembly in a defined set. For example:

expand set "Assembly Components" to;

This might give you objects that contain buttons, plastic housings, printed circuit boards, etc. All related objects defined as the FROM connection end are listed.

Relationship Clause

This clause displays all objects connected in a specific relationship. This is useful when you are working with an object that may use multiple relationship types. If the starting object can only connect with one relationship, this form has the effect of listing all the connection ends used by the starting object. These ends may be defined as a TO end or a FROM end—it does not matter. Only the relationship type is of importance.

The Relationship clause uses the syntax:

```
expand set NAME [from|to] [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]
 [SET_CLAUSE|DUMP CLAUSE];
```

The Relationship clause of the Expand Set command returns all objects connected to the starting object with a specific relationship. The command can be made more particular by specifying the direction of the relationship and/or the types of objects to return.

For a description of the clause, see *Expanding From*.

For example, assume you have a set of drawing objects. These drawings may use a number of relationships such as a User Manual Relationship, Design Relationship, Marketing Relationship, Drawing Storage Relationship, etc. You may want to examine all objects that use a particular type of relationship. For example, you might want a list of all objects that have used each drawing in some marketing way. To do this, you might enter a command similar to:

```
expand set "Drawing Collection" relationship "Marketing Relationship";
```

This command expands each object contained within the set named “Drawing Collection.” As each object is expanded, MQL lists any related objects that have the Marketing Relationship type. It searches through all the object connections for the Marketing Relationship type. If MQL finds a relationship with the type that uses the object being expanded, the other connection end is displayed and the end’s directional relationship is identified. It does not matter whether the related objects are defined as the FROM end or the TO end of the relationship. Only the relationship type is of importance.

Type Clause

This clause displays all related objects of a specific object type. This is useful when you are working with a set of objects that may be connected to multiple object types. If the starting object can only connect with one object type, this form is similar to the Relationship form using a wildcard pattern. The Type form uses the syntax:

```
expand set NAME [from|to] [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]
 [SET_CLAUSE|DUMP CLAUSE];
```

For a description of the clause, see *Expanding From*.

For example, assume you have a set that contains training course objects. To each object, you might have related objects that are of type Evaluation, Student, Materials, Locations, etc. You may want to examine all Evaluation type objects in order to trace a course’s progress in meeting student needs. To do so, enter a command similar to:

```
expand set “Training Courses” type Evaluation;
```

This command expands each of the objects contained in the “Training Courses” set. It lists any related objects that have the Evaluation type. Those objects might belong to multiple relationship types (such as “Professional Evaluation Relationship” or “Student Evaluation Relationship”). It does not matter if the related objects are defined as the FROM end or the TO end of the relationship. Only the object type matters in the location and display of the output objects.

Withroots Keyword

You can include the root object by including the `withroots` keyword with all selectables to make it easier for client apps to identify a hierarchy.

Examples Without the `withroots` Keyword	Examples With the `withroots` Keyword

`MQL<9>expand set set_1 recurse to 2;`
`1 r2 to t2 t2-2 0`
`2 r2 to t2 t2-3 0`
`2 r3 to t2 t2-2-2 0`
`2 r3 to t2 t2-2-3 0`
`2 r1 from d12 d12-2 0`
`2 r1 from d11 d11-3 0`
	`MQL<10>expand set set_1 recurse to 2 withroots;`
	`0 t2 t2-1 0`
	`1 r2 to t2 t2-2 0`
	`2 r2 to t2 t2-3 0`
	`2 r3 to t2 t2-2-2 0`
	`2 r3 to t2 t2-2-3 0`
	`2 r1 from d12 d12-2 0`
	`2 r1 from d11 d11-3 0`
	`2 r2 from t2 t2-1 0`

`MQL<19>expand set set_1 recurse to 2 select bus id dump;`
`1 r2 to t2 t2-2 0 20768.23721.38088.36415`
`2 r2 to t2 t2-3 0 20768.23721.38088.39031`
`2 r3 to t2 t2-2-2 0 13576.55552.30896.29472`
`2 r3 to t2 t2-2-3 0 6384.39445.50345.7641`
`2 r1 from d12 d12-2 0 20768.23721.38088.32444`
`2 r1 from d11 d11-3 0 20768.23721.38088.24862`
	`MQL<20>expand set set_1 recurse to 2 withroots select bus id dump;`
	`0 t2 t2-1 0 20768.23721.38088.35693`
	`1 r2 to t2 t2-2 0 20768.23721.38088.36415`
	`2 r2 to t2 t2-3 0 20768.23721.38088.39031`
	`2 r3 to t2 t2-2-2 0 13576.55552.30896.29472`
	`2 r3 to t2 t2-2-3 0 6384.39445.50345.7641`
	`2 r1 from d12 d12-2 0 20768.23721.38088.32444`
	`2 r1 from d11 d11-3 0 20768.23721.38088.24862`

Filter and Activefilter Clauses

This clause allows you to specify an existing filter(s) that is defined within your context to be used for the expansion. You can use wildcard characters or an exact name. For example:

```
expand set Assembly 12345 1 filter OpenRecs;
```

In addition, you can use the activefilter clause to indicate that you want to use all filters that are enabled in within your context. For example:

```
expand set Assembly 12345 1 activefilter;
```

Recurse Clause

Once you have a list of related objects, you may also want to expand these objects. The Recurse clause of the Expand Businessobject command expands through multiple levels of hierarchy by applying the Expand command to each business object found:

```
recurse to [N|all]
```

`N` is any number indicating the number of levels that you want to expand.

`all` indicates that you want to expand all levels until all related business objects are found.

Set Clause

Once you have a list of related objects, what do you do with them? In some cases, you can simply search for a particular object and you will not need to reference the output object again. In that case, you might want to display the expansion output on your terminal. However, in other cases, you may want to capture the output and save it. That is the reason for the Set clause of the Expand Set command.

The Set clause uses the following syntax:

```
into|onto set SET_NAME
```

`SET_NAME` is a valid name value that will be assigned to the set.

When the Set clause is included within the Expand Set command, the related objects are placed within a set and assigned a set name. This set may already contain values or it may be a new set created for the purpose of storing this output.

When it is an existing set, the previous values are either replaced or added onto depending on the keyword you use to begin the Set clause:

`into`	The existing set contents are discarded and only the current output is saved.
`onto`	The new output is appended onto the existing set contents. This is the same as when working with queries and sets.

The Set clause is optional. If no Set clause is included with the Expand Set command, the output listing of related objects is displayed.

Dump and Output Clauses

You can specify a general output format for listing the expanded information to a file and for output. This is done with the Dump clause:

```
[dump "SEPARATOR_STR"] [output FILENAME]
```

`SEPARATOR_STR` is a character or character string that should appear between the field values. It can be a tab, comma, semicolon, carriage return, etc. If you do not specify a separator string value, a space is used.

`FILENAME` identifies a file where the print output is to be stored.

The Dump clause specifies that you do not want to print the leading field name (a space) and that you want to separate the field names with the separator string you provide.

Separator strings can make the output more readable. If many of the business object have similar field values, using tabs as separators will make the values appear in columns.

The Output clause prints the expanded information to a file that you specify (FILENAME).

Terse Clause

You can specify the terse clause so that object IDs are returned instead of type, name, and revision. This is done with the Terse clause. For example, the following command returns a list of object IDs for objects connected to the specified Part:

```
expand set Part "35735" A terse;
```

Limit Clause

Since you may be accessing very large databases, you should consider *limiting* the number of objects to display. Use the Limit clause to define a value for the maximum number of objects to include in the expansion. For example, to limit the expansion to 25 objects, you could use a command similar to the following:

```
expand set Part "35735" A limit 25;
```

Print Set

You can view the definition of a set using the Print Set command. This command enables you to view all the clauses used to define the set name:

```
print set NAME [user USER_NAME] [SELECT] [start START_RANGE] [end
END_RANGE] [dump [SEPARATOR_STR]] [recordseparator
SEPARATOR_STR] [tcl] [output FILENAME];
```

NAME is the name of the set you want to view. You can include the User clause if you are a business administrator with person access, or if you have visibility to another user's set.

SELECT specifies the fields you want to print.

SEPARATOR_STR specifies the character or character string that will be used to separate field values from a single business object.

FILENAME outputs the results of the command to an external file rather than displaying the information on your output terminal.

When you enter this command, MQL displays the business objects (if any) that make up the set. For example, to see the definition for the set named "Seat Components," you would enter:

```
print set "Seat Components";
```

Depending on your system setup, names may be case sensitive.

If the set name is not found, an error message will result. If that occurs, use the List Set command to check for the presence and spelling of the set name.

Each clause is described in the sections that follow.

Select Clause

This clause enables you to obtain more information than just the type, name or revision of the object.

To first examine the general list of field names, use this command:

```
print businessobject selectable;
```

The selectables for sets are the same as for business objects. Refer to *Select Clause* for this list.

The Selectable clause is similar to using the ellipsis button in the graphical applications—it provides a list from which to choose.

The Select clause enables you to list all the field names whose values you want to print.

```
select [+] FIELD_NAME [ [FIELD_NAME] ... ]
```

When this clause is included in the Print Set command, the values of these fields are printed for each business object contained within the set. For example, assume you have four objects within a set named Components. You can see the names and descriptions of each object with the following command:

```
print set Components select name description;
```

Pagination

Whether a set is sorted or not, it can always be paged through. In MQL, you can print a subset of a set using:

```
print set SET_NAME [start START_RANGE] [end END_RANGE];
```

The parameter START_RANGE is inclusive and zero-based. The parameter END_RANGE is exclusive and zero-based. To scroll through an entire set, you would use commands that resemble:

```
print set SET_NAME start 0 end 100;
print set SET_NAME start 100 end 200;
print set SET_NAME start 200 end 300;
```

Methods for performing these operations are also available in the Studio Customization Toolkit (Set.subset, Set.subsetSelect). Refer to the Studio Customization Toolkit Reference Guide (JavaDocs) for details.

Expressions on Sets

You can evaluate expressions against a set of business objects as described in *Formulating Expressions for Collections*.

Sort Set

By default, Live Collaboration presents business objects in sets sorted alphabetically by Type, then Name, then revision. You can disable sorting or specify other “Basic” properties by which objects in sets should be sorted, by using the MX_SET_SORT_KEY environment variable, which can set any number of basic select items (type, name, revision, owner, locker, originated, modified, current, and policy) by which objects in sets will be presented. Refer to the *Installation Guide* for details.

In addition to this sorting, you can explicitly sort a set by any list of select values in MQL using the following syntax:

```
sort set SET_NAME [into set SET_NAME] [select VALUE1 VALUE2...];
```

As with the sorting environment variable, you can specify ascending or descending order by prefixing the select with + or - (ascending (+) is the default.) For instance:

```
sort set x select +name -owner;
```

This command will perform a sort on both `name` and `owner` fields. Names will be sorted in ascending order, then owners will be sorted in descending order.

Delete Set

If you are a Business Administrator with person access, you can delete sets in any person's workspace (likewise for groups and roles). Other users can delete only their own sets.

You must be a business administrator with group or role access to delete a set owned by a group or role.

If you decide that a set is no longer desired, you can delete it using the Delete Set command:

```
delete set NAME [user USER_NAME];
```

`NAME` is the name of the set to be deleted. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.

Searches the list of existing sets. If the name is found, that set is deleted. If the name is not found, an error message will result.

For example, assume you have a set named "Auto Repairs" that you no longer need. To delete this set from your area, you would enter the following command:

```
delete set "Auto Repairs";
```

When a set is deleted, there is no effect on the business objects. Grouping business objects together in a set is not the same as connecting them. While connecting business objects makes a global *link* between the objects, sets provide only a local linkage that has no value outside the local user's context.

set system Command

Description

The Set System command allows you to control system-wide settings, and to enable or disable FCS compressed synchronization. Password requirements can be set for the entire system, as described in *System-Wide Password Settings*. Triggers may be turned on or off with the Triggers Off command. For conceptual information on this command, see *Maintaining the System* in Chapter 9. For more information on compressed synchronization, see *FCS Compressed Synchronization* in Chapter 2.

User Level

System Administrator

Syntax

System Administrators can control certain other settings that affect the system as a whole using the Set System command:

```
set system SYSTEM_SETTING;
```

Where SYSTEM_SETTING is:

```
casesensitive |on |;
              |off|
```

```
changelattice update set |on |;
                         |off|
```

```
changevault update set|on |;
                      |off|
```

```
constraint |index [indexspace SPACE]|;
           |normal|
           |none  |
```

```
dbservertimezonefromdb |on |;
                       |off|
```

```
decimal |.|;
        |,|
```

```
emptyname |on |;
          |off|
```

```
fcsextensions  [FILE_EXTENSION{,FILE_EXTENSION}];
```

```
fcssettings bucketsync |on |;
                       |off|
```

```
fcssettings modbusoncheckouthistory |on |;
                                    |off|
```

```
fcssettings syncmaxfilenum |NUMERIC_VALUE|;
```

```
fcssettings syncmaxsize |NUMERIC_VALUE|;
```

```
fcssettings zipsync |on |;
                    |off|
```

```
globaluniqueTNR |on |;
                |off|
```

```
history |on |;
        |off|
```

```
indexspace DBINDEXSPACE;
```

```
persistentforeignids |on |;
                     |off|
```

```
priviledgedbusinessadmin |on |;
                         |off|
```

```
searchindex file FILENAME;
```

```
tablespace SPACE;
```

```
tidy |on |;
     |off|
```

```
truncatehistory |on |;
                |off|
```

These settings are descibed in detail in *Maintenance* in Chapter 9.

The Set System FCSSettings ZipSync command is used to enable or disable FCS compressed synchronization.

```
set system fcssettings zipsync |on |
                               |off|
```

The Set System FCSExtensions command is used to configure file extensions indicating compressed files to FCS.

```
set system fcsextensions  [FILE_EXTENSION{,FILE_EXTENSION}]
```

For more information, see *FCS Compressed Synchronization* in Chapter 2.

The Set System FcsSetting BucketSync command is used to activate or deactivate FCS bucket replication. The Set System FcsSetting SyncMaxFileNum command sets the bucket replication parameter `fcssettings.syncmaxfilenum`. The Set System FcsSetting SyncMaxSize command sets the bucket replication parameter `fcssettings.syncmaxsize`. For more information, see *FCS Bucket Replication* in Chapter 2.

The Set System GlobalUniqueTNR command checks the current state of the TNR uniqueness to see how a system was upgraded.

```
print system globaluniqueTNR
```

The above command shows "On" if TNR uniqueness is being enforced across all vaults (the default). It shows "Off" if TNR uniqueness is being enforced on a per-vault basis. If this setting is changed, this creates indices on the mvTNR table, which potentially is a very expensive operation. If you must, you can globally change the setting using:

```
set system globaluniqueTNR [on|off]
```

All new databases are created with the `globaluniqueTNR` flag set to On.

The Set System Searchindex command creates schema for the Full-text Search Server with Exalead. This command requires that the Cloudview Server be up and running before it is issued. If the Cloudview Server is not available, this command produces the following error: "Unable to connect to index server. Config file saved, but "set system searchindex" command will need to be run again to use index server."

```
   set system searchindex file FILENAME
```

The Set System TableSpace command is used to configure the system to use a new tablespace and indexspace for any new administration tables and indices as follows:

```
   set system tablespace DBTABLESPACE
```

```
   set system indexspace DBINDEXSPACE
```

When using SQL Server, the casesensitive setting is Off by default. To work with SQL Server in case-sensitive mode, you must configure case-sensitivity by executing the following commands:

```
   set context user creator;
   set system casesensitive on;
```

See also *To configure SQL Server to work in case-sensitive mode* in Chapter 9.

List System

The List System command is used to display all system settings.

```
   list system ;
```

Your output will be similar to:.

```
   History=On
   change vault update sets setting =Off
   DecimalSymbol=.
   TidyFiles=Off
   Foreign constraint setting = Normal
   privilegedBusinessAdmin=On
   empty name allowed=Off
   CaseSensitive=On
   DbServerTimezoneFromDb=Off
```

Print System

The Print command prints the specified setting to the screen:

```
print system |casesensitive           |
             |changevault             |
             |constraint              |
             |dbtimezonefromdb        |
             |decimal                 |
             |emptyname               |
             |fcsextensions           |
             |globaluniqueTNR         |
             |history                 |
             |persistentforeignids    |
             |privilegedbusinessadmin|
             |tidy                    |;
```

For example:

```
print system changevault;
```

Your output will be similar to:

```
updateSets=Off
```

For the command to print the list of file extensions for FCS compressed files:

```
print system fcsextensions;
```

Your output will be similar to:

```
jpg,giff,tiff
```

For more information, see *FCS Compressed Synchronization* in Chapter 2.

site Command

Description

A *site*, which is a set of locations, can be added to a person or group definition to specify location preferences.

For conceptual information on this command, see *Locations and Sites* in Chapter 2.

User Level

System Administrator

Syntax

```
[add|modify|delete]site NAME {CLAUSE};
```

- NAME is the name you assign to the site. Site names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the site.

Add Site

Sites are nothing more than a set of locations. A site can be associated with a person or group object. When associated with a person, the site defines the list of locations preferred by a particular person. When associated with a group, the site defines the list of locations preferred by all members of the group.

In addition, the enovia.ini file for the Studio Modeling Platform or Live Collaboration Server may contain the following setting:

```
MX_SITE_PREFERENCE = SITENAME
```

where SITENAME is the name of the site object.

This setting overrides the setting in the person or group definition for the site preference. It is particularly designed for use in the enovia.ini file for the Live Collaboration Server, where all Web clients should use the site preference of the Live Collaboration Server to ensure optimum performance. Refer to the *Installation Guide* for more information.

A site is defined with the Add Site command:

```
add site NAME [ADD_ITEM {ADD_ITEM}]
```

NAME is the name of the site you are defining. All sites must have a name unique in the database. The name can contain spaces. You should assign a name that has meaning to both you and the user. For more information, see *Administrative Object Names*.

ADD_ITEM is an Add Site clause that provides more information about the site you are creating. The Add Site clauses are:

```
[!|not]hidden
description VALUE
member location NAME
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

Member Clause

Use this clause to add locations to the site definition. The location named must exist in the database and the spelling and case must match exactly. If the location name contains embedded spaces, use quotation marks. For example:

```
add site "US Company" member location "Western Division";
```

History Clause

The history keyword adds a history record marked “custom” to the site that is being added. The STRING argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Modify Site

After a site is defined, you can change the definition with the Modify Site command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify site NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the site to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Site clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Site Clause	Specifies that…
name NAME	The site name is changed to the new name.
description VALUE	The description is changed to the new value specified.
icon FILENAME	The image is changed tp the new image in the field specified.
add location NAME	The named location is added to the site definition.
remove location NAME	The named location is removed from the site definition.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.

Modify Site Clause	Specifies that…
`property NAME [to ADMINTYPE NAME] [value STRING]`	The existing property is modified according to the values specified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	The `history` keyword adds a history record marked “custom” to the site that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

As you can see, each modification clause is related to the clauses and arguments that define the site.

Delete Site

If a site is no longer required, you can delete it. Since deleting a site affects all locations within that site, it is recommended that you make a backup prior to deletion. This protects you if you need access to files from that site at a later date. You may need to involve the DBA in the backup and restore process (if a restore is necessary).

To delete a site, use the Delete Site command:

```
delete site NAME;
```

- `NAME` is the name of the site to be deleted.
- Searches the list of defined sites. If the name is not found, an error message is displayed. If the name is found, the site is deleted.

store Command

Description

A *store* is a storage location for checked-in files. All files checked in and used by Live Collaboration are contained in a file store. These files can contain any information and be associated with any variety of business objects. A file store defines a place where you can find the file.

For conceptual information on this command, see *Stores* in Chapter 2.

User Level

System Administrator

Syntax

```
[add|modify|tidy|inventory|validate|delete|purge|rechecksum]
store NAME {CLAUSE};
```

- NAME is the name you assign to the store. Store names must be unique. This means a store and location cannot have the same name. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the store.

Add Store

There are several parameters that can be associated with a store. Each parameter enables you to provide information about the new store. Some parameters are common to all stores, others depend on the type of store you are creating. While only the Name and Type clauses are required, the other parameters can further define the store, as well as provide useful information about the store.

Before defining a DesignSync store, configure the DesignSync server to:

- Allow BrowseServer access to the system administrator user that will create the store.
- The communication protocol used by the DesignSync server needs to match the protocol in use on the Live Collaboration Server. If you want to use secure protocol (SSL/HTTPS) to communicate between DesignSync and the Live Collaboration Server, you must configure SSL on your DesignSync server and install the digital certificate onto the Live Collaboration Server. Refer to the *Configuration Guide : DesignSync File Access* in the online documentation.

A file store is defined with the Add Store command:

```
add store NAME [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the store you are defining. All stores must have a name unique in the database. The name can contain spaces. Do not include the @ sign in the name, as that is used internally by Live Collaboration. You should assign a name that has meaning to both you and the user. For more information, see *Administrative Object Names*.

- ADD_ITEM provides more information about the store you are creating. The only required clause for store creation is the Type clause. Other clauses can help to further define the store. The Add Store clauses are:

description VALUE
fcs FCS_URL
port PORT_NUMBER
url VALUE
[un]lock
type captured|designsync|external
filename [not]hashed
[!|not]hidden
path PATH_NAME
prefix PREFIX
host HOST_NAME
password PASSWORD
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]
property NAME [to ADMINTYPE NAME] [value STRING]
protocol PROTOCOL_NAME
user USER
deletetrigger DELETE_TRIGGER_IMPLEMENTATION
replicatetrigger REPLICATE_TRIGGER_IMPLEMENTATION
checksum [on|off]
checksumwarnonly [on|off]

The Path clause is required only for captured files. Although all other clauses are optional, the following defaults are assumed:

Clause	Default
filename	hashed
host	localhost
[un]lock	unlocked
path	MATRIXHOME
permission	rw, rw, rw

Each clause and the arguments they use are discussed in the sections that follow.

FCS Clause

For captured and external stores, type in the FCS URL in the **FCS URL** text box. You must define the FCS URL for FCS file replication. The FCS URL is not required for file checkin and checkout operations.

When an FCS URL is not defined, the system uses the URL of the local MCS. Since there are two ways to connect to the Live Collaboration Server, it is important to note that the MCS URL can only be inferred when connecting through the application server, usually at port 8080. The other way to connect is through the server, usually at port 1099, but an MCS URL cannot be inferred when connecting this way because an MCS URL is not defined. This means if you are connecting through the server and need to perform FCS operations that require an FCS URL, you must manually define the FCS URL.

Include the Web application name. The syntax is:

```
fcs http://host:port/WEBAPPNAME
```

For example:

```
fcs http://host:port/ematrix
```

If using Single Sign On (SSO) with FCS, the FCS URL should have the fully qualified domain name for the host. For example:

```
fcs http://HOSTNAME.MatrixOne.net:PORT#/ematrix
```

You can also specify a JPO that will return a URL. For example:

```
fcs ${CLASS:prog} [-method methodName] [ args0 … argN]
```

You can also specify settings in framework.properties or web.xml to control various parts of the FCS system such as: the length of time a ticket or receipt is valid, and how many tickets or receipts the system retains before deleting them. For information about FCS processing, see the *File Collaboration Server Guide*.

DesignSync stores do not require an FCS URL and one should NOT be defined for them.

Type Clause

This clause identifies how the files are stored and accessed. You can specify one of the three store types: captured, designsync, or external.

Once you decide how to store and control your files, you can specify the store type by adding a Type clause to your Add Store command. All file store definitions must have a Type clause to be usable. The following example creates a store to hold captured files:

```
add store Drawings
   description "Storage for electronic drawings"
   host RELIABLE
   path ${MATRIXHOME}/Drawings
   type captured;
```

File Name Hashing

Captured stores generate hashed names for checked in files based on a random number generator and timestamp. If a name collision occurs, it will retry with a new hashname up to 100 tries, then

return an error. Since the files for captured stores are physically stored on disk, the names are hashed to be recognized by Live Collaboration only.

A file copied from a store to a location goes through the same file naming algorithm.

When a file is checked into a captured store (or location) a 2-level directory structure is created below the store/location's directory path. Each subdirectory name is derived from the hashed file name and is 2 characters long. The first 2 characters of the hashed name are the first level subdirectory name; the 3rd and 4th characters are the second level subdirectory name. For example, a captured and hashed store named Docs is defined to use the directory `Docs`. A file is checked in that uses this store and the hashed name generated is `dd24c8c236e6875e.c06`. The file is put into the following directory on the store's host machine:

`Docs`**`/dd/24/`*`dd24`***`c8c236e6875e.c06`

The database includes the subdirectories in the file name. The `format.file` select output includes this information in the `locationfile[ ]` and `capturedfile` sub-select output as shown below:

```
format.file.locationfile[Docs] = dd/24/dd24c8c236e6875e.c06
format.file.capturedfile = dd/24/dd24c8c236e6875e.c06
```

Lock Clause

This clause locks a store from the user for all write activities. Business objects with a policy using a locked store cannot have files checked in (written), but files can be checked out (read) or deleted. This is useful when backups are being made or when a store is full. The default is unlocked.

For example:

```
add store "Vellum CAD Drawings"
   locked
  type captured;
```

A locked store prevents file checkin, but still allows file checkout or deletion. You can prevent files from being deleted or checkedout by adding a trigger on the RemoveFile or Checkout events. Refer to the *Configuration Guide : Triggers* in the online documentation.

Path Clause

This clause is required for captured and DesignSync stores. For captured stores, the path identifies where the file store is to be placed on the host. All files and folders referencing a store will be relative to its path. Paths should not include the @ sign, as that is used internally by Live Collaboration. A store should NOT point to the same directory as a location since this may cause unnecessary file copying.

Stores should use actual host names and paths, not mounted directories. For captured stores, the path must be exported to all users that need access to the files.

A captured store will create a directory for the captured files. When a file is checked into a captured store (or location), a 2-level directory structure is created below the store/location's directory path. Each subdirectory name is derived from the hashed file name and is 2 characters long.

The following Add Store command creates a file store located in MATRIXHOME\Training on the Reliable host:

```
add store Training
   description "Storage for training information"
   host Reliable
   path ${MATRIXHOME}/Training
   type captured;
```

For DesignSync stores, the path should indicate where files, folders, and modules should be looked for and put in the DesignSync system (or “repository”, as it is sometimes called), and correspond to part of the string used in the “setvault” DesignSync command.

A path is not required when you create a DesignSync store and can instead be defined later when a file, folder or module is checked into the store. DesignSync stores should only be created without a path if you plan to use the store for files and folders. If you plan to use the store for modules, you should define a path of “/Modules/*” since modules can ONLY be checked into stores with a path of “/Modules.” Files and Folders can be check into stores with any path except “/Modules.

For example, to set up a store for a DesignSync project that uses this setvault command:

```
dss setvault sync://src.matrixone.net/Projects/Documents/Release
```

the path would be:

```
Projects/Documents/Release
```

Prefix Clause

A prefix can be added to the path defined for a captured store or location to enhance file management operations. A prefix adds a subdirectory to the path. For example, when you add the prefix project1 to a store with a path of c:\enovia\stores\store1\, the c:\enovia\stores\store1\project1 directory is created.

```
add store store1 prefix project1;
```

Prefix names are limited to 64 characters and can only include upper and lower case letters and numbers. Adding “\” and “/” to the prefix is not allowed. A store or location can only have one prefix defined at a time.

A prefix separates the path into the main path and the path with the prefix. For example, if a folder has a file limit and that limit is reached, you can modify the store’s prefix to redirect file checkins to the new directory. On UNIX platforms, if the current file storage is full, you can modify the prefix to use a new mounted drive. Modifying the prefix will affect where files are stored in future checkin operations.

```
modify store NAME prefix PREFIX;
```

You can only add and modify prefixes using MQL.

Import and export operations are not aware of prefixes. When you import or export from the current version or versions with the prefix option, you need to manually migrate the prefixes using an MQL or Tcl script.When you import from older versions without the prefix option, the prefix is defaulted to an empty string.

Host Clause

This clause identifies the host system to contain the file store being defined. If a host is not specified, the current host is assumed and assigned. If the store is to be physically located on a PC

and accessed through a network drive, the store host name must be set to `localhost`. Host names should not include the @ sign, as that is used internally by Live Collaboration.

For DesignSync stores, the Host should be set to the name of a DesignSync server. For example:

```
src.matrixone.net
```

File stores can be created and can exist across networks. Depending on who uses a file store, you might install it on a system local to its users. That speeds up access and avoids placing additional loads on network communications.

File stores that are frequently used by all nodes within a network should be contained on a centrally located node. If a file store is used extensively by one system, you may want to place the store on that system to improve access time and communications requirements.

For example, the following command defines a store (Video) that resides on the system called `Reliable` in a directory defined by the environment variable `MATRIXHOME`:

```
add store Video
   description "Storage for video data"
   host Reliable
   path ${MATRIXHOME}
   type captured;
```

If files will need to be checked in and out via the Web, you should install a file Live Collaboration Server on the file store host machine, as described in the *File Collaboration Server Guide*.

Protocol and Port Clauses

When creating a captured or designsync store, you can include the parameters `protocol` and `port`. The protocol defined for a captured store determines the file access given by the controlling FCS.

```
protocol PROTOCOL_NAME [port PORT_NUMBER]
```

- `PROTOCOL_NAME` is file for captured stores. DesignSync stores use a protocol of either http or https.
- Each protocol has a default port that is used if not specified in the store definition. Include the port subclause to specify a port other than the default.

This protocol is not related to the protocol used for file transfers between FCSs during synchronization operations. The HTTP/S protocol is always used for these operations.

For example:

```
add store MyStore type captured host Reliable
path ${MATRIXHOME}/MyStore protocol file user Engineering
password secret;
```

Permission Clause

File permissions are no longer supported for captured stores, and have no effect on files and folders that are managed by the FCS.

Warning: Do not directly modify or change folders and files managed by FCS in any way (e.g., rights, size, etc.).

User and Password Clauses

For DesignSync stores, the User and Password clauses are used to specify a valid DesignSync user and password; therefore, you must adhere to DesignSync naming conventions.

The initial connection to DesignSync is made by the user defined here, and access privileges for non-file operations in DesignSync are based on this user account. However, file (and folder) operations are performed by the current context user, not this user. Therefore, the DesignSync store user specified must be set up with the "SwitchUser" privilege in the AccessControl file in DesignSync, and this must be done before creating the store. Refer to *Access controls* in Chapter 2 for other access controls that must be configured.

For file operations, the context user is subject to access controls in effect for both Live Collaboration and DesignSync. For example, if a checkin is attempted via the Semiconductor Accelerator, and the user does not have both Live Collaboration and DesignSync checkin access, the operation will fail. If the checkin is successful, in DesignSync the author is listed as the user that checked in the file.

Since a Live Collaboration username may contain characters not permitted in a DesignSync username, such as spaces and Japanese characters encoded in UTF-8, there is an encoding/decoding that occurs between the 2 systems. For example, a space in a Live Collaboration username is represented in DesignSync with a "+".

Vault Clause

When adding an external store you must include the Vault clause to specify the Web Service Adaplet (external) vault to be used in conjunction with the external store. The Vault clause only has meaning if both it and the store are marked as external.

Refer to the *Web Services Adaplet Programming Guide* for details.

File or Param Clause

When adding an external store you can use either the File or Params clause to include optional parameters for use with the external vault. For example:

```
add store wsStore type external vault wsVault file
paramsFile.xml;
```

Refer to the *Web Services Adaplet Programming Guide* for details.

DeleteTrigger Clause

The DeleteTrigger clause allows you to implement alternate behavior for when files are being deleted from a store or location. To add a no-delete policy on a store, you can use the out of the box implementation in com.matrixone.fcs.backend.NoFileDelete. For example:

```
add store MyStore type captured deletetrigger
com.matrixone.fcs.backend.NoFileDelete;
```

You can also write your own Java class that implements the com.matrixone.fcs.mcs.DeleteTrigger interface to insert any customized behavior, such as altering the file delete list. It can be either a JPO or a regular Java class that can be found in the classpath.

The DeleteTrigger that is set for a store automatically applies to all locations that are associated with that store, unless it is overridden at a particular location. The DeleteTrigger clause is only valid on captured stores (and locations).

ReplicationTrigger Clause

The ReplicationTrigger clause allows you to implement alternate behavior when files within the store are replicated based on file replication rules set in the governing object's policy. For example, you could write your own Java class that implements the com.matrixone.fcs.mcs.ReplicationTrigger interface to insert any customized behavior such as blocking the replication or getting a file from a different source. It can be either a JPO or a regular Java class that can be found in the classpath.

The ReplicationTrigger clause is only valid on captured stores, and not locations.

Checksum Clause

The Checksum clause allows you to activate or deactivate the usage of checksums, which FCS uses to perform a file integrity check at checkout to determine if the file to be checked out has been modified (corrupted) since it was last checked in. When checksums are activated, FCS will compute and send a checksum to the MCS when a file is checked in. Then when a file is checked out, FCS computes a checksum of the file to be checked out and compares it with the checksum saved on the MCS for the previously checked in version of the file. If the checksums differ, FCS returns an error and the file checkout fails.

The default is off.

Using checksums to check file integrity has a performance cost. The checksum computation is based on streamed data, which increases FCS checkin and FCS checkout time slightly, but the increase is a reasonable tradeoff for the increased data integrity.

ChecksumWarnOnly Clause

The ChecksumWarnOnly clause allows you to set the behavior of FCS when it detects a corrupted file by comparing checksums during checkout. By default, FCS returns an error and the checkout process fails. When checksumwarnonly is turned on, the invalid checksum is logged, but no exception is thrown, so the checkout succeeds, although the user is warned.

The default is off.

This checksumwarnonly parameter applies only when the checksum parameter is turned on.

Modify Store

After a file store is defined, you can change the definition with the Modify Store command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify store NAME [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the store you want to modify.
- `MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Store clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Store Clause	Specifies that…
`add location LOCATION_NAME`	The named location is added to the store.
`remove location LOCATION_NAME`	The named location is removed from the store.
`description VALUE`	The current description, if any, is changed to the value entered.
icon FILENAME	The image is changed tp the new image in the field specified.
`name NAME`	The current file store name is changed to that of the new name entered.

`filename [not]hashed`	The file name is either encoded (`hashed`) or not (`nothashed`).
	Modifying a captured file store from a `hashed` to a `nothashed` file name is not allowed. Attempting to do so will result in this error:
	`MQL<n>mod store STORE filename nothashed;`
	`Error: #1900068: mod store failed`
	`Error: #1900618: Store modification from hashed to nothashed filename is not allowed`

`filter FilterClass`	The current filter on input and output data stream is changed.
`params paramsString | file paramsFile`	The current parameter file or string (xml file) is changed to the new one entered.
`path PATH_NAME`	The name of path to the file store is changed to the value entered. Note: If you change the path of a store, the files are not accessible. When you change the path, the system assumes you are also going to move the files.
`prefix PREFIX`	The prefix is changed to the value specified.
`protocol PROTOCOL_NAME`	The protocol is changed to the value specified.
`port PORT_NUMBER`	The port is changed to the number specified.
`host HOST_NAME`	The host containing the file store is changed to the host named.
`[un]lock`	The store is either locked from the user for all write activities (lock) or available (unlock).
`permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]`	*File permissions are no longer supported for captured stores, and have no effect on files and folders that are managed by the FCS.* **Warning:** Do not directly modify or change folders and files that are managed by the FCS in any way (e.g., rights, size, etc.)
`user USER`	The current DesignSync username is changed to the new name entered.

Modify Store Clause	Specifies that…
`password PASSWORD`	The current DesignSync password is changed to the new one entered.
`url VALUE`	The current URL is changed to the new one entered.
`fcs FCS_URL`	The current FCS URL is changed to the new one entered.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`deletetrigger DELETE_TRIGGER_IMPLEMENTATION`	Captured File delete behavior is modified. DELETE_TRIGGER_IMPLEMENTATION implements the com.matrixone.fcs.mcs.DeleteTrigger interface. It can be either a JPO or a regular Java class that can be found in the classpath. *The deletetrigger clause is only valid for captured stores and locations.* See also *DeleteTrigger Clause*.
`replicatetrigger REPLICATE_TRIGGER_IMPLEMENTATION`	Captured File replication behavior is modified. REPLICATE_TRIGGER_IMPLEMENTATION implements the com.matrixone.fcs.mcs.ReplicationTrigger interface and provides alternate behavior for the replication transaction. It can be either a JPO or a regular Java class that can be found in the classpath. *The replicatetrigger clause is only valid for captured stores.*
`checksum [on|off]`	The checksum computation at checkin and the checksum integrity check at checkout is activated (on) or deactivated (off). The default is off.
`checksumwarnonly [on|off]`	The "exception thrown upon invalid checksum detected at checkout" behavior is modified. When checksumwarnonly is turned off, and an invalid checksum is detected at checkout, the system throws an exception and the checkout fails. When checksumwarnonly is turned on, the invalid checksum is logged and the user is warned, but no exception is thrown, so the checkout succeeds. The default is off. Applies only when the checksum parameter is turned on.

As you can see, each modification clause is related to the clauses and arguments that define the file store. The only modifications that you cannot make to a file store definition is a change in the store type or the tablespaces used. To change the store type, you must create a new file store. Once a store type is assigned, it remains in place as long as the file store exists. When you modify a store,

you first name the store and then list the modifications. For example, the following command changes the name of the Drawings store and the path:

```
modify store Drawings
   name "CAD Drawings"
   path ${MATRIXHOME}/CADDrawings;
```

When this command is executed, the name of the store changes to “CAD Drawings” and the path is changed to `${MATRIXHOME}/CADDrawings`.

Multiple Directories in Hashed Captured Stores

When the MQL `upgrade` command is run for version 10.5 or higher, all existing captured and hashed stores and locations are updated to use multiple directories. Files checked into any store from that point on use the multiple-directory algorithm for file storage. Files existing in a pre-10.5 database remain in the 8.3 format, in a single-directory file system unless they are re-checked into a captured and hashed store that has the `multipledirectories` setting turned on. If you want to re-distribute files based on the multiple-directory algorithm, you must create MQL scripts to check out and then check in all files. You can then run these scripts as time and computer resources allow.

You can disable the setting if required, with one of the following commands:

```
mod store STORE_NAME [!|not]multipledirectories];
mod location LOCATION_NAME [!|not]multipledirectories];
```

Hashed stores/locations have only one possible setting:

- `multipledirectories` generates 16.3 format hashed filenames in a 2-level subdirectory structure.

The `notmultipledirectories` setting is not supported.

Print Store

The Print Store command prints the store definition to the screen allowing you to view it. When a Print command is entered, MQL displays the various clauses that make up the definition.

```
print store STORE select FIELD;
```

- `STORE` is the name of the specific store instance.
- `FIELD` lets you specify data to present about the store being printed.

Tidy Store

The Tidy Store command can be used in a captured store situation to clean up obsolete copies of files that might exist in a replicated environment.

```
tidy store NAME;
```

`NAME` is the name of the file store you want to tidy.

Inventory Store

You can list the file contents of a store by using the `Inventory Store` command. Additional clauses provide the ability to specify a subset of locations to inventory. Memory usage remains at a low fixed amount regardless of the size of the store. The syntax is:

```
inventory store NAME [location LOCATION_TARGET{,LOCATION_TARGET}] [store]
[fcsdbchecksum];
```

`NAME` is the name of the file store.

`LOCATION_TARGET` is one or more location names that are associated with the specified store. Locations are separated by a comma but no space.

It is highly recommended that inventories of all stores and locations are performed nightly as part of the backup procedure.

If files names are hashed, a list is presented containing a mapping of the hashed names to the original names.

If MQL is running in verbose mode, the output includes:

- The full path of file origination
- The date and time of creation (if the file name is hashed)
- The format of the file and the owning business object.

You can also include the keyword store after the storename to list files stored at the store itself. For example, the command:

```
inventory store Assemblies store location Dallas,London;
```

might result in the following output:

```
store Assemblies file original
C:\Designs\widget\SHEET1.PRT captured /matrix/prod/stores/Assemblies/
31b49a45.784 on Tue Apr 4, 1994 4:19:17 PM format Cadra of business
object Assembly 12345 A
```

The creation date and time for hashed file names correlates to when the database received control of the file and hashed the name. Keep in mind that there are a number of ways a file can get into Live Collaboration: through checkin, import, cloning, or revisioning.

You can also include the keyword `fcsdbchecksum` to list the current checksums for the files in the store.

If the output indicates that files have been orphaned, contact Dassault Systemes Customer Service for help in diagnosing the error.

Validate Store

The Validate Store command can be used to identify files in a captured store (or location) that are not associated with any business object in the database. The command requires an input file that lists all files relative to the store's (or locations's) directory. For multiple directory stores/locations (the default for captured and hashed stores), the input file can be created from the captured store directory using the following command from the store directory on Unix:

```
find -type f -print
```

For example, your file might look like:

```
./00/b3/00b30d7774254606.b27
```

```
./0036a01d.6ff
./01/11/01119e1b826a714e.0fb
```

From a DOS prompt on Windows, the input file can be created from the output of:

```
dir /s /b /a-d
```

And to send the output to a file:

```
dir /s /b /a-d  > c:\temp\files.txt
```

However, on Windows you will need to remove the drive and store directory from each file listed. For example, for a store named “distribs” you might have output similar to:

```
D:\distribs\00\b3\00b30d7774254606.b27
D:\distribs\0036a01d.6ff
D:\distribs\01\11\01119e1b826a714e.0fb
```

You would need to search on “D:\distribs\” and delete.

Once the input file is created, System Administrators can use the following command:

```
validate store NAME file FILENAME;
```

Where:

`NAME` is the name of a captured store.

`FILENAME` is an input file that lists all files in a captured store or location directory.

The `validate store` command creates a file called FILENAME.out (the same name as was used for the input filename, except with a file extension of .out), which lists files that are not associated with business objects. You can then cleanup the store or location directory by deleting these files.

Monitoring Disk Space

As a System Administrator, you should monitor the disk space available to your stores carefully. If disk space is running low, you could simply change the path or host of the store, but then you would have to move all checked-in files to the new directory, or users would receive errors when trying to access the files. A better solution is to change the store that the policies use.

To replace an existing store

1. Use the following MQL command:

   ```
   modify store STORE_NAME lock;
   ```

2. Create a new store with a host and path that has ample disk space.
3. Determine which policies use the store that needs to be replaced.

   a) In MQL, run the following command:

   ```
   list policy * select name store dump |;
   ```

   The results will look something like:

   ```
   CMM|Object
   Model|Product
   Critical Report|Submission
   Incident|Submission2
   Sequencer|Object
   Media|Product
   Task|Object
   ```

```
Marketing Documents|Document
Kit|Product
...
```

b) Copy the list and paste it into a text editor.

c) Create a table and sort by the policy column.

4. Modify the policies that use the filled up store to use the newly defined store.
5. Unlock the original store.

New files checked into objects governed by these policies will be put in the new store. Files that were checked in previously can still be checked out and opened for view, but if new files are checked in, the new store is used. Refer to *Implications of Changing Stores* in Chapter 2 for more information.

Delete Store

If a file store is no longer required and is not in use, you can delete it with the Delete Store command:

```
delete store NAME;
```

`NAME` is the name of the file store to be deleted.

Searches the list of existing file stores. If the name is found, and business objects have files in the store, the store deletion is stopped with the following message:

```
Error: #1900068: delete store failed
Error: #1400005: Object has references
```

For example, to delete the empty Income Taxes store, enter the following MQL command:

```
delete store "Income Taxes";
```

After this command is processed, the store is deleted and you receive an MQL prompt for another command.

The delete will fail if business objects have files in the store.

Purge Store

The Purge Store command is used to purge files from store locations. The syntax is:

```
purge store STORE_NAME [before DATE] [beforeAtime DATE]
[commit N][continue] [location
LOCATION_TARGET{,LOCATION_TARGET}];
```

If the Purge Store command is used with just a store name, it does a purge on all locations, leaving files only in the default place defined in the store object. For example, if you wanted to purge all files in the store called Maps, you would enter the following MQL command:

```
purge store Maps;
```

Optional clauses provide more control over which files are deleted. In addition, you can trace the purge event by using the `trace type store` command that logs the store's events. Refer to *trace Command* for more information. Each `purge store` clause is described in the sections that follow.

Before Clause

For more flexibility, you can add a date restriction to the Purge Store command to remove only files that were last checked in prior to the date specified in the command. For example, if you want to purge only files last checked in prior to February 28, 2014 at 9:57:32 AM EST in the store called Maps, enter the following MQL command:

```
purge store Maps before "Thu Feb 28, 2014 9:57:32 AM EST";
```

BeforeAtime Clause

Include the `BeforeAtime` clause to remove files that were accessed before the date specified in the command. For example:

```
purge store Maps beforeAtime "Thu Feb 28, 2014 9:57:32 AM
EST";
```

If you use a date without specifying a time, the default time is set to 12:00:00 AM.

Continue Clause

Include the keyword `continue` if you don't want the command to stop if an error occurs. If the log file is enabled, failures are listed in the file. Refer to *trace Command* for more information. For example:

```
purge store "Engineering-Dallas" continue;
```

If an error occurs when using the `Continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next business object. For this reason, when using the `Continue` clause you should also include the `Commit` clause, described below.

Commit N Clause

Include the `Commit N` Clause when purging large stores. The number N that follows specifies that the command should commit the database transaction after this many objects have been purged. The default is 10. For example:

```
purge store "Engineering-Dallas" continue commit 20;
```

Location Clause

To specify a subset of locations that you want purged, include the `Location` clause. For example:

```
purge store "Engineering-Dallas" continue commit 20 location
London,Paris,Milan;
```

When listing locations, delimit with a comma but no space.

Rechecksum Store

The Rechecksum Store command is used to create a checksum for each file in a store. Typically you would use this on migrated files. The syntax is:

```
rechecksum store STORE_NAME [continue] [commit N] ;
```

The checksum for a file is calculated only if it has not been previously calculated. If a checksum is calculated, it will be based on an up-to-date copy of the file at an arbitrary location. The checksum value will be propagated to all non-obsolete copies of the file.

You can trace the rechecksum event by using the `trace type store` command that logs the store's events. Refer to *trace Command* for more information.

This method of creating checksums for files that were migrated and not individually checked in (and thus did not have a checksum created at checkin) assumes the files are not already corrupted. You might find it helpful to run the `validate` command first. Refer to *Validate Store* for more information.

Note that the `rechecksum store` command applies only when `checksum` has been turned on at the store level. Refer to *Add Store* for more information.

Using the `rechecksum` command during migration may consume a lot of time. For large stores and locations, it may be impractical to do a `rechecksum store` command. In this case, you can either use `rechecksum buslist` to migrate only active objects, or do not use `rechecksum` at all. If you do not use `rechecksum` at all, only files that are newly checked in (with the checksum option on) will have their checksums verified on checkout. See *Rechecksum Business Object or Business Object List*.

Each `rechecksum store` clause is described in the sections that follow.

Continue Clause

Include the keyword `continue` if you don't want the command to stop if an error occurs. If the log file is enabled, failures are listed in the file. Refer to *trace Command* for more information. For example:

```
rechecksum store "Engineering-Dallas" continue;
```

If an error occurs when using the `Continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next business object. For this reason, when using the `Continue` clause you should also include the `Commit` clause, described below.

Commit N Clause

Include the `Commit N` Clause when creating checksums for large stores. The number N that follows specifies that the command should commit the database transaction after this many objects have had a checksum created for them. The default is 10. For example:

```
rechecksum store "Engineering-Dallas" continue commit 20;
```

table Command

Description

Tables can be defined to display multiple business objects and related information. Each row of the table represents one business object. Expressions are used to define the columns of data that are presented about the business objects in each row. When you define a table, you determine the number and contents of your table columns.

In 3DEXPERIENCE Platform, there are two kinds of tables: user table or a system table.

For conceptual information on this command, see *Tables* in Chapter 8.

User Level

System tables are created by Business Administrators that have Table administrative access, and are displayed when called within a custom application.

Syntax

```
[add|copy|modify|evaluate|delete|print]table NAME {CLAUSE};
```

- NAME is the name you assign to the table. Table names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the table.

Add Table

To define a table from within MQL use the Add Table command:

```
add table NAME user USER_NAME [ADD_ITEM {ADD_ITEM}];
```

Or

```
add table NAME system [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the table you are defining. Table names cannot include asterisks.
- USER_NAME can be included with the user keyword if you are a business administrator with person access defining a table for another user. I
- system refers to a table that is available for system-wide use, and not associated with the session context.

You must include either the user or system keyword.

- ADD_ITEM provides additional information about the table. The Add Table clauses are:

```
[!|in|notin|not]active
```

```
column [label STRING_VALUE] COLUMN_TYPE_DEF
[COLUMN_DEF_ITEM]
```

description VALUE
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING] units | [picas] | points | inches |
visible USERNAME{,USERNAME}
history STRING

- The column clause must be used for each required column of the table, to specify the COLUMN_TYPE_DEF. All other clauses and subclauses are optional.
- The history clause applies only to system tables; it does not apply to user tables.

Each clause and the arguments they use are discussed in the sections that follow.

Units Clause

This clause specifies the units of page measurement. There are three possible values: picas, points, or inches.

```
 units picas
Or
units points
Or
units inches
```

Without a unit of measurement, Live Collaboration cannot interpret the values of any given header, footer, margin, or field size. Because picas are the default unit of measurement, Live Collaboration automatically assumes a picas value if you do not use a Units clause.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character “I” may not occupy the same amount of space as the characters “E” or “O.” To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit across a table page? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

Column Clause

This clause defines each column in the table. When a column is added to a system table, it will be propagated to all tables derived from that table.

The Column clause is made up of several subclauses. Each Column clause must have a Businessobject, Relationship, or Set subclause, unless the table column is usable only on the Web (see note below). All other subclauses are optional:

```
column [label STRING_VALUE] COLUMN_TYPE_DEF
[COLUMN_DEF_ITEM]
```

- If the label keyword is not used, the column heading is the expression.
- `STRING_VALUE` is the text that is to appear in the heading of the column.
- `COLUMN_TYPE_DEF` can be any of the following:

```
businessobject QUERY_WHERE_EXPRESSION
```

```
relationship QUERY_WHERE_EXPRESSION
```

```
set QUERY_WHERE_EXPRESSION
```

- `QUERY_WHERE_EXPRESSION` is an expression that is evaluated on each object in the table with the results placed in the Table cell. This expression is constructed according to the syntax described in *Queries* in Chapter 6.
- The `QUERY_WHERE_EXPRESSION` is used with `businessobject, relationship,` or `set` keyword. When applied to relationships, the information presented in the column will only be available from an *indented* table, where relationships are apparent. Relationship information will be blank in table cells that represent business objects and vice verse. A *flat* or regular table shows only the properties of the business objects it contains as expressed in the column definition. While the same table definition can be used as both a flat and indented table, generally the usage determines how the table is defined.

Omit the `businessobject`, `relationship`, *or* `set` *keyword for a table column that is to be used only on the Web. Other column settings will indicate whether it is a check box, image, etc..*

- `COLUMN_DEF_ITEM` is a Column subclause that provides additional information about the value to be printed. These subclauses define information such as the size and scale of the columns, the order in which the columns should be placed on the page, and the links used within the columns.

Column Definition	Meaning
`name COLUMN_NAME`	The name to assign to the column, for example: `column name colname1`
`size WIDTH HEIGHT`	The default size of a column is determined by its contents, and the font size used for dialogs. The defaults are recommended; however, you can set column sizes using the other column subclauses. The width and height can be explicitly set using the Size clause with width and height values respectively:
`minsize MIN_WIDTH MIN_HEIGHT`	The minimum width and/or height of the column, for example: `column units picas minsize 20 12`
`scale PERCENTAGE_VALUE`	Percentage of the entire table to be used for this column. For example, use `scale 25` for a 4-column table of equal column width.
`href HREF_VALUE`	The link data to the JSP. The Href link is evaluated to bring up another page. Many table columns will not have an Href value at all. The Href string generally includes a fully qualified JSP filename and parameters, which can contain embedded macros and expressions for mapping to database schema. Refer to *Using Macros and Expressions in Configurable Components* in Chapter 8 for more details.
`alt _ALT_VALUE`	Alternate text displayed until any image associated with the column is displayed and also as “mouse over text.”
`range RANGE_HELP_HREF_VALUE`	For use in Web tables only to specify the JSP that gets a range of values and populates the column with the selected value.
`update UPDATE_URL_VALUE`	The URL address for updating the column.
`program SORT_PROGRAM_NAME`	Defines a program for sorting the table columns.
`order NUMBER`	The order of the column within the table. For example, `order 3` would place the column as the third in the table. Please note that if the column is modified for any reason, this column may not remain third in the table even if the order number is still 3. See *Understanding how column order is processed* for details.
`sorttype alpha|numeric|other|none`	Determines how the column is sorted.
`add user [USER_NAME|all]`	For use in Web forms only to specify who will be allowed access to the column.
`setting NAME VALUE`	For use in Web forms only. Settings are general name/value pairs that can be added to a column as necessary. They can be used by JSP code, but not by hrefs on the Link tab. Also refer the *Business Modeler Guide : Using Macros and Expressions in Dynamic UI Components* for more details.
`remove user [USER_NAME|all]`	For use in Web forms only to specify who will not be allowed access to the column.
`remove setting NAME VALUE`	For use in Web forms only to remove settings.

Column Definition	Meaning
`autoheight [true|false]`	Autoheight and autowidth are both `false` by default, even if height/width have not been specified. If you set the size with the size clause, you can change back to the default size by setting the autoheight and autowidth clauses to `true` .
`autowidth [true|false]`	
`edit [true|false]`	Determines whether users can edit cells in the column.
`hidden [true|false]`	Determines whether the column is hidden.

For example, each of the following are valid column clauses:

```
column businessobject attribute["Target Cost"] - attribute["Actual Cost"]
```

```
column relationship name
```

```
column relationship attribute[Quantity]
```

Notice that selectable items can be operated on as numerical expressions.

Understanding how column order is processed

The order in which columns appear in a table do one always follow the order number given to the column. It is important to note that column modifications are processed in the order that they are specified in the modify table command. This means if a column is added with an order number, K, that is *greater* than the total number (N) of columns in the table, K is reset to N+1. If a column is added with an order number, K, that is *less* than N, all of the columns with order numbers greater than K will be bumped up by 1. The order numbers of columns in a table are always consecutive and can never have gaps.

For example, if we start with an 11 column table and:

- A column A is added with order number 14 which becomes column 12.
- A column B is added with order 21 which becomes column 13.
- A column C is added with order 15 which becomes column 14.
- A column D is added with order 13 which becomes column 13, bumping B to 14 and C to 15.

You might expect the columns to be DACB but in fact they are ADBC. Table columns must be modified in order.

History Clause

The `history` keyword adds a history record marked “custom” to the system table that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

The history clause applies only to system tables; it does not apply to user tables.

Example Add Table command

The following is an example of a complete Add Table command:

```
add table TestTable system
column name colname1
sorttype none
setting "Column Type" Image
setting "Image Size" format_mxThumbnailImage
column name colname2
label emxProduct.Table.Name
businessobject name
href "${COMMON_DIR}/emxTree.jsp?treeMenu=type_PLCProductLine"
sorttype none
setting "Registered Suite" ProductLine
setting "Show Type Icon" true
setting "Target Location" content
column name colname3
label emxProduct.Table.Description
businessobject description
sorttype none
setting "Registered Suite" ProductLine
column name colname4
label emxProduct.Table.State
businessobject current
sorttype none
setting "Admin Type" State
setting "Registered Suite" ProductLine
column name colname5
label emxProduct.Table.Owner_GlobalPM
businessobject owner
sorttype none
setting Export true
setting "Registered Suite" ProductLine
setting format user
column name colname6
sorttype none
setting "Column Type" File
setting "Registered Suite" Components
column name colname7
href "${COMMON_DIR}/
emxTree.jsp?mode=replace&treeMenu=type_PLCProductLine"
sorttype none
setting "Column Icon" "images/iconNewWindow.gif"
setting "Column Type" icon
setting "Popup Modal" false
setting "Registered Suite" ProductLine
setting "Target Location" popup;
```

Copy Table

You can modify any table that you own, and copy any table to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, Business Administrators can change their workspace to that of another user to work with tables that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

After a table is defined, you can clone the definition with the Copy Table command.

If you are a Business Administrator with table access, you can copy system tables. If you are a Business Administrator with person access, you can copy tables in any person's workspace (likewise for groups and roles). Other users can copy visible workspace tables to their own workspaces.

This command lets you duplicate table definitions with the option to change the value of clause arguments:

```
copy table SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the table definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any table of the same name belonging to the user specified in the `Touser` clause.

The order of the `Fromuser`, `Touser` *and* `Overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

To copy a system table use "system" as both the fromuser and touser name. For example:

```
copy table PartTable SpecTable fromuser system touser system;
```

History Clause

The `history` keyword adds a history record marked "custom" to the system table that is being copied. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

The history clause applies only to system tables; it does not apply to user tables.

Modify Table

If you are a Business Administrator with table access, you can modify system tables. If you are a Business Administrator with person access, you can modify tables in any person's workspace (likewise for groups and roles). Other users can modify only their own workspace tables.

You must be a business administrator with group or role access to modify a table owned by a group or role.

Use the Modify Table command to add or remove defining clauses and change the value of clause arguments:

```
modify table NAME user USER_NAME [MOD_ITEM {MOD_ITEM}];
```

Or

```
modify table NAME system [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the table you want to modify. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.
- `system` refers to a table that is available for system-wide use, and not associated with the session context.
- `MOD_ITEM` is the type of modification you want to make. Each is specified in a Modify Table clause, as listed in the following table. Note that you need specify only the fields to be modified.

Modify Table Clause	Specifies that…
`name NEW_NAME`	The current table name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
`units | [picas] | points | inches`	The current units definition is changed to the new units specified.
icon FILENAME	The image is changed tp the new image in the field specified.
`column delete COLUMN_NUMBER`	The column identified by the given column number is removed from the table. To obtain the column number for a specific column, use the Print table command. When the table definition is listed, note the number assigned to the column to delete.
`column delete name COLUMN_NAME`	The column identified by the given column name is removed from the table.
`column modify COLUMN_NUMBER [label STRING_VALUE] COLUMN_TYPE_DEF [COLUMN_DEF_ITEM]`	The column identified by the given column number is modified. To obtain the column number for a specific column, use the Print table command. When the table definition is listed, note the number assigned to the column to delete.
`column modify name COLUMN_NAME [label STRING_VALUE] COLUMN_TYPE_DEF [COLUMN_DEF_ITEM]`	The column identified by the given column name is modified.
`column COLUMN_DEF`	A new column can be defined according to the column definition clauses and placed at the end of the table.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`visible USER_NAME{,USER_NAME};`	The object is made visible to the other users listed.

Modify Table Clause	Specifies that…
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the system table that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*. The history clause applies only to system tables; it does not apply to user tables.

Each modification clause is related to the arguments that define the table. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a table, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire command.
- If you are working from a script, group the changes together in a single Modify Table command.

Deriving a User Table from a System Table

If you are a user with access to a system table, you can copy (clone) the system table to create a new table in your workspace. This new user table inherits all the columns of the system table. Any subsequent changes made to the system table are reflected in the derived user table as well.

Once derived, the user table can be customized to add new columns, hide existing columns and to change the order of the columns. However, other aspects of the user table, like expressions and settings of columns inherited from the system table, cannot be altered.

A user table that is derived from a system table cannot be used to derive another user table.

You can only derive a user table from a system table in MQL. Once the derived user table is created, you can see it the same as your other tables.

The Copy Table command lets you derive a user table by copying a system table:

```
copy table SRC_NAME DST_NAME [derived] [COPY_ITEM
{COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the table definition (source) to be copied. Must be a system table if your are deriving from it.
- `DST_NAME` is the name of the new definition (destination); that is, the user table being created.
- `derived` is an optional keyword that should be given just after the `DST_NAME` (name of the new user table) to indicate that the new table is derived. When the derived keyword is included, the SRC_NAME table must be a system table. The default is `notderived`.

- COPY_ITEM can be:

fromuser USERNAME	USERNAME is the name of a person, group, role or association.
touser USERNAME	
overwrite	Replaces any table of the same name belonging to the user specified in the Touser clause or the current workspace.

The order of the Fromuser, Touser *and* Overwrite *clauses is irrelevant, but* MOD_ITEMS, *if included, must come last.*

- A Business Administrator with person access can derive a system table to another user's workspace with the Touser clause.
- MOD_ITEM is the type of modification you want to make. Refer to the table in *Modify Table* for a complete list of possible modifications.

Derived Table Behavior

When you modify a system table after a user table has been derived from it, the modifications are not apparent in the table definition until the user table is modified. However, when applied to objects in a Matrix Navigator browser or product page, the changes are seen immediately.

Similarly, if a derived user table is modified with new columns added, and then the system table it was derived from is modified by adding new columns, the order of the columns in the derived table may not be as expected. You will need to readjust the column order numbers in MQL or rearrange the columns.

If the system table (source) is deleted, the user table derived from it is also deleted.If you want to keep the derived user table, you can first modify it to make it a standalone table. For example:

```
modify table test user1 notderived;
```

If you need to export a derived table, all columns must have a name. If they do not, you will get a warning when exporting. You must add names to any un-named column and re-export the table or import will fail.

Evaluate Table

Integrators can use the evaluate table command to include information from Live Collaboration tables in other applications. Information is returned for a single business object or relationship at a time. The business object ID or relationship ID is required and can be obtained using the print command.

```
evaluate table NAME [user USER_NAME | system] businessobject
ID;
```

Or

```
evaluate table NAME [user USER_NAME | system] relationship
ID;
```

- NAME is the name of the table you want to evaluate.

- USER_NAME refers to a person, group, role, or association.

system refers to a table that is available for system-wide use, and not associated with the session context.

When a table is evaluated, all fields contained within the table for the specified business object ID or relationship ID are listed on your screen. For example, the following command looks for information from a table named “RPO:”

```
evaluate table RPO businessobject 68104.11481.34617.772;
```

This command might produce results similar to the following:

```
sismuth  9/12/98  62750  TKA Netting Co.  Pr.1
```

Evaluating Tables on Collections of Objects

The evaluate table command also allows a SEARCHCRITERIA (as defined in *SEARCHCRITERIA Clause*) to be given as an alternative to using the businessobject or relationship keywords:

```
evaluate table NAME [user USER_NAME | system] [list bus |
rel] SEARCHCRITERIA;
```

When using a SEARCHCRITERIA in an evaluate table command, the table columns are evaluated as summary data across the entire collection, returning a single row of information. As such, the column definitions should contain the set keyword, and use set-oriented expressions (that is: count, average, maximum, minimum, sum, product, standarddeviation, correlation). Any column definitions that include single business object expressions are ignored.

You can optionally include the List Businessobject or List Relationship clause with a SEARCHCRITERIA to indicate that it applies to 1 or the other.

Delete Table

If you are a Business Administrator with table access, you can delete system tables. If you are a Business Administrator with person access, you can delete tables in any person’s workspace (likewise for groups and roles). Other users can modify only their own workspace tables.

You must be a business administrator with group or role access to delete a table owned by a group or role.

If a table is no longer required, you can delete it using the Delete Table commands

```
delete table NAME [user USER_NAME | system];
```

- NAME is the name of the table to be deleted. If you are a business administrator with person access, you can include the User clause to indicate another user’s workspace object.
- USER_NAME refers to a person, group, role, or association.
- system refers to a table that is available for system-wide use, and not associated with the session context.

Searches the list of defined tables. If the name is found, that table is deleted. If the name is not found, an error message is displayed.

For example, to delete the table named “Income Tax Table,” enter the following:

```
delete table "Income Tax Table";
```

After this command is processed, the table is deleted and you receive an MQL prompt for another command.

Print Table

Use the Print Table command to print information about the attributes of a specific table, including the number and characteristics of each table column.

```
print table NAME [user USER_NAME | system][SELECT];
```

- `NAME` is the name of the table to be printed.
- `USER_NAME` refers to a person, group, role, or association.
- `system` refers to a table that is available for system-wide use, and not associated with the session context.

`SELECT` specifies a subset of the list contents.

Searches the list of defined tables. If the name is found, that table information is printed. If the name is not found, an error message is displayed. For example, to print details about the table named “DescNote,” enter the following:

```
print table "DescNote";
```

The following is sample output:

```
MQL<28>print table 'DescNote';

table           DescNoteRes
 inactive

 #100000 column
   label      Description
   businessobject description
   size       11 2
   minsize    0 0
   autoheight true
   autowidth  true
   editable   true
   hidden     false
   sorttype   none
   user       all

 #100001 column
   label      Notes
   businessobject attribute[Notes]
   size       10 2
   minsize    0 0
   autoheight true
   autowidth  true
   editable   true
   hidden     false
   sorttype   none
   user       all
```

```
nothidden
created Wed Oct 31, 2001 2:57:09 PM EST
modified Wed Feb 20, 2002 2:47:56 PM EST
```

Since tables have additional uses in support of dynamic UI modeling, the MQL print command suppresses the output of data that is not used. For example, if you print a table that is defined as a system object used for Web applications, the following selects will not be printed:

`size, minsize, scale, font, minwidth, minheight, absolutex, absolutey, xlocation, ylocation, width, and height.`

Conversely, when printing non-Web tables, parameters used only for Web-based tables are suppressed from the output:

`href, alt, range, update, and settings`

Example

When using a `SEARCHCRITERIA` in an `evaluate table` command, the table columns are evaluated as summary data across the entire collection, returning a single row of information. As such, the column definitions should contain the `set` keyword, and use set-oriented expressions (that is: count, average, maximum, minimum, sum, product, standarddeviation, correlation). Any column definitions that include single business object expressions are ignored.

Example:

```
add table settable
     column label Sum set ' sum ( attribute[Priority] ) '
     column label Average set ' average ( attribute[Priority] ) '
     column label P1Count set 'count ( attribute[Priority] == 1 ) ';
evaluate table 'settable' set Escalate;
18.0  1.5  6
# Add a column whose expression is a "single bus obj" expression:
mod table settable add column label Priority set ' attribute[Priority] ';
# Which yields exactly the same evaluation:
evaluate table 'MySetTable' set Escalate;
18.0  1.5  6
```

If you want to perform an evaluate table with single business object expressions against each member of a collection, you need to create your own loop.

```
# Create a list of busId's for the set
set lObject [mql print set escalate select id dump |]
set lBusId [split $lObject \n]
# loop through the list of busId's and run evaluate table for each
# NOTE: use the keyword 'bus' to indicate a bus obj, not a rel ID.
foreach sBusId $lBusId {
  set sRow [mql evaluate table 'MyBOTable' bus $sBusId]
  puts $sRow
}
```

thread Command

Description

MQL has a notion of a *thread*, where you can run multiple MQL sessions inside a single process. Each thread is like a separate MQL session, except that all threads run within the same MQL process. This concept was introduced into MQL to support the MSM idea of multiple “deferred commit” windows.

User Level

System Administrators

Syntax

The syntax for the thread command is:.

```
[start|resume|print|kill] thread;
```

Start Thread Command

By default, when you enter MQL, you are in thread number 1. You can start a new thread by running the `start thread` command. The command returns the ID (an integer) of the new thread. You are now in the new thread.

```
start thread;
```

If, in the previous thread, you had started a transaction and made a change to the database but had not committed it yet, the change will not be visible in the new thread. This behavior is the same as you would have if you had multiple MQL processes running and in one of them a transaction had been started and a change made.

Because application/framework/custom programs cannot guarantee that all programs have been precompiled, it is important to adhere to the following rules when using secondary threads:

1. Keep the operations performed on the secondary thread to an absolute minimum.
2. Never modify any objects that could possibly be in an update state due to modifications on any other transaction thread, including the main thread.
3. Never perform any operation that could cause a JPO to be compiled, since this may cause implicit write operations for JPOs that need compiling.

Resume Thread Command

You can switch back to an existing thread using the `resume thread` command. Indicate the ID of the thread you want to resume. For example, if you are in the third thread and want to return to the first thread, use:

```
resume thread 1;
```

Print Thread Command

You can find out the current thread by running `print thread`. The ID number of the current thread is returned.

```
print thread;
```

Kill Thread Command

A thread other than the current one can be killed (like killing an MQL process) by using the `kill thread` command. Indicate the number of the thread that you want to terminate.

```
kill thread 2;
```

tip Command

Description

Tips are not supported by web applications and should no longer be used.

Object Tips are small pop-up windows that appear from Matrix Navigator when you hold the cursor briefly over any object in a browser. They are similar to the *Tool Tips* that appear over toolbar buttons in many programs to tell you what the button is for. However, unlike tool tips, you can define the contents of the Object Tip window to display the information you need most often about the objects you use. You can also use tips instead of Type Name and Revision to identify objects in Matrix Navigator.

For example, you could include any attribute or basic property for an object that you frequently need to know, such as the current owner, or the last date the object was changed. This gives you a quick way to check basic data about an object without having to select Basics or Attributes from the menu or toolbar.

You can save Object Tip definitions by name (as personal settings) to turn on or off as you need them. A single tip may become part of several Views, being activated in one, and available in another. Refer to the *Matrix Navigator Guide : Using View Manager* for more information on Views.

From Matrix Navigator browsers, tips that quickly display pertinent information about objects can be very useful. Each user can create her/his own tips from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar tips consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Tips can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. Once the tips are defined, individual users can turn them on and off from the Matrix Navigator browsers as they are needed.

In the Matrix Navigator browsers, they display on the Tips tab within the Visuals Manager window and in the View menu.

It is important to note that tips are all Personal Settings that are available and activated only when context is set to the person who defined them.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]tip NAME {CLAUSE};
```

- NAME is the name of the tip you are defining. Tip names cannot include asterisks.
- CLAUSEs provide additional information about the tip.

Add Tip

To define a new tip from within MQL, use the Add Tip command:

```
add tip NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the tip you are defining.

USER_NAME can be included with the user keyword if you are a business administrator with person access defining a tip for another user. If not specified, the tip is part of the current user's workspace.

ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the tip, you cannot have the same name for two tips. If you use the name again, an error message will result. However, several different users could use the same name for different tips. (Tips are local to the context of individual users.)

After assigning a tip name, the next step is to specify the conditions (ADD_ITEM) that each object must meet in order to display when the tip is turned on (activated). The following are Add Tip clauses:

[!|in|notin|not]active
appliesto businessobject|relationship
type TYPE_PATTERN
name PATTERN
minorrevision REVISION_PATTERN
vault PATTERN
owner PATTERN
where QUERY_EXPR
expression EXPRESSION
[!|not]hidden
visible USER_NAME{,USER_NAME};
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Applies To Clause

This clause indicates that the tip applies to business objects or relationships.

Type Clause

This clause assigns a tip for a particular type of business object or relationship.

```
type TYPE_PATTERN
```

TYPE_PATTERN defines the types for which you are assigning a tip.

The Type clause can include more than one type by using multiple values to define the pattern. The Type clause can also use wildcard characters.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, Live Collaboration reads the value as the next part of the tip definition.

Name Clause

This clause assigns the names of objects to include in the tip.

```
name PATTERN
```

PATTERN defines the name(s) of objects to include in your tip.

The Name clause can include more than one name by using multiple values to define the pattern. The Name clause can also use wildcard characters. For example, the following definition displays a tip to display all business objects with names that start with “Inter”, or include the letters IBM, or include the words “International Business Machines”:

```
add tip“IBM”
name Inter*,*IBM*,”International Business Machines”;
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, "International Business Machines"). If you include spaces, Live Collaboration reads the value as the next part of the tip definition, as described in the Type clause.

Minorrevision Clause

This clause assigns a tip for a particular revision of business objects.

```
minorrevision REVISION_PATTERN
```

REVISION_PATTERN defines the revision for which you are assigning a tip.

The Minorrevision clause can include more than one revision value and wildcards as in the other clauses that use patterns. Creating a tip that shows the most current change to an object can be very useful. This gives you a quick way to identify only the objects that have recently changed.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, Live Collaboration reads the value as the next part of the tip definition.

Vault Clause

This clause assigns a tip for business objects that are in a particular or similar vault:

```
vault PATTERN
```

PATTERN defines the vault(s) for which you are assigning a tip.

The Vault clause can use wildcard characters. For example, the following definition displays a tip for all business objects that reside in the “Vehicle Project” vault:

```
add tip “Vault Search”
   businessobject * * *
   vault “Vehicle Project”
   owner *;
```

Expression Clause

This clause can be used to obtain or use information related to a tip, including business object data and also administrative object information.

EXPRESSION can be any select expression available for business objects. The object tip will display up to 100 characters.

Copy Tip

You can modify any tip that you own, and copy any tip to your own workspace that exists in any user definition to which you belongor that is defined as visible to you. As an alternative to copying definitions, you can use the `set workspace` command to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with tips that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

After a tip is defined, you can clone the definition with the Copy Tip command. Cloning a tip definition requires Business Administrator privileges, except that you can copy a tip definition to your own context from a group, role or association in which you are defined.

This command lets you duplicate tip definitions with the option to change the value of clause arguments:

```
copy tip SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the tip definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

COPY_ITEM can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any tip of the same name belonging to the user specified in the `touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Tip

Use the Modify Tip command to add or remove defining clauses and change the value of clause arguments:

```
modify tip NAME [user USER_NAME] [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the tip you want to modify. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.
- MOD_ITEM is the type of modification you want to make. With the Modify Tip command, you can use these modification clauses to change a tip:

Modify Tip Clause	Specifies that…
active	The active option is changed to specify that the object is active.
notactive	The active option is changed to specify that the object is not active.
appliesto businessobject|relationship	The appliesto option is changed to reflect what the tip now affects.
type TYPE_PATTERN	The type criteria is changed to the named pattern.
name PATTERN	The name criteria is changed to the named pattern.
revision REVISION_PATTERN	The revision criteria is changed to the named pattern.
vault PATTERN	The vault criteria is changed to the pattern specified.
owner PATTERN	The owner criteria is changed to the pattern specified.
where QUERY_EXPR	The query expression is modified.
expression EXPRESSION	The select expression is changed to the expression specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
visible USER_NAME{,USER_NAME};	The object is made visible to the other users listed.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

These clauses are essentially the same ones that are used to define an initial tip except that Add property and Remove property clauses are included. When making modifications, you simply substitute new values for the old.

Although the Modify Tip command allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the tip name or remove the tip entirely, you must use the Delete Tip command and/or create a new tip.

Delete Tip

If a tip is no longer needed, you can delete it using the Delete tip command:

```
delete tip NAME [user USER_NAME];
```

- `NAME` is the name of the tip to be deleted. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.
- Searches the local list of existing tips. If the name is found, that tip is deleted. If the name is not found, an error message results.

For example, assume you have a tip named "Current Docs" that you no longer need. To delete this tip from your area, you would enter the following MQL command:

```
delete tip "Current Docs";
```

After this command is processed, the tip is deleted and you receive the MQL prompt for the next command.

When a tip is deleted, there is no effect on the business objects or on queries performed by Live Collaboration. Tips are local only to the user's context and are not visible to other users.

toolset Command

Description

Toolsets are not supported by web applications and should no longer be used.

The Business Administrator creates programs to perform specific functions. Some programs can run automatically when certain trigger events occur, such as the promotion of an object to a new state. Other programs could be executed to automate and standardize a common task, such as creating a cost analysis for a project, or creating a report on a project. Business Administrators create three types of programs:

- *Programs*, which can be executed without first selecting an object. This type of program might perform a query and generate a report on the found objects.
- *Methods*, which are programs that are associated with particular object type. For example, a method on a Product object might create a User Guide object and connect it to the Product automatically.
- *Wizards*, which are programs with a user interface that ask a series of questions and then execute their code. Wizards are similar to many Windows installation programs, and can be used to simplify or standardize a complex process. Wizards may be either stand-alone, like other Programs, or require a business object on which to act, and so become a method. Wizards are a special type of program and can be used as a program or a method .

When creating toolsets, you can access the list of all available Programs, Methods, and Wizards, then add a toolbar with buttons to perform these functions. You may want to consult your Business Administrator when configuring your toolsets to determine the optimal selection for your individual use.

You can save Toolset definitions by name (as personal settings) to turn on or off as you need them. A single toolset may become part of several Views, being activated in one, and available in another. Refer to the *Matrix Navigator Guide : Using View Manager* for more information aboutViews.

From Matrix Navigator browsers, toolsets that provide easy execution from toolbar buttons can be very useful. Each user can create her/his own toolsets from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar toolsets consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Toolsets can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. Once the toolsets are defined, individual users can turn them on and off from the Matrix Navigator browsers as they are needed.

In the Matrix Navigator browsers, toolsets display on the Toolsets tab page within the Visuals Manager window and in the View menu.

It is important to note that toolsets are all Personal Settings that are available and activated only when context is set to the person who defined them.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]toolset NAME {CLAUSE};
```

- NAME is the name of the toolset you are defining. The toolset name cannot include asterisks.
- CLAUSEs provide additional information about the toolset.

Add Toolset

To define a new toolset from within MQL, use the Add Toolset command:

```
add toolset NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the toolset you are defining.
- USER_NAME can be included with the user keyword if you are a business administrator with person access defining a toolset for another user. If not specified, the toolset is part of the current user's workspace.
- ADD_ITEM specifies the characteristics you are setting.
- When assigning a name to the toolset, you cannot have the same name for two toolsets. If you use the name again, an error message will result. However, several different users could use the same name for different toolsets. (Remember that toolsets are local to the context of individual users.)
- After assigning a toolset name, the next step is to specify the conditions (ADD_ITEM) that must be met in order to display the toolbar button when the toolset is turned on (activated).

The following are Add Toolset clauses:

[!|in|notin|not]active
program | NAME {,NAME} |
method
[!not]hidden
visible USER_NAME{,USER_NAME};
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Program Clause

This clause allows you to define which programs and/or wizards should be added to the Toolset. You can include multiple programs/wizards.

Method Clause

This clause allows you to define the method to be added to the Toolset. Only one method can be included in a toolset.

Copy Toolset

You can modify any toolset that you own, and copy any toolset to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, Business Administrators can change their workspace to that of another user to work with toolsets that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

After a toolset is defined, you can clone the definition with the Copy Toolset command.

If you are a business administrator with person access, you can copy toolsets to and from any person's workspace (likewise for groups and roles). Other users can copy visible toolsets to their own workspaces.

This command lets you duplicate toolset definitions with the option to change the value of clause arguments:

```
copy toolset SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- `SRC_NAME` is the name of the toolset definition (source) to be copied.
- `DST_NAME` is the name of the new definition (destination).
- `COPY_ITEM` can be:

`fromuser USERNAME`	USERNAME is the name of a person, group, role or association.
`touser USERNAME`	
`overwrite`	Replaces any toolset of the same name belonging to the user specified in the `touser` clause.

The order of the `fromuser`, `touser` *and* `overwrite` *clauses is irrelevant, but* `MOD_ITEMS`, *if included, must come last.*

- `MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Toolset

Use the Modify Toolset command to add or remove defining clauses and change the value of clause arguments:

```
modify toolset NAME [user USER_NAME] [MOD_ITEM {MOD_ITEM}];
```

- `NAME` is the name of the toolset you want to modify. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.
- `ITEM` is the type of modification you want to make.
- With the Modify Toolset command, you can use these modification clauses to change a toolset:

Modify Toolset Clause	Specifies that…
`active`	The active option is changed to specify that the object is active.
`notactive`	The active option is changed to specify that the object is not active.
`add program NAME {,NAME}`	The named program is added to the toolset.

Modify Toolset Clause	Specifies that…
`add method NAME {,NAME}`	The named method is added to the toolset.
`remove program NAME {,NAME}`	The named program is removed from the toolset.
`remove method NAME {,NAME}`	The named method is removed from the toolset.
`remove all`	All methods and programs are removed from the toolset.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`visible USER_NAME{,USER_NAME};`	The object is made visible to the other users listed.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

These clauses are essentially the same ones that are used to define an initial toolset except that Add property and Remove property clauses are included. When making modifications, you simply substitute new values for the old.

Although the Modify Toolset command allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the toolset name or remove the toolset entirely, you must use the Delete toolset command and/or create a new toolset.

Delete Toolset

If a toolset is no longer needed, you can delete it using the Delete toolset command:

```
delete toolset NAME [user USER_NAME];
```

`NAME` is the name of the toolset to be deleted. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.

Searches the local list of existing toolsets. If the name is found, that toolset is deleted. If the name is not found, an error message results.

When a toolset is deleted, there is no effect on the business objects or on queries. Toolsets are local only to the user's context and are not visible to other users.

trace Command

Description

Server diagnostic tools allow the following *trace* information to be sent either to a file or to standard output (the “destination”).

For conceptual information on this command, refer to the *Configuration Guide : About Tracing* in the online documentation.

User Level

System Administrator

Syntax

Tracing can be turned on (and off) using the following MQL commands:

```
trace type TYPE{,TYPE}|filename FILENAME | [not full];
                      |on                |
                      |off               |
                     |text STRING        |
```

Each clause is described in the sections that follow.

Type Clause

`TYPE` is the type of tracing to affect and is one of the following:

`ds`
`fcs`
`ftp`
`index`
`jpo`
`ldap`
`logwriter`
`memory`
`mql`
`nul`
`OTHER_TYPE`
`portlet`

searchindex
smtp
sql
store
trigger
verbose

Particular care should be taken in the use of SQL and VERBOSE tracing. Both are low level, and liable to generate a large amount of output data, which will affect performance.

The keyword `jpo` enables JPO compilation or invocation messages to be output. It is important to note that enabling only the `verbose` keyword will not produce this particular output.

The keyword `memory` can be used to turn on memory manager tracing. You can enable more verbose memory tracing via an environment variable only. Refer to the *Installation Guide : Server Diagnostics* chapter for more information on memory tracing options.

The keyword `verbose` enables more details to be output. It also enables Studio Customization Toolkit Origin tracing when classes to trace are defined with MX_ADK_TRACEALL environment variable or using the name as the keyword. The keyword `logwriter` may be used to turn on Studio Customization Toolkit tracing. If no filename is included it will create ematrix.log in the MX_TRACE_FILE_PATH directory. Use of OTHER_TYPE is recommended over use of logwriter. The output below shows the following calling stack to illustrate both the timing and automatic Studio Customization Toolkit origin tracing: a JSP page calls to the kernel using the Studio Customization Toolkit interface MQLCommand.executeCommand() with the command 'exec program JPOGetAttr'. The Java Program Object JPOGetAttr is called from within the kernel. The JPOGetAttr makes further Studio Customization Toolkit calls to:

1. Context.reset to establish 'creator' as current context
2. BusinessObject.open to open the object 't3 t3-1 0'
3. BusinessObject.getAttributes to get the object's attributes
4. Context.getClientTask to retrieve ClientTasks, if any.

For readability, the lengthy session id has been shortened, and the full path name of the JSP has been shortened from org.apache.jsp.common.emxRunMQL_jsp to show only the last level:

```
19:04:27.902 VERB t@2672 stateless dispatch for
executeCmd.bosMQLCommand
19:04:27.902 VERB t@2672 allocate context for session
49XX:mxYY:(emxRunMQL_jsp:521)
19:04:27.902 VERB t@2672   input params: cmd=exec prog  JPOGetAttr;
19:04:27.902 MQL  t@2672 Start MQLCommand: exec prog  JPOGetAttr;
19:04:27.912 MQL  t@2672 Session: 49XX:mxYY:(emxRunMQL_jsp:521)
19:04:27.912 MQL  t@2672 Program: JPOGetAttr
19:04:27.912 MQL  t@2672    args:

19:04:27.912 VERB t@2672 stateless dispatch for
allocExternalContext.bosInterface
19:04:27.912 VERB t@2672   input params:
sessionId=49XX:mxYY:(emxRunMQL_jsp:521), stackTrace=
```

```
19:04:27.912 VERB t@2672 dispatch complete since 19:04:27.912, 0.000
secs (0.000 direct 0.000 nested)

19:04:27.912 VERB t@2672 stateful dispatch for reset.bosContext
19:04:27.912 VERB t@2672   input params:
(session=49XX:mxYY:(emxRunMQL_jsp:521)), user=creator, passwd=,
lattice=
19:04:27.922 VERB t@2672   output params: returnVal=creator
19:04:27.922 VERB t@2672 dispatch complete since 19:04:27.912, 0.010
secs (0.010 direct 0.000 nested)

19:04:28.012 VERB t@2672 stateless dispatch for
openTNRV.bosBusinessObject
19:04:28.012 VERB t@2672 allocate context for session
49XX:mxYY:(emxRunMQL_jsp:521)
19:04:28.012 VERB t@2672   input params: name=t3, type=t3-1, rev=0,
vault=unit1
19:04:28.012 VERB t@2672   output params: returnVal
objectid=62869.36236.33703.1576
19:04:28.012 VERB t@2672 dispatch complete since 19:04:28.012, 0.000
secs (0.000 direct 0.000 nested)

19:04:28.012 VERB t@2672 stateless dispatch for
getAttributes.bosBusinessObject
19:04:28.012 VERB t@2672 allocate context for session
49XX:mxYY:(emxRunMQL_jsp:521)
19:04:28.012 VERB t@2672   input params: id=62869.36236.33703.1576,
getHidden=0
19:04:28.032 VERB t@2672 dispatch complete since 19:04:28.012, 0.020
secs (0.020 direct 0.000 nested)
>>>> End of program execution:
19:04:28.052 MQL  t@2672 End Program: JPOGetAttr since 19:04:27.912,
0.140 secs (0.110 direct 0.030 nested)
19:04:28.062 MQL  t@2672 End MQLCommand
>>>> End of invocation of MQLCommand.executeCommand:
19:04:28.062 VERB t@2672 dispatch complete since 19:04:27.902, 0.160
secs (0.160 direct 0.000 nested)

19:04:28.062 VERB t@2672 stateful dispatch for
getClientTask.bosContext
19:04:28.062 VERB t@2672   input params:
(session=49XX:mxYY:(emxRunMQL_jsp:521))
19:04:28.062 VERB t@2672   output params: returnVal length=0
19:04:28.062 VERB t@2672 dispatch complete since 19:04:28.062, 0.000
secs (0.000 direct 0.000 nested)
```

The keyword `logwriter` may be used to turn on Studio Customization Toolkit tracing. If no filename is included it will create ematrix.log in the MX_TRACE_FILE_PATH directory. Use of OTHER_TYPE is recommended over use of logwriter.

The keyword `store` can be used with the `trace type` command, so that when synchronizing or purging stores a log file can be created that indicates the success or failure of every file and business object.

OTHER_TYPE allows the definition of a user-defined tracing type, for example, “My Trace.” Programmers can embed tracing messages in their implementation code to provide strings to be output to the trace file of type OTHER_TYPE using the following:

```
trace type OTHER_TYPE text STRING;
```

With these types of messages within programs, you would then enable the tracing with one of the following:

```
trace type OTHER_TYPE on;
trace type OTHER_TYPE filename FILENAME;
```

File Clause

Specifying a file with the `file FILENAME` clause turns the specified type of tracing on, and redirects the output to the specified file. There is no need to use the keyword “on”. If a filename has already been specified for another type of tracing within the current session, that filename will be used, and the one specified here will be ignored, but the tracing will be enabled.

If the filename specified for any tracing already exists in the directory MX_TRACE_FILE_PATH, that existing file will be copied to a backup whose name is constructed by prepending a time-specific prefix to the filename, in the form “yyyymmddhhmmss__FILENAME.”

On Clause

The `on` modifier will turn the specified tracing on and send the trace information to stdout, unless a trace file is already in use by another type of tracing.

The On clause for the Trace command is not supported in the Windows Studio version of MQL in versions V6R2009x and later. This applies only to the 3DEXPERIENCE Platform (PC rich client). Tracing does work for UNIX and for the console version that is shipped together with the Server.

Off Clause

The `off` clause turns the specified type of tracing off. If other tracing types have been turned on, they will stay on. Tracing that is turned on via .ini file settings can only be turned off using the keyword `all`. For example:

```
trace type SQL off;
trace type all off;
```

When all types of tracing directed at the same file are turned off, the file is closed. In order to resume tracing to a file, any subsequent command must specify a filename. If it doesn’t, tracing will be resumed with stdout as the destination.

Instead of having to turn trace types off type-by-type, you can use ‘off’ with no trace type specified to turn all currently active tracing off and to close and unset the destination.

```
trace off [thread];
```

Pause/Resume Clause

You can use pause to make all the currently active trace types inactive. No tracing output will be produced until a ‘trace resume’ command is issued. Both the list of trace types and the destination

are retained, so resume will pick up tracing the same types of information to the same place as it was before pausing. If tracing is not paused, trace resume has no effect.

```
trace |pause  |;
      |resume |
```

Not Full Clause

Trace output includes timestamp information, which you can turn off with the `notfull` clause. You can also use `!full`. To re-enable, use the `full` clause. This is helpful with SQL tracing so that you can use the SQL without editing it.

Print trace

The `print trace` command outputs information for all trace settings, as illustrated by the following trace commands:

```
MQL<3>trace type mql,sql filename mqlsql.log;
MQL<4>trace pause;
MQL<5>print trace;
All thread trace:
   type = MQL,SQL
   pathname = c:\ematrix9\logs
   filename = mqlsql.log <paused>
   full = TRUE
```

You will notice in the above example that when tracing is paused, it is marked by <paused> after the filename.

If tracing is turned off, the type and filename fields will be empty.

Errors

If tracing is currently active with a defined destination and you turn on additional tracing with a different destination, the system will inform you of the current defined destination in a warning, and your added tracing will be streamed to the current destination as well. For example:

```
MQL<16>trace type sql filename sql.log;
MQL<18>print trace;
Trace:
   type = SQL
   pathname = c:\ematrix9\logs
   filename = sql.log
   fulllabel = TRUE
MQL<19>trace type mql filename mql.log;
Warning: #1600078: Filename mql.log ignored. Tracing already active to
file c:\ematrix9\logs/sql.log
```

transaction Command

Description

A *transaction* involves accessing the database or producing a change in the database. All MQL commands involve transactions.

For conceptual information on this command, see *Working with Implicit and Explicit Transactions* in Chapter 1.

User Level

System Administrators

Syntax

Several MQL commands enable you to explicitly start, abort, and commit transactions:

`abort transaction [NAME];`
`commit transaction;`
`print transaction;`
`start transaction [read];`
`set transaction wait|nowait|savepoint [NAME];`

These commands enable you to extend the transaction boundaries to include more than one MQL command.

- If you are starting a transaction, use the `read` option if you are only reading.
- Without any argument, the `start transaction` command allows reading and modifying the database.

transition Command

Description

The Transition command supports the migration of a database from one revision to another. The Transition command does not change base-level schema definitions, but supports cases where it is necessary to combine a change to administration definitions (policy, relationship) with a corresponding adjustment of the data stored for business objects and/or connections controlled by those definitions.

Three migration tools are provided for performing specialized migration operations with the Transition command:

- One for transitioning revisions
- One for transitioning `published` flags
- One for transitioning to dynamic relationships

User level

Business Administrator

Syntax

The MQL syntax for the Transition command is:

```
transition [TRANSITION_ITEM];
```

TRANSITION_ITEM can be one of the following:

Transition Item	Description
`revisions`	• Creates major versions from existing VPLM data that has been mapped to minor revisions by the `upgrade` command (see *Migration 1: Transition Revisions command*).
`published-flags`	• Fixes the published/best-so-far bits in the mxIdent or lxRev field based on the state of the object (see *Migration 2: Transition Published-Flags command*, and *Published States* in Chapter 4).
`dynamic-relationship`	• Mutates existing PLMInstance relationship types and instances to be *dynamic* relationships (see *Migration 3: Transition Dynamic-Relationship command*, and *Dynamic Relationships* in Chapter 4).

Migration 1: Transition Revisions command

The Transition Revisions command creates major versions out of existing VPLM data by performing the following operations:

1. Fix the lxBO.lxRev field to concatenate existing revision strings with the new major-minor revision string using the first string from the new major-minor sequence.

2. Calculate majororder-minororder and update mxIdent based on previous/next pointers. (If VPLM data has resulted in v_order values that are in conflict with the previous/next order, this would be considered a mismanagement of the attribute.)
3. Set the lxBO next pointers to NULL.
4. Adjust the Ixprev/Ixnext chaining.

Syntax

The MQL syntax for the Transition Revisions commands is:

```
transition revisions policy NAME_PATTERN majorsequence STRING
   delimiter STRING;
transition revisions policy NAME_PATTERN minorsequence STRING
   delimiter STRING;
transition revisions policy NAME_PATTERN major;
transition revisions policy NAME_PATTERN minor;
```

This command allows the modification of a policy to add a major or minor sequence definition, and to update all business objects that are governed by the policy to update their policies.

The `majorsequence` or `minorsequence` keyword is mandatory. The delimiter keyword is required only if the desired end result is intended to allow for both major and minor revision sequences.

The NAME_PATTERN argument can be either the name of a single policy or a pattern to specify a list of policies (either a comma-separated list or a wildcard pattern). When a pattern is specified, all policies matching that pattern are transitioned atomically. If during processing business objects are found that are in a policy that is not in the list specified by NAME_PATTERN, the command errors out and leaves the data untouched.

Using the `minorsequence` keyword adds minor revision control to the policy, so this assumes that the existing sequence in the policy is to be mapped to the major sequence. It also implies that you want existing revision sequences to be broken into a series of major revisions (single-object minor revision families). The command combines the current revision of each object with the initial value of the new minor sequence provided in the command to generate a new major-minor revision string to store in the database. It also defines separate majorids for each single-object family (possibly reusing the object's physicalid) and toggles the MAJORID-has-been-resolved bits on the lxBO row.

Using the `majorsequence` keyword adds major revision control to the policy, so this assumes that the existing sequence in the policy is to be mapped to the minor sequence. It also implies that you want existing revision sequences to be kept as minor revision families. This form of the command calculates the initial value of the new major sequence provided in the command to generate a new major-minor revision string to store in the database.

In both the `majorsequence` and `minorsequence` cases, the command looks for `published` flags on the state definition in the policy and if they are there, propagates the `published` flag to all objects in those states.

The command also modifies the access masks on all states to ensure that any access rule that specifies revise access is updated to specify both `revisemajor`/`reviseminor` access in the same way. It also examines access filters to determine whether they use the "revise" string to refer to an access. If so, it replaces this reference with "revisemajor" or "reviseminor."

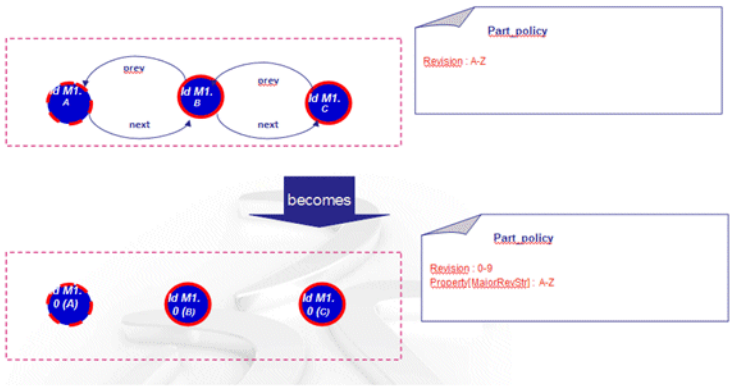
The `transition revisions policy NAME_PATTERN majorsequence STRING` *flavor of the command is not supported in V6R2013 because the VPLM application only needs the* `minorsequence` *flavor.*

The `transition revisions policy NAME_PATTERN minorsequence STRING` *flavor of the command can work only with policies that only have a minor sequence. This is because the whole point of the command is to migrate pre-V6R2012 data, which cannot possibly have a majorsequence.*

Since the Transition Revisions command changes the revision string for the business objects in the specified policy, this would require an update of any unique keys (including mxTNR) and indices that contain the revision selectables. In V6R2012, the Transition Revision command detected any such problematic unique keys and indices and refused to work until these were explicitly disabled. In V6R2013, the command behaves identically, but accepts an additional `force` modifier, which when used indicates that Transition Revision should itself temporarily disable the unique keys and indices, run the transition, and then re-enable everything that was disabled. This ensures that unique keys and indices are kept in synchronization with the updates to the business object's revision strings.

If the `transition revisions policy NAME_PATTERN minorsequence STRING` command is run twice, the second execution will error out because the policy already has a minorsequence.

You can use `transition revisions policy NAME_PATTERN major` to convert a minor-only sequence into a major-only sequence. The you can use `transition revisions policy NAME_PATTERN minor` to do the reverse conversion.



Migration 2: Transition Published-Flags command

The Transition Published-Flags command fixes flags on business objects depending on their current state. After a policy has been modified to have "published" states, existing business object data in those states is out of date. It needs to have the best-so-far/published bits turned on in lxBO.lxFLAGS.

Syntax

The MQL syntax for the Transition Published-Flags command is:

```
transition published-flags policy NAME;
```

This command finds business objects governed by the specified policy that are in published states, sets the new bit in lxBO.lxFLAGS, and creates mxFamily entries for the business objects' minor revision families.

Migration 3: Transition Dynamic-Relationship command

The Transition Dynamic-Relationship command makes existing relationship types and instances dynamic. A relationship type cannot be mutated to become dynamic with the Modify Relationship command. The new administration-only MQL Transition command can perform this change for VPLM data.

The MQL syntax for the Transition Dynamic-Relationship command is:

```
transition dynamic-relationship type ROOT_REL_TYPE;
```

This command converts the administration definition of the relationship type, and also migrates all connections that are of that type. It toggles the `dynamic` bit on the MXRELTYPE entry. It also removes the data from LXRO and creates new data in LXDRO/MXFAMILY to make those PLMInstances point to the now major versions.

Persistent Data

In addition to changes to tables described in the preceding sections, the mxStateReq table also has a new upgrade step that is used to detect that a new bit is being used in its MXFLAGS column.

The dynamic relationship feature adds an additional requirement for the definition of the database user for Oracle installations: the user must now have Create View privileges.

Transition commands produce no history records. Apart from the fact that a policy is transformed in such a way as to give it a new majorsequence and a new minorsequence, there is no way to tell that the policy was migrated using a Transition command.

type Command

Description

A *type* identifies a kind of business object and the collection of attributes that characterize it. When a type is created, it is linked to the (previously defined) attributes that characterize it. It may also have methods and/or triggers associated (refer to *Programs* in Chapter 7 and the *Administration Guide : Triggers* for more information). Types are defined by the Business Administrator and are used by users to create business object instances.

A type can be *derived* from another type. This signifies that the derived type is of the same kind as its parent.

For conceptual information on this command, see *Types* in Chapter 4.

User Level

Business Administrator

Syntax

```
[add|copy|modify]type NAME {CLAUSE};
```

- NAME is the name you assign to the type. Type names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the type.

Add Type

An object type is created with the Add Type command:

```
add type NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides additional information about the type:

description VALUE
attribute NAME {, NAME}
derived TYPE_NAME
abstract [true|false]
method PROG_NAME {, PROG_NAME}
form NAME {,NAME}
trigger EVENT_TYPE [minorrevision] {action|check|override} PROG_NAME [input ARG_STRING]
[!|not]hidden

```
property NAME [to ADMINTYPE NAME] [value STRING]
history STRING
```

All these clauses are optional. You can define a type by simply assigning a name to it. If you do, Live Collaboration assumes that the type is non-abstract, does not contain any explicit attributes, and does not inherit from any other types. If you do not want these default values, add clauses as necessary.

You may notice performance problems when adding a Type to a very large type hierarchy. If you experience this, from Oracle, turn off "cost-based optimization" and the situation should improve. Refer to Oracle documentation for more information.

Attribute Clause

This clause assigns explicit attributes to the type. These attributes must be previously defined with the Add Attribute command. (See *Add Attribute*.) If they are not defined, an error message is displayed.

Adding an attribute to a business type should not be included in a transaction with other extensive operations. This is a "special" administrative edit, in that it needs to update all business objects of that type with a default attribute.

If you add an attribute that is part of an index, the index is disabled. Refer to Working with Indices in Chapter 9 for more information.

For the Matrix Navigator user, the display of attributes (when creating objects or viewing attributes) will appear in the reverse order of the programmed order. Therefore, you should put the attribute you want to see first last in the MQL script.

A type can have any combination of attributes associated with it. For example, the following Add Type command assigns three attributes to the Shipping Form type:

```
add type "Shipping Form"
   description "Use for shipping materials to external locations"
   attribute "Label Type"
   attribute "Date Shipped"
   attribute "Destination Type";
```

Three explicit attributes are associated with this type definition: Label Type, Date Shipped, and Destination Type. When the user creates a business object of a Shipping Form type, s/he will be required to provide values for these attributes. For example, the user might enter the following values at the attribute prompts:

Prompt:	User Response:
Label Type	Overnight Express
Date Shipped	December 22, 1999
Destination Type	Continental U.S.

These values are then associated with that instance along with any files the user may want to check in.

Derived Clause

Use the Derived clause to identify an existing type as the parent of the type you are creating. The parent type can be abstract or non-abstract. A child type inherits the following items from the parent:

- all attributes
- all methods
- all triggers
- governing policies

  For example, if two policies list the parent type as a governed type, then those two policies can also govern the child type. Note that in such a case, the child type is not listed as a governed type in the policy definitions.
- allowed relationships

  For example, if a relationship allows the parent type to be on the FROM end, then the child type can also be on the FROM end of the relationship. The child type is not listed in the relationship definition.

Assigning a parent type is an efficient way to define several object types that are similar because you only have to define the common items for one type, the parent, instead of defining them for each type. The child type inherits all the items listed above from the parent but you can also add attributes, programs, and methods directly to the child type. Similarly, you can (and probably will) assign the child type to a policy or relationship that the parent is not assigned to. Any changes you make for the parent are also applied to the child type.

For example, suppose you have a type named “Person Record”, which includes four attributes: a person’s name, telephone number, home address, and social security number. Now, you create two new types named “Health Record” and “Employee Record:”

```
add type "Health Record"
   derived "Person Record";
add type "Employee Record"
   derived "Person Record";
```

Both new types require the attributes in the Person Record type. Rather than adding each of the four Person Record attributes to the new types, you can make Person Record the parent of the new types. The new types then inherit the four attributes, along with any methods, programs, policies, and relationships for parent.

Abstract Clause

An abstract type indicates that a user will not be able to create a business object of the type. An abstract type is helpful because you do not have to reenter groups of attributes that are often reused. If an additional field is required, it needs to be added only once. For example, the “Person Record” object type might include a person’s name, telephone number, home address, and social security number. While it is a commonly used set of attributes, it is unlikely that this information would appear on its own. Therefore, you might want to define this object type as an abstract type.

Since this type is abstract, there will never be any actual instances made of the Person Record type. However, it can be inherited by other object types that might require the attribute information. Even though a user may never be required to enter values for the attributes of a Person Record object, they may have to enter values for these attributes for an object that inherited the Person Record attributes (such as an Employee Record or Health Record).

Use one the following clauses:

```
abstract true
Or:
abstract false
```

If the user can create an object of the defined type, set the abstract argument to `false`. If not, set the abstract argument to `true`. If you do not use the Abstract clause, false is assumed, allowing users to create instances of the type.

For example, in the following definition of Federal Individual Income Tax Form, the user can create an object instance because an Abstract clause was not included in the definition:

```
add type "Federal Individual Income Tax Form"
   derived "Federal Tax Form";
```

This definition is equivalent to:

```
add type "Federal Individual Income Tax Form"
   derived "Federal Tax Form"
   abstract false;
```

Method Clause

The Method clause of the Add Type command assigns a method to the type. A method is a program that can be executed from Live Collaboration when it is associated with the selected object. Programs selected as methods require a business object as a starting point for executing. For example, the following adds the existing program named "calculate tax" to the Federal Individual Income Tax Form.

```
add type "Federal Individual Income Tax Form"
   derived "Federal Tax Form"
   method "calculate tax";
```

A user can then select any Federal Individual Income Tax Form object and execute the program "calculate tax."

Form Clause

The Form clause of the Add Type command assigns a form design to the type. This form must be previously defined with the Add Form command. (See *Add Form*.) If it is not defined, an error message is displayed.

A type can have any number of forms associated with it. For example, the following Add Type command assigns three forms to the Employee types:

```
add type "Employee Record"
   description "employees of Acme, Inc."
   form "insurance information"
   form "work history"
   form "pension and savings plans;
```

Trigger Clause

Event Triggers provide a way to customize behavior through Program objects. Triggers can contain up to three Programs, (a check, an override, and an action program) which can all work together, or each work alone. The Trigger clause specifies the program name, which event causes the trigger to execute, and which type of trigger program it is. Types support triggers for many events. For more information on Event Triggers, refer to the *Configuration Guide* : *Triggers*.

The format of the Trigger clause is:

```
trigger [minorrevision] EVENT_TYPE TRIGGER_TYPE PROG_NAME [input
ARG_STRING]
```

- `EVENT_TYPE` is any of the valid events for Types:

changevault	changename	changeowner
changepolicy	changetype	checkin
checkout	connect	copy
create	delete	disconnect
grant	lock	modify*
modifyattribute	modifydescription	removefile
minorrevision	revoke	unlock
*** The modify EVENT_TYPE only supports action triggers.**		

For a discussion of the modify events, refer to *Configuration Guide : Triggers* in the online documentation.

- `TRIGGER_TYPE` is Check, Override, or Action.
- `PROG_NAME` is the name of the program to execute when the event occurs.
- `ARG_STRING` is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.
- Environment variable “0” always holds the program name and is set automatically by the system.
- Arguments following the program name are set in environment variables “1”, “2”,… etc.

History Clause

The `history` keyword adds a history record marked “custom” to the type that is being added. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the addition. See also *Adding History to Administrative Objects*.

Copy Type

After a type is defined, you can clone the definition with the Copy Type command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy type SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- SRC_NAME is the name of the type definition (source) to copied.
- DST_NAME is the name of the new definition (destination).
- MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

History Clause

The history keyword adds a history record marked “custom” to the type that is being copied. The STRING argument is a free-text string that allows you to enter some information describing the nature of the copy operation. See also *Adding History to Administrative Objects*.

Modify Type

You can change the definition with the Modify Type command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify type NAME [MOD_ITEM {MOD_ITEM}];
```

- NAME is the name of the type you want to modify.
- MOD_ITEM is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Type clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Type Clause	Specifies that…
name NEW_NAME	The current type name changes to that of the new name entered.
description VALUE	The current description, if any, changes to the value entered.
icon FILENAME	The image is changed tp the new image in the field specified.
add attribute NAME	The named attribute is added to the type’s list of explicit attributes.
add form NAME	The named form is added to the type.
add method PROG_NAME	The named method is added to the type.
add trigger EVENT_TYPE PROG_TYPE PROG_NAME	The named trigger program name is added. This Modify clause must obey the same construction and syntax rules as when defining a type.
remove attribute NAME	The named attribute is removed from the type’s list of explicit attributes.
remove form NAME	The named form is removed from the type.
remove method PROG_NAME	The named method is removed from the type.
remove trigger EVENT_TYPE PROG_TYPE	The named trigger program is removed from the type. This clause must obey the same construction and syntax rules as when defining a type.

Modify Type Clause	Specifies that…
`derived TYPE_NAME`	The type being modified inherits attributes, methods, triggers, governing policies, and allowed relationships of the type named.
`abstract true`	Business object instances of this type cannot be created.
`abstract false`	Business object instances of this type can be created.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.
`history STRING`	Adds a history record marked “custom” to the type that is being modified. The `STRING` argument is a free-text string that allows you to enter some information describing the nature of the modification. See also *Adding History to Administrative Objects*.

Each modification clause is related to the arguments that define the type. For example, the following command changes the name and attribute list of the Shipping Form type:

```
modify type “Shipping Form” name “Shipping Label”
add attribute “Label Color”;
```

uniquekey Command

Description

The kernel does not enforce the Type Name Revision (TNR) constraint to ensure object uniqueness within a vault. Instead, a uniquekey can be optionally defined. The uniquekey is a predefined selection of IPLM dictionary attributes. If defined, it is enforced by the kernel and behaves as follows:

- Inheritance is mandatory
- Inherited association = N; direct association = N
- Unique-key driven (must be explicitly assigned to a type or relationship)
- Uniquekeys are unique only for a single vault.

Starting in V6R2014, the implementation of uniquekeys has been changed to create a cyrptographic hash and store the hash value in the database. For more information, see the *Installation Guide : Upgrading to V6R2014* in the online documentation.

When creating objects in Matrix Navigator, if a unique key exists in the database, errors may occur when creating objects. The error occurs after you create an object using all default attribute values and then try to create another object.

User level

Business Administrator

Syntax

The Uniquekey command has the following syntax:

```
[add|copy|modify|delete|enable|disable] uniquekey {CLAUSE};
```

Clauses provide additional information about the uniquekey.

Add Uniquekey

The Add Uniquekey command has the following syntax:

```
add uniquekey NAME |type TYPE_NAME         | [ADD_ITEM {ADD_ITEM}];
                   |relationship REL_NAME|
```

- ADD_ITEM provides additional information about the uniquekey being created:

```
icon FILENAME

description VALUE

[!|not]global

[interface INTERFACE_NAME]

attribute NAME {,NAME}
```

```
field FIELD_NAME [size SIZE]

[!|not]hidden

property NAME [value STRING]

property NAME to ADMIN [value STRING]
```

where ADMIN is one of:

```
|TYPE NAME         |

|RELATIONSHIP NAME|
```

Restricting the Index Size for SQL Server

SQL Server restricts the size of the index to 900 bytes. For example, if you try to create two string attributes with length 255, since they are nvarchar columns, the size of the index for string attributes is calculated as 255*2 (for Unicode), which would go over the limit. You can use the size modifier with the Add Uniquekey command to restrict the size so that indexing is right.

Copy Uniquekey

The Copy Uniquekey command has the following syntax:

```
copy uniquekey SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

- MOD_ITEM provides additional information about the uniquekey being copied:

```
name NAME

description VALUE

icon FILENAME

[!|not]hidden

add |attribute NAME                          |
    |field FIELD_NAME [size SIZE]            |
    |property NAME [to ADMIN] [value STRING]|

remove |attribute NAME          |
       |field FIELD_NAME        |
       |property NAME [to ADMIN]|

modify |field FIELD_NAME [size SIZE]

property NAME [to ADMIN] [value STRING]|
```

Modify Uniquekey

The Modify Uniquekey command has the following syntax:

```
modify uniquekey NAME [MOD_ITEM {MOD_ITEM}];
```

See MOD_ITEM table above for information on arguments to the Modify Uniquekey command.

Delete|Enable|Disable Uniquekey

The Delete, Enable, and Disable Uniquekey commands have the following syntax:

```
[delete|enable|disable] uniquekey NAME;
```

upload Command

Description

The Upload and Download commands are available for use in programs to be executed by a Web client through a Server. They allow client-side files and directories to be manipulated by server-side programs (and vice versa). See *download Command* for information on the Download command.

Generally the download and upload commands would be used in conjunction with a wizard 'file text box' widget. See *Widget Subclause* for details.

User level

Business Administrator

Syntax

Use the `upload` command when a file resides on a client and must be processed by a server side program.

```
upload sourcefile SOURCE targetfile TARGET [[!|not]delete] [[!|not]overwrite];
```

`SOURCE` specifies the full path and name of the file to be uploaded from the client.

`TARGET` specifies the destination path and file name on the server, relative to `WORKSPACEPATH`. If the specified directories do not exist, Live Collaboration creates them. Refer to *Using ${} Macros* for more information on `WORKSPACEPATH`.

When `delete` is specified, the source file is deleted on the client after the upload. The default is FALSE.

When `overwrite` is specified, if `TARGET` file name already exists, the uploaded file replaces it. The default is TRUE.

Note that the defaults for the delete and overwrite options are different between upload and download in order to maintain consistency with the default behavior of checkin/checkout.

For example:

```
upload sourcefile /reports/report.doc targetfile report.doc delete;
```

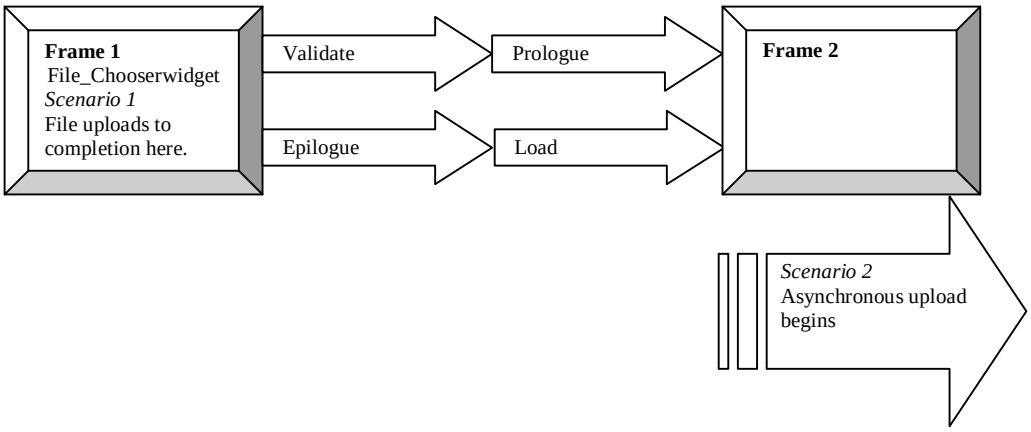
Upload Command vs. Widget Upload Option

The upload file widget option and the `upload` command are intended to solve two different problems:

- The file chooser with `upload` flag allows the user to upload one file before the server-side wizard programs are run. The file would generally contain data to be parsed and used by the wizard. For example, you could populate an object's attributes or even a wizard list box based on a file selected by the wizard user.
- The `upload` command allows the user to upload one or more files for server side processing. For example, you could preserve the client directory structure during file checkin so that a checkout on a different client would maintain the same relative directories (such as for use

with CAD files that generally contain an inherent directory hierarchy). This could be achieved by parsing the directory structure and list of files from a user selected text file (using a wizard file chooser with upload option), and then uploading the files for the server-side checkin.

When running a wizard, control passes between the applet running on the client (drawing frames, accepting user input) and the Live Collaboration kernel running on the server (executing wizard programs and their embedded MQL commands). The frame transition programs (load, validate, epilogue, prologue) run on the server, and control is not returned to the client until they have all been run. Since the upload/download commands are client-side tasks that need to be controlled from the client, they are not processed synchronously within the execution of the wizard program. Rather, they are processed when control next returns to the client; that is, after all the frame transition programs have run to completion. Consider the following scenarios, illustrated below:



- *Scenario 1*: File_Chooser widget has the `upload` flag set so, before the frame transition programs are run, the file selected on Frame 1 is uploaded and available to these programs.
- *Scenario 2*: File_Chooser widget does NOT have the `upload` flag set. Instead, the Epilogue has the command “mql upload …”. In this case, after ALL the frame transition programs are run, an asynchronous task, running in the background, is launched to upload the file selected on Frame1, so it is NOT guaranteed to be available for Frame 2.

Files uploaded to the server are automatically deleted when the session that put them there is finished, if they have not been deleted before that programmatically.

vault Command

Description

A *vault* is a grouping of similar objects within the database, as well as a storage location for metadata which identifies those objects.

The Business Administrator determines what the vault is for, while the System Administrator defines where the vault is located on the network. Vaults should use actual host and path names, not mounted directories. Paths must be exported on the host to all users who require access to the vaults.

For conceptual information on this command, see *Vaults* in Chapter 2.

User Level

Business Administrator

Syntax

```
[add|modify|clear|index|delete]vault NAME {CLAUSE};
```

- NAME is the name you assign to the vault. Vault names must be unique. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the vault.

Add Vault

Use the Add Vault command to define a vault:

```
add vault NAME [ADD_ITEM {ADD_ITEM}];
```

- ADD_ITEM provides more information about the vault you are creating. Although the clauses are not required to make the vault usable, they are used to define a vault location other than the current default host and path. In addition, the clauses can help users understand the purpose of the vault.

The Add Vault clauses are:

description VALUE
file MAPFILENAME
[!|not]hidden
indexspace SPACE
interface LIBRARYFULLPATH
map STRING
property NAME [to ADMINTYPE NAME] [value STRING]

```
server SERVERNAME
```

```
tablespace SPACE
```

Each clause and the arguments they use are discussed in the sections that follow.

File Clause

This clause is used to define the map or parameters filename for foreign or external vaults respectively.

The scott.map file is placed in the ENOVIA_INSTALL/api/mxff/ directory when the Studio Federation Toolkit is installed. The map file can be specified with the File clause. For example:

```
add vault scott
   interface d:\enoviaV6R2011\studio\api\mxff\mxff
   file d:\enoviaV6R2011\studio\api\mxff\scott.map;
```

Vault Types

There are three types of vaults: local, foreign and external. Most vaults are local. Foreign vaults are used with Adaplets™, which allow data from virtually any source to be modeled as objects. External (Web Service Adaplet) vaults are used with External stores that contain data maintained by external servers.

When defining a vault in MQL, you don’t need to specify which type it is and there is no clause that allows you to specify the type. The system knows which type of vault you are defining by the parameters you specify for the vault. For local vaults, you define Oracle tablespaces using the Tablespace and Indexspace clauses. For foreign vaults, you specify tablespaces, and Interface and Map fields (using the Interface and Map clauses). External vaults need only a parameters file. All these clauses are described in the following sections.

Tablespace Clause

This clause is used to specify the data and index storage areas in which the data tables for the vault are stored.

The names of the tablespaces and their associated storage are defined by the database administrator (DBA). This must be done prior to defining vaults. If you do not specify a tablespace name, the default data tablespace is used. Consult the database administrator about which tablespace you should use.

DB2 does not allow a separate index tablespace unless you specify an associated DMS tablespace. Therefore, if you specify an index tablespace for a vault when using DB2, you cannot use the default Data Table Space. You must specify the associated DMS data tablespace.

For SQL server, file groups are roughly the equivalent of the tablespaces used in other databases.

Refer to the *Installation Guide* for additional information.

Indexspace Clause

This clause is used to specify the Oracle tablespace in which the index and constraint information for the vault is stored.

The names of tablespaces and their associated storage are defined by the database administrator (DBA). This must be done prior to defining vaults. If you do not specify a tablespace name, the default index tablespace is used.

Interface Clause

This clause is used to define the full path name of the library for a foreign vault. The Interface should be specified as ENOVIA_INSTALL/api/mxff/mxff, with no extension. The .dll or shared library file is then used to access the vault, depending on whether the client is a Windows or a UNIX client.

Note that this field cannot be modified once the vault has been created. If changes are required, the vault must be deleted and then recreated. Deleting foreign vaults deletes only the database tables associated with it. The data from the foreign federation is left intact.

The interface can be used in more than one vault definition, but only by making a copy of it with a different name. For example, if the mxff interface is used to link a second database with Live Collaboration (presumably with a different mapping), a copy of mxff should be created and renamed, and then referenced in the second foreign vault definition.

Use of the same interface (of the same name) by more than one vault will cause problems when accessing business objects. Each Foreign vault must use a uniquely named interface. The mxff library can be copied and renamed for different vaults.

Map Clause

This clause is used to specify a string which defines the schema map for a foreign vault.

The schema map indicates which metadata in the foreign data maps to what types of data in Live Collaboration. It becomes part of the vault definition and is always referred to when accessing the data. The Map clause specifies the server, mode, and each table in the data source. Refer to the *Adaplet Programming Guide : Mapping the Data* for more information.

Param Clause

This clause is used to specify a string of XML to define the parameters for an external vault. You can alternatively use the File clause to indicate a file that provides the XML parameters.

Refer to the *Web Service Adaplet Programming Guide* for more information.

Modify Vault

After a vault is defined, you can change the definition with the Modify Vault command. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify vault NAME [MOD_ITEM {MOD_ITEM}];
```

`NAME` is the name of the vault you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Vault clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Vault Clause	Specifies that…
`description VALUE`	The current description, if any, changes to the value entered.
icon FILENAME	The image is changed tp the new image in the field specified.
`name NAME`	The current vault name changes to the new name entered.
`file MAPFILENAME`	The name of the map file is changed.
`map STRING`	The map schema information is replaced with the new string.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

Each modification clause is related to the clauses and arguments that define the vault. When modifying a vault, you first name the vault to change and then list the changes to make. For example, the following command assigns new name and description to the vault called The Cleveland Project.

```
modify vault “The Cleveland Project”
   name "The San Diego Project"
   description "Includes data for San Diego area."
   file c:\enoviaV6R2011\studio\scott.map;
```

Clear Vault

While maintaining vaults and their associated databases and tables, you may notice the following:

In Version 10.5, support for large file sizes was added, and so the LXSIZE column in the LXFILE_* table for each new vault is defined as type FLOAT, instead of type NUMBER as in previous releases. So new installations of Version 10.5 and up define the table columns as FLOAT, but databases upgraded from versions older than 10.5 will use the existing table columns (which can support large files, too) for upgraded vaults, but any vaults added with Version 10.5 or higher will define the table column as FLOAT.

The Clear Vault command is used to delete all business objects and associations in a vault:

```
clear vault NAME [minorrevision] [relationship] [full]
[size NUMBER];
```

NAME is the name of the vault you want to clear.

NUMBER is the size of the transaction before a database commit. The Size clause only applies when any of the other optional clauses are included.

Once a Vault has been established and used, you should NOT use the Clear Vault command. In addition, you should not use it when users are online.

Minorrevision Clause

Include this clause before actually clearing the vault to have first remove the business objects in the named vault from revision chains of objects in other vaults. This involves modifying rows in the vaults lxBO table (marking rows that need to be cleaned up). Upon successful execution of this command, all revision chains that need adjustment prior to the clear vault command are fixed.

Relation Clause

Include this clause before actually clearing the vault to first disconnect business objects in the named vault from objects in other vaults. This involves modifying rows in the vaults lxRO table (marking rows that need to be cleaned up). Upon successful execution of this command, all connections to objects in other vaults are removed.

Full Clause

Include this clause before actually clearing the vault to have first perform both a “clear vault relationship” and a “clear vault revision.” Additionally all tables of the vault are dropped and recreated, cleaning up the marked rows.

Size Clause

Include this clause of the clear vault command to indicate a transaction size before the relation or revision changes are committed to the database. For example, to remove all connections to or from objects in other vaults and commit the changes 100 at a time use the following:.

```
clear vault MyVault relation size 100;
```

When used without the Revision, Relation or Full clauses, the Size clause is ignored.

Index Vault

The Index command should be used periodically, after modifications and deletions, to clean up the database indices. .

```
index vault NAME  [table TABLE_NAME [indexspace TABLESPACE_NAME]];
```

NAME is the name of the vault to index. You can run the index vault command against any vault, including ADMINISTRATION.

Re-indexing vaults can improve find performance whether or not transaction boundaries have been used in the data loading process. If data is loaded from a sequentially sorted data file, the resulting index will be less than optimal. Re-indexing *randomizes* the index, making find performance noticeably better. Indexing a vault in this manner rebuilds the system indices that must be present for locating objects by name, type, and owner, as well as other SQL convertible fields.

To show the SQL commands for a particular index vault command, without actually changing the indices, use the `validate index vault command` as follows:.

```
validate index vault NAME [table TABLE_NAME [indexspace TABLESPACE_NAME]];
```

This is helpful to use on very large databases, where indexing a vault may take many hours. The validate output shows the SQL commands that need to be run. You could then manually run the commands in order, to make progress with minimal disruption.

Each clause is described below.

Table Clause

Include tthis clause to indicate which database tables should be re-indexed. Only those columns that have indexing defined will be re-indexed. You should include up to and including the "_" in a table name, since what follows is specific to the vault specified. For example:

```
index vault "Engineering-1" table lxbo_;
```

This command might generate and execute SQL similar to:

```
alter index lxBO_abbe6b7a_lxOid_Index rebuild;
alter index lxBO_abbe6b7a_lxName_Index rebuild;
alter index lxBO_abbe6b7a_lxOwner_Index rebuild;
alter index lxBO_abbe6b7a_lxPolicy_Index rebuild;
```

The Engineering-1 vault is associated with the `abbe6b7a` table.

Indexspace Clause

Use this clause to specify an alternate database tablespace to use for processing this command. For example:

```
index vault "Engineering-1" table lxbo_ indexspace
USER_DATA;
```

This SQL generated is as follows:

```
alter index lxBO_abbe6b7a_lxOid_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxName_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxOwner_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxPolicy_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxState_Index rebuild tablespace USER_DATA;
```

This command adds indices to all columns of lxBO_ table (of the vault Engineering-1) that have indexing defined, and the command would use tablespace USER_DATA to hold the index data. You must also include the table clause with using the indexspace clause.

Fixing Fragmented Vaults

As objects are deleted from a vault, storage gaps will occur in the vault database file. These gaps represent wasted disk space and can cause an increase in access time. MQL provides the `tidy vault` command to fix fragmentations in the database file of the vault.

```
tidy vault NAME [commit N];
```

- `NAME` is the name of the vault you want to fix. You can specify the ADMINISTRATION vault to remove unused records of deleted administration objects.
- Consolidates the fragmented database file. It deletes rows in the database tables that are marked for deletion.

Commit N Clause

Include this clause when tidying large vaults. The number N that follows specifies that the command should commit the database transaction after this many objects have been tidied. The default is 1000. For example:

```
tidy vault "Engineering" commit 200
```

The Commit N clause cannot be used for the ADMINISTRATION vault.

Updating Sets With Change Vault

When an object's vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data
- The original business object is deleted from the "old" vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object's vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless.

To address this issue, the change vault command includes the following additional functionality:

- Add the object clone from the new vault to all IconMail messages and user sets in which the original object was included.

The additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails. For this reason, the default functionality has not been changed, but business administrators can execute the following MQL command to enable/disable this functionality for system-wide use:

```
set system changelattice update set | on  | off |;
```

For example, to turn the command on for all users, use:

```
set system changelattice update set on;
```

Once this command has been run, when users change an object's vault through any application (that is, all products, 3DEXPERIENCE Platform), all IconMails that reference the object are fixed.

You can also fix IconMail at the time the change vault is performed (in MQL only). For example:

```
modify bus Assembly R123 A vault Engineering update set;
```

Printing a Vault Definition

The Print Vault command prints the vault definition to the screen allowing you to view it. When a Print command is entered, MQL displays the various clauses that make up the vault definition and their current values.

```
print vault NAME;
```

- If the `NAME` contains embedded spaces, use quotation marks.

- A print vault command lists only objects that have been updated (those objects for which there is a database table entry).

Delete Vault

If a vault is no longer required, you can delete it with the Delete Vault command:

```
delete vault NAME;
```

- `NAME` is the name of the vault to be deleted.
- Only empty vaults can be deleted. To remove all objects from a vault, use the `clear vault` command. See *Clear Vault*.

Searches the list of existing vaults. If the name is found and the vault contains no business objects, the vault is deleted. If the name is not found, an error message is displayed. If you attempt to delete a vault that contains business objects, an error message is displayed.

For example, to delete the vault named “1965 Bank Loans,” enter:

```
delete vault “1965 Bank Loans”;
```

After this command is processed, the vault is deleted and you receive an MQL prompt for another command.

view Command

Description

Views are not supported by web applications and should no longer be used.

Using the Matrix Navigator, customized Views offer a flexible and convenient way to *package* frequently used sets of Visuals (filters, cues, tips, toolsets and tables). For example, for each new revision of a product component, you might want to:

- Identify the latest change made with a red arrow cue
- List who performs work on it and who owns it with a quick object tip
- Filter out all the old revisions
- Always use the BOM Details table in the Details browser
- Run a program to create a new revision on command (with a tool button)

In Matrix Navigator you can use View Manager to make your filter, cue, tip, toolset and table selections *only once*, name the *View* and save it. The next time you needed to create a new product revision, you select the object, select the named View and do your work.

In Matrix Navigator, these Views are listed in the View menu by the names you give them, for you to use over and over again on any object/s you select. In MQL, you can use the List View command to see all the available views.

Views can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. You cannot create a named View until you have several filters, cues, tips, toolsets or tables available from which to choose. Once the views are defined, individual users can turn them on and off from Matrix Navigator browsers as needed.

From Matrix Navigator, views that customize or standardize a display can be very useful. Each user can create her/his own views from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar views consistently, it may be easier to create an MQL program, and have each user run the program.

In the Matrix Navigator, the list of named views displays in the View Manager and in the in the View menu.

It is important to note that views are all Personal Settings that are available and activated only when context is set to the person who defined them.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]toolset NAME {CLAUSE};
```

- `NAME` is the name of the toolset you are defining. The toolset name cannot include asterisks.
- CLAUSEs provide additional information about the toolset.

Add View

To define a new view from within MQL, use the Add View command:

```
add view NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the view you are defining. View names cannot include asterisks.

USER_NAME can be included with the user keyword if you are a business administrator with person access defining a view for another user. If not specified, the dataobject is part of the current user's workspace.

ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the view, you cannot have the same name for two views. If you use the name again, an error message will result. However, several different users could use the same name for different views. (Remember that views are local to the context of individual users.)

After assigning a view name, the next step is to specify which visuals or other conditions should be include in the view (ADD_ITEM) when the view is turned on (activated). The following are Add View clauses:

filter NAME
cue NAME
tip NAME
toolset NAME
table NAME
[!|in|notin|not]active MEMBER_TYPE NAME {,NAME}
[!|not]hidden
visible USER_NAME{,USER_NAME};
property NAME on ADMIN [to ADMIN] [value STRING]

The filter, cue, tip, toolset and table clauses require only the accurate name(s) of each one to be entered.

Copy View

After a view is defined, you can clone the definition with the Copy View command.

If you are a business administrator with person access, you can copy views to and from any person's workspace (likewise for groups and roles). Other users can copy visible views to their own workspaces.

Copying a View from one user to another is more complicated than for other Visuals because Views have members that need to be copied as well.

This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy view SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}]
[changename VISUAL OLD_NAME NEW_NAME] [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the view definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

COPY_ITEM can be:

fromuser USERNAME	USERNAME is the name of a person, group, role or association.
touser USERNAME	
overwrite	Replaces any view of the same name belonging to the user specified in the touser clause.

VISUAL can be any one of the following: filter, tip, cue, toolset, table.

OLD_NAME is the current name of the member visual.

NEW_NAME is the name that the member visual will have after the copy command is executed.

The order of the fromuser, touser, overwrite, *and* changename *clauses is irrelevant, but* MOD_ITEMS, *if included, must come last.*

MOD_ITEMs are modifications that you can make to the new definition. Refer to the table in *Modify View* for a complete list of possible modifications.

When a View is copied, all its members are copied also. If a member of the same name exists in the target user, the command aborts, unless overwrite or changename is specified. If the command aborts, the database is left unchanged.

To avoid naming conflicts, you can specify a change of name when a member is copied. This is done with the changename keyword. For example:

```
copy view View1 ViewTable fromuser purcell touser Engineer
changename filter Bugs PurcellBugs changename tip
RelationName PurcellRelationName;
```

where View1 is a View belonging to person purcell containing a filter named Bugs and a tip named RelationName. This command will create a View named ViewTable in role Engineer. The filter Bugs will be copied to a filter named PurcellBugs. The tip RelationName will be copied to a tip named PurcellRelationName.

All other members of the View, if any, will be copied to the Role with their names unchanged. If a View with that name already exists or if a Filter of that name exists, etc., the command will abort and the database will be left unchanged.

Modify View

You can modify any view that you own, and copy any view to your own workspace that exists in any user definition to which you belong or that is defined as visible to you. As an alternative to copying definitions, Business Administrators can change their workspace to that of another user to work with views that they do not own. See *Working with Workspace Objects* in Chapter 6 for details.

Use the Modify View command to add or remove defining clauses and change the value of clause arguments:

```
modify view NAME [user USER_NAME] [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the view you want to modify. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.

MOD_ITEM is the type of modification to make. Each is specified in a Modify View clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify View Clause	Specifies that…
`add filter NAME {,NAME}`	The named filter is added.
`remove filter NAME {,NAME}`	The named filter is removed.
`add cue NAME {,NAME}`	The named cue is added.
`remove cue NAME {,NAME}`	The named cue is removed.
`add tip NAME {,NAME}`	The named tip is added.
`remove tip NAME {,NAME}`	The named tip is removed.
`add toolset NAME {,NAME}`	The named toolset is added.
`remove toolset NAME {,NAME}`	The named toolset is removed.
`add table NAME {,NAME}`	The named table is added.
`remove table NAME {,NAME}`	The named table is removed.
`active MEMBER_TYPE NAME {,NAME}`	The named MEMBER_TYPE (filter, cue, tip, toolset, table) is made active.
`inactive MEMBER_TYPE NAME {,NAME}`	The named MEMBER_TYPE (filter, cue, tip, toolset, table) is made inactive.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`visible USER_NAME{,USER_NAME};`	The object is made visible to the other users listed.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the view.

Although the Modify View command allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the view name or remove the view entirely, you must use the Delete View command and/or create a new view.

Delete View

If a view is no longer needed, you can delete it using the Delete View command:

```
delete view NAME;
```

`NAME` is the name of the view to be deleted. If you are a business administrator with person access, you can include the user clause to indicate another user's workspace object.

Searches the local list of existing views. If the name is found, that view is deleted. If the name is not found, an error message results.

When a view is deleted, there is no effect on the business objects or on queries. Views are local only to the user's context and are not visible to other users.

webreport Command

Description

A *webreport* is a workspace object that can be created and modified in MQL or using the WebReport class in the Studio Customization Toolkit. Webreports are used to obtain a set of statistics about a collection of business objects or connections.

For conceptual information on this command, see *Webreports* in Chapter 7.

User Level

Business Administrator

Syntax

```
[add|copy|modify|evaluate|temporary|delete]webreport NAME
{CLAUSE};
```

- NAME is the name of the webreport you are defining. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object. (By default the webreport is added to the current workspace.)
- CLAUSEs provide additional information about the webreport.

Add Webreport

To define a webreport from within MQL use the add webreport command:

```
add webreport NAME [user USER_NAME][ADD_ITEM {ADD_ITEM}];
```

- NAME is the name of the webreport you are defining. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object. (By default the webreport is added to the current workspace.)
- ADD_ITEM is an add webreport clause that provides additional information about the webreport. The add webreport clauses are:

description STRING_VALUE
appliesto | businessobject | | relationship | | both |
searchcriteria STRING
notes STRING
reporttype STRING
[!|not] checkaccess

```
groupby EXPR_ITEM
data EXPR_ITEM
summary | NAME [APPLIESTO] SUM_ITEM {SUM_ITEM} |
        | all  [APPLIESTO]                     |
visible USER_NAME{,USER_NAME}
[!|not]hidden
property NAME [to ADMIN] [value STRING]
```

Appliesto Clause

Include this clause to indicate if the webreport collects statistics on business objects, relationships, or both. The default is business object. For example:

```
add webreport UsedParts appliesto both;
```

The ability to report on relationships as well as or instead of business objects is applicable only when SEARCHCRITERIA is an expand. (When SEARCHCRITERIA is not an expand, no relationships are returned, so there are none to evaluate against.) In an expand, the information desired may be some summary of data from the relationships found and not the end objects. Or it could be you need more than one statistic and that one is properly obtained from the objects and the other from the relationships.

In addition to an overall `appliesto` flag for the webreport, a webreport's groupbys, datas, and summaries may use this flag. Their flags are only relevant (and looked up) if the `appliesto` flag for the entire webreport has the value "both".

Searchcriteria Clause

`SEARCHCRITERIA` is defined recursively as one of:

```
set NAME
query NAME
temp set OBJECTID [{,OBJECTID}]
temp query businessobject TYPE NAME REVISION
[TEMP_QUERY_ITEM {TEMP_QUERY_ITEM }]
expand [EXPAND_ITEM {EXPAND_ITEM}]
[( {( }]SEARCHCRITERIA [ ){ )}]
```

`TEMP_QUERY_ITEM` is one of:

```
owner NAME
vault NAME
```

[!|not]expandtype
where WHERE_CLAUSE
limit NUMBER
over SEARCHCRITERIA
querytrigger

OBJECTID is the Type Name and Revision of the business object.

The values of TEMP_QUERY_ITEM have the same meaning in this context as they have for the temp query command, as described in *query Command*.

EXPAND_ITEM is one of:

from
to
type PATTERN {,PATTERN}
relationship PATTERN {,PATTERN}
fixed points ID [,Id]
withroots
filter PATTERN
activefilters
reversefilters
view NAME
recurse [to | N |]
recurse to |all|
[into|onto] set NAME
SELECT_BO
where WHERE_CLAUSE
SELECT_REL
DUMP

The values of EXPAND_ITEM have the same meaning in this context as they have for the expand bus command, as described in *businessobject Command*, with a few exceptions:

- In the case of “select,” it indicates whether a subsequent Where clause applies to business objects or to relationships. In expand bus select has an additional meaning that is not applicable here.

- Leaf can be used with recurse in a SEARCHCRITERIA expand to specify that only “leaf” nodes of the expand should be returned. What counts as a leaf node differs depending on whether “recurse to all” or “recurse to n” is issued. In the “all” case, a leaf node is one that has no further connected objects that satisfy the criteria. With a recursion level N indicated, the leaf nodes consist of the same objects as recurse to all, PLUS all objects on level N.

  Leaf is not allowed in the expand bus command, but is allowed here.

An “expand” searchcriteria does not make sure that returned objects are unique. That is, any objects that are connected more than once to the object(s) being expanded (or recursed to) will be listed more than once. This affects webreports, eval expression, and other commands using searchcriteria. For example, the following might return a list containing duplicate entries:

```
MQL<18>eval expres "count TRUE" on ( expand bus t2 t2-1 0 );
```

To avoid the duplication, you could change the searchcriteria to the following:

```
temp query * * * over expand bus t2 t2-1 0
```

SEARCHCRITERIAs can be linked with binary operators, which follow simple, intuitive rules:

- and—an object is in both collections
- or—an object is in one or the other collection
- less—an object is in the result if it is in the left-hand collection but not the right-hand one.

If there is more than one binary operator in a `SEARCHCRITERIA`, parentheses must be used to disambiguate the clause. (There is no implied order of operations.) You must include a space before and after each parenthesis. In addition, the number of left and right parentheses must match each other. For example:

```
( set A + set B ) - set C
```

would probably evaluate differently than:

```
set A + ( set B - set C )
```

Notes Clause

This clause is used to store a string of any length. It is designed for use by applications to store any information they need to maintain with the webreport. (Of course, properties could be used, but they are limited to no more than 255 characters.) For example, the Business Metrics application stores data in the notes field that is required to update the JSP page used to create a webreport with values used to evaluate the webreport.

Reporttype Clause

This clause can be used to store a string up to 255 characters. The Business Metrics application uses this field to identify a webreport as being one of 4 types of webreports:

- Object Count
- Object Count in State Over Time
- Object Count Over Time
- LifeCycle Duration Over Time.

NotCheckaccess Clause

This clause indicates if show and read access should be checked when the webreport is evaluated; that is, when the data expressions are evaluated. There are two reasons you may want to turn off these accesses:

- Checking access can be a big performance penalty.
- Turning off the access checking allows the counting and summarizing of data on objects that can't otherwise be seen or read; that is, objects to which you don't have show or read access are included in the statistics returned.

The default is that read and show access are checked (for show, that's assuming show access is on).

This option does not control what happens when the business objects or connections found by a webreport result are printed, which can be done with options in "evaluate webreport" and "temp webreport", as well as through a Select clause in "print webreport". In that case, show and read access will apply as they normally do regardless of this setting.

Groupby Clause

This clause is used to define a hierarchy of the data in the results of a webreport. A groupby has one expression, which is either a string or a pointer to an expression object. A webreport can have any number of groupbys.

The expression used with groupby should be an expression that applies to a *single business object or connection*. All of the groupby expressions will be evaluated against each object found by the search criteria. The values of these evaluated expressions will be used to organize the objects into a number of subsets. There will be one subset for each unique combination of groupby values after evaluating them all across all the objects.

For example, suppose the attribute Priority can take on values 0,1,2,3 and the attribute Material can take on values Metal, Wood, Rubber, Plastic, Other. Then, if the webreport includes these groupby expressions:

```
groupby value 'attribute[Priority]'
groupby value 'attribute[Material]'
data value 'average attribute[Cost of Rework]'
```

it could result in as many as 20 distinct subsets, corresponding to the 20 unique combinations of (Priority, Material), and it would report the average Cost of Rework for the objects in each of those ~20 subsets. The term 'cell' is used to refer to these subsets in the webreport selectables described below.

The `Maxgroupings N` clause limits the number of subsets created for distinct values of this groupby expression to be no greater than N. It will keep the subsets which have the largest number of associated objects, and will create an additional 'other' subset of the remaining objects.

The objects returned by the search criteria are grouped by the values of the first groupby expression, then sub-grouped by values of the second groupby expression, and so on to separate the objects into their appropriate cells. Each groupby can have a label associated with it, and the labels need not be unique. The groupby syntax is one of:

```
groupby [APPLIESTO] object EXPR_NAME [label STRING]
[maxgroupings N];
```

```
groupby [APPLIESTO] value STRING [label STRING]
[maxgroupings N];
```

- Where APPLIESTO is a valid Appliesto Clause. The appliesto setting of the groupby is only looked at if the Appliesto clause on the entire webreport is "both."

For example:

```
add webreport UsedParts groupby value current label Status
groupby object WhereUsedExpression label "Used In";
```

Data Clause

This clause has the same syntax as the Groupby clause:

```
data [APPLIESTO] object EXPR_NAME [label STRING];
```

```
data [APPLIESTO] value STRING [label STRING];
```

- Where APPLIESTO is a valid Appliesto Clause. The appliesto setting of the data is only looked at if the Appliesto clause on the entire webreport is "both."
- However, the expressions used with data clauses are those appropriate for evaluation on *a collection of business objects or connections* (called "collection expressions"), such as averages, sums, standard deviations, etc. Refer to Working with Expressions chapter for details.
- Each data clause has one expression, which is either a string or a pointer to an expression object. A webreport can be defined with any number of data clauses.
- Each data expression is evaluated against each of the subsets determined by the search criteria together with the groupbys. To evaluate the data expressions against combinations of these subsets, such as the entire collection or with some groupby ignored, summaries should be created. See the *Summary Clause* section for more information.

Each data can have a label associated with it, and the labels need not be unique.

For example:

```
data value dateperiod ym (maximum (current.actual));
```

Summary Clause

A webreport can include a list of uniquely named summaries. A summary specifies a subset of the datas and groupbys of the webreport, which allows you to collapse groupings and report on only a subset of the datas. Use the `Summary` clause of the `add webreport` command to specify the summaries. The syntax is:

```
 summary | NAME [APPLIESTO] SUM_ITEM {SUM_ITEM} |
         | all  [APPLIESTO]                     |
```

- APPLIESTO is a valid *Appliesto Clause*. The appliesto setting of the summary is only looked at if the Appliesto clause on the entire webreport is "both."
- Where SUM_ITEM is one of:

```
groupby INDEX{,INDEX}
```

```
data | INDEX{,INDEX} |
     | all           |
```

- INDEX specifies one of the webreport's groupbys or datas indicated by order added to the summary. The first groupby (and data) added has an index of 1, with the last one added having an index of the total number of groupbys (or datas).
- If "summary all" is specified, a summary is created for every possible subset of groupbys and each summary will have all data expressions. You cannot use "summary all" when the number of groupbys is more than 8.
- You can indicate "data all" when you list a subset of groupbys.

For example, using the example above with groupbys on Priority and Material, you could specify a Summary report with ignores the Material groupby to report the average Cost of Rework by sorting by Priority alone:

```
summary groupby 1 data 1
```

The 1's refer to the first groupby and first data expressions defined for this webreport.

Copy Webreport

If you are a Business Administrator with person access, you can copy webreports in any person's workspace (likewise for groups and roles). Other users can copy visible webreports to their own workspaces.

After a webreport is defined, you can clone the definition with the Copy Webreport command. Cloning a webreport definition requires Business Administrator privileges, except that you can copy a webreport definition to your own context from a group, role or association in which you are defined.

This command lets you duplicate webreport definitions with the option to change the value of clause arguments:

```
copy webreport SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

- SRC_NAME is the name of the webreport definition (source) to be copied.
- DST_NAME is the name of the new definition (destination).
- COPY_ITEM can be:

fromuser USERNAME	USERNAME is the name of a person, group, role or association.
touser USERNAME	
overwrite	Replaces any webreport of the same name belonging to the user specified in the Touser clause.

The order of the fromuser, touser *and* overwrite *clauses is irrelevant, but* MOD_ITEMS, *if included, must come last.*

- MOD_ITEMs are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Webreport

If you are a Business Administrator with person access, you can modify webreports in any person's workspace (likewise for groups and roles). Other users can modify only their own workspace webreports.

You must be a business administrator to modify a webreport owned by a group or role.

Use the Modify webreport command to add or remove defining clauses and change the value of clause arguments:

```
 modify webreport NAME [user USERNAME][MOD_ITEM
{MOD_ITEM}];
```

- `NAME` is the name of the webreport you want to modify. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.
- `MOD_ITEM` is the type of the modification you want to make. Each is specified in a Modify webreport clause, as listed in the following command. Note that you need specify only the fields to be modified.

Modify Webreport Clause	Specifies that…
`name NEW_NAME`	The current webreport name is changed to the new name entered.
`description VALUE`	The current description value, if any, is set to the value entered.
`APPLIESTO`	The webreport applies to the new setting - business objects, relationships or both.
`searchcriteria STRING`	The search criteria is changed to STRING.
`notes STRING`	The notes are changed to STRING.
`reporttype STRING`	The report type is changed to STRING.
`[!|not]checkaccess`	The checkaccess setting is changed as indicated.

`summary NAME MOD_SUM_ITEM {MOD_SUM_ITEM}`
`where MOD_SUM_ITEM is:`
`| APPLIESTO |`
`| add | groupby | INDEX |`
`| | data | |`
`| remove | groupby | INDEX |`
`| | data | |`
	The named summary is modified as specified.

`add ADD_SUMMARY`	The new summary is added as specified.
`remove summary NAME`	The existing summary is removed.
`copy result to archive`	The existing evaluation result is saved in an archive.
`copy archive INDEX to result`	The specified archive is copied to the current evaluation result, thereby overwriting any existing one.

Modify Webreport Clause	Specifies that…
result ARCHIVE_ITEM {ARCHIVE_ITEM} where ARCHIVE_ITEM is: | label NAME | | description VALUE | | remove objects | | [un]lock | | [!|not]hidden |	The current evaluation result is modified as indicated.
archive INDEX ARCHIVE_ITEM {ARCHIVE_ITEM}	The specified archive is modified as indicated.
remove result	Deletes the existing results.
remove archive INDEX	Deletes the specified archive results.
remove archive all	Deletes all archive results.
remove archive label PATTERN	Deletes all archive results that fit the label PATTERN.
append | groupby EXPR_ITEM | | data EXPR_ITEM |	New groupby or data is added to end of list (index is total number of groupbys or datas). EXPR_ITEM is a valid definition of the groupby or data.
add groupby INDEX EXPR_ITEM	New groupby is added with index as specified. EXPR_ITEM is a valid definition of the groupby.
remove groupby INDEX{,INDEX}	Groupbys listed by index are removed.
add data INDEX EXPR_ITEM	New data is added with index as specified. EXPR_ITEM is a valid definition of the data.
remove data INDEX{,INDEX}	Datas listed by index are removed.
add visible USER_NAME{,USER_NAME}	Named users are added to visible list so they can use the webreport.
remove visible USER_NAME{,USER_NAME}	Named users are removed from visible list so they can no longer use the webreport.
[!|not]hidden	The webreport hidden setting is set as specified.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the webreport. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

Evaluate Webreport

Use the evaluate webreport command in MQL to generate results:

```
evaluate webreport NAME [user USER_NAME][EVAL_ITEM
{EVAL_ITEM}] [OUTPUT_ITEM {OUTPUT_ITEM}];
```

NAME is the name of the webreport you are evaluating. You can include the user USER_NAME clause to clarify to which workspace the webreport belongs. (By default USER_NAME is assumed to be the current context user.)

You can evaluate the webreport as defined, or you can specify EVAL_ITEMs to override settings in the defined webreport. The EVAL_ITEMS are:

no count
data INDEX{,INDEX}
summary NAME{,NAME}
no data
no summary
objects [select SELECT_ITEM{,SELECT_ITEM}]

OUTPUT_ITEMs can be included to specify how to handle the results. The clauses include:

[unlock] save [SAVE_ITEM {SAVE_ITEM}] where SAVE_ITEM is: | label NAME | | description VALUE | | archive | | objects | | lock |
xml
output FILENAME
separator STRING
groupbyseparator STRING

Each clause is described in the sections that follow.

No Count Clause

By default, a count of the objects that meet the search criteria is included when the webreport is evaluated. Include no count to remove it from the webreport's results.

[No] Data Clause

Include the Data clause in the evaluate webreport command to include only the results of the webreports datas specified with INDEX. Include no data to remove all datas from the webreport's results.

[No] Summary Clause

Include the Summary clause in the evaluate webreport command to include only the webreport summary specified with NAME. Include no summary to remove all summaries from the webreport's results.

Objects Clause

Include the Objects clause in the evaluate webreport command to return the list of business object ids that meet the search criteria, in addition to the webreport results. Include Select clauses in brackets to return this information about the objects found.

Save Clause

Include the Save clause to save the results in the database. You can include any of the archive subclauses, including label, description, archive, and objects. If save is issued without archive, the results are saved as "the result."

If you use archive, the computed results are stored as an archive, rather than as the result. The difference between the result and an archive is in how you read or manipulate them.

You can also lock or unlock the results. When the result is locked it cannot be overwritten without unlocking it.

xml Clause

Include the xml clause so that the output is formatted as xml.

Separator Clause

Include the Separator clause to specify how fields are separated in the results. The separator is used to separate a list of groupby values from the count and data values in a given line of output.

Groupbyseparator Clause

Include the Groubyseparator clause to specify how groupby output is separated in the webreport results. The groupby separator is used to separate groupby values.

Temporary Webreport

You can use the temp webreport command to define and evaluate webreport in 1 step:

```
temp webreport [TEMP_ITEM {TEMP_ITEM}];
```

The TEMP_ITEMs are:

searchcriteria STRING
[!|not]checkaccess
groupby EXPR_ITEM
data EXPR_ITEM
ADD_SUMMARY
count
objects [select SELECT_ITEM{,SELECT_ITEM}]
separator STRING
groupbyseparator STRING
xml
output FILENAME

Delete Webreport

If you are a Business Administrator with person access, you can delete webreports in any person's workspace (likewise for groups and roles). Other users can delete only their own workspace webreports.

You must be a business administrator with group or role access to delete a webreport owned by a group or role.

If a webreport is no longer required, you can delete it using the delete webreport command

```
delete webreport NAME [user USERNAME];
```

NAME is the name of the webreport to be deleted. If you are a business administrator with person access, you can include the User clause to indicate another user's workspace object.

Searches the list of defined webreports. If the name is found, that webreport is deleted. If the name is not found, an error message is displayed. For example, to delete the webreport named "ProjectStatus" enter the following:

```
delete webreport "ProjectStatus";
```

After this command is processed, the webreport is deleted and you receive an MQL prompt for another command.

Example

This section shows how some of reports would be represented using the functionality described above.

Lifecycle Duration Over Time Report

This example will find all objects with type=Nut* which are currently in the Release state, and last entered the Release state between Jan 2002 and April 2005. Notice that single-quotes are used to enclose the entire search criteria, and double-quotes are used to enclose the Where clause that is part of the search criteria.

It will group these objects by week according to the date they entered the Release state. Notice the use of 'yw' in the dateperiod expression. This insures that the weeks (1-53) in different years be grouped separately.

For each such subset, it will report on the average duration for each of the states in the policy:

```
add webreport MyTclReport notcheckaccess
       searchcriteria 'temp query bus Nut* * * where
   "policy ~~ EC* && current == Release
                            && current.actual >= 01/01/2002
                            && current.actual <= 04/01/2005"'
groupby value "dateperiod yw current.actual"
data value "average state[Preliminary].duration"
     data value "average state[Review].duration"
     data value "average state[Approved].duration"
     data value "average state[Release].duration"
     data value "average state[Obsolete].duration"
```

Object Count Report

This example will find all objects with type=Nut*. It will group them by their current state and by the value of the attribute 'Part Classification'. With life-cycle states as in the previous example, and with values for 'Part Classification' such as Molded, Machined, Extrusion,..., these groupbys will result in subsets such as "Molded parts in state Review", "Extrusions in state Release", "Machined parts in state Obsolete", and so on.

The data reported is a simple count of the objects in each subset.

```
add webreport MyTclReport notcheckaccess
          searchcriteria 'temp query bus Nut* * *'
        groupby value current
        groupby value attribute[Part Classification]
        data value "count TRUE"
```

Object Count in State Over Time Report

This example again finds all objects with type=Nut* which are currently in the Release state, and last entered the Release state between Jan 2002 and April 2005. It groups them into the weeks in which they entered their current state, and reports a simple count of the objects in each dateperiod.

```
add webreport MyTclReport notcheckaccess
          searchcriteria 'temp query bus Nut* * * where
                "policy ~~ EC* && current == Release
                            && current.actual >= 01/01/2002
                               && current.actual <= 04/01/
2005"'
          groupby value "dateperiod yw current.actual"
          data value 'count TRUE'
```

If we wanted further information on the costs of the parts released in these time periods, we could add additional data to be reported:

```
data value 'count (attribute[Actual Cost] >= 100)'
data value 'count (attribute[Actual Cost] > 10 &&
attribute[Actual Cost] < 100)'
data value 'count (attribute[Actual Cost] <= 10)'
```

The results with these additional Data clauses would look like this:

```
Business Objects
      200205=2|2|1|0|1|
      200208=50|50|10|22|18|
      200307=51|51|15|27|9|
      200324=60|60|20|20|20|
      200333=61|61|21|20|20|
      200334=53|53|24|26|3|
      200335=120|120|45|53|22|
      200409=60|60|15|15|30|
      200410=53|53|6|27|20|
      200438=61|61|21|14|26|
```

Webreport evaluation presents the groupby values whose unique combinations define each cell in the webreport result delimited by the groupbyseparator character '|', followed by the separator character (=) and the count of objects for the cell. After the count for this cell, the data expressions are listed, delimited again by the groupbyseparator character '|'.

In this example, the first data expression is 'count TRUE', which is going to repeat the object count for each cell.

So, the 2nd row (starting with 200208) tells us that in the 8th week of 2002, 50 parts were released. Of those 50 parts, there were 10 costing more than 100, 22 costing between 10 and 100, and 18 costing less than 10.

Object Count Over Time Report

This example will find all objects with type=Nut*. It will group them by the week in which they were originated AND the value of the 'Part Classification' attribute. It will report a count of the objects in each subset.

```
add webreport MyTclReport notcheckaccess
          searchcriteria 'temp query bus Nut* * *'
          groupby value "dateperiod yw originated"
          groupby value attribute[Part Classification]
          data value 'count TRUE';
```

A similar webreport could report on the average cost of parts released during the months of last year. This is defined as a temp webreport:

```
temp webreport notcheckaccess
          searchcriteria 'temp query bus Nut* * * where
                "policy ~~ EC* && current == Release
                            && current.actual >= 01/01/2004
                             && current.actual < 1/1/2005"'
          groupby value "dateperiod ym
state[Release].actual"
          data value 'average attribute[Actual Cost]';
```

wizard Command

Description

When you create a Business Wizard, you choose the number of frames and define the contents of each frame, customizing the wizard for a specific purpose relating to your database.

A *wizard* is a program which asks the user questions and executes a task based on the information received. It consists of a one or more dialog boxes, called *frames*, which are presented sequentially to the user. Generally, each frame provides some explanation or asks a question, then requires that the user either type a response or make a choice from a list. When all information has been collected, the wizard program performs an action based on the information.

Business Wizards allow you to create a user interface to simplify repetitive tasks performed by users.

Suppose, for example, that a number of users are required to check reports into the database on a weekly basis. A wizard could ensure that reports are named according to a standard report naming convention, prevent the accidental replacement of existing files, and track which users have submitted a report, even sending IconMail or email to the manager who reads the reports to advise that the reports have been checked in.

A wizard is a type of Program object. This means that a wizard can be launched most of the places that a Program can be launched, for example, stand-alone, or as a method. However, due to the graphical nature of wizards, they cannot be executed using MQL.

Wizards are composed of *frames*. Each frame contains one or more *widgets*. Each term is explained in detail below.

Frames

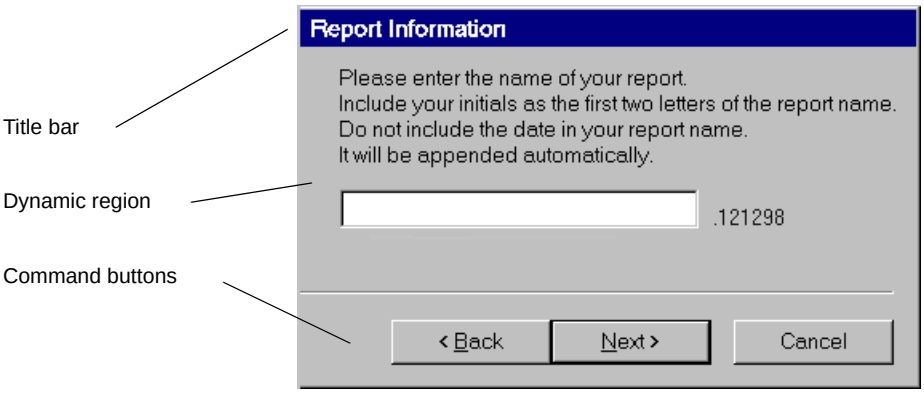
A *frame* is a dialog box that contains instructions and/or asks for information from the user. The information gathered in one frame can be checked for correctness before moving to the next frame and will often dictate the choices loaded into the next frame.

Frames are designed by the Business Administrator using a layout editor. The basic layout of a frame consists of a fixed title in the title bar at the top, a fixed set of three active command buttons along the bottom, and a dynamic region (sub-frame) in the middle. Only the dynamic region can change from one frame to the next.

In order to be effective, a wizard should gather information over a series of simple steps with each step responsible for a single piece of information. This allows for focused activity during each frame and eliminates the need for large complex dialogs.

The frame-specific dynamic region typically has a multiple-line text box near the top that describes the step being performed. The rest of the region is set up to gather the information needed for the step to be completed. The region is called “dynamic” because choices provided in the region can be filled during run time.

The following shows a frame whose dynamic region contains a multiple-line textbox which explains the format for the name of a report. It also provides an input field where the user can type the report name:



For most frames, three command buttons are available: **Back**, **Next**, and **Cancel**. After providing information or making a selection, the user clicks the **Next** button to proceed to the next frame. Clicking the **Back** button returns to the previous frame, where changes can be made to the information supplied. Clicking the **Cancel** button exits the wizard. The first frame of any wizard has a deactivated (grayed-out) **Back** button. The final frame of the sequence has a **Finish** button in the place of the **Next** button. When a user selects the **Finish** button in the final frame, the task is carried out by executing the code associated with the Business Wizard. A single optional status frame can be used to return feedback to the user about the task just performed (or provide the reason for the task not being performed).

Widgets

Anything that is included in the dynamic region of a frame is called a *widget*. The designer of the wizard chooses how many and what types of widgets will be included in each frame.

A frame can include any number of widgets, but it is most effective to limit their numbers in order to reduce the complexity of each frame.

There are ten different types of widgets. The following tables lists the widget types, which are explained in more detail in *Widget Subclause*.

Widget	Description
label	a non-editable text field
command button	a button which, when clicked, executes a program
text box	a text field, which can be defined to be either editable or non-editable
image	a picture file containing a GIF image
list box	a list of items from which a single or multiple selections can be made
combo box	a combination of an editable text box and a list box—users can type their choice or select from the list
radio button	a group of one or more mutually exclusive options, each with a circle beside it; only one can be selected
check box	a group of one or more options, each with a check box beside it; multiple options can be selected
directory text box	a text field with an ellipsis button next to it, which launches the directory chooser.
file text box	a text field with an ellipsis button next to it, which launches the file chooser.

Runtime Program Environment

Business Wizards require a mechanism for passing information between the Program objects that can be used within the wizard. The *Runtime Program Environment* (RPE) is provided to hold program environment variables. The RPE, which is a dynamic heap in memory that is used to hold name/value pairs of strings, makes it possible to pass parameters between internal programs and scripts.

For Business Wizards, the RPE allows information to be passed between frames, between programs that control the individual widgets, and to the wizard program that processes the information gathered from the user. For example, after all data-gathering frames have been displayed, you may want to present a summary frame where the user can check for errors. Also, if the user clicks the **Back** button, you would want the frames to display just as they did when the user clicked the **Next** button, including the choices the user made.

See *Runtime Program Environment* for additional information on the RPE.

Business Wizard Internal Programs

The basic functions of the Business Wizard require no specific programming since they are handled internally by the system. These basic functions include displaying frames and widgets, including responding to the Next, Back and Cancel buttons.

The Business Administrator customizes each Business Wizard by writing programs that control the individual widgets and that execute a task based on the information collected from the user. The programs that can be used to customize the Business Wizard include the following:

- **Prologue** – the program that executes just before a frame is displayed. See *Prologue Subclause*.
- **Epilogue** – the program that executes when a frame is closed. See *Epilogue Subclause*.
- **Load Program** – the program that loads values into the widget field. See *Load Subclause*.
- **Validate Program** – the program that tests the validity of the widget's state. See *Load Subclause*.
- **Button Program** – the program that controls what happens when a command button located in the dynamic area of a frame is selected.
- **Wizard Code** – the program that controls the task that is executed when the user clicks the Finish button in the Business Wizard. See *Code Clause*.

Note that all of the programs listed above (except for the actual wizard code) must be defined as independent program objects before they can be used by a Business Wizard. You can edit existing programs while defining a wizard but you cannot create a new program object while editing a wizard.

Business wizard internal programs should not launch external applications, such as a word processing program. Attempts to do so will cause unexpected results, including crashing the wizard.

Using ${} Macros

Macros provide a relatively easy way to work with variables. The main limitation to macros however is that they are available only for the program that calls them. When working with nested programs, the RPE enables you to pass values between programs. ${} macros are placed into the

RPE so that they can be used by nested programs. Just remember to drop the "${}" characters when referencing the macro variable in the RPE.

Refer to the *Configuration Guide : Macros* in the online documentation for more information.

Planning the Wizard

Before you start creating a Business Wizard, you should plan what the wizard will accomplish and how you want it to look.

- Decide which task(s) the Business Wizard will perform, for example, checking in a report.
- Determine what information is needed to perform the task, for example, user name, department name, report name, etc.
- Based on the amount of information that needs to be collected, decide how many frames are needed and how each frame should look.
- Decide what platform from which users will be running the Wizard. The wizard should be designed with the lowest screen resolution in mind and the Master Frame should be sized to fit the screen. Also, if the wizard will be executed from the Web, it is recommended that default fonts and colors are used. The `font.properties` file located in the browser's directory specifies what fonts are displayed, but this file is generally used for specifying fonts for alternate character sets for Asian languages. For more information on the font.properties file, and setting fonts on the web, see http://java.sun.com/j2se/1.4.2/docs/guide/intl/fontprop.html.

User level

Business Administrator

Syntax

```
[add|copy|modify|delete]wizard NAME {CLAUSE};
```

- `NAME` is the name of the wizard you are creating. All wizards must have a name assigned. When you create a wizard, you should assign a name that has meaning to both you and the user. For more information, see *Administrative Object Names*.
- CLAUSEs provide additional information about the business wizard.

Add Wizard

Creating a Business Wizard involves setting up basic Wizard parameters, defining frames and widgets (the components of frames), and specifying the underlying program code.

Wizards are defined using the Add Wizard command:

```
add wizard NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name of the wizard you are creating.

ADD_ITEM is an Add Wizard clause which provides additional information about the wizard. The Add Wizard clauses are:

code CODE
description VALUE
external|mql
file FILENAME
[!|not]needsbusinessobject
[!|not]downloadable
execute immediate|deferred
frame FRAME_NAME FRAME_ITEM {FRAME_ITEM}
[!|not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

All of these clauses are optional. You can define a wizard by simply assigning a name to it. You will learn more about each Add Wizard clause in the sections that follow.

After you create the wizard, refer to this sections to learn how to run the new wizard: *Program Wizard*, *MQL Download/Upload Commands*, and *Run/Test Wizard*.

Code Clause

This clause defines the commands for the wizard. The code provides the instructions to act on the information collected from the user during the execution of the frames belonging to the Business Wizard.

Include MQL/Tcl program commands if you specify MQL as the wizard program type.

If you specify external as the wizard program type, include the command necessary to start the external program. Include path information, if necessary. For example:

```
c:\enovia\studio\programs\buswiz3.exe
```

Because Business Wizards are made up of a number of programs, there must be a way to pass information between the programs. The method used is the Runtime Program Environment (RPE), a dynamic heap in memory that is used to hold name/value pairs of strings. See the *Runtime Program Environment* section for additional information.

Wizard Program Type (External or MQL)

You can specify the source of the code as MQL or external. An MQL program can be run from any machine with the 3DEXPERIENCE Platform installed; it is not necessary for MQL to be available on the machine.

```
add wizard NAME mql;
```

An external wizard program consists of commands in the command line syntax of the machines from which they will be run. Since MQL can be launched from a command line, MQL code could

be specified in an external program. This would spawn a separate MQL session that would run in the background. In this case, MQL would have to be installed on every machine which will run the wizard.

```
add wizard NAME external;
```

File Clause

This clause specifies the file that contains the code for the wizard. The code provides the instructions to act on the information collected from the user during the execution of the frames belonging to the Business Wizard.

This is an alternative to use the `Code` clause to define commands for the wizard.

Needs Business Object Clause

You can specify that the wizard must function with a business object. This selection assumes that you add the Business Wizard to a business Type as a method.

For example, you select this option if the wizard promotes a business object. If, however, the wizard creates a business object, the wizard is independent of an existing object and this option would not apply.

```
add wizard NAME needsbusinessobject;
```

Live Collaboration runs any wizard specified as “needs business object” with the selected object as the starting point. If a wizard does not require a business object, the selected object is not affected. The following indicates that a business object is not needed:

```
add wizard NAME !needsbusinessobject;
```

When defining a type or format, you can specify program information:

Downloadable Clause

If the wizard includes code for operations that are not supported on the Web product (for example, Tk dialogs or reads/writes to a local file) you can include the `downloadable` clause. If this is included, this program is downloaded to the web client for execution (as opposed to running on the Server). For wizards not run on the Web product, this flag has no meaning.

```
add wizard NAME downloadable;
```

If the downloadable clause is not used, `notdownloadable` is assumed.

If the downloadable clause is used, then `deferred` program execution is assumed (see the *Execute Clause*). If the downloadable clause is used, and the execute clause is `immediate`, an error will be generated that reads:

```
A program that is downloaded cannot execute immediately.
```

Execute Clause

Use the `execute` clause to specify when the wizard should be executed. If the `execute` clause is not used, `immediate` is assumed.

`Immediate` execution means that the program runs within the current transaction, and therefore can influence the success or failure of the transaction, and that all the program's database updates are subject to the outcome of the transaction.

`Deferred` execution means that the program is cued up to begin execution only after the outer-most transaction is successfully committed. A deferred program will not execute at all if the outer transaction is aborted. A deferred program failure only affects the new isolated transaction in which it is run (the original transaction from which the program was launched will have already been successfully committed).

However, there are a number of cases where deferring execution of a program does not make sense (like when it is used as a trigger check, for example). In these cases the system will execute the program immediately, rather than deferring it until the transaction is committed.

There are four cases where a program's execution can be deferred:

- Stand-alone program
- Method
- Trigger action
- State action

There are six cases where deferred execution will be ignored:

- Trigger check
- Trigger override
- State check
- Range program
- Wizard frame prologue/epilogue
- Wizard widget load/validate

There is one case where a program's execution is always deferred:

- Format edit/view/print

A program downloaded to the web client for local execution (see *Downloadable Clause*) can be run only in a deferred mode. Therefore, if you use the `downloadable` option, the program is automatically deferred.

Frame Clause

A Business Wizard is composed of one or more *frames*. Each frame is a window that requests and stores information in order to complete a task. You can include any amount of information in a frame, but in order to be effective, each frame should be responsible for gathering a single piece of information, for example, a department name or the name of an object in the database.

The Frame clause of the Add Wizard command defines all information related to a wizard frame including: units and color of the frame, prologue and epilogue programs, status, and widget types. The Frame clause uses the following syntax:

```
frame FRAME_NAME [FRAME_ITEM {FRAME_ITEM}]
```

`FRAME_NAME` is the name of the frame you are defining. All frames must have a named assigned. This name must be unique within the wizard and should have meaning for both you and the user. For more information, see *Administrative Object Names*.

The following are some examples of names you might use:

Department Name
Get Codeword
Get Filename
Availability Choices

`FRAME_ITEM` is a Frame subclause which provides additional information about the frame. The Frame definition subclauses are:

`units [picas|points|inches]`
`color FOREGROUND [on BACKGROUND]`
`size WIDTH HEIGHT`
`prologue PROG_NAME [input ARG_STRING]`
`epilogue PROG_NAME [input ARG_STRING]`
`status [true|false]`
`icon FILENAME`
`widget WIDGET_TYPE WIDGET_ITEM {WIDGET_ITEM}`

The sections that follow describe these clauses and the arguments they use.

Units Subclause

This subclause specifies the units of frame measurement. There are three possible values: picas, points, or inches.

```
units picas
Or:
units points
Or:
units inches
```

If you do not use a Units clause, a picas value is automatically assumed.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character “I” may not occupy the same amount of space as the characters “E” or “O.” To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum

amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit on a frame? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

When planning the wizard, consider the operating systems of the those who will use the wizard. The wizard should be designed with the lowest screen resolution in mind and the Master Frame should be sized to fit the screen.

Color Subclause

This subclause specifies color values used as the default foreground and background for the frame.

```
color [FOREGROUND] [on BACKGROUND]
```

`FOREGROUND` is the name of the color for the foreground printed information (any vertical or horizontal lines of information).

`BACKGROUND` is the name of the color used as an overall background for the frame. Note that the word `on` is required only if a background color is specified.

For a list of available colors, refer to:

- Windows— \$MATRIXHOME\lib\winnt\rgb.txt.
- UNIX— /$MATRIXHOME/lib/ARCH/rgb.txt.
  $MATRIXHOME is where Business Process Services is installed.
  ARCH is the UNIX platform.

Size Subclause

This subclause defines the dimensions of the frame. A frame can be any size.

To define a frame size, you need two numeric values. One represents the width (`COL_SIZE`) and one represents the length (`ROW_SIZE`). Both of these values must be provided and entered according to the following syntax:

```
size ROW_SIZE COL_SIZE
```

The values reflect the Units value defined for the frame (picas, points, or inches). For example, for an 7 by 5 inch frame, the following are equal:

`size 66 30`	Measured in picas.
`size 504 360`	Measured in points.
`size 7 5`	Measured in inches.

When selecting a frame size, keep in mind the monitor sizes of the users who will be running the wizard. For example, a wizard frame does not display the same amount of information on a screen with a resolution of 640x480 as it does on a screen with a resolution of 1280x1024. Use the lowest

common denominator of screen resolution to design wizards and the master frame should be fit to size.

Prologue Subclause

The frames's *prologue* is a program which executes immediately before a frame is displayed and before any other action is taken. Prologue programs can be used to influence the loading of widgets. They can also be used to skip frames. Each frame within a Business Wizard can have its own prologue.

The prologue program can be used to prepare for the loading of the widgets. This might include setting values in the RPE, and even redefining widget load programs. This allows widget load functions to be generic, with specific behavior being controlled at run time by the owning frame.

The prologue function can also cause the frame to be skipped. Whenever a prologue program returns a non-zero value, the frame is skipped.

The following syntax is used:

```
prologue PROG_NAME [input ARG_STRING]
```

`ARG_STRING` defines arguments to be passed into the program. The arguments can be referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

- Environment variable "0" always holds the program name and is set automatically by the system.
- Arguments from `ARG_STRING` are set in environment variables "1", "2",… etc.

See *Runtime Program Environment* for additional information on program environment variables.

One common use of the prologue program is to skip a frame within a wizard. If a frame's prologue exits with a non-zero return code, that frame will be skipped. This works either going forward or backward within a wizard. In the following example, the user is looking at Frame C and tries to click the **Back** button. You don't want the user to be able to go back to Frame B until certain information has been filled in on Frame C.

To accomplish this, you could include the following code in the prologue for all frames before Frame C (in this case, Frame B and Frame A, since you don't want to go back to either).

```
tcl;
eval {
set sFrameMotion [string toupper [ mql get env FRAMEMOTION ] ]
if { $sFrameMotion == "BACK" } {
    set iExitCode 1
} else {
    set iExitCode 0
}
exit $iExitCode
}
```

Since a frame's prologue is executed before the frame displays, if it returns an exit code of 1, the frame is skipped. In this case, the user would continue to see Frame C.

Epilogue Subclause

The frame's *epilogue* is a program which is executed whenever a frame is closed. The epilogue program can be used to undo what the prologue program did, and perform any other cleanup activity. Since data is passed between frames using RPE, you can use the epilogue program to clean up widget variables before moving on to the next frame.

If an epilogue program returns non-zero after pressing the **Next** button, the current frame is redisplayed. Thus, even if all of the widget validate programs return zero, the epilogue program can still keep the next frame from appearing. Although the epilogue program runs when the **Back** button is pressed, the return code is not checked.

When the epilogue program for the last frame in the frame sequence has been executed, programmatic control of the wizard is then handled by the wizard code, defined on the Code tab of the Business Wizard. See*Code Clause* for additional information.

The following syntax is used:

```
epilogue PROG_NAME [input ARG_STRING]
```

`ARG_STRING` defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

- Environment variable "0" always holds the program name and is set automatically by the system.
- Arguments from `ARG_STRING` are set in environment variables "1", "2",… etc.

See *Runtime Program Environment* for additional information on program environment variables.

Status Subclause

Each wizard can have an optional *status frame*, which is generated after the user has clicked the Finish button and the Business Wizard code has been executed. This option can be applied to the last frame of the sequence only.

The status frame returns feedback to the user about the task just performed or provides the reason why the task was not performed. The status frame contains a single **Close** button in place of the three regular frame buttons (Back, Next, Cancel). The dynamic region would typically contain only a multiline text box, though it could also contain image or label widgets.

Status frame processing includes executing the prologue and the load programs, but no input from the user is accepted or processed. The status frame programs should not perform any database operations.

The `status` subclause uses two arguments: true and false:

```
status [true|false]
```

When the status argument is set to `true`, the frame is used as a status frame. If the status argument is set to `false` (this is the default), no status frame is generated at the conclusion of the Business Wizard.

Widget Subclause

The term *widget* refers to any component of the frame. The Widget subclause of the Frame clause uses the following syntax:

```
widget WIDGET_TYPE WIDGET_ITEM {WIDGET_ITEM}
```

`WIDGET_TYPE` refers to one of ten types of fields that make up the dynamic region of the frame. Widget types include the following:

Widget Type	Meaning
`label`	a non-editable text field.
`button`	a button which launches a program object.
`textbox`	a text field which can be defined to be either editable or non-editable.
`image`	a picture file containing a GIF image.
`listbox`	a box containing a list of items from which multiple selections can be made.
`combobox`	a box containing a list of items from which a single selection can be made.
`radiobutton`	a group of one or more options, each with a circle next to it. A blank circle indicates that it is not selected (or “off”); a filled in circle indicates that it is selected (or “on”). A radio group is used for mutually exclusive choices, that is, only one option can be selected.
`checkbox`	a group of one or more options, each with a check box next to it. A blank check box indicates that it is not selected (or “off”); a box with a check in it indicates that it is selected (or “on”). Multiple options can be selected.
`directorytextbox`	a text field with an ellipsis button next to it, which launches the directory chooser.
`filetextbox`	a text field with an ellipsis button next to it, which launches the file chooser.

`WIDGET_ITEM` is a Widget subclause which provides additional information about the widget. The Widget subclauses are:

```
name WIDGET_NAME
value VALUE [selected value|input ARG_STRING]
load PROG_NAME [input ARG_STRING]
validate PROG_NAME [input ARG_STRING]
observer PROG_NAME [input ARG_STRING]
color FOREGROUND [on BACKGROUND]
start XSTART YSTART
size WIDTH HEIGHT
font FONT_NAME
autoheight [false|true]
```

```
autowidth [false|true]
drawborder [false|true]
multiline [false|true]
edit [false|true]
scroll [false|true]
password [false|true]
upload [false|true]
```

The widget subclauses are described in the sections that follow.

Widget Name Subclause

The Name subclause is used to define a unique name for the widget. Widget names can be any length of alphanumeric characters; spaces can be included.

Since widgets are named so that they can be identified by the system, a unique name is automatically created by the system for each widget. Therefore, this subclause is optional. If you do not specify a name for the widget, the system-supplied name is used.

If you want to include the widget on multiple frames within the Business Wizard and retain the value within each frame, you should replace the system-supplied name with a name of your own choosing. Widgets that share the same name will share the same value.

The widget Name subclause has the following syntax:

```
widget WIDGET_TYPE name WIDGET_NAME
```

Value Subclause

This subclause identifies the contents of the widget. The Value subclause has the following syntax:

```
value VALUE [selected value|input ARG_STRING]
```

For all widgets except the image and command button widget types, VALUE specifies the text that will be included in the widget, for example, the string of characters to be included as a label. For textbox or listbox widgets, VALUE is the default text that is included in the box. For combobox, checkbox, or radiobutton, VALUE is the text included in the group.

Several widgets need two pieces of information: a list of choices and one or more selections. `Selected value` is an optional argument which identifies the item you want selected or highlighted. For example, if you have a radio group that contains the days of the week and you want “Wednesday” to be selected by default, use the following syntax:

```
value Monday Tuesday Wednesday Thursday Friday selected value Wednesday
```

For the image widget type, VALUE is the name of the .gif file used to represent the image. For example:

```
value hands.gif
```

For the command button widget type, VALUE is the name of the program file used to represent the graphic. ARG_STRING is an optional argument(s) that can be passed into the program.

Load Subclause

The load program, if defined, contains code to load values into the widget field. `Load` is an option for all widgets except for label and image. The load program can also be used to define alternate values for the widget depending on the context, information gathered in a prior frame, etc. The following syntax is used:

```
load PROG_NAME [input ARG_STRING]
```

`ARG_STRING` defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.
- Environment variable “1” always holds the widget name and is also set automatically by the system.
- Arguments from the Input field are set in environment variables “2”, “3”,… etc.

If a command button program returns non-zero, the frame is refreshed. This means that all widgets in the frame will have their load program run. This allows a command button program to modify variables in the RPE and then force the widgets in the current frame to reload themselves.

Validate Subclause

Validate programs check if the values added by the user (to an input field, for example) are within an acceptable range of parameters. `Validate` is an option for all widgets except for label and image.

The validate program, if defined, is used to test the validity of the widget's state. By returning a non-zero value, the validate program causes the system to redisplay the frame and place focus on the offending widget. It is good style to have the validate program additionally display a message box that explains the error.

Validate programs are normally used for editable text box widgets and combo box widgets.

---

*Note that the validate program does not execute when the user clicks the **Back** button.*

The following syntax is used:

```
validate PROG_NAME [input ARG_STRING]
```

`ARG_STRING` defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.
- Environment variable “1” always holds the widget name and is also set automatically by the system.
- Arguments from the Input field are set in environment variables “2”, “3”,… etc.

Observer Subclause

This subclause defines a program to be executed when the user makes a selection from a wizard list box, check box, or radio button.

The following syntax is used:

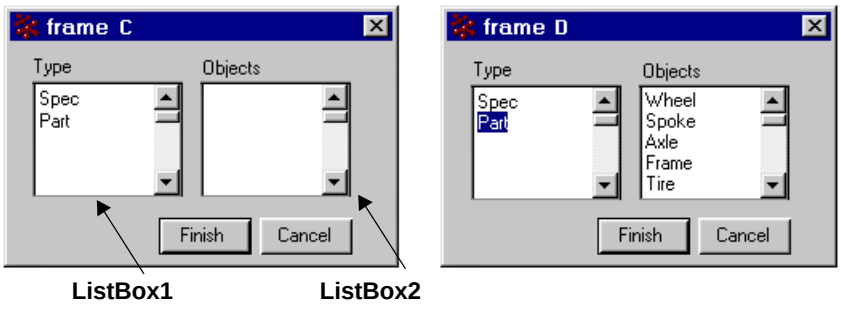
```
observer PROG_NAME [input ARG_STRING]
```

ARG_STRING defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2… etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.
- Arguments from ARG_STRING are set in environment variables “1”, “2”,… etc.

See *Runtime Program Environment* for additional information on program environment variables.

For example, the selection of Type “Part” from the ListBox1 calls an observer program that populates the Business Objects in ListBox2, as illustrated below.



The observer program has all the side effects of clicking a button within a wizard; that is, it is executed immediately and if it exits with a return code of 1, each of the widgets in the current frame has its values (re)read and (re)populated by the RPE variables associated with the frame. The INVOCATION macro is populated with the value “observer.” If the widget was a multi-selection list box, then `mql get env <widget name>` returns a space-delimited string containing current values.

Performance is an important consideration. When calling an observer program, every widget on a frame is examined to get the current value of that widget. For a frame with many widgets, this could be time consuming. Also, since the observer program is actually run on the server, there is lag time associated with the round trip from client to server and back. The developer of an observer program needs to evaluate whether the time it takes for the observer program to complete is acceptable for a user to wait.

Color Subclause

This subclause specifies color values used as the default foreground and background for the widget.

```
color [FOREGROUND] [on BACKGROUND]
```

FOREGROUND is the name of the color for the foreground printed information (any vertical or horizontal lines of information).

BACKGROUND is the name of the color used as an overall background for the widget. Note that the word on is required only if a background color is specified.

A list of available colors is contained on Windows in \$MATRIXHOME\lib\winnt\rgb.txt.

Start Subclause

This subclause is used to define the placement of the widget in the frame. The syntax is:

```
widget WIDGET_TYPE start XSTART YSTART
```

The XSTART (horizontal) and YSTART (vertical) coordinates identify the widget's starting point—where the first character of the field value is displayed. For example, to place a label widget in the upper left corner of the frame, the starting position would be 0, 0:

```
widget label start 0 0
```

Size Subclause

This subclause is used to define the size of the widget. Specify height and width values to define the widget size. The syntax is:

```
widget WIDGET_TYPE size WIDTH HEIGHT
```

The widget's size is dependent on the Units (picas, points or inches) specified for the widget. The `width` value defines the horizontal size of the widget. The `height` value defines the vertical size of the widget. For example, to define a textbox widget 12 picas long by 3 picas high, use:

```
widget textbox units picas size 12 3
```

Font Subclause

Use the Font subclause to **s**pecify the font in which the text of a widget field appears. This option is available for all widget types except for the image widget. The syntax is:

```
widget WIDGET_TYPE font FONT_NAME
```

Fonts available are based on the computer's defined system fonts.

Although it is possible to change the fonts used in wizards, it is best to let a wizard use the default font and colors which have been set up by the end user, particularly if they will be used across the Web. See the *Business Modeler Guide : Wizard Fonts on the Web*.

Autowidth and Autoheight Subclauses

Specify `true` or `false` for the Autowidth and Autoheight subclauses of the Widget subclause if you want the program to select the appropriate height and/or width based on the contents of the widget. The syntax is:

```
widget WIDGET_TYPE autoheight [true\false] autowidth [true|false]
```

Drawborder Subclause

Use the `drawborder` subclause to include a border around the field. This option is available for the text box, image, check box, radio button, file text box, and directory text box widget types. The syntax is:

```
widget WIDGET_TYPE drawborder [false|true]
```

Multiline Subclause

Use the `multiline` subclause to have text displayed on more than one line in a text box or to allow more than one choice to be selected in a list box. The syntax is:

```
widget WIDGET_TYPE multiline [false|true]
```

Edit Subclause

Use the `edit` subclause to allow the user to change the value while viewing the frame. This option is available only for the text box widget type. The syntax is:

```
widget WIDGET_TYPE edit [false|true]
```

Scroll Subclause

Use the `scroll` subclause to have scroll bars displayed so that the user can scroll text up and down. This option is available only for the text box widget type. The syntax is:

```
widget WIDGET_TYPE scroll [false|true]
```

Password Subclause

Use the `password` subclause to have all text the user types into the field masked with the asterisk character. This option is available only for the text box widget type. The syntax is:

```
widget WIDGET_TYPE password [false|true]
```

Upload Subclause

Use the `upload` subclause to allow the user to select a client-side file and copy it to the server. This option is available only for the file text box widget type. See *Upload Command vs. Widget Upload Option* for additional information. The syntax is:

```
widget WIDGET_TYPE upload [false|true]
```

Program Wizard

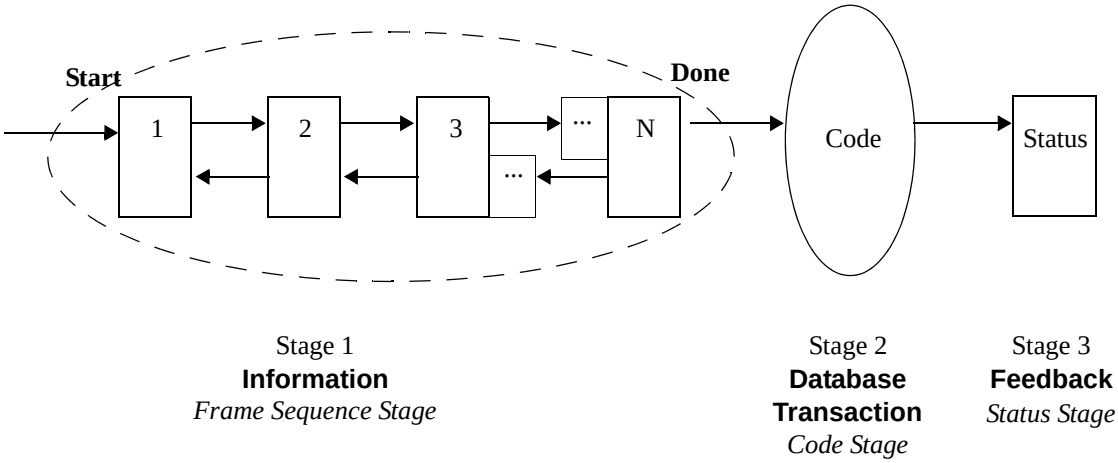
In creating a Business wizard, you will need to write at least one program, which will perform the database transaction(s) for which the wizard is created. You can also write programs to control the loading of frames and widgets and to check the validity of widget values.

Before you begin to create a Business wizard, you should be familiar with the information contained in *Programs* in Chapter 7, including information about the Runtime Program Environment (RPE).

As you plan the project, keep in mind that a wizard has three stages:

- **Stage 1** — Information. This stage consists of a series of frames, which gather information from the user. The information is stored in the RPE, which is used to pass data between frames. **No database updates should be performed during this stage.** This is important so that the **Cancel** function can work properly.
- **Stage 2** — Database transaction. The wizard code, defined in the New Wizard dialog box Code tab, is executed. This stage uses the information from Stage 1 to perform all necessary database transactions. Results can be placed in the RPE for use by the optional Status Frame.

- **Stage 3** — Feedback. A single status frame is displayed to inform the user of the status of the transaction performed by the wizard. This stage is optional. It is important to note that this stage follows stage 2 and is not part of the sequence of frames in Stage 1.



Strategy for Creating Business Wizards

The following is one strategy that can be used in creating a Business wizard. The order of the steps can, of course, be changed according to your own programming style.

1. **Plan the project.** Decide what questions you want to ask the user and what format they will use to supply answers. Will they type in their answers, or choose from a list or group? Input fields provide the most flexibility, but it is easier to control the user's choices and prevent errors by having them make a selection from a fixed number of items.
2. **Design the wizard.** Decide the number of frames and the layout of the widgets in each frame. It is most effective to have each frame request a single piece of information.

   The frame sequence should only be used for the task of gathering information from the user, deferring any database work until the internal code of the wizard is executed.
3. **Design the master frame.** Master frames allow you to easily create a polished look for the Business wizard by making each frame the same size with the same color scheme. You can also add widgets that you want to appear in each frame, for example a logo or a horizontal rule. These default attributes can be overridden, as needed, on a frame-by-frame basis.
4. **Create the wizard**, positioning widgets on the frames. As you add frames, they inherit the characteristics of the master frame. If, however, you subsequently make changes to the colors or the dimensions of the master frame, these changes will *not* be reflected in any frames already added. Therefore, be sure to set up the master frame before adding any new frames.
5. **Write code to load and validate widgets**, if the widgets require load or validate programs.
6. **Write code for prologue and epilogue programs**, if frames require prologue or epilogue programs.
7. **Add program names** (for load, validate, prologue, epilogue) on the Edit Widget and Edit Frame dialog boxes of the Business wizard you created.
8. **Write code to execute the task** based on the information collected in the wizard. Include the code on the New Wizard dialog box of the Business wizard.

Program Environment Variables

Because business wizards are made up of a number of independent programs that may need to share data, there must be a way to pass information between the programs. The method used is the Runtime Program Environment (RPE), a dynamic heap in memory that is used to hold name/value pairs of strings.

In using the RPE to pass data between program objects, careful attention must be paid to the naming of program environment variables and to the cleaning up of these variables. All data is passed in the form of strings. Here are some guidelines:

- The program name is automatically associated with a variable named “0”.
- The widget name is automatically associated with a variable named “1”.
- Program input parameters are named “1”, “2”, etc. except within a load or validate program for a widget. Since the variable “1” is reserved for the widget name, program input parameters are named starting with the number “2”.
- The calling program can manually place the expected local variables into the RPE (using the naming scheme above) before executing the called program, or simply add them to the end of the execute command; the system will automatically place them in the RPE with the proper names.
- Programs should return their data in a variable with the name identified with the 1st input parameter.
- Lists should be returned using a Tcl form (space separated, with curly braces used to surround items that contain spaces or special characters).
- Since the RPE is shared memory, any variable can be overwritten by a variable of the same name. It is good practice to always capture wizard variables immediately into custom local variables to be assured that the values will be available where necessary within the wizard. RPE variables can be saved as local variables (in Tcl: `set localUser [mql get env USER]`) or saved back into the RPE under a different name that won’t be overwritten (for example: `programName.USER` where programName is the name of the program).

  Any program or trigger that runs at any time before the wizard completes has the potential to alter RPE variables. For example, a user starts a wizard, then checks the attributes of another object in the database. Depending on how the object is configured, a program or trigger may fire automatically without the user’s knowledge that could overwrite existing RPE variables.

  For this reason, saving the Type, Name, Revision, and Objectid into wizard-specific RPE variables should always be done in the prologue of the first frame. This is the only way to preserve this information. A properly written wizard/program should *always* fetch out of the RPE all variables needed before proceeding.

How the System Displays Frames and Widgets

With the previous naming conventions in mind, here is how the frame and widget functions work:

1. When a frame is displayed, its prologue function is first executed. The prologue should establish any parameters in the RPE that will be used by the load functions of the widgets belonging to the frame. This allows widget load functions to be generic, with specific behavior being controlled at run time by the owning frame. The prologue function can also cause the frame to be skipped by returning a non-zero value.
2. Each widget belonging to the frame will then be constructed and set using the following steps:

a) If an RPE variable exists with the widget's name, the system uses its value. The variable may already exist if a previous frame has a widget of the same name, or if the frame's prologue program created the variable.

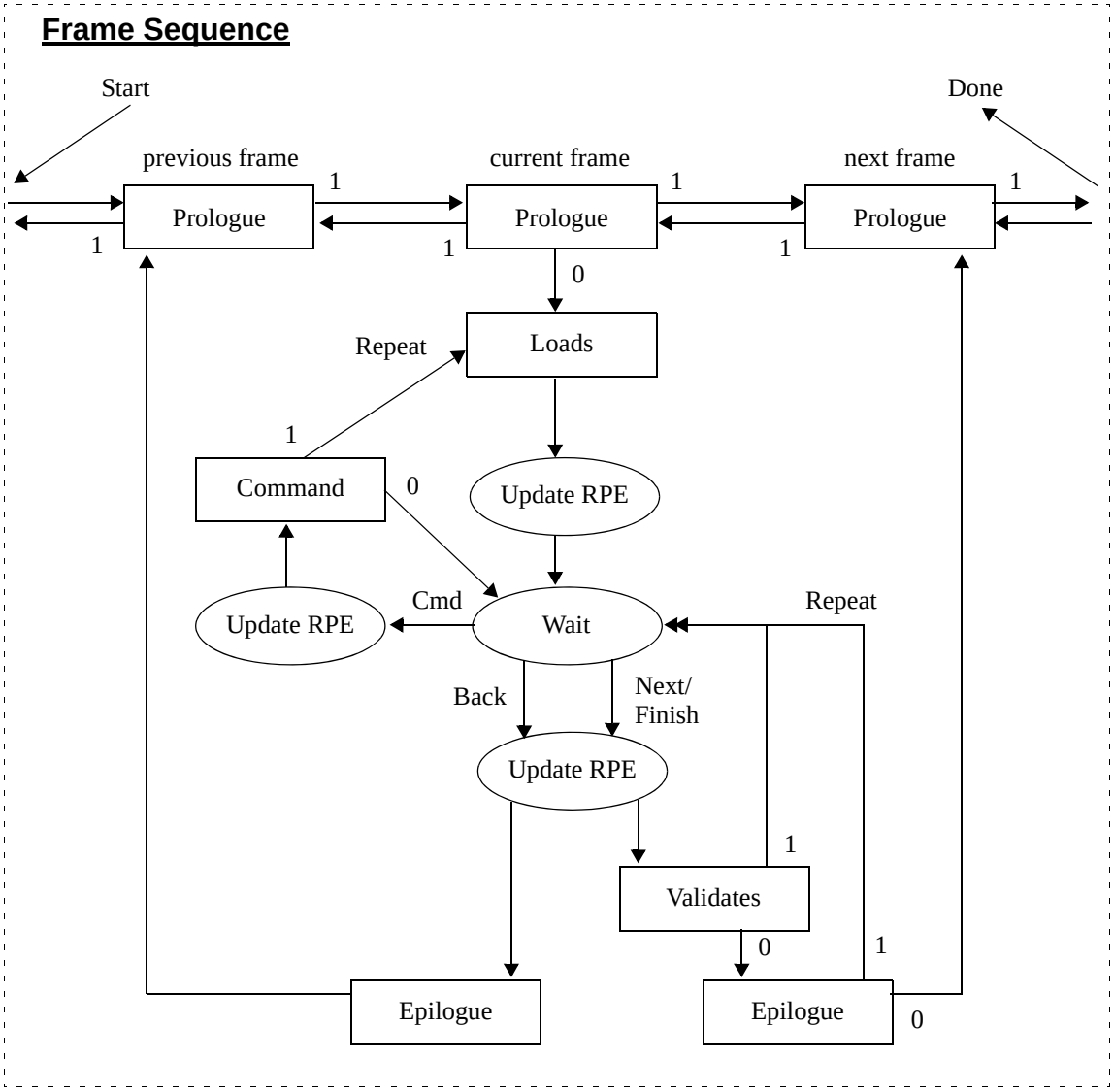
b) This ensures that any redisplay of a frame retains the previously-set value of the widget. It also allows you to include in a summary frame all items selected by the user.

c) Assuming no RPE variable is found, the system next checks to see if the widget has a load program. If so, the load program is executed, and again a search is made for an RPE variable with the widget's name. (Each load program is given the widget's name in argument 1.)

d) Assuming no RPE variable is found, the system uses the widget's default value found in the value fields on the Edit Widget dialog box.

3. All widgets, except Label and Graphic, have an RPE variable that has the widget's name and has a value equal to its current state. For example, a widget named `answer` is a radio group made up of the choices `Yes` and `No`. This is represented in the RPE with a variable named `answer`; its value is either `Yes` or `No` depending on the widget's state. As the state of the widget changes, the value of the widget variable changes.
4. If the **Back** button is pressed, the current frame's epilogue function is called (performing any necessary cleanup), and steps 1 through 3 are repeated for the previous frame.
5. If the **Cancel** button is pressed, the Business wizard immediately ends.
6. If the **Next** or **Finish** button is pressed, each widget (except Labels and Graphics) belonging to the frame executes its validate function. After each save function is executed, the exit code is checked and if it is non-zero, the frame remains displayed and focus is placed on the offending widget. This should happen only with an editable text box or combo box widget since all other widgets should provide the user only with legal choices. The current frame's epilogue function is called, and if there is a next frame, steps 1 through 3 are repeated for that frame.

7. If the **Finish** button is pressed, then the body of the Business wizard code is executed. If a status frame is defined, it is displayed after the code is executed and only steps 1 and 2 are performed (the user will need to press the **Close** button to remove the frame from the screen – essentially step 5).



For each widget type there is a defined format for the widget variable that holds state information. These formats are as follows:

Widget types	Widget variable values
Text box	The actual text (including newlines)
List box	List of selections
Combo box	
Radio button	
Check box	

Loading Complex Widgets

Several widgets need two pieces of information: a list of choices, and one or more selections from the list. For example you might have a check box group with the choices `Weekdays`, `Saturday`, and `Sunday`, and want the choice `Weekdays` to be selected by default.

Choices and selections can be defined on the Edit Widget dialog box in the fields **Choices** and **Selected**. List choices (and selections) in a single string with a space used as a separator.

When using MQL, set the widget's stored database value to the following:

```
VALUE Weekdays Saturday Sunday SELECTED Weekdays
```

As it reads from the database, the system will load choices into the widget until it finds `selected`. It then assumes that the tokens that follow are default selections.

Another approach is to use a load program. By convention, the choices go into an RPE variable whose name is the same as the load program (obtained through argument 0) and the selections go into an RPE variable whose name is the same as the widget name (obtained through argument 1). Remember that the widget variable in the RPE holds its state, and the state of a complex widget is its current selections. So the load program has identified the state of the widget by loading the variable associated with the argument 1.

Using Spaces in Strings

Lists of widget choices and selections are given in a single string with spaces used as separators. This means that any choice containing spaces should be quoted.

However, due to the extensive use of the Tcl language for manipulating lists, the Tcl notation for lists is recognized by the system. Therefore curly braces can be used to quote single list items. This is true for all arguments being passed into program objects and all values returned by program objects. The system also does a better job of supporting nested use of curly braces than it does with quote characters.

Using ${} Macros

For compatibility, ${} macros continue to be supported. ${} macros are placed into the RPE so that they can be used by nested programs. Just remember to drop the "${}" characters when referencing the macro variable in the RPE.

The following macros are used in wizards:

Macro	Meaning
`WIZARDNAME`	Name of the Business wizard.
`FRAMENAME`	Name of the current frame.
`FRAMENUMBER`	Number of the current frame in the frame sequence (excluding master and status frames). Frame numbers begin with 1. The status frame returns a value of 0.
`FRAMETOTAL`	Total number of frames in the frame sequence (excluding master and status frames).

Macro	Meaning
FRAMEMOTION	Current state of wizard processing: `start` The wizard has been started; the user has not yet clicked a command button. `back` The user clicked the Back command button. `next` The user clicked the Next command button. `finish` The user clicked the Finish command button. `repeat` The current frame has been redisplayed. `status` The current frame is the status frame of the wizard.

SELECTEDOBJECTS	List of currently selected objects to be passed to any program. This macro is populated whenever a program or method is invoked from the toolbar, right-mouse menu or Methods dialog. The macro's value consists of a single string of space-delimited business object IDs for the objects that are selected at the time the program or method is invoked. (For a method, this macro is redundant—it is equivalent to the OBJECTID macro—but it is populated nevertheless for consistency.)
	The SELECTEDOBJECTS macro can be read into a Tcl string or list variable as follows:
	`# this reads the macro as a single Tcl string`
	`set sObjs [mql get env SELECTEDOBJECTS]`
	`# this reads the macro into a Tcl list.`
	`set env lObjs [split [mql get env SELECTEDOBJECTS]]`
	When no objects are selected, the macro is created, but holds an empty string.

WORKSPACEPATH	Directory that is used as the base directory for upload/download. For programs/wizards run through the server, it is evaluated by combining the enovia.ini settings MX_BOS_ROOT and MX_BOS_WORKSPACE, and appending a unique temporary directory for the session that runs it to avoid collisions across sessions. For desktop client configurations, it is set to the enovia.ini file for the Studio Modeling Platform setting TMPDIR. These settings guarantee that the client has access to the files on the server.

Although the operating system determines the valid syntax of the commands, the typical syntax used is:

```
command ${MACRO 1} ${MACRO 2} ...
```

Using Eval Commands

You can wrap the code in an `eval` command to prevent the 3DEXPERIENCE platform output window from popping up. Placing tcl code in an `eval` command causes output to be suppressed when executed within 3DEXPERIENCE platform.

MQL Download/Upload Commands

The download and upload MQL commands are available for use in programs to be executed by a Web client through an Server. They allow client-side files and directories to be manipulated by server-side programs (and vice versa). See *download Command* and *upload Command* for syntax.

Generally the download and upload commands would be used in conjunction with a wizard 'file text box' widget. See *Widget Subclause* for details.

Run/Test Wizard

Any user with appropriate access can run a wizard program from within Matrix Navigator or from the system command line, including from a Windows desktop icon. A wizard can also be launched from within a program.

MQL trace can be used to test timing and debug wizard programs. For information, see the *Installation Guide* : *Server Diagnostics*.

A wizard can be run by any of the following methods:

- including its name in an Toolset (see *toolset Command*)
- executing the wizard as a Method, which requires an object as its starting point

Command Line Interface

From the system command line, the syntax of the command to run a wizard program is:

```
matrix -wizard NAME
```

where NAME is the name of the wizard program to launch.

The syntax of the command to run a wizard as a method is:

```
matrix -wizard businessobject OBJECTID WIZARD_NAME
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

where `WIZARD_NAME` is the name of the wizard program to launch.

When a wizard is run from the system command line, the user sees only the wizard user interface and is insulated from the rest of the user interface. The application is started, and when the wizard code has been executed, the application shuts down. For this reason, a Business Administrator testing the wizard throughout its creation process would probably test from within Matrix Navigator instead of using this method.

For example, a user may have to check in a report each week. A wizard could be created to gather information such as the name and type of the report and the user/department submitting it. Because the user does not need any of the other features to check in the report, the wizard could be run from the system command line or a desktop icon in Windows.

Copy Wizard

After a wizard is defined, you can clone the definition with the Copy Wizard command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy wizard SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM};
```

`SRC_NAME` is the name of the wizard definition (source) to copied.

`DST_NAME` is the name of the new definition (destination).

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

`MOD_ITEM`s are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modify Wizard

After a wizard is defined, you can change the definition with the Modify Wizard command. When modifying a wizard program that is used to launch an application, however, consider upward and downward compatibility between software versions.

The following command lets you add or remove defining clauses and change the value of clause arguments:

```
modify wizard NAME [MOD_ITEM] {MOD_ITEM};
```

`NAME` is the name of the wizard you want to modify.

`MOD_ITEM` is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Wizard clause, as listed in the following table. Note that you only need to specify fields to be modified:

Modify Wizard Clause	Specifies that…
`code CODE`	The current code definition changes to that of the new code entered.
`description VALUE`	The current description, if any, changes to the value entered.
`external|mql`	The specification of the wizard program type (external or MQL) changes as entered.
`file FILENAME`	The current file name changes to the new name entered.
icon FILENAME	The image is changed tp the new image in the field specified.
`name NAME`	The current name of the program changes to the name entered.
`[!]needsbusinessobject`	The status of the need for a business object changes as indicated here: `needsbusinessobject` is used when a business object is needed. `!needsbusinessobject` is used when a business object is not needed.
`[!]downloadable`	The status of downloadable changes as indicated here: `downloadable` is specified when the program includes code for operations not supported on the Web product (for example, Tk dialogs or reads/writes to a local file). `!downloadable` is specified when the program does not include code for operations not supported on the Web product.
`execute immediate`	The status of wizard execution changes so the program runs within the current transaction.
`execute deferred`	The status of wizard execution changes so the program runs only after the outermost transaction is successfully committed.
`hidden`	The hidden option is changed to specify that the object is hidden.
`nothidden`	The hidden option is changed to specify that the object is not hidden.
`frame FRAME_NAME FRAME_MOD_ITEM {FRAME_MODE_ITEM}`	The named frame item(s) are changed to the new item(s) specified.
`add FRAME FRAME_NAME [before FRAME] [FRAME_ITEM {FRAME_ITEM}]`	The named frame is added.

Modify Wizard Clause	Specifies that…
`remove FRAME FRAME_NAME`	The named frame is removed.
`property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is modified.
`add property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is added.
`remove property NAME [to ADMINTYPE NAME] [value STRING]`	The named property is removed.

As you can see, each modification clause is related to the arguments that define the wizard frame.

Modifying Wizard Frames

The following subclauses are available to modify the existing frames in a wizard:

Modify Wizard Frame Subclauses	Specifies that…
`FRAME_ITEM`	The named frame item(s) are changed to the new item(s) specified.
`name FRAME_NAME`	The frame name is changed to the new name specified.
`widget WIDGET_NAME WIDGET_ITEM {WIDGET_ITEM}`	The existing widget is modified with the WIDGET_ITEM(S) specified.
`add widget WIDGET_TYPE [WIDGET_ITEM {WIDGET_ITEM}]`	The named widget is added.
`remove widget WIDGET_NAME`	The named widget is removed.

Modifying Widgets

The following subclauses are available to modify the existing widgets in a wizard frame:

Modify Widget Subclauses	Specifies that…
`name WIDGET_NAME`	The widget name is changed to the new name specified.
`value VALUE [selected value|input ARG_STRING]`	The existing widget value is modified.
`load PROG_NAME [input ARG_STRING]`	The load program for the named widget is changed.
`validate PROG_NAME [input ARG_STRING]`	The validate program for the named widget is changed.
`color FOREGROUND [on BACKGROUND]`	The foreground and/or background color(s) of the named widget are changed to the new color(s) specified.
`start XSTART YSTART`	The position of the named widget is changed.
`size WIDTH HEIGHT`	The dimensions of the named widget are changed.

Modify Widget Subclauses	Specifies that…
`font FONT_NAME`	The font for the text in the named widget is changed.
`autoheight [false|true]`	The value of `autoheight` for the named widget is changed. `True` indicates that the height of the widget is determined by the system; false indicates that the height of the widget is coded.
`autowidth [false|true]`	The value of `autowidth` for the named widget is changed. `True` indicates that the width of the widget is determined by the system; false indicates that the width of the widget is coded.
`drawborder [false|true]`	The `drawborder` value for the named widget is changed. `True` indicates that the widget will have a border; `false` indicates that the widget will not have a border.
`multiline [false|true]`	The `multiline` value for the named widget is changed. `True` indicates that a text box widget can contain multiple lines or a list box widget can allow multiple selections; `false` indicates that the widget will contain a single line or allow a single selection.
`edit [false|true]`	The `edit` value for the named widget is changed. `True` indicates that the widget field is editable by the user; `false` indicates that the field is not editable.
`scroll [false|true]`	The value of the `scroll` option for the named widget is changed. `True` indicates that the widget field is scrollable; `false` indicates that the field is not scrollable.
`password [false|true]`	The `password` value for the named widget is changed. `True` indicates that a password is required; `false` indicates that no password is required.
`upload [false|true]`	The `upload` value for the named file text box widget is changed. `True` indicates that the widget is designed to allow the user to select a client-side file and copy it to the server; `false` indicates that the wizard programmer will include the MQL upload command in one of the wizard programs.

As you can see, each modification clause is related to the arguments that define the wizard frame widgets.

Delete Wizard

If a wizard is no longer required, you can delete it with the Delete Wizard command:

```
delete wizard NAME;
```

`NAME` is the name of the wizard to be deleted.

Searches the list of wizards. If the name is not found, an error message is displayed. If the name is found, the wizard is deleted.

Index

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | Y

Symbols

Numerics

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X