

Intershop 7 Application Programming Guide



Application Programming Guide

Document ID: ENF7-00-05-05

Publication date 2012-04-05

These materials are subject to change without notice. These materials are provided by Intershop Communications AG and its affiliated companies ("Intershop Group") for informational purposes only, without representation or warranty of any kind, and Intershop Group shall not be liable for errors or omissions with respect to the materials. The only warranties for Intershop Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

This document and all of its parts are protected by copyright. All rights, including those of duplication, reproduction, translation, microfilming, storage on electronic media and processing in electronic form are expressly reserved.

Intershop® and Enfinity™ are trademarks or registered trademarks of Intershop Communications AG. All other company, product and brand names are trademarks or registered trademarks of their respective owners.

Copyright © 2005-2012 Intershop Communications. All Rights Reserved.

Table of Contents

Chapter 1: Getting Started	22
About this Guide	22
Knowledge Assumed	22
Typographical Conventions	23
Other Documentation Available	23
Developing on Intershop 7	23
Setting Up Intershop Studio	24
Installing and Configuring Intershop Studio	24
Set Additional Intershop Studio Preferences	28
Optimize Intershop Studio Performance	34
Updating Intershop Studio	36
Intershop 7 Architecture	37
Cluster Perspective	37
Architectural Layers	39
Programming Artifacts	41
Intershop 7 Test Framework	43
What Is a Unit Test?	43
Why Run Unit Tests?	43
The JUnit Test Framework	44
Package Naming Conventions	45
The Test Scenario	45
PA-DSS Compliance	47
Sensitive Authentication Data	47
Key Store	48
Audit Trail	48
Payment Information Transfer	48
Chapter 2: Cartridge Development	49
Cartridges	49
What Are Cartridges?	49
Cartridge Components	49
Cartridge Definition	50
Cartridge Controller Class	52
Cartridge Development Cycle	52
Overall Development Process	52
Components of an Intershop 7 Development Environment	53
Overview of the Cartridge Development Cycle	54

Planning a Cartridge Project	55
Cartridge Layer Model	56
Base Cartridges	59
Package Structure	59
Cartridge Class Path Container	60
Cartridges and Intershop Studio	61
Cartridge Build and Deployment	62
Building the Cartridge	62
Deployment Process	64
Cartridge Startup Process	65
Apps and Cartridges	66
Defining Apps	66
Adding Cartridges To Apps	67
Cartridge Development Overview	67
Creating New Cartridges	69
Open the Cartridge Wizard	69
Step 1: Define Cartridge Properties	69
Step 2: Define Cartridge Template, Structure, additional Properties and Dependencies	71
Finish the Cartridge Creation Process	72
Set the Code Generator Version for the Project	72
Importing Cartridges	73
Importing Source and Server Cartridges	74
Importing Existing Projects	77
Working with Cartridges	79
Explore Cartridge Resources	79
Manage Cartridge Properties	79
Cartridge Editor	79
Cartridge Properties View	80
Register Cartridges	81
Managing Classpath Settings	81
Understanding Classpath Containers	81
Set Classpath for Cartridge Project	81
Set Server Classpath	82
Automatic Update of Classpath Container	83
Building a Cartridge	83
Overview of Build Process	83
Preparing Properties Files	83
isbuild.properties	84

<IS_HOME>\tools\build\shared\build.properties	84
<IS_SOURCE>\<cartridge_name>\build\build.properties	85
Preparing Build Files	85
Overview of Build Files	85
Build File Configuration	86
Configure ANT Settings and Run Build	86
Run Default Build Target	87
Run Build Process with Modified Settings	87
Run Selected ANT Targets	87
Run ANT Manually	89
Localizing Templates During the Build Process	89
Cartridge Editor	90
Cartridge Editor: Overview	91
Cartridge Editor: Dependencies	92
Cartridge Editor: Runtime	93
Cartridge Editor: Database Initialization	94
Cartridge Editor: Cartridge Pipeline Hooks	95
Cartridge Editor: Properties	96
Development Directory	97
Build Directory	98
Chapter 3: Business Object Development	100
Business Objects	100
The Business Object Layer	100
Business Object Layer Terms	101
Aggregates	101
Entity Objects	101
Value Objects	102
Repositories	103
Extensions	103
Business Object Context	104
Persistence Layer Mapping	104
Business Object Layer Implementation	105
Business Object Framework	105
Repository Types	105
Business Object Custom Attributes	106
Business Object Lifecycle	106
Business Object Change Listener	107
Transaction Handling	107
Identifiers	107
Caching	108

Cache Invalidation	108
Persistent Objects	109
Persistent Objects and Modeling	109
API Structure and Interface Types	110
Enfinity Definition Language (EDL)	110
Persistent Objects	110
Persistent Object Classes	110
Object-Relational Mapping	110
Managers - Access to Persistent Objects	111
Base Classes for Persistent Objects	111
Modeling Example for Persistent Object Classes	112
Native Attributes	112
What Are Native Attributes?	112
Data Types and Type Mapping	113
Extensible Object Attributes	114
What Are Extensible Object Attributes?	114
AttributeValue Tables	115
Localizable Extensible Object Attributes	116
Replicated Extensible Object Attributes	116
Data Types	116
Direct Custom Attributes	117
What Are Direct Custom Attributes?	117
DCA Definition	117
DCA Access	118
Relationships Between Persistent Objects	119
Relationship Types: Associations and Weak Relation	119
Multiplicity of Relationships	120
Relationship Names and Roles	120
One-to-Many Relationships	120
Weak Relations	121
Many-to-Many Relationships	123
Managers for Persistent Objects	123
What Are Managers?	123
Base Class	124
Manager Methods	124
Modeling of Managers and the respective Implementations	124
CAPI Interfaces	124
What Are CAPI Interfaces?	124
Inbound and Outbound CAPI Interfaces	125
CAPI Interfaces and Persistent Objects	126
CAPI Interfaces and Managers	126
Providers	126

Creating Models	127
Start the Model Wizard	127
Set General Model Properties	128
Define the Initial Model Content	129
Import additional EDL files	131
Working with EDL Models	131
Add New Model Elements	131
Add Attributes and Operations	131
Add Relationships and Dependencies	132
Delete Model Elements	133
Compare Data Object Models	133
Compare Object Model Versions	133
Compare Different Object Models	134
Modeling Persistent Objects for Intershop 7	134
Create Business Interfaces	134
Create a Persistent Object	134
Create Native Attributes	136
Create Extensible Object Attributes	136
Create Association Relationships	137
Create Weak Relations	138
Create Manager Implementation Classes	138
Modeling of Managers and the respective Implementations	138
Generating Code	139
Validate the Model	139
Generate Code	139
Generate Database Resource Files	139
How To	139
Handle Foreign Key Constraints	139
Handle Index Definitions	140
Declaring an Index	141
Generating an Index	141
Configure the Naming Manager	142
Sample Configuration	142
Fallback Mechanism	143
Access Manager Classes and Business Interfaces	143
Access Factories and POs	144
Use Finder Methods	144
Use Bind Variables	145
Java Source Code Conventions	146
Source Code Formatting	146

Editor Settings	146
Language	147
Line Breaks	147
Nesting Levels	147
Method Declarations	147
Import Statements	148
If-Statements	148
Switch Statements	148
Blank Spaces	149
Wrapping Lines	149
Source Code Naming	151
General Rules	151
Class Names	151
Method Names	151
Package Names	152
Loop Counters	153
Arguments/Parameters, Attributes/Properties, and Local Variables	153
Constant Names	153
Source Code Documentation	153
Non-Javadoc Comments	153
Javadoc Comments	154
Javadoc Examples	155
Java Editor	156
Outline View for Java Editor	157
Automatic Code Validation	157
Content Assist	158
Pipelet Assist	158
EDL Model Editor	160
User Interface	160
Outline View for EDL Model Editor	160
Content Assist	161
EDL Model Comparer	161
Object Path Expressions	162
Object Path Expressions: Overview	162
Object Path Syntax	163
Object Path Elements	163
Parameters	163
Literals	163
Null Values	164
Mapping of Object Paths to Java Objects	164
Lookup Methods	164

Lookup Strategies	165
Object Paths in ISML	166
ISML Expressions	166
ISML Loops	167
Object Paths in Pipelines	167
Pipelet Input Aliases	167
Pipeline Decision Nodes	168
Pipeline Loop Nodes	168
Web Services	168
Object Paths in Intershop Studio	169
EDL Reference	169
EDL Files	169
Core Language	170
Comments	170
Imports	170
Primitive Types	170
External Types	170
Namespaces	171
Complex Types	171
Attributes	171
Relationships	172
Modifiers	172
Properties	172
Constraints	173
Literals	173
Reserved Keywords	173
ORM Models	174
ORM Classes	174
ORM Attributes	175
ORM Relations	176
ORM Dependencies	177
CAPI Models	178
CAPI Interfaces	178
CAPI Attributes	178
CAPI Constants	179
CAPI Relations	179
CAPI Operations	179
XML Models	180
XML Classes	180
XML Attributes	180
XML Containments	181
RAPI Models	181

RAPI Interfaces	181
RAPI Operations	182
Chapter 4: Pipelet Development	183
Pipelets	183
What Are Pipelets?	183
Pipelet Class	184
Pipelet Descriptor Files	184
Creating Pipelets	185
Start the Pipelet Wizard	185
Set General Pipelet Properties	185
Define Class Name and Package Structure	187
Set Pipelet Properties	188
Working with Pipelets	191
Edit General Pipelet Properties	191
Add Additional Pipelet Properties	191
Working with Pipelet Property Editors	191
Explore Pipelet References	192
Using the Pipelet Assist	192
Using the Quick Fix Function	195
Pipelet Reload	197
Editing Pipelets on Remote Servers	198
Styleguide	198
Quick Overview	198
Pipelet Modeling	199
Pipelet Naming	199
Pipelet Class Naming	199
Pipelet Group Naming	201
Pipelet Parameter Naming	201
Pipelet Parameter Types	202
Pipelet Parameter Fallbacks	204
Pipelet Form Data Access	206
Pipelet Error Handling	206
Chapter 5: Pipeline Development	208
Pipelines	208
What Are Pipelines?	208
Pipeline Elements	209
Public and Private Pipelines	209
Pipeline Dictionary	210
What Is the Pipeline Dictionary?	210

What Does the Pipeline Dictionary Contain?	210
Pipeline Types	210
Pipeline Execution	211
Pipeline Loading	213
Internal Pipeline Representation	214
Pipeline Overloading and Pipeline Inheritance	214
Overloading vs. Inheritance	214
Defining an Overload Relation	216
Pipeline Overloading: Basic Properties	217
Defining a Pipeline Inheritance Relation	217
Pipeline Inheritance: Basic Properties	218
Pipeline Inheritance and Call/Jump Node Resolution	218
The "Super:" Operator	220
Pipeline Parameters and Pipeline API	221
What Is the Pipeline API?	221
Pipeline API and Pipeline Dictionary Processing	222
Aliasing	223
Overloading the Pipeline API	224
Using Pipelines With and Without Pipeline API	224
Pipeline Debugger	225
Creating Pipelines	225
Start the Pipeline Wizard	225
Set General Pipeline Properties	225
Add Pipelets and Flow Control Elements	227
Add Transitions	227
Manage Pipeline Layout	228
Manage Node and Transition Properties	229
Manage Pipeline Parameters	229
Manage Pipeline Access Control	229
Defining Basic Pipeline Patterns	231
A Simple Pipeline	231
Changing the Pipeline Execution Flow	231
Using Multiple Start Nodes	231
Calling a Sub-Pipeline	232
Ending Pipelines With a Jump	232
Using Named End Nodes and Multiple Call Node Exits	233
Using Loop Nodes	233
Working with Pipelines	234
Check Pipeline and Data Flow	234
Pipeline Reload	235
Compare Pipelines	236

Pipeline References and Dependencies	241
Using the Content Assist	241
Using the Quick Fix Function	242
Run Pipeline From Intershop Studio	242
Editing Pipelines on Remote Servers	244
Debugging Pipelines	245
Preparing to Debug	245
Set Breakpoints	245
Define Connection and Site Context	246
Launch Debug Session	246
Step Through a Pipeline in Debug Mode	247
Working With Debug Views	247
Pipeline Security	248
Public and Private Start Nodes	248
Prefix Pipelines	248
CorePrefix Pipeline	250
Pipeline Access Control Lists	251
ACL Lookup Process	252
ACL Syntax	252
Styleguide	253
Quick Overview	253
Pipeline Security	254
General	254
Pipeline Access Control Lists	254
Process Pipelines	254
Separating View and Process Pipelines	254
Input Parameters	255
Process Pipelines in Business Components vs. in Application Cartridges	256
Migration of Process Pipelines	257
Pipeline Naming Guidelines	257
Presentation Pipelines	257
Process Pipelines	259
Visual Pipeline Editor	260
User Interface	260
Palette	262
Index Map	262
Push Action	262
Quick Outline View	262
Customize the Pipeline Editor	264
Content Assist	264
Pipeline Comparer	266

Pipeline Component Reference	267
Start Node	268
Error Node	271
End Node	272
Stop Node	273
Decision Node	273
Loop Node	275
Interaction End Node (Interaction Node)	276
Interaction Continue Node	277
Call Node	279
Jump Node	281
Join Node	281
Transitions	282
Connectors	283
Chapter 6: ISML Template Development	284
Templates and ISML	284
What Are Templates?	284
Templates and ISML	284
Template Types	285
Template Deployment	285
Templates and Localization	286
Locales	286
The Default Directory	286
Fallbacks for the Template Lookup Process	287
The Template Conversion Process	287
Converting ISML Templates to Java Classes	288
Converting Java Classes to HTML	288
Static and Dynamic Content Conversion	288
Character Encoding in the Conversion Process	288
Character Encoding for Reading ISML Templates	288
Character Encoding in ISML-to-JSP Conversion	289
Character Encoding in Java Source and Class File Conversion	291
Decoding Post Data	291
Page Caching	292
Basic Principles	292
Page Cache Administration	293
Page Caching Restrictions	293
Personalized Page Caching	294
Selective Page Cache Deletion	294
Intershop Markup Language (ISML)	295
What is ISML?	295

ISML Language Elements	295
Expression Example	297
Expression Conventions	297
Expression Syntax	297
Template Variables	302
Constants	303
Operators	304
ISML Functions	306
Creating Templates	306
Start the Template Wizard	307
Set General Template Properties	307
Working with Templates	308
Creating Template Folders	308
Using the Content Assist	308
Using the Quick Fix Function	309
Open Referenced Elements	309
Explore References and Dependencies	309
Using Pre-Defined Code	310
Editing Templates on Remote Servers	310
Styleguide	311
Source Code Formatting Conventions	311
ISML	311
Formatting Rules	312
XHTML Standard	313
Template Naming	314
Web Forms	315
Template Editor	316
User Interface	316
Outline View for Template Editor	317
Snippet View for Template Editor	318
Content Assist	319
Properties View Cell Editors	322
ISML Reference	323
ISML Tags	323
<ISBINARY>	323
<ISBREAK>	324
<ISCACHE>	324
<ISCACHEKEY>	326
<ISCOMMENT>	327
<ISCONTENT>	327

<ISCOOKIE>	330
<ISIF>, <ISELSEIF>, <ISELSE>	332
<ISFILE>	333
<ISFILEBUNDLE>, <ISFILE>, <ISRENDER>	333
<ISINCLUDE>	335
<ISLOOP>, <ISNEXT>, <ISBREAK>	338
<ISMODULE>	341
<ISPIPELINE>	344
<ISPLACEHOLDER>	345
<ISPLACEMENT>	348
<ISPRINT>	348
<ISREDIRECT>	351
<ISRENDER>	352
<ISSELECT>	352
<ISSET>	353
<ISNEXT>	355
<ISTEXT>	355
<IS{Module Name}>	357
ISML Functions	357
action()	357
ContentURL()	358
ContentURLEx()	360
existsTemplate()	361
getCookie()	362
getHeader()	362
getText(), getTextEx()	363
getValue()	363
hasElements(), hasLoopElements()	364
hasNext()	365
isDefined()	365
isSSEnabled()	365
Icase()	366
len()	366
pad()	366
parameter()	367
paramMap(), paramEntry()	367
replace()	368
servlet()	368
sessionlessurl(), sessionlessurlex()	369
split()	369
stringToHtml()	369
stringToWml()	370
stringToXml()	370

trim()	370
ucase()	371
url()	371
urlex()	372
val()	373
WebRoot()	373
WebRootEx()	374
Chapter 7: Pagelet Development	376
Content Management System (CMS) Development with Pagelets	376
Introduction	377
CMS Layer Model	377
Content Model Development	379
Content Model Elements	379
Pagelet Definitions	379
Content Entry Point Definition	380
Slot Definition	381
Relationship of Content Instances	381
Page and Page Variant	381
Pagelet (Page Variant/Component) and Slot	383
Slot and Component	383
Component and Component Entry Point (Include)	386
Advanced Concepts	386
Context Object Relation (COR)	386
Other CMS related Resources	387
Pipelines	387
ISML Templates	387
Content Templates	387
Pagelet Model Editor	388
User Interface	388
Pagelet Development Perspective	388
Central Pagelet Model View	389
Properties View	390
Outline View	390
Open a Pagelet2 Model	390
Create a new Pagelet2 Model	391
Edit a Pagelet2 Model	393
Pagelet Definition (Content Unit)	393
Slot Definition	398
Call Parameter Interfaces	398
Page Entry Point Definition (Page Definition)	400
Component Entry Point Definition (Include Definition)	401

Context Object Relation Definition	402
Configuration Parameter Type	402
Delete a Pagelet2 Model	403
Copy a Pagelet2 Model	403
Move a Pagelet2 Model	403
Rename a Pagelet2 Model	404
Chapter 8: Creating Extensions	405
Extensions	405
What are Extensions?	405
How the Extension Mechanism Works	405
Pipeline Extensions	407
ISML Template Extensions	407
Java Extensions	407
Defining New Extension Types	407
Extending Pipelines	408
Create a Pipeline Extension Point	408
Create and Implement an Extension Pipeline	409
Bind the Extension Pipeline to the Extension Point	411
Extending ISML Templates	413
Define an ISML Extension Point	413
Create and Implement an Extension Template	413
Bind the Extension Template to the Extension Point	413
Extending Java Code	415
Define a Java Extension Point	416
Create and Implement an Extension Class	416
Bind the Extension Class to the Extension Point	417
Chapter 9: Query Development	419
Queries	419
Usage Patterns	420
Query Files	422
User Search Expressions	423
Public APIs and Usage	423
Pipelets	423
VerifySearchTerm	423
LoadQuery	424
UpdateDictionary	424
ExecuteCountQuery	425
ExecuteObjectsQuery	426

ExecutePageableQuery	427
ExecuteUpdateQuery	428
Query Files	429
Parameter Declaration	429
Return Mapping	430
Query Processor Declaration	431
Query Templates	431
Public Java Interfaces	433
SearchExpression	433
QueryMgr	434
Query	434
QueryStatement	434
QueryContext	434
QueryParameterHandler	434
QueryResult	435
Row	435
QueryProcessor	435
Creating and Editing Query Files	437
Creating Query Files	437
Editing Query File Details	438
General Query File Details	439
Execution Processor	439
Parameters	440
Return Values	441
SQL Statements	442
Query Editor	443
User Interface	443
Outline View for Query Editor	444
Chapter 10: Webform Development	445
Overview	445
General Process for Using Webform	446
Set Up a Webform Model	447
General	447
Creating Webform Models	448
Editing Webform Models	449
Optional: Implement Validators and Formatters	450
Validators	451
Formatters	452
Create a Pipeline to Create a Form Instance	453

Create a Pipeline to Handle the Input	454
Set Up a Display Template	455
Webforms Reference	456
Webform Related Classes and Pipelets	456
Object Path Expressions	458
Chapter 11: Web Service Development	460
 Web Services	460
What Are Web Services?	460
Web Service Technologies	460
Web Service and Intershop 7	461
Technical Infrastructure	461
Accessing WSDL Files for Intershop 7 Web Services	463
Data Types	463
Primitive Types	463
Java Object Types	464
JavaBeans	464
 Preparing a Web Service	465
Planning the Web Service	465
Implementing Web Service Functionality	465
 Publishing a Web Service Pipeline	467
Configuring the Web Service	467
Configuring the Web Service Operations	469
Accessing the Web Service	471
 Creating a Web Service Client	471
Configuring the Web Service Client	472
Configuring the Java Stub Class Generation	474
Configuring the Pipelet Generation	475
Chapter 12: Intershop Studio Reference	478
 Preferences	478
Intershop 7 Development Environment	478
Appearance Preferences	479
Cartridge Source Code Locations	480
ISML Template Preferences	481
Localization Preferences	482
Pipeline Editor Preferences	483
Problem Annotation Preferences	484
Remote Server Preferences	485
Server Classpath Preferences	486

XML Descriptor Files Preferences	487
Workbench and Perspectives	488
Workbench, Views and Editors	488
Perspectives	491
Views	492
The Cartridge Explorer	492
The Toolbar	492
The Status Icons	493
The Drop-Down Menu	494
Drag & Drop Operations	495
The Package Explorer	495
The Toolbar	495
The Status Icons	495
The Drop-Down Menu	496
Pipelet, Pipeline and Template Views	496
The Toolbar	497
The Status Icons	497
The Drop-Down Menu	497
Pagelet View	498
The Toolbar	498
The Status Icons	498
The Drop-Down Menu	499
Components View	499
Content Component Definitions View	501
Query View	503
The Toolbar	503
The Status Icons	504
The Drop-Down Menu	504
The Properties View	504
The Toolbar	504
The Problems View	505
The Outline View	505
The Pipeline Dictionary View	505
The Status Icons	505
Intershop 7 Dependency Hierarchy View	506
Webforms View	507
Search	509
Global Search Capabilities	509
Intershop 7 Cartridge Search	509
Intershop 7 Element Search	510
Finding Elements In Graphical Editors	510

Searching Element References	511
Searching Similar Elements	512
Searching Elements Used	512
Remote Server Configuration	513
Server Connection Parameters	513
Logon Parameters	514
Common Parameters	515
Chapter 13: API Tools	517
API Tools Introduction	517
Supported Artifacts	517
Main Tasks	517
Using API Tools	518
Configuring API Tools	518
Creating API Models	519
Exporting API Documentation	520
Viewing API and API Modifications	520
Updating API Model	521
Exporting Change Reports	522
Creating Migration Reports	522
Command Line API Tools	522
API Definition Exporter	523
API Definition Updater	524

Getting Started

About this Guide

This guide is addressed to developers intending to create solutions on top of the Intershop 7 platform, or to extend or customize an existing Intershop 7 solution. The book focuses on the following topics:

■ Technologies and standards

Which existing technologies and standards does Intershop 7 use? How do developers leverage these technologies into an Intershop 7 application?

■ Programming components and layers

Which programming layers are part of the Intershop 7 platform component? What are the basic building blocks that these programming layers contain, out of which an application is composed?

■ Basic programming and application design steps

Which basic programming steps do developers take to assemble building blocks into deployable application components? How is custom functionality implemented, and how is a new piece of functionality made available on a server? Which general design considerations have to be taken into account when planning an application?

■ Available programming tools

Which programming tools are available to aid in the development of Intershop 7 applications?

Reading through the chapters that follow, keep in mind that this book sometimes takes a high-level approach, offering a strategic overview. For more information on particular topic, the reader is often referred to other books of the Intershop 7 documentation set in which individual topics are discussed in more detail.

Knowledge Assumed

The book assumes basic familiarity with the following topics:

■ The Java programming language and XML

The Intershop 7 platform component is based on the Java EE 6 standard. Creating an Intershop 7 application typically involves substantial coding in Java. Moreover, XML (eXtensible Markup Language) is used virtually everywhere in

Intershop 7, in order to impose a structured representation on data processed by Intershop 7 applications.

■ The Java EE Platform

The Java Platform, Enterprise Edition (Java EE) is the industry standard for enterprise Java computing as defined in the Java EE specification (<http://www.oracle.com/technetwork/java/javaee/overview/index.html>)

The Intershop 7 programming model leverages many of the components and services defined in the Java EE specification.

■ Multi-tier Client-server Architectures

Intershop 7 applications are typically distributed across multiple components and layers. Therefore, familiarity with design principles and communication technologies involved in the client-server architecture model is required.

Typographical Conventions

Throughout this book the following formatting conventions are used to denote special elements in the text:

- *italic* typeface

Is used to highlight cross-references to other parts of this guide or to external resources, such as other documents of the ∁ documentation set or Web sites. Is also used for special words, e.g. names of files and directories, cartridges, and packages.

- monospaced typeface

Is used within paragraphs for reserved and special words, e.g. names for database tables, Java classes, methods, and attributes. Also, for any commands or options to be typed at a command prompt. Whole paragraphs set in a monospaced font highlight sample code or excerpts from configuration files.

- the hash symbol (#)

If the hash symbol precedes a shell prompt, it indicates that the current user is root.

- <placeholder>

Text enclosed within angle brackets (<text>) are placeholders.

Placeholders designate text in path names or code samples that is context dependent and needs to be replaced with a real name or value, e.g. <current-date>

Other Documentation Available

Customers and partners with valid support contract can download Intershop 7 guides at through the Intershop Customer Support Knowledge Base at <http://support.intershop.com>.

Developing on Intershop 7

Apart from providing business applications to serve sales, procurement and content management processes, Intershop 7 provides a powerful platform for developing

additional applications for unified commerce systems that are precisely tailored to the specific needs of a particular business model. Unified commerce applications based on Intershop 7 offer unique integration capabilities by sharing business data and business logic across multiple sites and multiple business units.

From a development perspective, Intershop 7 offers a range of distinct features which greatly facilitate application development and significantly shorten time-to-market. Important features include:

- **Rich pre-built functionality**

Intershop 7 comes with a set of powerful and feature-rich frameworks. These frameworks encode a wide range of objects and business logic connecting these objects, providing a solid base for every unified commerce application. When creating new applications based on Intershop 7, developers can directly integrate the functionality made available by the Intershop 7 frameworks, which greatly simplifies and accelerates the development process and shortens time-to-market.

- **Fully integrated set of development tools**

Intershop 7 features various tools designed to aid in application development. Using these tools facilitates seamless integration of new and existing application components.

- **Open standards and technologies**

Intershop 7 is based on open standards, including many of the standards that are part of the J2EE platform such as Servlet technology, JSP, JNDI, JDBC, JAAS, and JTA. In addition, Intershop 7 leverages Internet technologies such as eXtensible Markup Language (XML) and Web services.

- **Component-based Application Design**

By design, Intershop 7 naturally supports implementation approaches which attempt to de-compose a complex application into a set of smaller, re-usable components or modules. This facilitates application development across teams and the re-use of code.

Setting Up Intershop Studio

The main development tool for the Intershop 7 platform is Intershop Studio an integrated development environment based on IBM's Eclipse platform (<http://www.eclipse.org/>).

The latest Intershop Studio revisions are based on the Eclipse Indigo release (<http://www.eclipse.org/org/press-release/20110622indigo.php>).

Installing and Configuring Intershop Studio

Intershop supports Intershop Studio on Windows only. It is packaged on the Intershop 7 installation medium for Windows in `setup\studio`.

NOTE: Irrespective of the OS version of Intershop 7, all customers with valid support contracts can obtain Intershop Studio releases and updates via Intershop Support.

Irrespective of the way Intershop Studio is distributed, as a customer with a valid support contract you can also obtain Intershop Studio releases and updates via the Intershop Support Web site (<http://support.intershop.com>).

The following instructions assume that Intershop Studio is obtained and installed separately from Intershop 7. Do the following:

1. Get Intershop Studio.

Log on to the Intershop Support Web site <http://support.intershop.com> and download the Intershop Studio package applicable for your platform (either 32- or 64-bit), e.g., IntershopStudio_3.0.0.9-win32.win32.x86_64.zip.

2. Extract the zip archive on your machine to a directory of your choice.

NOTE: since Intershop Studio is not installed in a literal sense, the extract directory will also be the location where your Intershop Studio instance is going to live from now on. We will refer to this directory as <IntershopStudio>.

3. Launch Intershop Studio.

Run the IntershopStudio.exe executable from the <IntershopStudio> directory.

4. Define a workspace location.

When an Intershop Studio instance is launched for the first time, it prompts you for a workspace location. The workspace is a directory on your hard disk where Intershop Studio stores information about loaded projects, preferences and configuration parameters, external tools, and any other custom settings.

The Intershop Studio default location for the workspace is in <IntershopStudio>\workspace. However, you may want to choose a location outside the Intershop Studio directory tree, to keep your data separated from Intershop Studio files. By this you can more easily upgrade Intershop Studio or backup your workspace data. To make this setting permanent, select the provided checkbox.

When working on several projects in parallel you may consider creating multiple workspaces, one for each project. To switch between workspaces during a running Intershop Studio session go to File | Switch Workspaces.

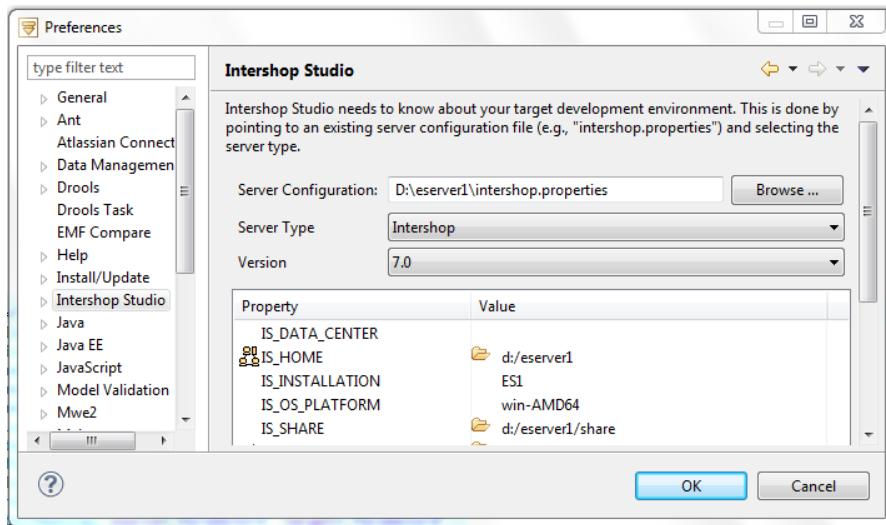
Another way of selecting a workspace is by creating multiple Intershop Studio shortcuts, one for each project, where you specify the workspace to be loaded via the a start parameter to the Intershop Studio executable:

```
<IntershopStudio>\IntershopStudio.exe -data d:\es_workspaces\project_1
```

5. Connect Intershop Studio to the target development environment

You need to tell Intershop Studio the location of your local development environment (with the directories build, source, and target). Intershop Studio can read this information from your server configuration, as defined in your %IS_HOME%\intershop.properties file.

Go to Window | Preferences | Intershop Studio and specify the according server configuration.

Figure 1. Setting the Target Development System in Intershop Studio

Make sure to select the correct Intershop 7 version.

See *Intershop 7 Development Environment* for more information on this preferences page.

6. Set the Default Code Generator Version.

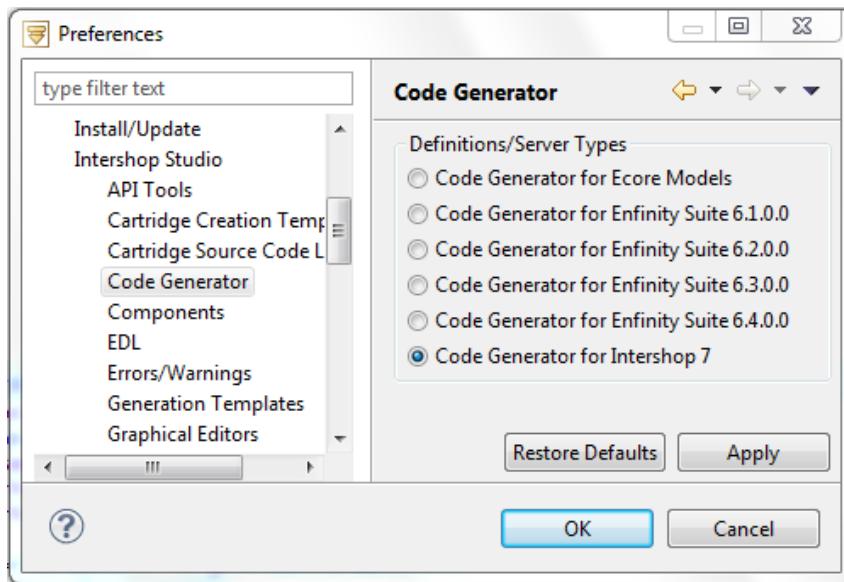
Intershop Studio includes a code generator for validating and generating Java object code from Enfinity Definition Language models (EDL). The code generator version must match the Intershop 7 version for which you intend to develop.

The code generator version can be set as a workspace and a cartridge property. The value of the workspace property is used as the default value which will apply as long as a cartridge does not overwrite this value.

To set the default code generator version which applies to the entire workspace:

a. **Go To Window | Preferences | Intershop Studio | Code Generator.**

Figure 2. Setting the Code Generator Version in Intershop Studio



b. **Set the code generator version according to your Intershop 7 target development system.**

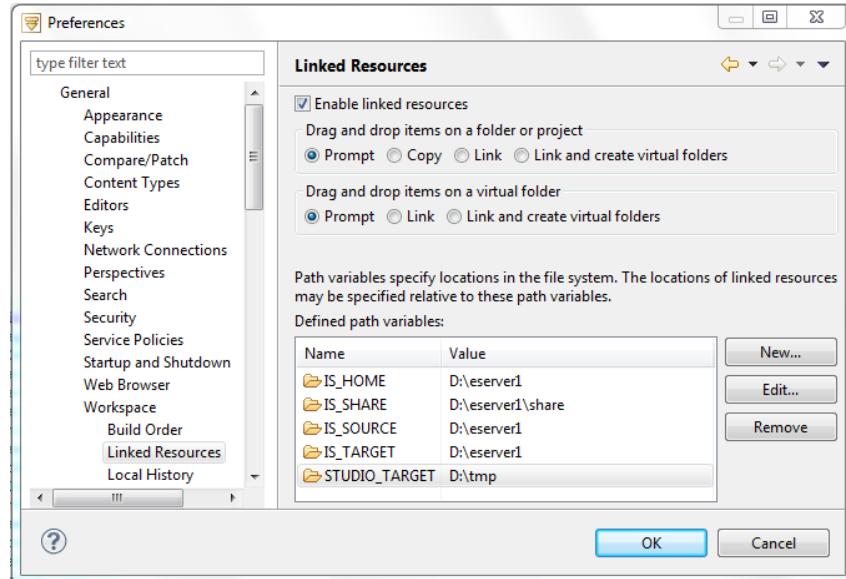
7. **Optional: Specify the Intershop Studio build target folder (ESTUDIO_TARGET).**

In order to show code errors and warnings, Intershop Studio re-builds projects when project resources are modified. The default target for these builds is a `bin` folder in the cartridge source directory.

If you don't want to clutter up your development environment with files generated by Intershop Studio, do the following:

- a. **Go to Window | Preferences | General | Workspace | Linked Resources.**
- b. **Add and set the STUDIO_TARGET variable accordingly.**

Figure 3. Setting the Intershop Studio Build Target



8. Import Cartridge Sources into Intershop Studio.

In order to edit cartridge sources, you need to import them first. Note, that the import is not done in a literal sense (no copying of physical files), Intershop Studio merely reads in the contents of the cartridges from the build environment and lets you browse cartridge resources in the package explorer view. When you edit cartridge sources in Intershop Studio any changes are written to the build environment.

To "import" cartridge sources into Intershop Studio:

- a. **Go to File | Import | Intershop Studio Cartridge Development | Existing Source and Server Cartridges into Workspace.**
- Opens the cartridge source import wizard.
- b. **Import cartridges.**

The Import wizard shows a list of the cartridges that are available for import. Select the one you wish to import, then click Finish.

NOTE: In order to show up in the list, a cartridge must have been checked out to the source directory of your development environment and must be included in the server's cartridgelist.properties file.

Set Additional Intershop Studio Preferences

To confirm to the Intershop conventions for formatting Java source code as outlined in *Java Source Code Conventions* you should go through the preferences pages under Window | Preference | Intershop Studio and set the line

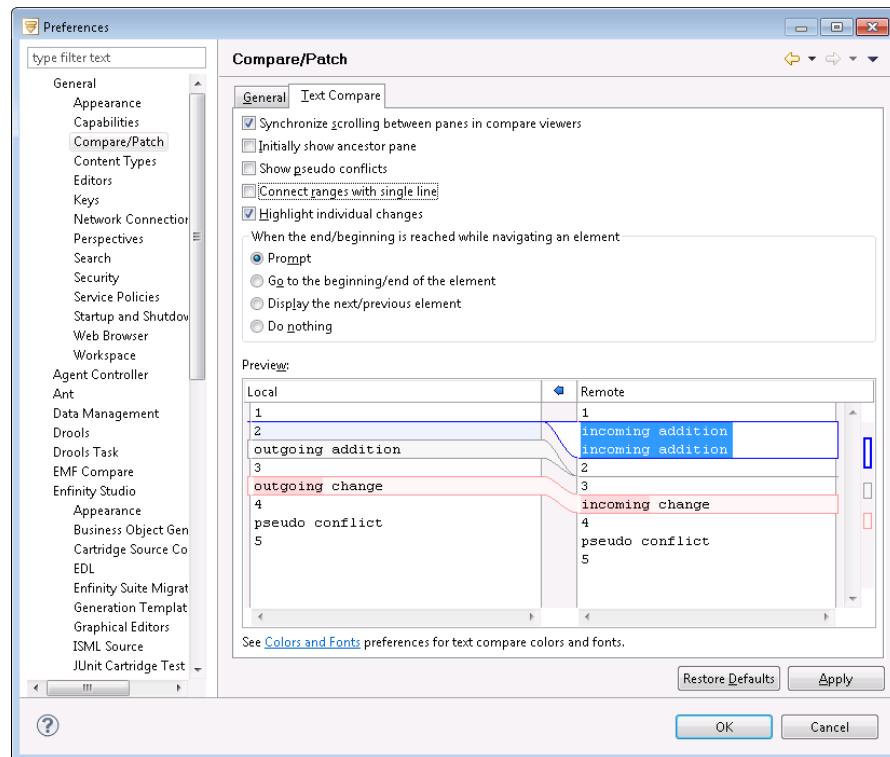
length to 120, enable tab-replacing by spaces and switch off all beautifier rules for editors (except rules for organizing imports).

Additionally, you may want to make the following adjustments to your Intershop Studio preferences:

1. Go to Window | Preferences | General | Compare/Patch.

On the Text Compare tab, To improve readability of the Intershop Studio compare view disable the "Connect ranges with single line" feature.

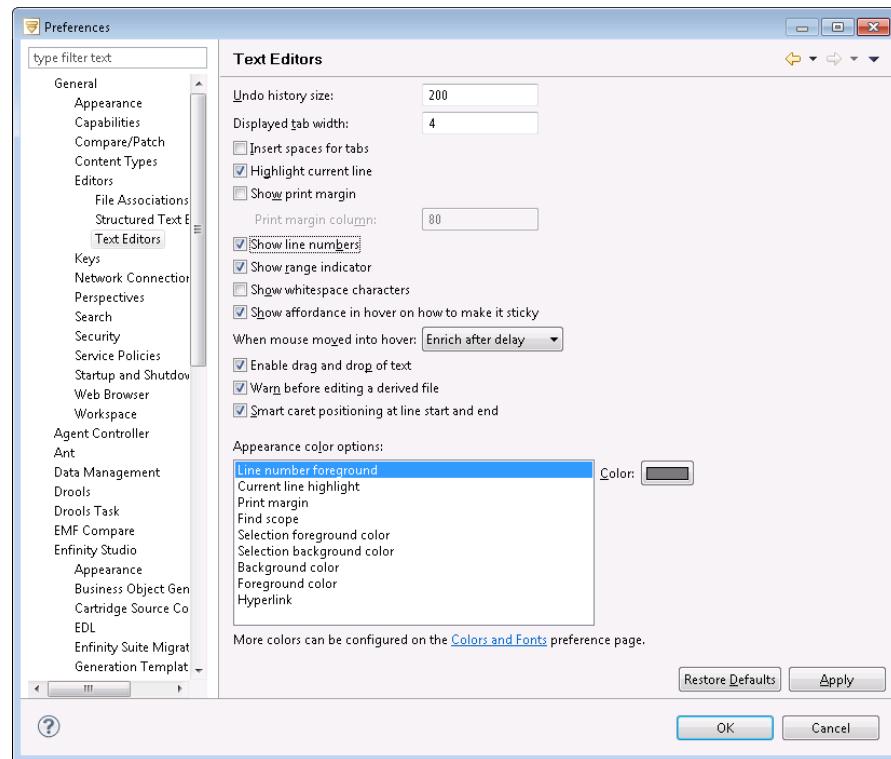
Figure 4. Intershop Studio Compare/Patch Preferences



2. Change text editor preferences.

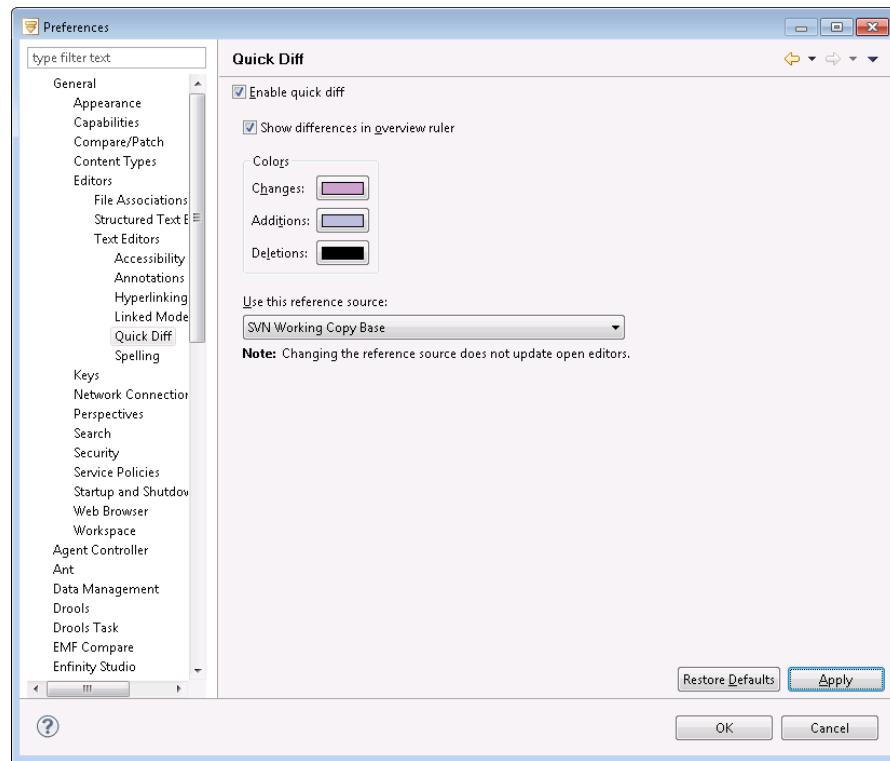
Go to Window | Preferences | General | Editors | Text Editors.

Enable "Show line numbers" and "Enable drag and drop of text".

Figure 5. Intershop Studio Text Editor Preferences

Go to Window | Preferences | General | Editors | TextEditors | Quick Diff.

Enable "Enable Quick Diff" and the "Show differences in overview ruler" options, change the reference source to "SVN Working Copy Base".

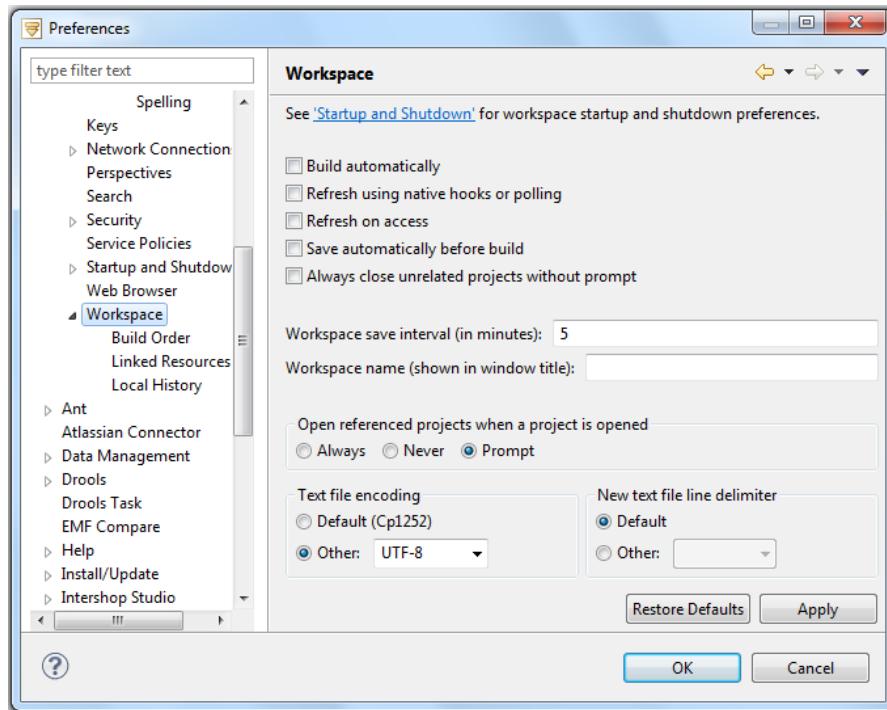
Figure 6. Intershop Studio Text Editor Quick Diff Preferences

3. Set workspace preferences.

Go to Window | Preferences | General | Workspace.

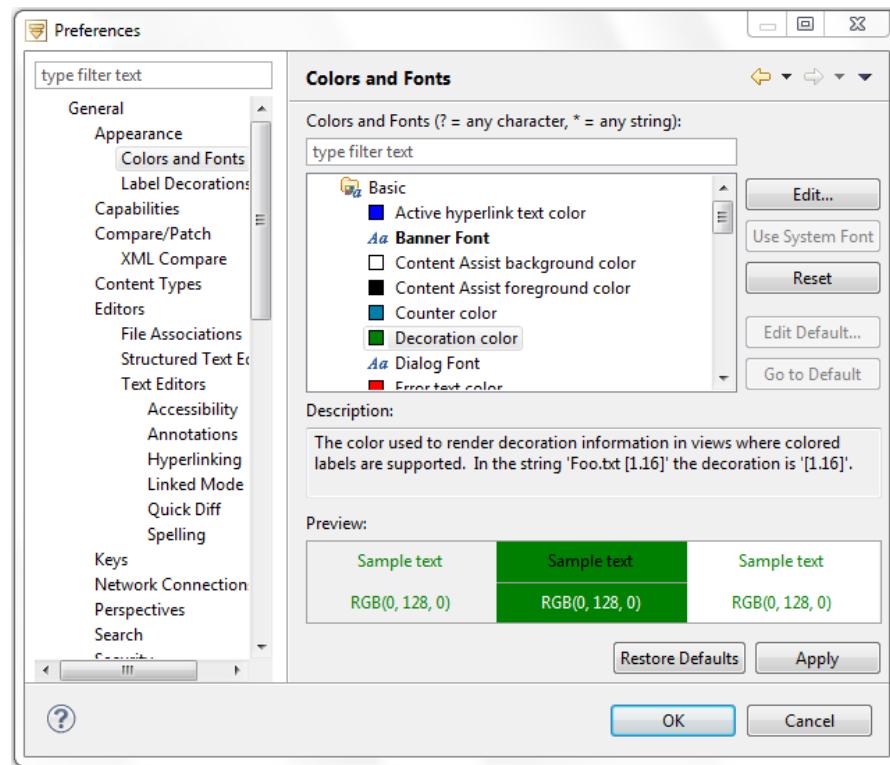
Make sure, all checkboxes are un-checked and the "Workspace save interval" is set to a proper value (e.g. 5 minutes).

If you run multiple Intershop Studio instances and/or have several workspaces (e.g. when developing in branch and trunk) assign a name to your workspace. This name will then be displayed in the workspace's title bar.

Figure 7. Intershop Studio Workspace Preferences

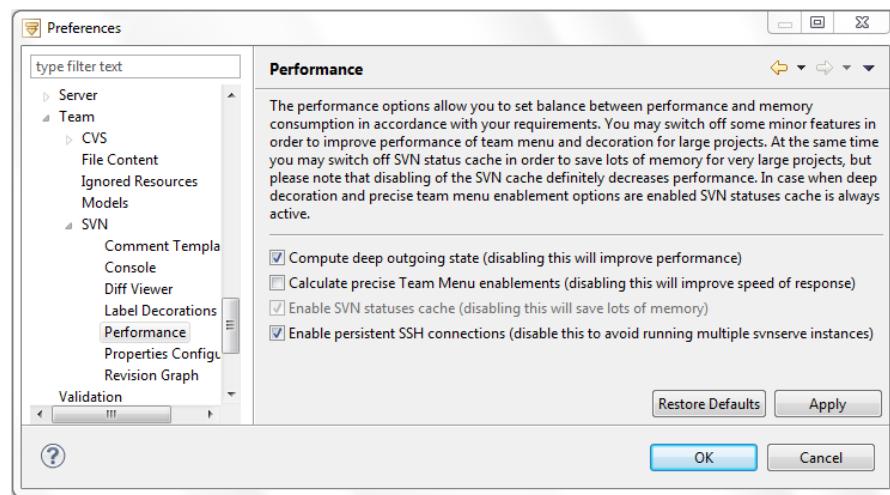
Also, when having multiple workspaces, you should choose different colors for different workspaces to make them better distinguishable. For example, change Basic Decoration Color to "green".

Go to Window | Preferences | General | Appearance | Colors and Fonts.

Figure 8. Intershop Studio Colors and Fonts Preferences

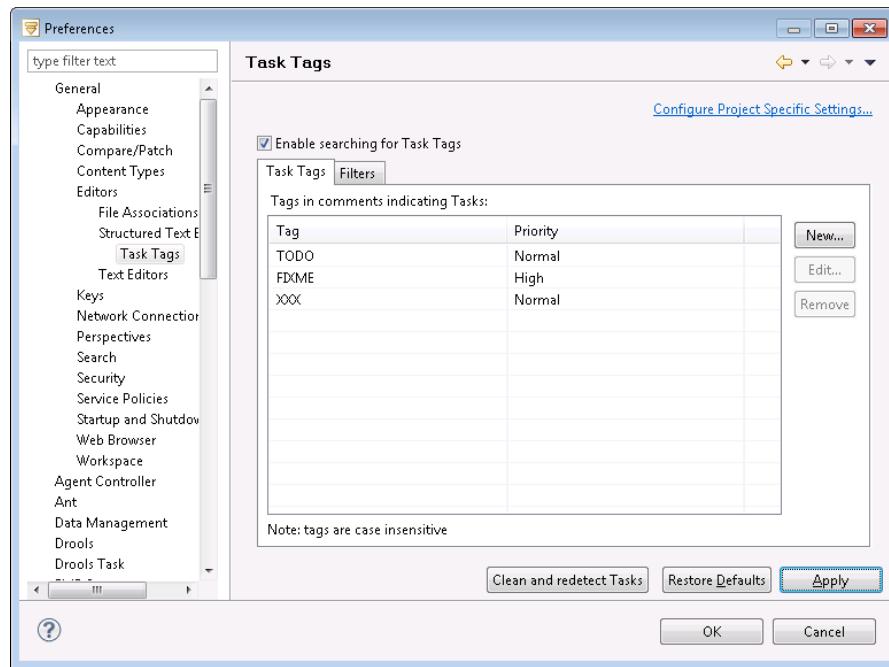
4. Go to Window | Preferences | Team | SVN | Performance.

Make sure, that the "Compute deep outgoing state" feature is enabled.

Figure 9. Intershop Studio SVN Performance Preferences

5. Go to Window | Preferences | General | Editors | Structured Text Editors | Task Tags.

To make the //TODO comment tag globally available and not only in Java source code, mark the "Enable searching for Task Tags" checkbox.

Figure 10. Intershop Studio Task Tags Preferences

Optimize Intershop Studio Performance

In order to improve the performance of Intershop Studio in general and the responsiveness of the Intershop Studio GUI in particular you can do the following:

■ Disable automatic builds

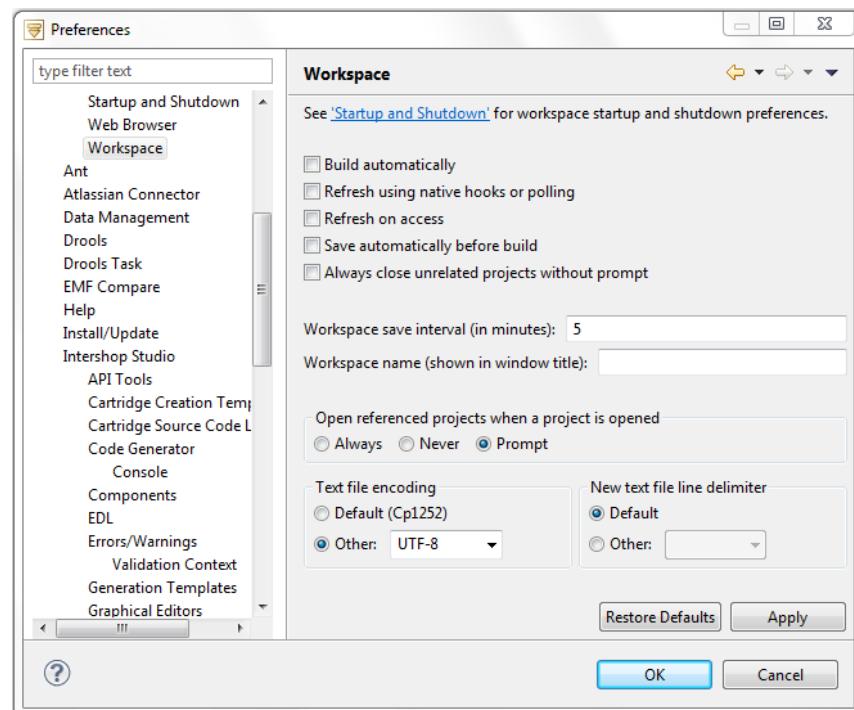
Intershop Studio default behavior is to automatically build a project when a resource file has been modified and saved. For this Intershop Studio uses its own build scripts and not the ones from the build environment. To save system resources you can disable the automatic build in the Intershop Studio preferences dialog.

If the automatic build is disabled, Intershop Studio does not automatically update warning and error markers in the code (including proposed quick fixes). To update these markers after modifying your code, build your project manually as described in *Building a Cartridge*. To update these markers for a single resource only (such as a pipelet or a pipeline), right-click the element in the Cartridge Explorer to open the context menu and select Check Element.

To disable automatic builds, do the following:

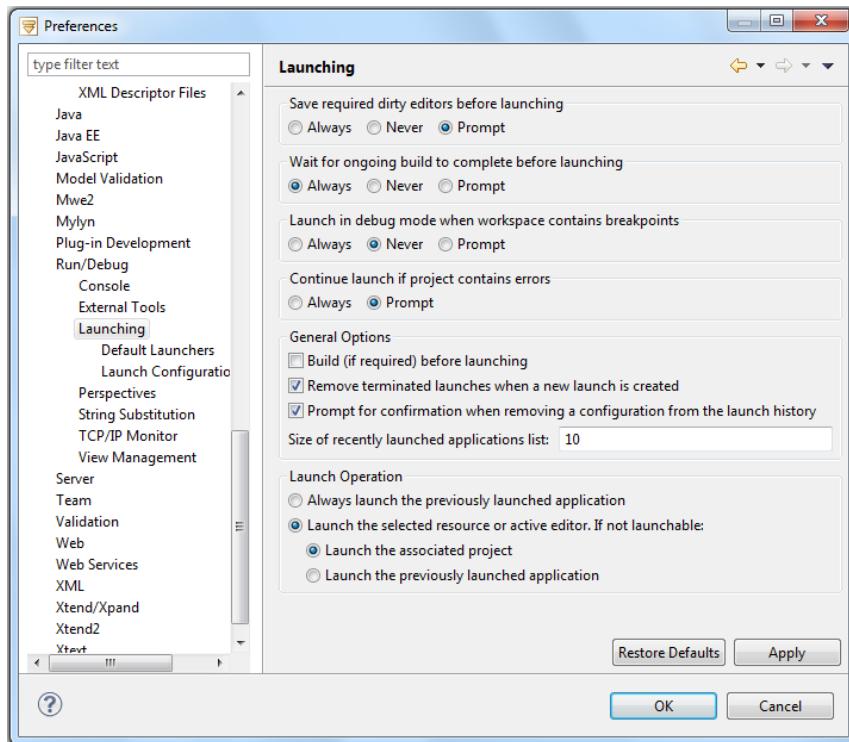
1. **Open the Intershop Studio preferences dialog (Window | Preferences).**
2. **Go to General | Workspace and uncheck the Build automatically checkbox.**

Figure 11. Workspace Preferences



- 3. Go to Run/Debug | Launching | General Options and uncheck the "Build (if required) before launching" checkbox.**

Figure 12. Launching Preferences



■ Allocate more RAM

The Intershop Studio default memory settings may be too low for your setup. To prevent running out of memory, you can increase the amount of RAM Intershop Studio has at its disposal.

Go to the <IntershopStudio> directory and edit the `IntershopStudio.ini` file. In this file, change the settings for minimal memory usage (`Xms{MB}m`), maximum memory usage (`Xmx{MB}m`), and the maximum memory for permanent generation of objects in the VM (`XX:MaxPermSize={MB}m`).

For example, for a 64bit operating system with 8GB RAM you could use the following values:

```
-Xms768m
-Xmx2408m
-XX:MaxPermSize=1536m
```

Make sure that the `XX:MaxPermSize` value lies between the other two values.

Updating Intershop Studio

For customers with a valid support contract, Intershop provides regular updates of the Intershop Studio development platform via its support Website.

There are two options to upgrade an Intershop Studio installation:

- Download a complete Intershop Studio from the Intershop support Website (<http://support.intershop.com>), then install and configure as described above (see *Installing and Configuring Intershop Studio*).
- Use Intershop Studio's built-in Update Functionality which connects to Intershop's update site.

This site can be used to update an already installed Intershop Studio with the latest features or to turn a bare eclipse installation into an Intershop Studio.

For the latter, do the following:

1. In Intershop Studio go to Help | Install New Software to open the update dialog.

2. Set general update preferences.

It is recommended to use the following settings:

- Show only the latest versions of available software (checked)
- Group items by category (checked)
- Contact all update sites during install to find required software (not checked)
- Hide items that are already installed (not checked)

3. From the "work with" drop-down list, select the pre-configured link to Intershop's update site. If in doubt, ask your Intershop support for the correct URL.

4. Enter login credentials.

The site will ask you for your login credentials. Enter a valid support contract number and a password, then click OK.

5. Select the components you wish to update, then click Finish.

Intershop Studio contacts the selected update site and shows components that are available for download.

NOTE: Warnings about downloading unsigned contents can be ignored.

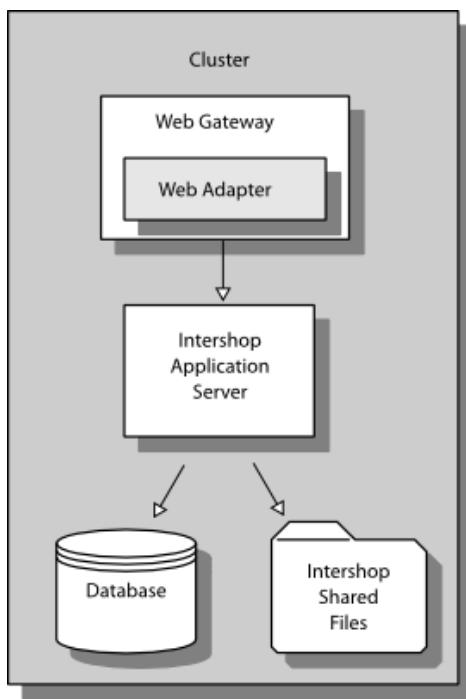
Intershop 7 Architecture

This section gives an overview of the software components, concepts, and code artifacts you need to know when writing applications for the Intershop 7 platform.

We first look at the stack of software components that make up a complete Intershop 7 system. We then drill down into the different architectural layers comprising each Intershop 7 application. And finally, we will discuss the main code artifacts and how they interact with each other in an application.

Cluster Perspective

A Intershop 7 cluster consists of the software components depicted in the following diagram:

Figure 13. Cluster View

These components include:

■ **Web Gateway (WGW)**

The Web Gateway component comprises a Web server and the Intershop 7 Web adapter.

The Web Gateway component dispatches HTTP client requests and responses between client applications (e.g. a user's Web browser) and the Intershop 7 application server.

■ **Application Server**

Provides the runtime environment for Intershop 7 cartridges.

■ **Database**

One possible backend for persisting business data.

■ **Intershop Shared File System**

An instance of Intershop 7's file-based data storage system that holds all the cartridges comprising an Intershop 7 application. It stores your applications business logic, pipelines, pipelets, and templates.

It also stores cluster-wide information necessary for running an application server.

Depending on the chosen installation scenario, the different Intershop 7 components may be installed on a single physical machines or distributed over multiple ones.

A single machine installation is the simplest and most straightforward installation scenario which is especially suited for development and testing purposes. In a production environment each component is usually installed on its own dedicated hardware.

The next section takes a closer look at the different architectural layers of each Intershop 7 application.

Architectural Layers

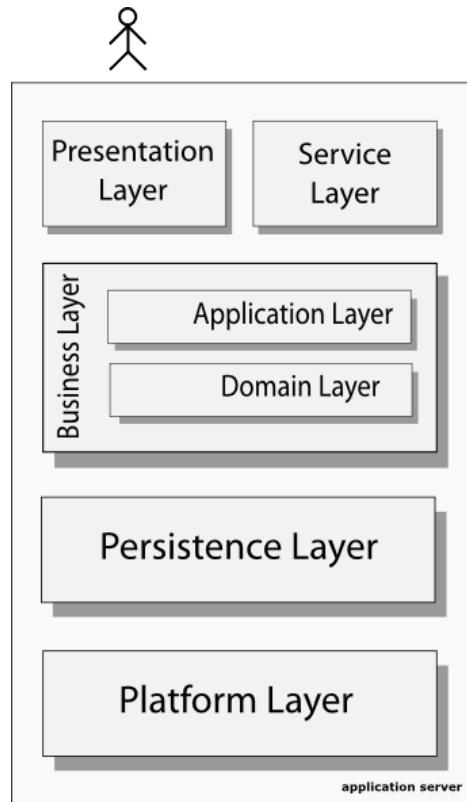
When developing applications for the Intershop 7 platform, you will create and modify code artifacts of various types, such as persistent objects (POs) for storing and retrieving business data from a database, business objects (BOs) for handling, manipulating, and processing business data, pipelines and pipelets to describe the application's business flows and processes, and templates for rendering business data into presentation output.

All these different artifacts comprising a Intershop 7 application can be assigned to different architectural layers where each layer has a distinct function, is populated by distinct artifacts, and exposes well-defined APIs.

The main advantage of separating functionality into different layers is that implementation details can be hidden from other layers allowing for ease of development, ease of code maintenance, and code reuse through software modularization.

In Intershop 7 applications we can distinguish between the architectural layers as depicted in the following diagram:

Figure 14. Intershop 7 Architectural Layers



These layers include:

■ Platform Layer

Provides technical functionality that is independent from certain business features. Functionality provided by the platform layer may be utilized by all other layers and includes the following:

- pipeline engine
- template engine
- servlet engine
- logging facilities
- cartridge startup facilities

■ Data/Persistence Layer

Contains POs, Queries, SQL scripts, and CAPI interfaces for POs.

POs provide the interface to the database. They model data that is persistently stored in the database and make these data available to higher order programming layers, namely the business layer.

SQL scripts are used to define foreign keys, constraints, indexes, etc. for the database.

■ Business Layer

The business layer defines the main API to be used by programmers writing applications on top of the Intershop 7 core product. The API models important domain concepts such as products, categories, or baskets and hides internal implementation details, like the mapping to the underlying persistence layer.

Whereas the persistence layer API must be optimized for maximum performance, the business layer API can be optimized for usability.

While the persistence layer implementation may be subject to frequent changes in order to improve its performance, the business layer API can remain stable.

The business layer defines generic business objects and their relations with each other and maps those business objects, their attributes, and their relations to the underlying data layer.

Additionally, the business layer comes with mechanisms to extend and customize the functionality of generic business objects for specific application needs. Therefore, the business layer can be logically subdivided into a domain layer and an application layer.

The *domain layer* defines reusable generic concepts belonging to a certain business domain. Examples for such generic concepts might be baskets, products, categories, or users.

These generic concepts are made available in form of interface classes and their implementations, e.g. the `bc_product` cartridge provides the class `ProductBOImpl` which is one possible implementation of the `ProductBO` interface.

The domain layer also defines related generic pipelets, e.g. a pipelet that calculates basket prices.

On the *application layer* the generic concepts defined on the domain layer can be extended and customized to meet the requirements of special use cases

and business domains. For example you might specialize the generic business object `ProductBO` to create a specialized business object `FooProductBO` with extra functionality that is only required in your specific business domain, e.g. the fashion or the automotive industry.

The application layer provides BO extensions, application-specific pipelets, and process pipelines.

■ **Presentation Layer**

The presentation layer contains functionality needed for rendering business data supplied by the underlying business layer into a presentation consumable by users. Such a presentation might be an HTML page as part of an online shop Web front, or the text of an automatically generated email message.

The presentation layer provides templates, view pipelines, pipelets, webforms, static content (e.g. CSS, images), and pagelets.

View pipelines are used to define the page flow of your Web application. Templates contained in view pipelines can access the data provided by pipelines. They define the layout and rules for rendering the data into presentational output.

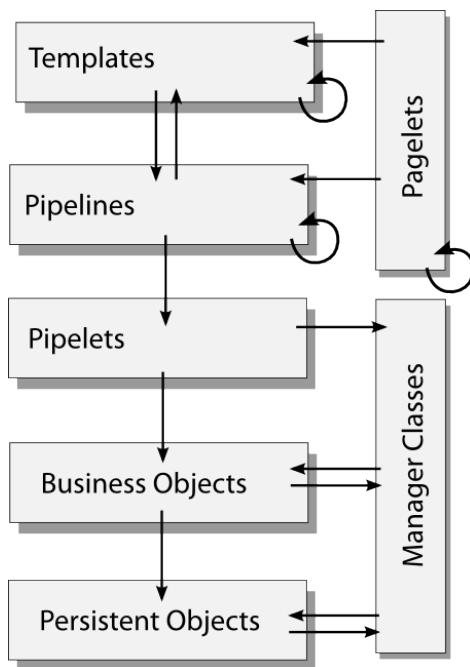
■ **Service Layer**

The service layer provides functionality for transforming business data supplied by the underlying business layer into a presentation consumable by other external computer systems. External systems may access the data using Web services conforming to standards like SOAP and REST.

The service layer contains artifacts like WSDL files, web service pipelines, and their related pipelets.

Programming Artifacts

When viewing an Intershop 7 application from a development perspective, we can distinguish between different layers of programming artifacts as depicted in the following diagram:

Figure 15. Intershop 7 Programming Layers

Programming artifacts include:

■ **Persistent Objects**

An Intershop 7 application operates on business data that is stored in a relational database. Currently Intershop 7 supports Oracle Database 11g as its persistent backend.

Persistent objects (POs) provide an object-oriented view on the data stored in the database.

■ **Business Objects**

Business objects

■ **Pipelets**

Based on business objects, pipelets can be programmed to solve a particular business function, such as determining the price of a product in a basket.
Pipelets are the building blocks of pipelines.

■ **Pipelines**

Pipelines are used to connect a series of pipelets and other pipeline nodes (e.g. flow control nodes) and describe business processes.

Pipelines can be dedicated to processing business data using the functionality provided by pipelets (process pipelines) or be used to transform business data into presentation output using templates (view pipelines).

■ **Templates**

Templates are invoked by view pipelines to present data to users. Templates may contain HTML, ISML, and references to static content (e.g. CSS and images). Via ISML object path expressions, templates can access data provided by the invoking view pipeline.

Intershop 7 Test Framework

What Is a Unit Test?

A unit test is a well-structured set of tests that exercises the methods of a Java class, or the functionality of a whole pipeline.

The Intershop 7 test environment is based on the JUnit framework and the Intershop 7 eTest cartridge:

■ **JUnit Framework**

Is a simple framework to write repeatable tests. Each test case is executed within a running application server in the same environment as the Intershop 7 business logic. Using the JUnit framework, you can focus on the test cases themselves before investing a lot of time with developing the actual test environment. See *The JUnit Test Framework* for more information. You can find an extensive coverage of the framework at www.junit.org [<http://www.junit.org>].

■ **Intershop 7 eTest Cartridge**

In Intershop 7, the unit test environment is implemented as a single cartridge called eTest. New test cases are always developed on the cartridge level. Each cartridge on your system provides its own test cases.

NOTE: The eTest cartridge is based on JUnit 3.8.1, which is provided by Intershop 7. Other JUnit versions should not be used.

Why Run Unit Tests?

There are several reasons to create and run unit tests. Running unit tests, you can:

■ **Measure the development progress**

Unit tests show if certain Java classes are working properly. Tests can be created for each class on your system. However, they can also run as a group of tests to regularly track the progress of your project and locate program bugs.

■ **Improve the reusability of tests**

Every test is exactly the same. You must be able to run tests multiple times.

■ **Show how to use Java classes**

If necessary, a unit test helps redefine the requirements for individual classes.

■ **Design Java classes so that they can be maintained easily**

If you want unit tests to run successfully, follow these guidelines:

- Run unit tests regularly
- Dedicate enough time to writing unit tests
- Refactor your code, if necessary

When you remove redundancies, eliminate unused functionality or recreate designs that you are refactoring. Refactoring improves the design of your code without adding new behavior. It helps to restructure code to reduce or remove duplication, and simplify the complicated. This keeps your code clean and concise so that it is easier to understand. Refactoring saves development time and increases quality.

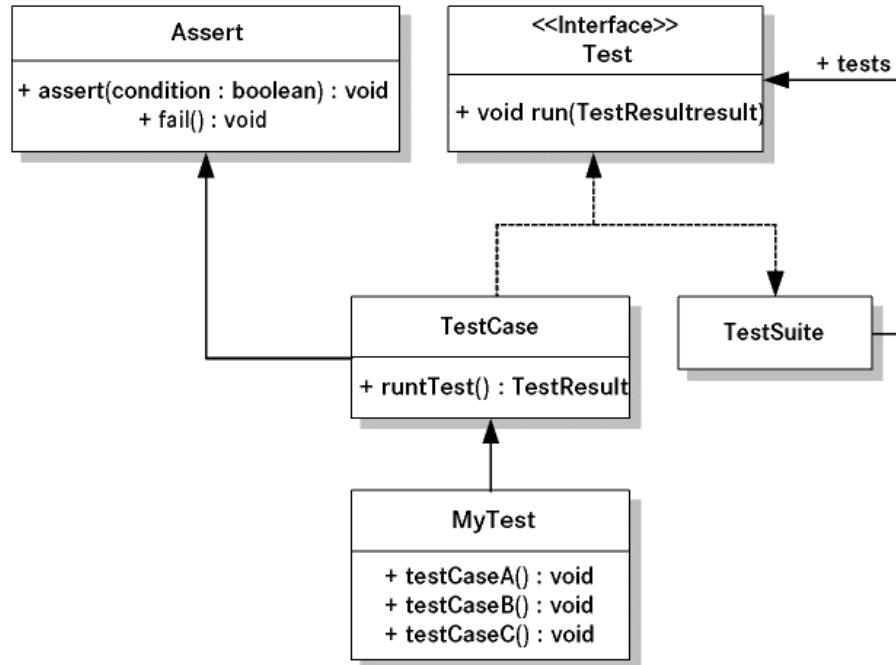
The JUnit Test Framework

JUnit is a simple unit-testing framework that makes developing for Intershop 7 a lot easier and helps to avoid errors. It consists of a number of Java classes that you can use to create and run unit tests.

The following two sections provide a brief overview about the main classes of the JUnit framework (see Overview, below) and the available TestRunner versions. The Intershop TestRunner is the main interface of the JUnit framework used to run tests.

The following figure shows a class diagram of the core of the JUnit framework:

Figure 16. Overview of JUnit Framework Classes.



The following table explains the classes:

Table 1. JUnit Framework Classes

Class	Description
Test	Is the interface that all types of test classes, e.g., TestCase and TestSuite , must implement.
TestCase	Is the core class of JUnit. It implements the Test interface and provides the methods to set up the test conditions, run the tests, and collect the test results. TestCase is the class you are extending when you write your tests. JUnit can run either a single test case or multiple test cases, and report the results in a TestResult .
TestSuite	Is an object provided by JUnit that allows you to run several tests at once. It implements the Test class and contains several test cases, other test suites, or any combination so that they can be run as a unit.
Assert	Is the super class of TestCase that provides all of the <code>assert()</code> methods that are available to you when you write your tests. The logic behind an <code>assert()</code> method is always the same: <code>assert("message if declaration is false", declaration);</code> . Because TestCase inherits from the Assert class, you can call the <code>assert()</code> method anytime during a test sequence.

Package Naming Conventions

Using the components mentioned above, you can test single units (classes, methods), features, or third-party products. To simplify unit tests, we recommend to adhere to the following naming conventions for packages:

```
<package name prefix>.<package name of tested element>
```

The following table lists all predefined package name prefixes:

Table 2. Package Naming Conventions

Package Prefix	Description
tests.units	Test specific units, e.g., classes, methods, pipelines, persistent objects
tests.features	Test specific features
tests.external	Test third party products, e.g., Oracle, PowerTier

After being packaged into a Java archive, Intershop recommends to use the name `testclasses.jar` as an archive name. However, you can also use other name schemes for Java archives to scan for, e.g., `*_tests.jar`. Use the `itestrunner` file in `<IS_HOME>\share\system\cartridges\etest\release` for configuration.

The appropriate section reads as follows:

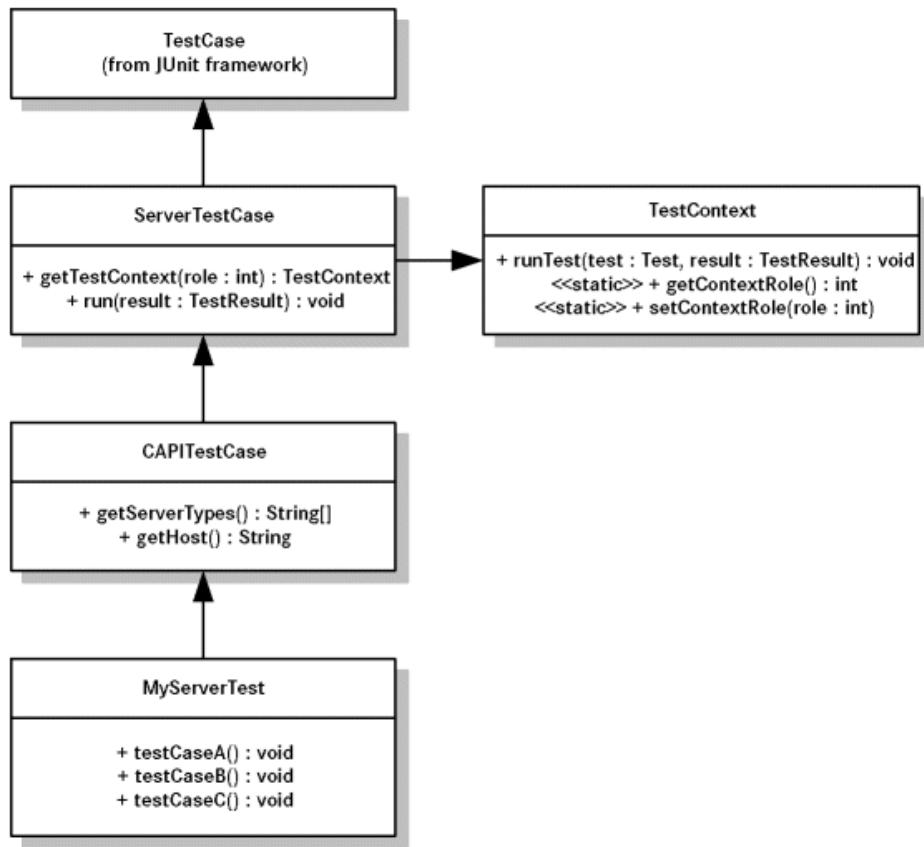
```
#cartridge scanner settings
scanning mode=test
.testjars=_tests.jar;testclasses.jar
```

If you want to test a specific unit within the `com.mycompany` package, for instance, you must create a test class that is part of the `tests.units.com.mycompany` package.

The Test Scenario

There are two scenarios for running test cases: a server-side and a client-side test scenario. This document focuses on server-side tests using the functionality of the eTest cartridge. Server-side tests are run in a full-blown application server environment which gives you access to typical server functionality, e.g., persistent objects or pipelines.

The following figure shows the main components of the server-side test scenario:

Figure 17. Components of a Server-Side Test.

The **CAPI TestCase** class is the common base class for server-side tests. It is implemented by the abstract **ServerTestCase** class and checks the given context of a specific test. The **CAPI TestCase** class hands the execution of a test case over to a **TestRunner** pipeline, which is part of the eTest cartridge.

The **TestRunner** pipeline is called using the following URL:

```
http://<server>/INTERSHOP/web/<server_group>/<site>/-/ /...  
TestRunner-Start?TestClassname=<test_class>&TestCase=<test_case>
```

The URL contains the following placeholders:

- `<server>` = the local host
- `<server_group>` = WFS
- `<site>` = eTest
- `<test_class>` = the test class
- `<test_case>` = the test case method

These methods are evaluated using the JUnit framework and Java reflection. The framework uses reflection to find and collect all of the test methods whose signatures match:

```
public void testWhatever()
```

PA-DSS Compliance

By default, Intershop 7 complies with the "Payment Application Data Security Standard" (PA-DSS), which constitutes a global security standard created by the Payment Card Industry Security Standards Council. PA-DSS has been implemented in an effort to provide the definitive data standard for software vendors that develop payment applications. The standard aims to prevent developed payment applications for third parties from storing prohibited secure data including magnetic stripe, card verification code/value, or PIN. In order to keep any customizations, extensions, etc. compliant with PA-DSS, Intershop strongly recommends to consider the requirements summarized in this section when developing with Intershop 7.

NOTE: For more information about PA-DSS and its requirements, refer to the *Intershop 7 PA-DSS Implementation Guide* and to <https://www.pcisecuritystandards.org/>.

Sensitive Authentication Data

Do not store sensitive authentication data after authorization (even if encrypted).

Sensitive authentication data includes the data as cited below.

■ Card verification value/card verification code

After authorization, do not store the card verification value or code (three-digit or four-digit number printed on the front or back of a payment card) used to verify card-not-present transactions.

■ Debugging or troubleshooting

Securely delete any sensitive authentication data (pre-authorization data) used for debugging or troubleshooting purposes from log files, debugging files, and other data sources received from customers, to ensure that magnetic stripe data, card verification codes/values, and PINs or PIN block data are not stored on software vendor systems. These data sources must be collected in limited amounts and only when necessary to resolve a problem, encrypted while stored, and deleted immediately after use.

■ Primary account number

Mask primary account numbers (PAN) when displayed (the first six and last four digits are the maximum number of digits to be displayed). Render PAN unreadable anywhere it is stored (including data on portable digital media, backup media, and in logs) by using any of the following approaches:

- One-way hashes based on strong cryptography (hash must be of the entire PAN)
- Truncation (hashing cannot be used to replace the truncated segment of PAN)
- Index tokens and pads (pads must be securely stored)
- Strong cryptography with associated key management processes and procedures

Key Store

Payment application must protect any keys used to secure cardholder data against disclosure and misuse. Payment application must implement key management processes and procedures for cryptographic keys used for encryption of cardholder data.

Intershop recommends to use the JavaTM Cryptography Extension to implement cryptographic processes.

■ Strong cryptographic key generation

Use the *PBEWithMD5AndDES* algorithm to generate a *SecretKey* object from the provided key specification.

■ Secure cryptographic key storage

Use a JCEKS type Java key store to store keys and protect the store with a password composed of three parts.

Audit Trail

Log messages for all security-sensitive operations.

■ Logged events

Log a message for the following events within the category "audit" using the info log level:

- All individual accesses to cardholder data from the application
- All actions taken by any individual with administrative privileges as assigned in the application
- Access to application audit trails managed by or within the application
- Invalid logical access attempts
- Use of the application's identification and authentication mechanisms
- Initialization of the application audit logs
- Creation and deletion of system-level objects within or by the application

■ Logged information

Record at least the following information for each event:

- User identification
- Type of event
- Date and time
- Success or failure indication

Payment Information Transfer

When sending cardholder data over public networks, make sure to use strong cryptography and security protocols (for example, SSL/TLS, Internet protocol security (IPSEC), SSH, etc.) to safeguard sensitive cardholder data during transmission over open or public networks.

Cartridge Development

Cartridges

What Are Cartridges?

In Intershop 7 the term "cartridge" refers to software module that encapsulate Java classes, persistent objects, pipelets, pipelines, templates and static content.

Cartridges are the building blocks and deployment containers of an Intershop 7-based application. Cartridges can be installed (deployed) on a Intershop 7 server in order to make the functionality implemented by the application units available on the server.

Thus cartridges are the standard mechanism for packaging and deploying Intershop 7 applications.

Any objects and entities defined at the different architectural layers (persistence layer, business object layer, presentation layer) are distributed over a set of cartridges.

The cartridge concept provides a uniform and easy-to-handle way of integrating new code components into Intershop 7. All new applications to extend and/or customize Intershop 7 must abide by the organizational principles that the cartridge concept imposes, no matter whether the application is simple or complex.

Cartridge Components

A cartridge bundles all components that are necessary for adding business logic into one portable and easy to isolate package. Cartridges specifically contain the following code artifacts:

- Java code (such as business objects, persistent objects, provider classes, or pipelets)
- Templates
- Pipelines
- Localization resources
- Queries
- Webforms
- Web services

- Images and static HTML files
- Descriptive and version information, and other resource files
- A cartridge configuration file

From a file system perspective cartridge components are organized within the following folders:

```
/build
/edl
/javasource
/javasource/resources
/model
/setup
/staticfiles
/staticfiles/cartridge
/staticfiles/cartridge/components
/staticfiles/cartridge/config
/staticfiles/cartridge/impex
/staticfiles/cartridge/lib
/staticfiles/cartridge/localizations
/staticfiles/cartridge/pipelines
/staticfiles/cartridge/queries
/staticfiles/cartridge/static
/staticfiles/cartridge/templates
/staticfiles/cartridge/urlrewrite
/staticfiles/cartridge/webforms
/staticfiles/cartridge/webservices
/staticfiles/share
```

From a Java package perspective cartridge components are grouped into the following packages:

- <cartridge base package>
Contains the optionally subclass of
`com.intershop.beehive.core.capi.cartridge.Cartridge`.
- <cartridge base package>.capi
Contains the public programming interface.
- <cartridge base package>.internal
Contains non public implementation.
- <cartridge base package>.dbinit.preparer
Contains database preparer for public use.
- <cartridge base package>.dbinit.data
Contains database settings required to run the cartridge.
- <cartridge base package>.pipelets
Contains pipelets. JavaDoc is generated for capi, preparers, rapi and pipelets.

Cartridge Definition

Each cartridge in an Intershop 7 system is defined by its cartridge properties file named `<cartridge_name>.properties`, where `<cartridge_name>` is the cartridge name in the `<IS_HOME>/share/system/config/cartridges` directory. For example, the core cartridge properties file is called `core.properties` and it contains the

cartridge properties for the core cartridge. The cartridge properties file contains two different types of information:

- A fixed set of general parameters which every cartridge must provide
- A variable set of cartridge specific parameters

General Information

This includes basic properties that define the cartridge. The basic properties are named and described in *Table 3, "General Information"*. Note that each property has the prefix `intershop.cartridges.<cartridge_name>`.

Table 3. General Information

Property	Description
<code>displayname</code>	Name of the cartridge for display purposes. The internal cartridge name remains constant and is identical to the cartridge directory name, making an alternative display name useful.
<code>build</code>	The internal build number of the cartridge.
<code>version</code>	The external version string for display purposes.
<code>classname</code>	The full name of the cartridge controller class. The cartridge manager instantiates this class during startup.
<code>archive</code>	Specifies the Java archive file containing all class files for primary cartridge components. A different archive is used for the class files used in database preparation.
<code>dependson</code>	Specifies cartridges that must be available for the current cartridge to work properly
<code>readonly</code>	Defines whether the cartridge modifies the database (false) or not (true).

Example content from the `bc_marketing.properties` file is listed below:

```
intershop.cartridges.bc_marketing.displayname = Component - Online Marketing
intershop.cartridges.bc_marketing.build = 242
intershop.cartridges.bc_marketing.version = 6.1.0.0
intershop.cartridges.bc_marketing.classname = com.intershop.component....
    marketing.MarketingCartridge
intershop.cartridges.bc_marketing.archive = bc_marketing.jar
intershop.cartridges.bc_marketing.dependson = core xcs bc.foundation bc_mvc
intershop.cartridges.bc_marketing.readonly = false
```

Cartridge Specific Information

Besides general information, the cartridge properties file may provide various cartridge specific settings. Some typical settings are listed in the following table:

Table 4. Cartidge-specific information

Property	Description
<code>Deletion Properties</code>	Defines pipelines used to remove domains or users.
<code>Staging Configuration</code>	Defines domain-specific and global tables to include in the data replication process.
<code>Staging Foreign Key Definitions</code>	Defines foreign keys of staging tables.

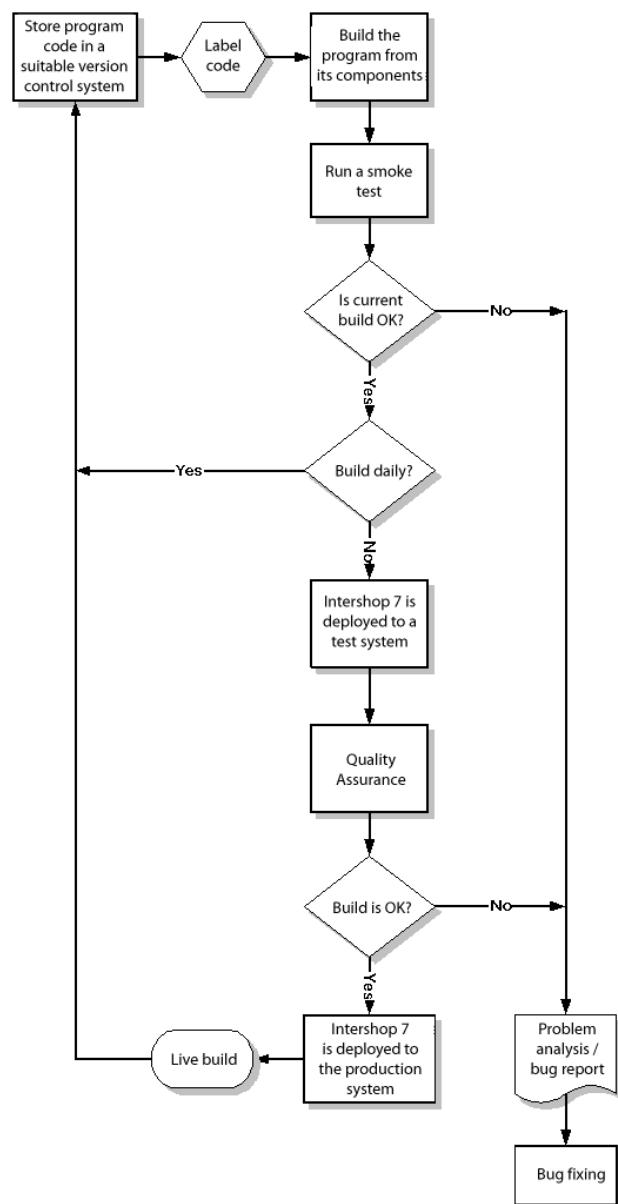
Cartridge Controller Class

Each cartridge is associated with a cartridge controller class which is stored in the folder <IS_SOURCE>\<cartridge_name>\javasource\com\...\<cartridge_name>. The controller class is instantiated by the cartridge manager on start-up. It defines initialization and shutdown methods. If your cartridge does not require special treatment during start-up or shutdown, you can use the pre-defined controller class com.intershop.beehive.core.capi.cartridge.Cartridge.

Cartridge Development Cycle

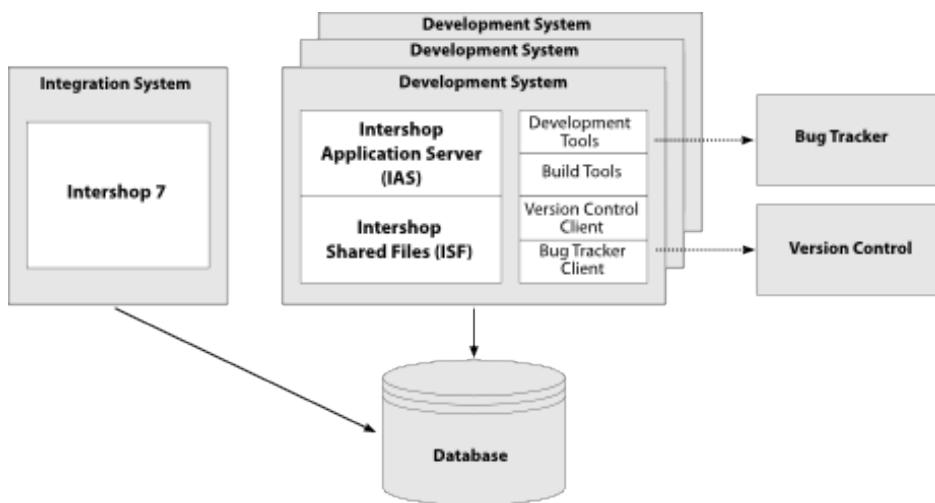
Overall Development Process

A typical development project comprises the main steps outlined in *Figure 18, "Overall development cycle"*.

Figure 18. Overall development cycle

Components of an Intershop 7 Development Environment

The Intershop 7 development environment is comprised of several components, as shown in *Figure 18, "Overall development cycle"*. Set up the same development environment on each development machine.

Figure 19. Components of a development environment

Development System

Install Intershop 7 locally on all development machines. This allows developers to work independently from each other. Each development system needs an application server instance and a local shared file system.

Database

Use a central database if possible. Each developer must have his or her own database account on the database server. If the database is administered from a central point in your system, the hardware requirements for the development machines are significantly lower than in a system with local database installations on each development machine.

NOTE: You can connect to a remote database to set up a development environment. You must, however, use a local application server instance and a local shared file system.

Integration System

Use a separate machine for testing purposes, for example to carry out integration and performance tests. Install an Intershop 7 cluster on that machine for testing purposes only. This test system should resemble the structure of the production system. It should consist of at least two, or even three application servers.

Build Scripts

Build scripts are an important part of the development environment. These scripts are contained in build files stored in the cartridge development build directory, e.g., <cartridge_name>\build. Each cartridge has the same set of build files. If you create a new cartridge, Intershop Studio automatically provides you with the build files and sets up the build directory.

Overview of the Cartridge Development Cycle

Cartridges may vary considerably with respect to their actual content and their complexity. All cartridge development, however, follows a general and unified

development process, no matter whether the cartridge functionality is simple or complex, generic or site-specific in nature.

The cartridge development cycle, which will be discussed in detail in this chapter, can be divided into four stages:

Step 1: Planning the Cartridge

Before embarking on a development project, the cartridge should be carefully planned. At the beginning, you should focus on some general questions. For example, you need to exactly delineate the functionality of the entire application and decide whether to package the entire application within a single cartridge or to distribute the functionality across a set of cartridges. You should also carefully examine which existing Intershop 7 functionality your project can be based on. See *Planning a Cartridge Project* for some hints.

Step 2: Creating the Cartridge Project

The actual development process starts out with several steps to create the cartridge project. These steps include:

- Creating the cartridge development directory structure
- Checking (and perhaps modifying) environment variables
- Declaring cartridge properties
- Writing the cartridge controller class

Step 3: Programming Cartridge Functionality

Programming new business functionality is at the heart of cartridge development. Programming typically cuts across several programming layers:

- Programming new business objects and manager classes
- Programming new pipelets
- Designing/Modifying templates
- Configuring pipelines

Step 4: Building and Deploying the Cartridge

For new business functionality to become available on a server, it needs to be built, packaged, and deployed. The cartridge build and deployment process comprises the following steps:

- Declaring pipelets made available by the cartridge
- Assembling additional resources required by the cartridge
- Building the cartridge
- Deployment and installation

Planning a Cartridge Project

When starting the development of a new application on the Intershop 7 platform it is recommended to identify required cartridges at the beginning of the development process.

Before embarking on a development project, you should carefully plan and design the cartridge. Careful planning and design will save development time and resources, and significantly contribute to the ultimate success of the project.

The following tasks are part of the design stage of a cartridge project:

■ **Define cartridge functionality**

Complex applications typically consist of more than one cartridge. It is essential to clearly define the overall functionality of the application and how the functionality is to be distributed across individual cartridges. Hence, cartridge development always starts out with a detailed and accurate definition of the functionality that the new cartridge is going to deliver. See *Cartridge Layer Model*, for a model that can provide orientation.

■ **Examine existing functionality**

Once the cartridge functionality has been defined, you should carefully examine connections to existing Intershop 7 functionality. The goal of this exercise is to identify where exactly the cartridge plugs in, and whether components of the functionality you want already exist. Is it really necessary to develop a new cartridge, or would it perhaps suffice to reorganize an existing pipeline? Do you need to extend functionality, or does the core functionality already exist, with only a few modifications necessary?

■ **Determine dependencies**

All cartridge projects are going to depend on functionality provided by other cartridges. For example, every cartridge depends on basic functionality implemented in the core cartridge, such as the cartridge controller class or the base classes for pipelets and business objects. If complex business functionality is implemented, typically many more dependencies exist. Check these dependencies carefully and make sure that all required cartridges are available on the development system.

■ **Define localization requirements**

It is important to be clear about the localization requirements of a cartridge. The cartridge should seamlessly integrate with Intershop 7's mechanisms for localization and internationalization.

Cartridge Layer Model

Cartridges are building blocks and deployment containers that encapsulate Java classes, persistent objects, pipelets, pipelines, templates and static content.

Cartridges may be classified by the need they fulfill. There are cartridges that implement user interfaces, deal with certain persistent objects or provide an integration with a backend system. Based on these characteristics cartridges can be categorized and described according the cartridge layer model as described below.

When developing on the Intershop 7 platform, the cartridge layer model can provide help in identifying and grouping functionality into cartridges. A well-thought cartridge design is the fundament for reusability and ease of development and maintenance.

In each layer multiple cartridges may reside. It is also possible that in a certain scenario no cartridge is developed for a certain layer. The layers describe the characteristics of cartridges and provide design guidelines. From a technical

perspective, the assignment of a cartridge to a certain layer is a matter of structuring and good design and not a technical necessity.

The different cartridge layers have the following characteristics:

■ Platform Layer

The platform layer includes all cartridges that represent the Intershop 7 core product.

■ Adapter Component Layer

Adapter components provide an interface or implement an integration with an external system or data source/sink. For example, an adapter component cartridge may implement an online connection to an ERP system, may provide pipelines and templates to export business objects in a specific format or may import data from a multimedia system.

In general, an adapter component should be implemented generically to allow for reuse in multiple solutions and projects. So the adapter must not contain any solution or project specific code or visualization. A single adapter component should always implement an integration to one external system only.

Adapter component cartridges typically contain pipelets, if necessary entity beans to store configurations, SOAP interfaces provided by session beans, pipelines for complex processes or integrations, but no user interface templates.

Adapter component cartridges have names starting with the prefix ac_. Intershop recommends to use the following pattern for naming Java packages: com.<mycompany>.adapter.<external_system_name>.

Example:

```
ac_siebel          // cartridge name  
com.intershop.adapter.siebel // Java package name
```

■ Business Component Layer

Business components typically implement processes and persistent objects. Good examples are a coupon cartridge or a cartridge providing special catalog extensions.

As for adapter components business components must be designed generically to allow for maximum reuse in multiple solutions and projects. Business components typically include additional persistent objects, manager classes for managing the persistent objects, pipelets, pipelines for complex processes, but no user interface templates.

Business component cartridges have names starting with the prefix bc_. Intershop recommends the following pattern for naming the according Java packages: com.<mycompany>.component.<business_component_name>.

Example:

```
bc_coupon          // cartridge name  
com.intershop.component.coupon // Java package name
```

■ Application Layer

An application is a view of the system with well-defined functionalities. An application always implements a user interface. An Intershop 7 solution or project will usually implement multiple user interfaces. Examples are the view of

the administrator to the system, the view of a storefront buyer or the view of an administrator of a buying organization.

Since user interfaces are implemented using pipelines, templates and static content the majority of pipelines, templates and static content will be found in application cartridges. Adapter components and platform cartridges are integrated using the pipelets and pipelines they provide.

Application cartridges typically bundle pipelines, templates and static content, rarely pipelets, and no persistent objects.

Intershop recommends to name application cartridges after the name of the project and the name of the view, separated by an underscore. Use the pattern `com.<companyname>.<projectname>.<viewname>` as naming pattern for the according Java package.

Example:

```
powershop_init          // cartridge name  
com.intershop.powershop.init // Java package name
```

■ Initialization Layer

A Intershop 7 system is initialized using the `DBInit` process. During that process the application/unit structure, initial business data, users, user groups and permissions etc. are created. The creation can be done using preparers, the site transporter and import processes. The role of the cartridges in the initialization layer is to combine and configure cartridges of other layers to form a solution or project implementation.

Cartridges in the initialization layer are only relevant during the `DBInit` process, as they are used as deployment containers. During runtime of the system these cartridges are not used.

Cartridges of the initialization layer should create or alter information within the context of the solution/project only. They should not alter global settings.

In the case of a solution cartridges of two different types can be found in the initialization layer: an init cartridge and a cartridge containing demo data.

While the init cartridge creates data structures that are always required for a solution, the demo cartridge provides sample data (e.g. sample buyers, sellers, products). So when implementing a project based on a solution the demo cartridge will usually be replaced by project specific initializations.

A solution will come with just one init cartridge and potentially multiple cartridges containing demo data (e.g. cartridges for multiple verticals).

In a project implementation that is not based on a solution just one initialization cartridge will be used. This cartridge will make sure that all project initializations are done and initial business objects are created.

Cartridges of the initialization layer usually include a long list of preparers to be triggered, optionally project specific preparers, parameterizations for the preparers (properties files) and import sources to create initial business objects (users, catalog data, etc.). Initialization cartridges may also deploy pipelines, templates and static content.

Intershop recommends the following naming scheme for initialization cartridges: project or solution name suffixed by `_init`. The Java package name is then `com.<companyname>.<projectname>.init`.

Example:

```
powershop_init          // cartridge name
com.intershop.powershop.init // Java package name
```

Base Cartridges

Two base cartridges are provided which store resources (such as Java code, pipelines and templates) that are shared between other Intershop 7 cartridges. The base cartridge concept reduces the need to copy identical resources across multiple cartridges. The following base cartridges exist:

- **sld_ch_base**

Base cartridge containing resources which are shared between at least two cartridges that serve as channel management plug-in for the sales or partner organization back office (`sld_enterprise_app`, `sld_ch_consumer_plugin`, `sld_ch_corporate_plugin`, `sld_ch_partner_plugin`, `sld_ch_information_plugin`).

- **buying_app_base**

Base cartridge containing resources which are shared between the business storefront (`sld_ch_corporate_app`) and the buying organization front end (`prc_buyer_site`, `prc_transaction_site`).

The following examples illustrate the usage of the base cartridges:

- Pipelines and templates of a feature which is equal in each channel are stored in the base cartridge. For example, managing order routing rules is equal in the Consumer, Business and Partner Channel. Therefore, all pipelines (except process pipelines) and templates of that feature should located in `sld_ch_base`.
- The base cartridges are invoked if a feature is only slightly different in each channel, but can be cleanly decomposed into reusable and specific parts. For example, the feature "Price Lists" is equal in the Consumer and the Business Channel, except that the user groups assigned to a price list are different. In a Consumer Channel, the user groups are represented by consumer groups. In a Business Channel, the Everyone user group of each organization is used. Hence, most pipelines and templates are located in `sld_ch_base`. Only the pipeline to determine the appropriate user groups are located in the respective channel management plug-in cartridges (`sld_ch_consumer_app`, `sld_ch_corporate_app`).
- Finally, the base cartridges are used to store identical template snippets which are shared across channels. The snippets can then be included by the templates which encode cartridge-specific functionality.

Package Structure

In addition to the naming schema for the cartridge base packages provided in *Cartridge Layer Model*, it is recommended to use the following sub-structure for cartridge components:

- <cartridge base package>
 - contains the optionally subclass of
`com.intershop.beehive.core.api.cartridge.Cartridge`
- <cartridge base package>.capi
 - contains the public programming interface

- <cartridge base package>.internal
contains non public implementation
- <cartridge base package>.dbinit.preparer
contains database preparer for public use
- <cartridge base package>.dbinit.data
contains database settings required to run the cartridge
- <cartridge base package>.pipelets
contains the pipelets

JavaDocs should be generated for the sub-packages `capi`, `preparers`, `rapi` and `pipelets`.

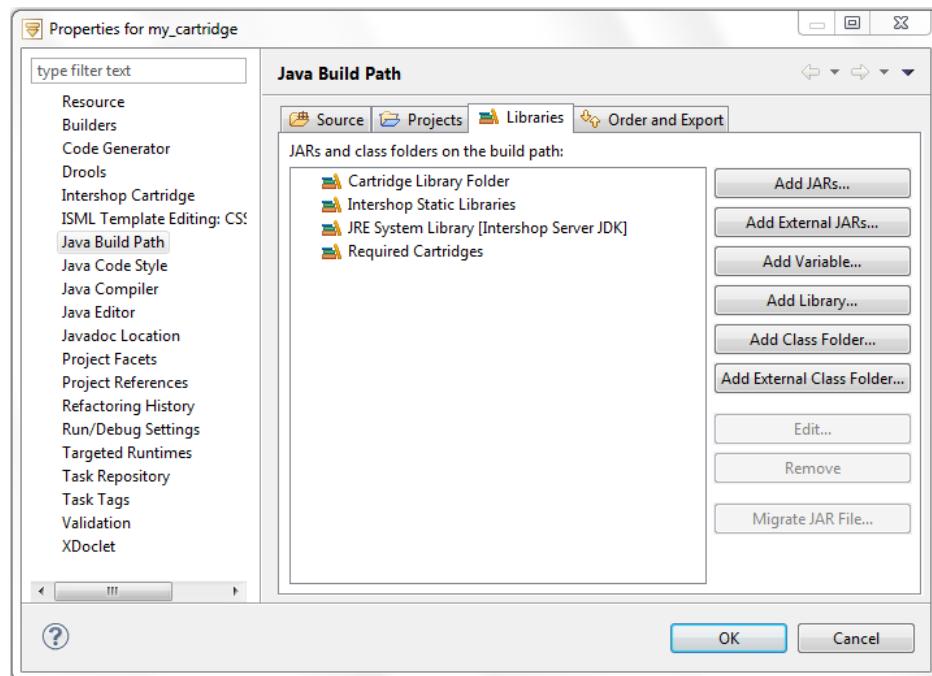
Cartridge Class Path Container

Intershop 7 cartridges provide a specific class path container, which includes all libraries of a cartridge. This container offers an efficient means to make Intershop 7 cartridge classes available to non-cartridge projects in Intershop Studio.

A wizard is available to ease adding cartridge libraries to a project build path.

- Right-click the cartridge.
Context menu opens.
- Select Configure Build Path... | go to Libraries tab | Add Libraries...
Add Library dialog opens.
- Select library type | set the required values | Confirm with Finish.

The wizard helps browsing for the required Intershop 7 cartridge and customizing its class path, if needed.

Figure 20. Adding cartridge library class path

Cartridges and Intershop Studio

There are two ways to start cartridge development using Intershop Studio. You can either set up a cartridge project from scratch or you can turn an already existing project into an Intershop Studio cartridge project. In both cases, the result will be that you have a cartridge project that allows you to modify and build the cartridge using the Intershop Studio environment.

To set up a cartridge project from scratch you use the Cartridge Wizard. The Cartridge Wizard will prompt you for the information required to set up the cartridge project, including general cartridge properties, cartridge dependencies, package configuration, and whether you want to overwrite the cartridge controller class. The Cartridge Wizard then creates the following components:

■ **Cartridge Development Directory**

The directory that you develop your cartridge in within the \javasource folder, including the appropriate package structure (see *Development Directory*). If requested, the wizard creates an Intershop 7 conform package structure containing additional sub-packages (see *Package Structure*).

■ **cartridge.properties file**

This file stores general cartridge properties, e.g. cartridge Id, cartridge name, cartridge dependencies list, etc (see *Cartridge Definition*).

■ **Cartridge Controller Class**

If requested, the wizard creates the cartridge controller class file, including method bodies for all life cycle methods you want to overwrite (see *Cartridge Controller Class*).

■ Build Files

A set of pre-configured ANT build files for cartridge-specific build tasks. See *Cartridge Build and Deployment* for details on the build process.

■ Links to external default libraries

By default, the following external libraries are linked to the cartridge project:

- any libraries found in the `/lib` folder of the cartridge
- libraries provided by cartridges that this new cartridge depends on.
- EJB-container libraries (if an EJB container is used)
- servlet engine libraries

If you already have a cartridge project, you can import the cartridge sources into Intershop Studio. It is important to keep in mind that you do not import the sources in the literal sense. In fact, what you do is make the cartridge project "known" to Intershop Studio and create an Eclipse project.

Cartridge Build and Deployment

Building the Cartridge

The build process makes cartridge components ready for deployment. It ensures that all components are compiled and packaged as required and stored in appropriate folders inside the cartridge's build directory. Depending on the exact content of the cartridge and the build configuration, the build process does the following:

- Creates the cartridge build environment (linking, copying templates and pipelines).
- Generates and compiles the cartridge's JAXB binding objects.
- Generates and compiles the cartridge's WSDL stubs.
- Compiles all source code components, creates the respective archive files and copies them to `<cartridge_build_dir>\release\lib`.
- Generates and compiles all necessary classes and files to publish Web services.
- Jars standard components.
- Generates the pipelet resource file.
- Generates the ORM object resource file.
- Pre-compiles and links ISML templates
- Creates JavaDoc for Java sources.

The ANT Tool

The build process for Intershop 7 cartridges is controlled and configured via the ANT tool. ANT stands for "Another Neat Tool" developed under the umbrella of the Jakarta Project. It is distributed freely and installed as default build tool under `<IS_HOME>\tools\ant`, including complete HTML-based documentation. For detailed information on ANT as part of the Jakarta Project, see <http://jakarta.apache.org/ant>.

The ANT tool is based on Java and XML. ANT uses XML build files to define and configure so-called targets, which in turn define a set of tasks to be executed during the build process. For example, the following sample sets up a target `compile.folder`. The target defines a task `<mkdir/>` with is further specified by XML attributes to create a directory. Note that the attribute values are partially defined via variables. Note also that because the variables that go into the `mkdir == tasks ==` need to be initialized first, the target `compile.folder` depends on another target (`compile.init`), i.e., it assumes that this other target has already been executed.

```
<target name="compile.folder"
        depends="compile.init"
        unless="release.lib.available">
    <mkdir
        dir="${is.target}/${component.name}/${foldername.release}/lib"/>
</target>
```

All tasks required by build tools are provided by ANT pre-defined tasks such as for compiling, copying, creating jar files, or deleting a tree.

NOTE: The ANT documentation is installed at the following location: `<IS_HOME>\tools\ant\docs\manuals`.

Building a Cartridge Using Intershop Studio

Intershop Studio supports the build process in various ways. First and foremost, Intershop Studio copies the central entry point for ANT (`build.xml`) and the cartridge build configuration file (`build.properties`) from `<IS_HOME>\tools\build\shared\cartridge` into the cartridge's build folder. This build script pre-configures standard tasks necessary to build a cartridge. The tasks references other build files stored into `<IS_HOME>\tools\build\shared` and its subfolder (like `compile.xml` or `jar.xml` and even `build.properties`).

Apart from generating a set of build files, Intershop Studio lets you run all or selected ANT tasks directly from within the development environment.

How Build Tools Work

As mentioned above, there are only two build files required in the cartridge's build folder: the main build script (`build.xml`) and the cartridge configuration file (`build.properties`). The main build script references other build files stored in `<IS_HOME>\tools\build\shared` and its subfolder (like `compile.xml` or `jar.xml`).

Each of the build files can be adjusted at the shared location to customize executed tasks needed for all cartridges. To implement cartridge-specific build tools, build files located in `<IS_HOME>\tools\build\shared\cartridge` must be copied into the cartridge's build folder and adjusted there. Don't forget to declare the new location in `build.xml`.

In addition to these customizations, existing build tools can be configured via `build.properties`.

NOTE: The build tools fully support file system links for all operating systems. Links are used to link folders between cartridge source directories and cartridge build directories. Do not manually delete your build target, as this will delete your static, template and pipeline source files.

Note also that the classpath used during building a component is built using the list of cartridges your cartridge depends on.

Deployment Process

This section provides a rough overview of the deployment process, with emphasis on the typical processes that every deployment has to consider. Note that the details of the deployment process are dependent on the architecture of the production system.

Deployment Overview

The deployment process differs depending on whether a complete new build is deployed, replacing the current system, or whether a new cartridge or a set of new cartridges is deployed into the current system.

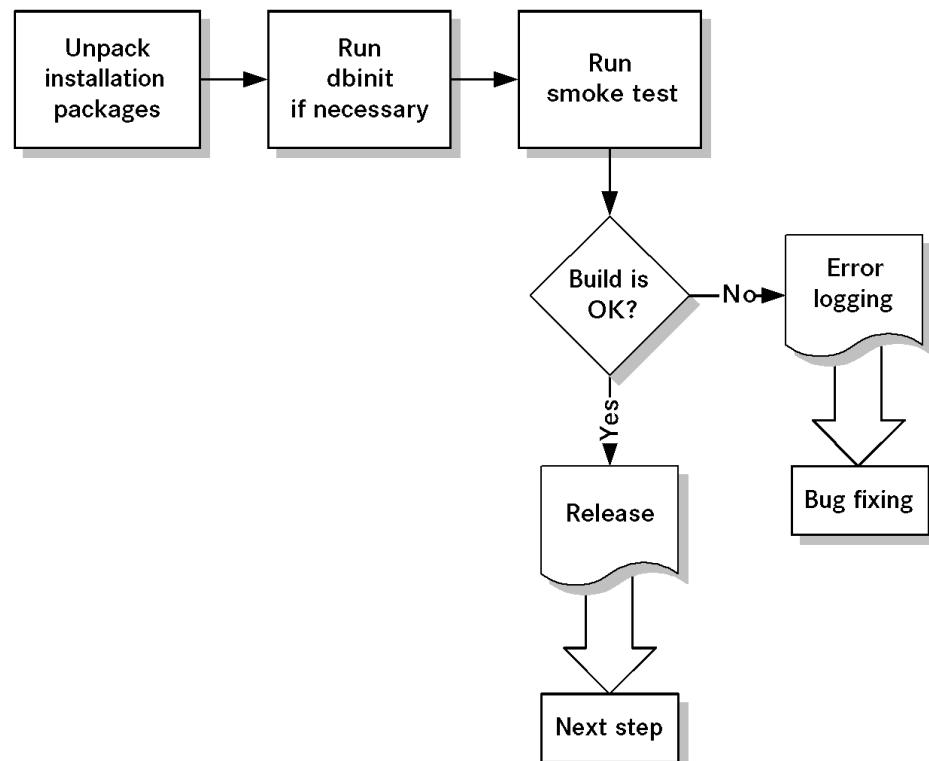
When deploying a complete application:

- 1. Copy the installation files to the production system**
- 2. Install the application**
- 3. Perform a dbinit and import required data.**
- 4. Perform a smoke test**
- 5. Start the application server**

CAUTION: Do not run dbinit on a live system, because this completely deletes and recreates all database tables.

A typical deployment process is depicted in the following diagram:

Figure 21. Deployment process overview



To deploy individual cartridges:

- 1. Copy the cartridge to the production system**
- 2. Change the `cartridgelist.properties` file**
- 3. Update the database**
- 4. Restart the application server**

Using Backup Systems

In the most simple case, deploying changes involves the following steps:

- 1. Shut down the Intershop 7 cluster.**
- 2. Deploy the cartridges or changes to the production system.**
- 3. Invalidate the Web adapter page cache if necessary.**
- 4. Restart Intershop 7.**

This simple process leads to a downtime of the production system. When deploying new cartridges or entire applications, however, it is often preferred or even necessary to avoid downtimes. To avoid downtimes, backup systems can be set up to handle all incoming traffic while the production system is being updated. To deploy changes without downtime:

- 1. Install a backup cluster, which can be smaller than the live cluster**
- 2. Switch load balancer or DNS mapping to the backup system**
- 3. Shut down your live cluster**
- 4. Deploy the new cartridge or application**
- 5. Invalidate the Web adapter page cache if necessary**
- 6. Restart the live cluster**
- 7. Switch the load balancer or DNS mapping back to the live cluster**

NOTE: The backup cluster can use the same database as the live cluster. This decision depends on whether database changes are applied.

Cartridge Startup Process

The startup of a cartridge is defined by its configuration and follows a specific startup process:

■ Cartridge Definition

All installed cartridges are listed in a single `cartridgelist.properties` file, located in the `<IS_HOME>/share/system/cartridges` directory. New cartridges are registered with the system by just adding them to the `cartridgelist.properties` file.

```
cartridges = core xcs bts ac_sap ac_oci bc_mvc ...
```

■ Startup Process of a Cartridge

A server process loads all cartridges in the order that is listed in the `cartridgelist.properties` file. Each registered cartridge is automatically started on all Intershop 7 application servers. For each cartridge, the managers,

service classes and persistent objects are registered with the naming service. The server process then scans the pipelets listed in all cartridges and registers them. For each cartridge, the cartridge controller class is called to run cartridge-specific initialization processes. An example for such an initialization process is the preloading of persistent objects into the PO-cache. See *Cartridge Controller Class* for details.

Apps and Cartridges

For the code artifacts like pipelines, queries, pagelets, etc. contained in cartridges to become available in applications (sites), the cartridges must be assigned to so called "apps". An app is a code artifact that defines the set of functionality for a specific application (the "application logic"). That is, cartridges must be associated with apps. To this end, each app includes a component file that defines the applicable cartridge list (but no lookup order).

NOTE: The file <IS.INSTANCE.SHARE>/system/cartridges/cartridgelist.properties defines all cartridges that are to be loaded at server startup and their according lookup order.

Defining Apps

Apps are defined in their corresponding "application cartridges". An application cartridge includes a *component* file, which defines the app instance and registers it in the system. The app must provide a cartridge list. Remember that the cartridges provide the code artifacts (pipelines, queries, ...) that are used by the app. In addition, the app must provide a URL identifier that must be unique within a site. A localized resource bundle contains the display name and description of the app.

The app must define following attributes:

■ **Identifier**

Unique within the entire system, should consist of <vendor>. <product>. <name>, e.g., intershop.enfinity.B2CWebShop.

■ **Resource bundle**

Specifies the Java resource bundle to load the localized values, e.g., resources.sld_ch_b2c_app.apps.B2CWebShop.

■ **cartridgeListProvider**

The instance that holds the cartridge list of the app. One cartridge lists can be used by multiple apps.

■ **urlIdentifier**

Specifies the URL that identifies an app within a site context.

The files in a cartridge that define an app include:

- <cartridge>/javasource/resources/<cartridge>/apps/<app_bundle_name>_<locale>.properties
Optional resource bundles to load localized values.
- <cartridge>/staticfiles/cartridge/components/apps.component

The app component file, which lists the associated cartridges and the properties mentioned above.

NOTE: *apps.component* files must be created and modified manually in each application cartridges file structure.

The contents of an *apps.component* file is illustrated below:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.intershop.de/component/2010">

    <!-- the cartridge list provider instance -->
    <instance name="intershop.enfinity.ExampleApp.Cartridges"
        with="CartridgeListProvider">
        <fulfill requirement="selectedCartridge" value="$cartridge"/>
        <fulfill requirement="selectedCartridge" value="cartridge1"/>
        <fulfill requirement="selectedCartridge" value="cartridge2"/>
    </instance>

    <fulfill requirement="app" of="AppEngine">
        <instance name="intershop.enfinity.ExampleApp" with="EnfinityApp">
            <fulfill requirement="id" value="intershop.enfinity.ExampleApp"/>
            <fulfill requirement="urlIdentifier" value="example"/>
            <fulfill requirement="resourceBundleName"
                value="resources.$cartridge.apps.ExampleApp"/>
            <fulfill requirement="cartridgeListProvider"
                with="intershop.enfinity.ExampleApp.Cartridges"/>
        </instance>
    </fulfill>
</components>
```

Adding Cartridges To Apps

In customization projects, existing apps are extended with new functionality. In addition, existing functionality may be replaced or modified. In these cases, one or more cartridges are added to an existing app. The new cartridges must be in the beginning of the lookup path for the particular code artifacts like pipelines, templates, etc.

Apps are extended by adding additional cartridges to the cartridge list provider. To do so, for each cartridge intended to be added to the app, an *app-extension.component* file like illustrated below must be created in *<cartridge>/staticfiles/cartridge/components/*:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.intershop.de/component/2010">
    <fulfill requirement="selectedCartridge"
        of="intershop.enfinity.ExampleApp.Cartridges"
        value="$cartridge"/>
</components>
```

Cartridge Development Overview

Intershop Studio includes the following features that support the cartridge development process. How to use these features will be described in the sub-sections below.

■ Create Cartridge wizard

The Create Cartridge wizard leads you through the steps required to set up a new cartridge project.

■ **Cartridge Import wizard**

The Cartridge Import wizard allows you to import the sources of existing cartridge projects. You can import:

- source cartridges located in your development environment
- server cartridges deployed on your server
- existing cartridge projects located elsewhere on your system

NOTE: Server cartridges can only be imported as 'binary' projects, which means that pipelines, templates, and static files will be managed by Intershop Studio, but no Java source code files)

■ **cartridge working sets**

Intershop Studio lets you define configurable set of cartridges, so called cartridge working sets. You can choose a working set as filter for most of the Intershop Studio views.

■ **actions to support cartridge building**

Intershop Studio provides the following actions to support you when building cartridges:

- create ant build files
- launch ant build files

■ **class path container**

Class-path containers simplify the build path handling for cartridge projects.

- INTERSHOP Server JDK: configured on Intershop Studio startup. Defines the jre variable
- Intershop 7 Server Libraries: contains libraries available in servers lib folder
- Cartridge Lib Folder: contains libraries provided by cartridges staticfiles/cartridge/lib folder
- Required Cartridges: contains the libraries provided by cartridges the cartridge directly or implicitly depends on.
- Cartridge: contains libraries provided by a cartridge. This container must be configured with the cartridge name

■ **cartridge perspective**

Intershop Studio comes with a pre-defined cartridge perspective that bundles most of the Intershop 7 element based views and actions.

■ **cartridge explorer**

The cartridge explorer displays a cartridge oriented view of the Intershop Studio workspace along with the Intershop 7 cartridge repository.

In the cartridge explorer, you can:

- view cartridges
- group cartridges by their membership to the server or workspace
- group cartridges by their site
- group cartridges that belong to a source project

The cartridge explorer contains pre-configured filters that let you:

- hide overridden elements
Only those elements are shown that are really visible in the current site context.
 - hide invisible elements
Shows only elements of cartridges that belong to the current site context.
 - show current cartridge only
Displays only the cartridge and its members that is directly connected to the currently selected element or editor.
 - hide pipeline members
Does not show pipeline nodes.
 - hide simple join nodes
- The cartridge explorer's user interface is sub-divided into the following areas:
- main area
Displays cartridges in a tree-like view.
 - resource area
Displays files and resources belonging to the currently selected element.
 - info area
Displays descriptive information about the currently selected element.

Creating New Cartridges

Open the Cartridge Wizard

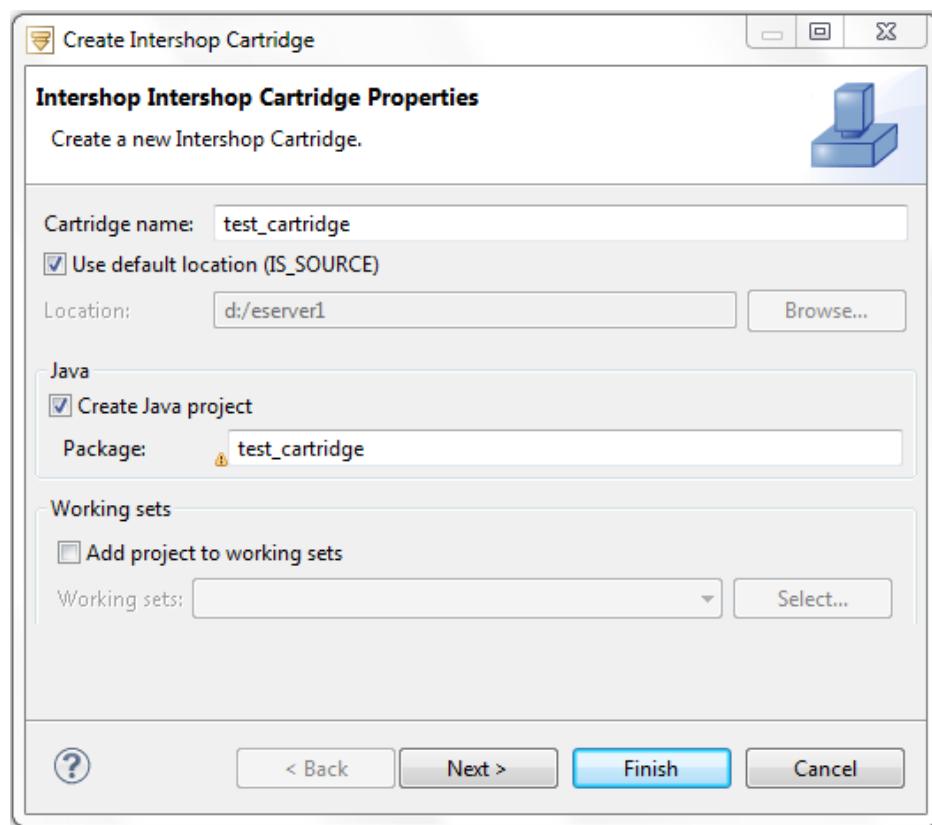
To open the Cartridge Wizard:

1. Select File | New | Cartridge from the menu.

Cartridge Wizard starts.

Step 1: Define Cartridge Properties

The first page of the Cartridge Wizard prompts you for general cartridge properties. These properties are stored in the `<cartridge_name>.properties` file. This file stores all information needed by the system to run the cartridge. The file is stored inside your cartridge in the `\staticfiles\share\system\config\cartridges` folder.

Figure 22. Cartridge Wizard: Defining Cartridge Properties

The following table describes the properties to be specified:

Table 5. General Cartridge Properties

Property	Description
Cartridge name	The cartridge name (ID) used internally by the system. The cartridge development directory is named after the cartridge name. The cartridge name must match the sub-package naming conventions (no spaces, no dots, no Java keywords, lower cases only etc.). Intershop recommends to use pre-defined prefixes to name cartridges. Via content assist available: ac_ (should be used for adapter component); app_ (should be used for an application); bc_ (should be used for business component); pf_ (should be used for platform component); service_ (should be used for a service component). Example: Intershop 7 cartridges use names such as "ac_addresscheck_std", "ac_bmecat" or "bc_shipping".
Location	Intershop 7 uses a default location to store cartridges. To edit location settings: uncheck the "Uss default location (IS_SOURCE)" checkbox and browse the desired location.
Java	Intershop recommends to create an Java project for every cartridge. The default name of the package is the cartridge name without the selected prefix.
Working sets	Check on the "Add project to working sets" checkbox to apply the cartridge to working sets. Click select and mark all desired working sets within "Select Working Sets" dialog. Working sets are important for filtering functions in a variety of views of different editors.

NOTE: The Cartridge Wizard enforces you to choose a unique cartridge name. If a cartridge development folder with the same name already exists under IS_SOURCE, or if there is already an Eclipse project of the same name, the Cartridge Wizard will prompt you for a new name.

Step 2: Define Cartridge Template, Structure, additional Properties and Dependencies

On the second page of the Cartridge Wizard, you can define a cartridge template, additional properties and dependencies to other cartridges if you want to use functionality provided by these cartridges. The cartridge dependency settings are also stored in the <cartridge_name>.properties file.

The following table describes the properties to be specified:

Figure 23. Second page of New Cartridge Wizard

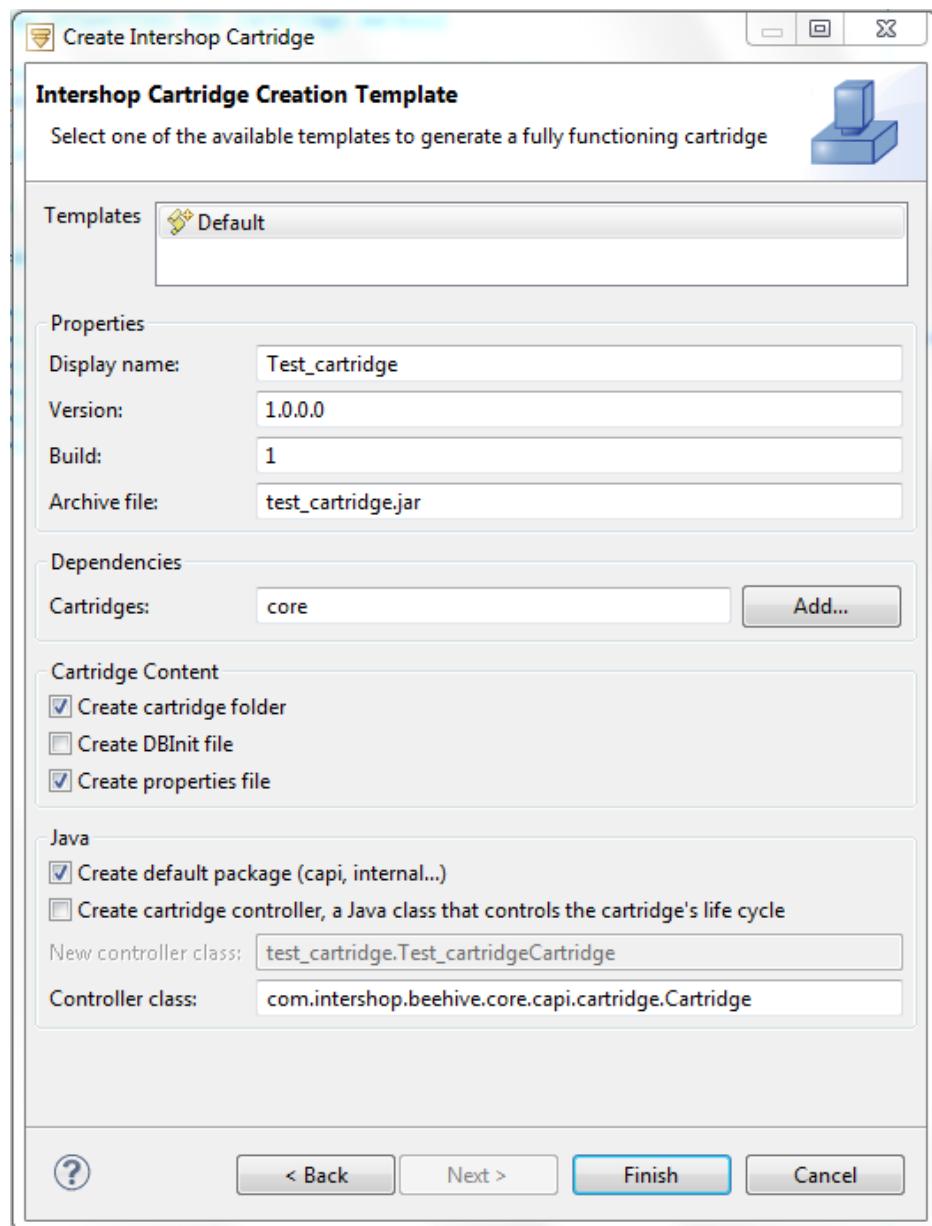


Table 6. Second page of New Cartridge Wizard

Property	Description
Template	Select a template to predefine cartridge's structure and dependencies.
Display name	The name of the cartridge used for display purposes.
Version	The external version string for display purposes.
Build	The internal build number of the cartridge.
Dependencies	Click Add and apply the desired cartridges from "Select Cartridge" dialog. All cartridges available on your development system can be selected. Note: Selecting a cartridge on which the new project depends often leads to indirect dependencies, as the selected cartridge may depend on other cartridges itself.
Create cartridge folder	Check on to create cartridge content folder.
Create DBinit file	Check on to create a DBinit file. DBinit file is stored within the cartridge <code>staticfiles\cartridge\dbinit.properties</code> . DBinit file stores data needed by preparer classes.
Create properties file	Check on to create a properties file. Intershop strongly recommends to create a properties file. Cartridge's properties file is stored within the cartridge <code>staticfiles\share\system\config\cartridges\<cartridge_name.properties></code> .
Create default package (capi, internal...)	Check on if default package should be created. Defines the package structure for your cartridge. Intershop Studio uses this information to generate an appropriate directory structure inside the <code>javadoc</code> folder of the cartridge development directory. This package contains files which store public business interfaces (API classes), implementation classes for public business interfaces and any other non-public classes and sub-packages used exclusively by the cartridge, pipelets belonging to the cartridge, including class files and pipelet descriptor files.
Create cartridge controller	If your cartridge needs special treatment during startup and shutdown, check the "Create cartridge controller" check box and type in the controller's name to modify the life cycle methods you wish to override and customize. Intershop Studio will then create an editable Java file within the <code>javadoc</code> folder of the cartridge.

Finish the Cartridge Creation Process

After defining the package structure and the cartridge controller class, click Finish to create the cartridge project. The new cartridge is now accessible through both the package explorer and the Cartridge Explorer view.

Starting with Intershop Studio, new or imported cartridges are assigned the "JavaScript" nature, in addition to the "beehive" and "Java" natures. JavaScript support is available when editing ISML files or JavaScript source files, enabling content assist functionality and detailed error notification, among others.

Set the Code Generator Version for the Project

The code generator version needs to be set as a property for each project independently. Intershop Studio will not permit you to edit a project unless this property has been set.

NOTE: The property will be stored in a file named <project_name>\.settings\com.intershop.studio.prefs. It is strongly recommended to keep this file and anything else in the .settings folder under revision control, otherwise developers would have to re-configure this property each time they check out and import the cartridge into Intershop Studio.

Set the code generator version in the project's properties:

- 1. In an explorer view right-click the project.**

A context menu is displayed.

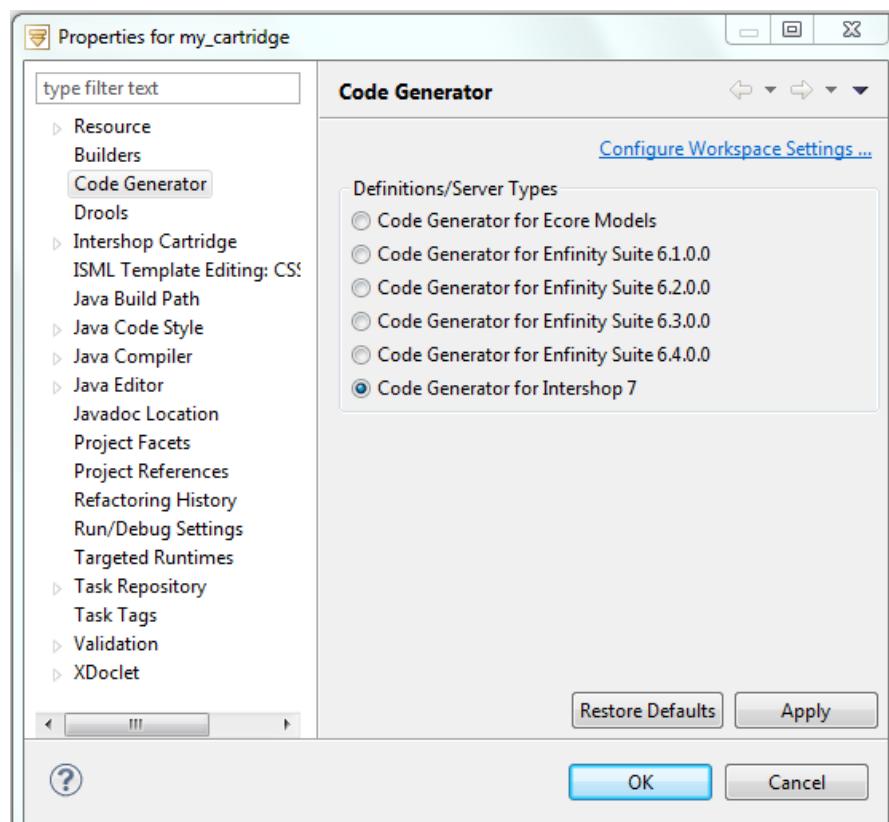
- 2. From the context menu, select Properties.**

The cartridge's properties dialog is displayed.

- 3. In the properties dialog, go to Code Generator.**

- 4. Set the appropriate code generator version.**

Figure 24. Setting the Code Generator Version in Intershop Studio



NOTE: Intershop Studio sets Code Generator for Intershop 7 as default code generator.

Importing Cartridges

The Cartridge Import Wizard allows you to import the following artifacts into the Intershop Studio workspace:

- source cartridges located in your development environment
- server cartridges located in your Intershop 7 installation
- existing cartridge projects located elsewhere on your system

Source cartridges are located in <IS_SOURCE>. Having been imported, a source cartridge can be modified and built in Intershop Studio, or can share resources with other cartridges.

Server cartridges are located in <IS_HOME>. Typically, server cartridges are imported to modify pipelines, templates or static content on a running system (this is also referred to as hot editing).

NOTE: Server cartridges are imported as binary projects with linked content. Java code cannot be modified on server cartridges. Any changes to pipelines, templates and static content are applied immediately.

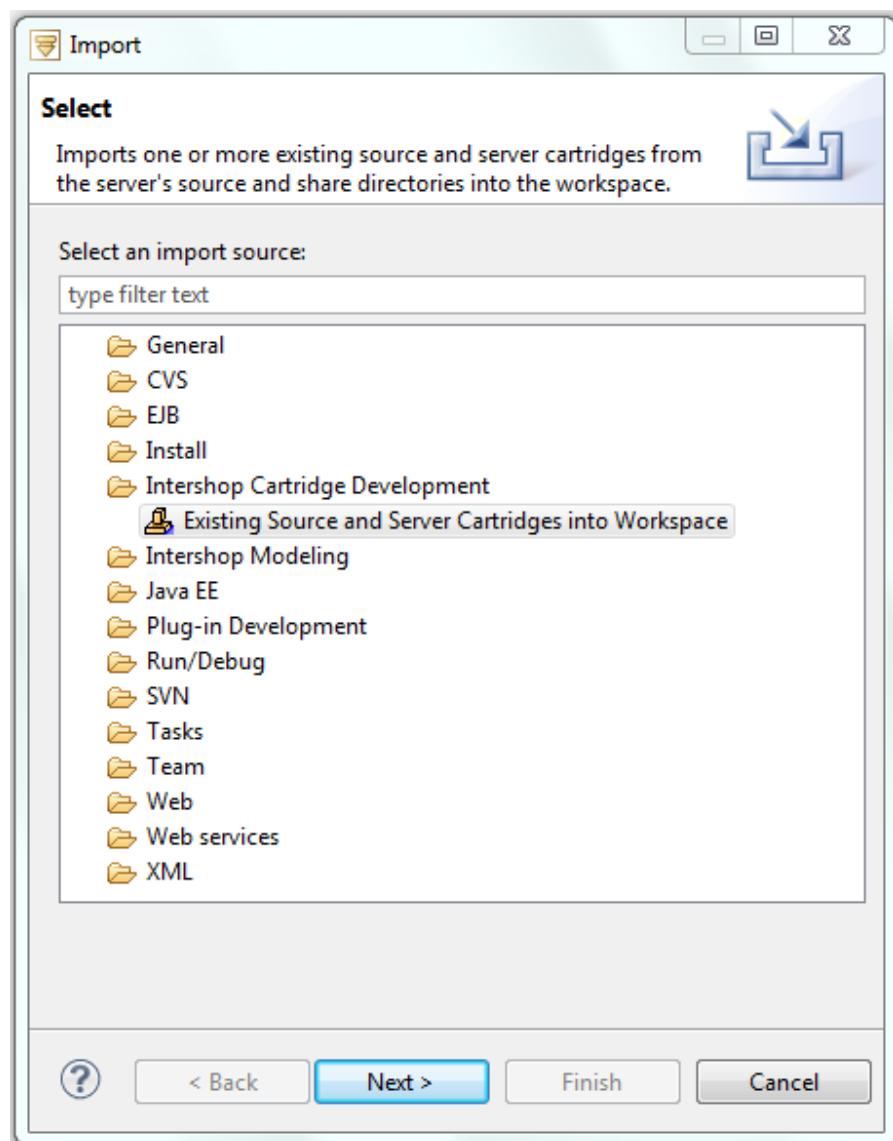
An existing cartridge project can be located anywhere on your system.

Importing Source and Server Cartridges

To import cartridges:

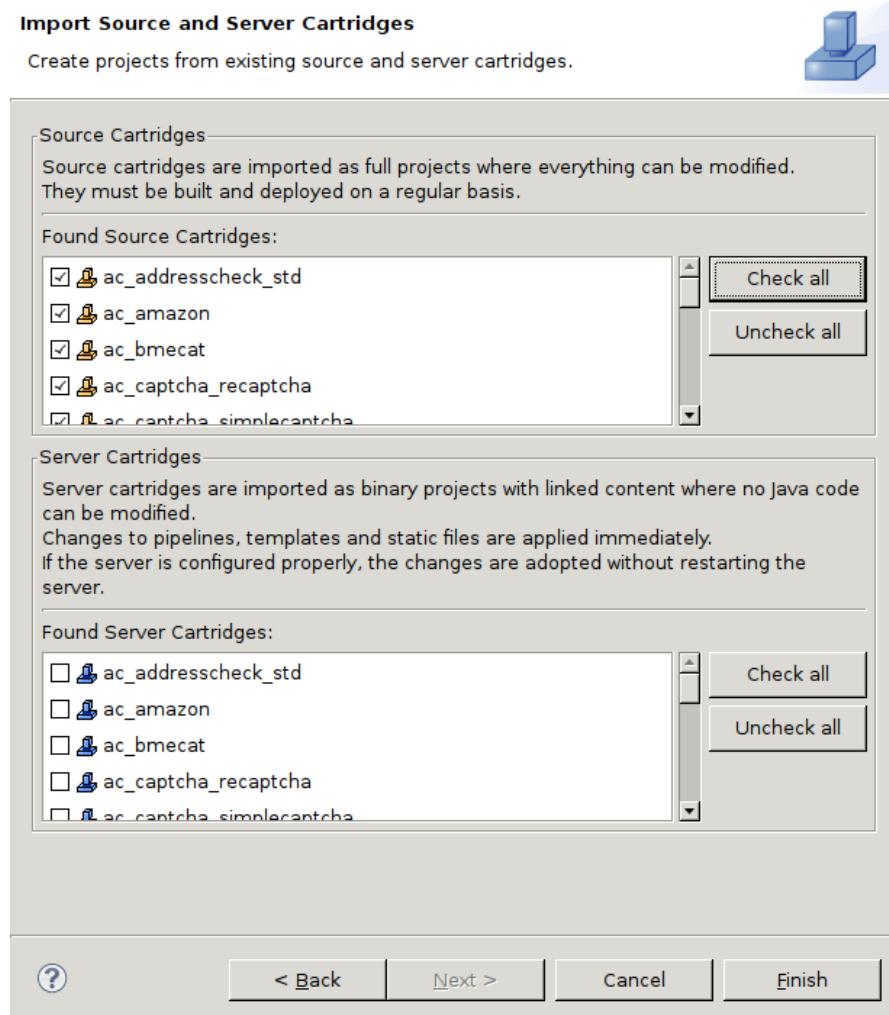
- 1. In the Workbench, select the File | Import menu to run the Import wizard.**

The first page of the Import wizard displays a list of source types you can import into Intershop Studio.

Figure 25. Select Import Source

2. From the list, choose **Intershop Cartridge Development | Existing Source and Server Cartridges into Workspace**, then click **Next**.
3. **Select cartridges for import.**

The second page displays a list of server and source cartridges found on your system.

Figure 26. Choose Cartridges for Import

Select the cartridges you want to import.

If the cartridge you want to import is not shown in the list, check the following points:

#+ATTR_ITEMIZEDLIST mark="disc"

- The Import Wizard does not list a cartridge project if the name is already used by another Eclipse project. To check for existing projects, open the Package Explorer.
- For cartridges that are registered with Intershop 7 in the `cartridgeList.properties` file, the Import Wizard looks for a cartridge development directory with the name of the cartridge ID in the `<IS_SOURCE>` folder. If the directory exists, the Import Wizard shows the cartridge in the import list.
- For any other cartridges that are not registered in `cartridgeList.properties`, the Import Wizard looks for a `cartridge.properties` file in `<IS_SOURCE>\<cartridge_name>\staticfiles`

\share\system\config\cartridges\ . If the properties file exists, the cartridge appears in the import list.

4. Click Finish to start the import process.

After a successful import, the new cartridge project is displayed in the Package Explorer. Use the Problems View to identify possible problems found during the import.

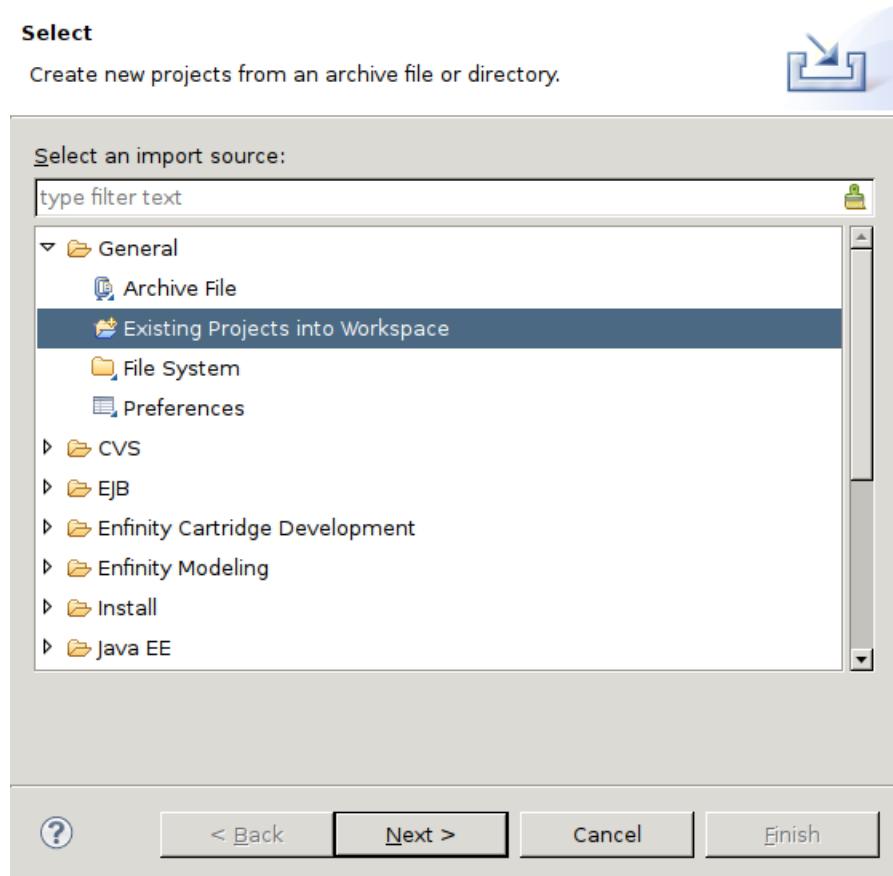
Importing Existing Projects

To import existing cartridge projects using the Import wizard:

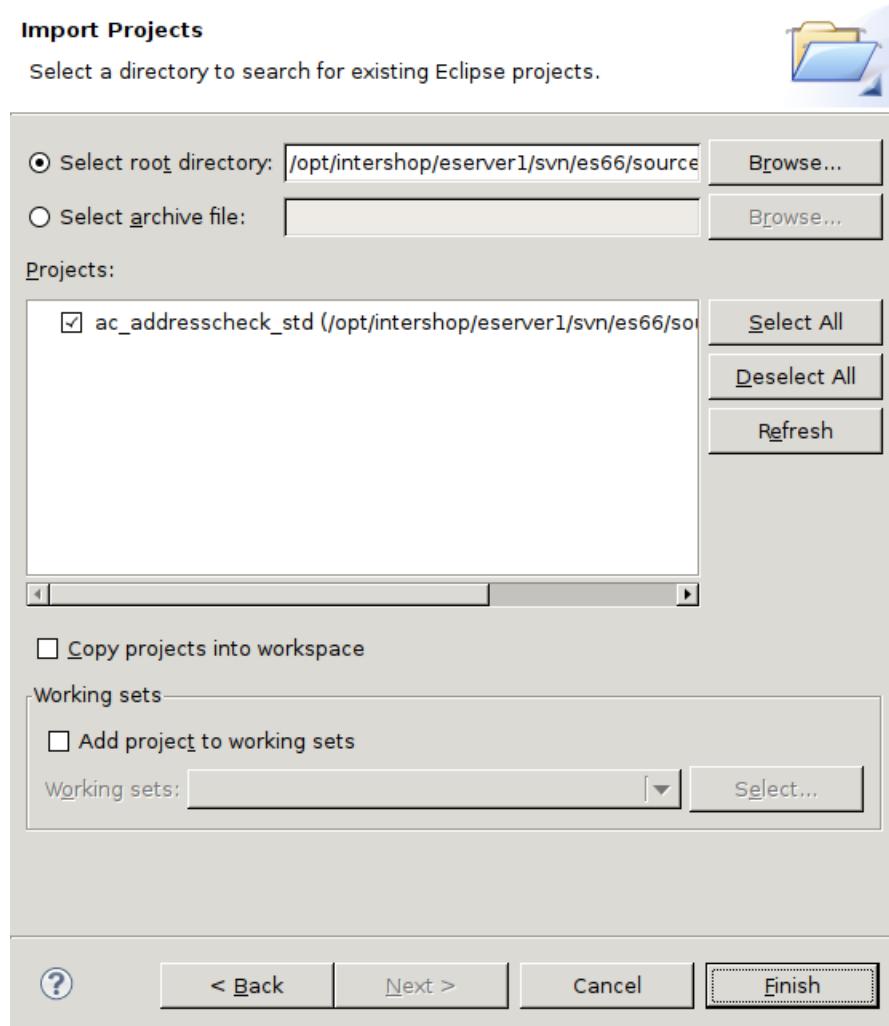
1. In the Workbench, select the File | Import menu.

The first page of the Import wizard displays a list of source types you can import into Intershop Studio.

Figure 27. Select Import Source



- 2. From the list, choose General | Existing Projects into Workspace, then click Next.**
- 3. Select the cartridge project you wish to import.**

Figure 28. Select Existing Project for Import

Browse to the directory where your existing cartridge projects reside.

From the list of found cartridge projects, select the ones you want to import.

If you want to copy all project resources into your workspace instead of keeping them in their original location, check the "copy projects into the workspace" checkbox.

If required you may add the imported projects also to an Intershop Studio working set.

- 4. Click Finish to import the project.**

Upon import, a cartridge is automatically assigned the "JavaScript" nature.

Working with Cartridges

Explore Cartridge Resources

Two main views are available to browse the contents of a cartridge project. Both views are part of the Intershop Studio Cartridge Development Perspective.

■ **Cartridge Explorer**

The Cartridge Explorer lists all cartridges deployed on your Intershop 7 system. Each cartridge is presented in a navigation tree that can be expanded by double-clicking the name of a cartridge. This way you navigate the entire cartridge and can locate cartridge resources you want to work with. See *The Cartridge Explorer* for more information.

NOTE: Cartridge projects that have been imported or newly created are visible in the Cartridge Explorer immediately, even if not registered with the application server. See here for registration details.

■ **Package Explorer**

The Package Explorer provides a view of the package structure of cartridge projects and allows you to open, close, and delete cartridge projects. By applying filters you can specify what is displayed in the Package Explorer. See *The Package Explorer* for more information.

Manage Cartridge Properties

Basically, there are two ways to edit the properties of a cartridge: using the Cartridge Editor or using the Properties View in the Cartridge Explorer.

Cartridge Editor

Intershop Studio includes the Cartridge Editor as the central tool for managing important cartridge details. The Cartridge Editor opens automatically upon creating a new Intershop 7 cartridge, i.e., after completing the cartridge wizard (see *Creating New Cartridges*). For existing cartridges, you can access the Cartridge Editor through selecting Open Cartridge from the cartridge's context menu in the Cartridge or Package Explorer.

The Cartridge Editor comprises of several tabs that group certain sets of cartridge details. These details tabs include:

■ **Overview**

Edit general cartridge properties, including ID, version, build, name, cartridge controller class.

■ **Dependencies**

Edit the list of cartridge dependencies, which are graphically visualized with immediate effect.

■ **Runtime**

Add or remove libraries that the cartridge uses or provides to other cartridges, and edit the project's class path.

■ Database Initialization

Manage the preparers that the cartridge requires to initialize the database.

■ Cartridge Pipeline Hooks

Edit cartridge pipeline hooks. This tab lists available hooks and their assigned pipeline start nodes.

■ Properties

Edit the `<cartridge>.properties` file directly.

For more information about the Cartridge Editor, see *Cartridge Editor*.

Cartridge Properties View

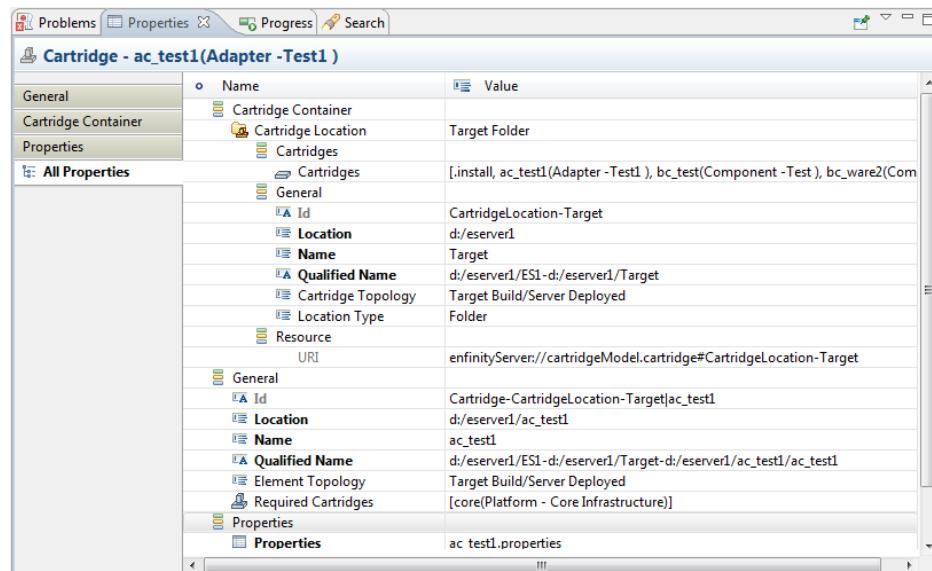
Cartridge properties can also be edited in the Properties View of the Cartridge Explorer. The Properties View of the Cartridge Explorer also shows general project properties and cartridge registration information.

NOTE: Make sure to work in the Cartridge Explorer. Cartridge properties are not displayed in the Properties View of the Package Explorer.

The values are read from the respective `<cartridge>.properties` file, such as `mytest.properties`. Modified values are written to the `<cartridge>.properties` file and become effective immediately.

Click the  icon from Properties view toolbar and deselect all checkboxes to expand the list of properties shown.

Figure 29. Cartridge Properties



For a description of the cartridge properties that can be configured, see:

- *Step 1: Define Cartridge Properties*
- *Step 2: Define Cartridge Template, Structure, additional Properties and Dependencies*

Register Cartridges

For a cartridge to become registered with the Intershop 7 server, it must be added to the `cartridgelist.properties` file which is stored in `<IS.INSTANCE.SHARE>/system/cartridges`. Restart the application server for the change to become effective. If cartridge registration has been successful, Intershop Studio sets the cartridge property "Registered On Server" to "true".

NOTE: Intershop Studio does not automatically register cartridges with the Intershop 7 server.

Managing Classpath Settings

Understanding Classpath Containers

Intershop Studio assembles the classpath from classpath containers. Classpath containers group the individual libraries included with the classpath according to their general function. Which libraries the classpath containers hold can be set independently for each project. The following classpath containers are available:

■ Intershop 7 Server Libraries

Includes libraries provided by the application server, e.g. EJB container (PowerTier) and servlet engine jar files. The set of server libraries available to cartridge projects is determined by the server classpath settings. See *Set Server Classpath* for details on how to set the server classpath.

■ JRE System Libraries (Intershop Server JDK)

Includes Intershop 7 server JDK and JRE libraries.

■ Cartridge Lib Folder Libraries

Includes all libraries found in the cartridge lib folder (`<IS_SOURCE>\<cartridge_name>\staticfiles\cartridge\lib`). Use the lib folder to link external libraries that are shipped with the cartridge. If the lib folder does not exist, you need to create it manually.

You can add libraries to the lib folder at any time. Intershop Studio detects changes in the lib folder and automatically updates the build path accordingly.

Always link external libraries by copying them to the lib folder, otherwise Ant or Intershop 7 will not be able to find them.

■ Required Cartridges Libraries

Includes all libraries provided by cartridges your project depends on.

Set Classpath for Cartridge Project

The classpath settings for a cartridge project can be checked and modified via the Cartridge Properties dialog. To open the Cartridge Properties dialog:

1. **In the Package Explorer, right-click the project you wish to modify.**

The context menu is displayed.

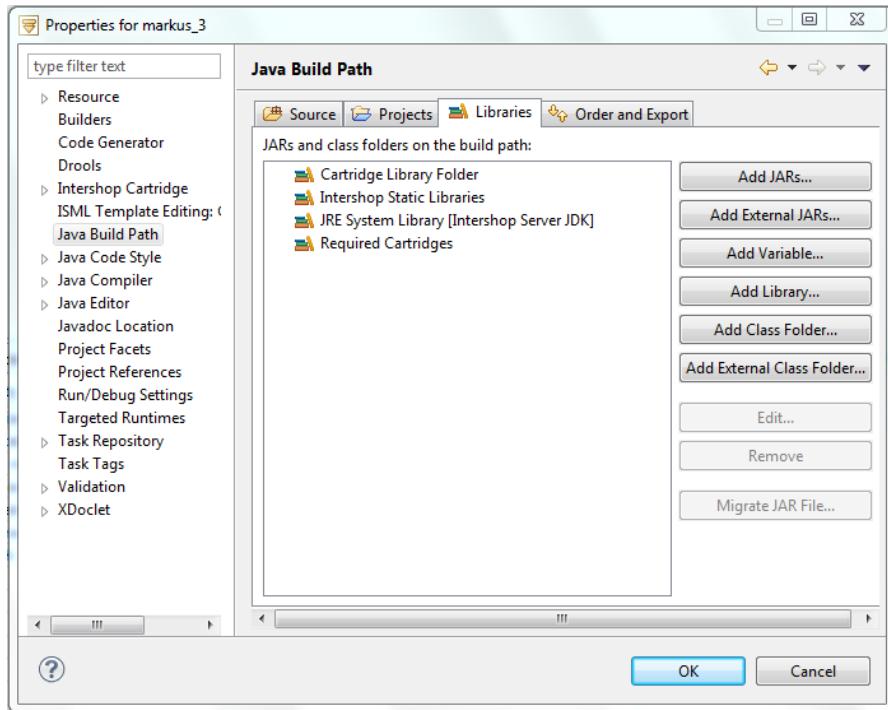
2. **From the context menu, select Properties.**

The Cartridge Properties dialog is displayed.

3. Select Java Build Path in the left tree pane, then click the Libraries tab.

The Libraries panel is displayed.

Figure 30. Cartridge Properties Dialog



4. Change the configuration as needed.

Add or modify jar files or libraries as required by your project.

5. Click Ok to save changes.

Set Server Classpath

The server classpath provides the default set of libraries included with the Intershop 7 Server Libraries classpath container, such as EJB container (PowerTier) and servlet engine jar files. The server classpath settings apply to all new cartridge projects. The settings determine the selection of server libraries available to all cartridge projects. However, you can enable or disable individual server libraries individually for each project by modifying the respective classpath container in the project's Java build path. See *Set Classpath for Cartridge Project* for details.

The server classpath comes with a default setting which fits the needs of typical cartridge projects. Proceed as follows to modify or restore the default settings:

- 1. From the menu bar, select Window | Preferences to open the Workbench Preferences dialog.**
- 2. Expand the tree belonging to the Intershop Studio page and select the Server Classpath page.**
- 3. Modify settings as required by your cartridge projects.**

To restore default settings, click Restore Defaults. To add or remove libraries, clear the Use Default Classpath check box and select libraries as required.

NOTE: It is recommended to keep the number of libraries on the server classpath as small as possible as this saves resources of your development system and improves overall performance of Intershop Studio (e.g. the opening of dialogs and search operations will be faster).

See *Server Classpath Preferences* for additional information on this page.

Automatic Update of Classpath Container

If you change the configuration of a cartridge, Intershop Studio detects the change and automatically updates the classpath containers.

For example, if you want to deploy pre-compiled Java libraries with your cartridge, you copy the jar archives to the cartridge \lib folder (<IS_SOURCE>\<cartridge_name>\staticfiles\cartridge\lib). Intershop Studio adds then the new archives automatically to the according classpath container.

This works also for modifications of the cartridge dependency list. If you add or remove cartridges to/from the list, Intershop Studio automatically updates the classpath container.

Building a Cartridge

Overview of Build Process

Building a cartridge with ANT takes three steps:

- 1. Preparing properties files**
- 2. Preparing ANT build files**
- 3. Run Build**

Typically, steps 1 and 2 are carried out only once during a cartridge development project. Each step is described below.

NOTE: ANT does not execute correctly if the value of your system's PATH environment variable contains a trailing slash. Before running ANT, check the value of the PATH variable and make sure it does not contain trailing slashes. For example, "Path=;D:\es6\eserver1\bin;D:\es6\eserver1\tools\ant\bin" is correct, whereas "Path=;D:\es6\eserver1\bin;D:\es6\eserver1\tools\ant\bin\" is not.

Preparing Properties Files

The build process makes use of property settings spread across various properties files. These properties are read and processed by build files as needed.

General system information is drawn from settings in `intershop.properties`. In particular, `intershop.properties` defines variables that point to the cartridge development root directory (`IS_SOURCE`) and to the cartridge build root directory (`IS_TARGET`).

Three more properties files are of importance as they are used to define general settings that apply to all cartridges built for a particular system (`isbuild.properties`, `<IS_HOME>\tools\build\shared\build.properties`) and cartridge specific settings (`<IS_SOURCE>\<cartridge_name>\build\build.properties`).

isbuild.properties

This file defines system-specific settings which apply to all cartridges built for and installed on a particular Intershop 7 system. The `isbuild.properties` file is not part of the cartridge development directory but is located directly under the Intershop 7 root directory (as pointed to by the `IS_HOME` variable of `intershop.properties`). The file is generated automatically during the installation process and is not set up individually for each cartridge. Changes to this file (which will be rarely necessary) should be applied with care.

The `isbuild.properties` file defines two basic types of information necessary for the build process. On the one hand, the file specifies directory information which is used to automatically generate `CLASSPATH` settings. In addition, the file contains cartridge definitions that are used to identify folders inside the cartridge development directory and the cartridge build directory.

```
#cartridge definitions
foldername.javasource = javasource
foldername.csource = csource
foldername.release = release
foldername.generated = generated
foldername.temp = generated
foldername.setup = setup
foldername.model = model
foldername.static = staticfiles
```

<IS_HOME>\tools\build\shared\build.properties

This file contains additional build properties for the overall build process. The properties are useful when creating JavaDocs or using the setup framework. The following settings should be set in case either JavaDoc is created or the setup framework is used:

- `product.name`
The name of your product
- `product.displayname`
The display name of your product
- `product.version`
The version number of your product
- `product.build`
The build number of your product
- `product.copyright.owner`
The copyright owner of your product
- `product.copyright.from`
The copyright-from year of your product
- `product.copyright.to`
The copyright-to year of your product
- `setup.vendor`
Your company's name

■ `setup.vendor.url`

Your URL

<IS_SOURCE>\<cartridge_name>\build\build.properties

This file defines cartridge specific settings for the build process. Like `build.xml`, it is copied from the `<IS_HOME>\tools\build\shared\cartridge` folder during cartridge creation. Most properties in this file have meaningful default values, so it is rarely necessary to adjust them. Intershop 7 uses settings in `build.properties` to provide configuration values for the build files. See the discussion of the respective build files below for information on these properties.

Preparing Build Files

The build process for a cartridge is driven by a set of pre-configured XML files. Each build file defines the targets pertaining to a particular domain, such as compiling Java source code (`compile.xml`) or packaging Java archive files (`jar.xml`).

Overview of Build Files

See *Table 7, “Intershop 7 build files”* for an overview of important build files. Note that only the main build script (`build.xml`) is stored in the `\build` folder inside the cartridge development directory, e.g., `<IS_SOURCE>\testcartridge\build`. Note also that this file always references other build files located at `<IS_HOME>\tools\build\shared\cartridge`. If a certain domain is not applicable to your project, the respective build file does nothing.

Table 7. Intershop 7 build files

Build File	Description
<code>build.xml</code>	Entry point for ANT build process.
<code>environment.xml</code>	Creates the cartridge build environment (linking, copying templates and pipelines).
<code>jaxb.xml</code>	Generates and compiles the cartridge's JAXB binding objects.
<code>wsdl.xml</code>	Generates and compiles the cartridge's WSDL stubs.
<code>compile.xml</code>	Compiles all source code components, creates the respective archive files and copies them to <code><cartridge_build_dir>\release\lib</code> .
<code>rapi.xml</code>	Generates and compiles all files necessary to publish Web services.
<code>jar.xml</code>	Jars standard components.
<code>pipelets.xml</code>	Generates the pipelet resource file.
<code>orm.xml</code>	Generates the ORM object resource file.
<code>precompile.xml</code>	Pre-compiles and links ISML templates
<code>javadoc.xml</code>	Creates JavaDoc for Java sources.
<code>localize.xml</code>	Creates a localized template set.

Build File Configuration

Custom cartridges typically do not necessitate any modifications to the build file settings. To check for build targets and available parameters:

- 1. Change to directory <IS_HOME>\tools\build\shared\cartridge.**

- 2. Prepare the environment for ANT.**

Run the environment.bat file, located in <IS_HOME>\bin.

- 3. To check for targets executed by specific build files, run for example:**

```
ant -buildfile <IS_HOME>\tools\build\shared\cartridge\compile.xml -projecthelp
```

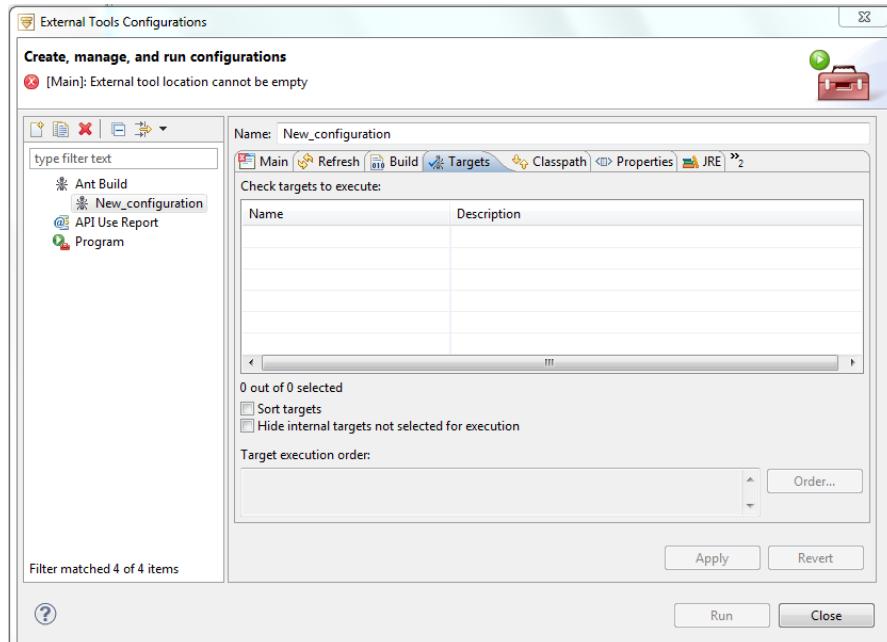
Configure ANT Settings and Run Build

Before you build the cartridge project the first time, you need to define the ANT configuration used for the build. The ANT configuration for a cartridge project is stored in a special *.xml configuration file. If necessary, you can modify ANT settings at later points in the project, for example to select special ANT targets to be executed, change the order in which targets are executed, or specify additional command line options to govern the ANT execution.

To configure ANT settings:

- 1. Select Run | External Tools | External Tools Configurations... from the Workbench menu to open the External Tools Configurations dialog.**
- 2. Select Ant Build | New configuration to open the configuration dialog.**
- 3. Select the project and edit settings as needed.**

Figure 31. ANT Configuration



NOTE: To generate the list of available ANT targets, Intershop Studio reads the names of all targets that are called for execution in the build.xml file. Therefore, when you edit the build file to add more targets for execution than those listed by default, these additional targets are

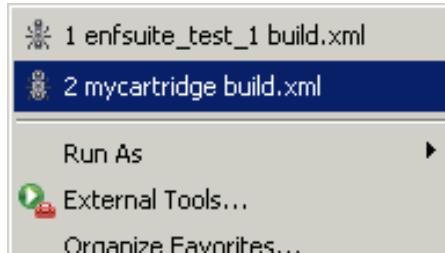
shown too. On the other hand, if your build file is not up-to-date, Intershop Studio may list ANT targets that are no longer required by your project. Intershop Studio ignores ANT targets without a description attribute.

Run Default Build Target

To run the default ANT target which builds the cartridge and creates jar archives:

- 1. Select Run | External Tools from the Workbench menu.**
- 2. Select the appropriate ANT configuration file for the project from launch history.**

Figure 32. Run Default Build Target



The output is captured and displayed in the Console view.

Run Build Process with Modified Settings

To build the cartridge with modified settings (e.g. to select a different build target):

- 1. In the Cartridge Explorer, right-click the cartridge project to open the context menu.**
 - 2. From the context menu, select Build Cartridge**
- The ANT configuration file for the project is displayed.
- 3. Modify settings as required and click Run to launch the build process.**

Run Selected ANT Targets

NOTE: In recent versions of Intershop 7, each ANT command is independent from all others. For example, ant jar only creates jar files but doesn't compile any files. To run all typical build steps for a cartridge, run ant, or ant jaxb, wsdl, compile, rapi, pipelets, orm, precompile, jar. Also note that the classpath used for various ANT targets is built using the dependson property of the cartridge property file.

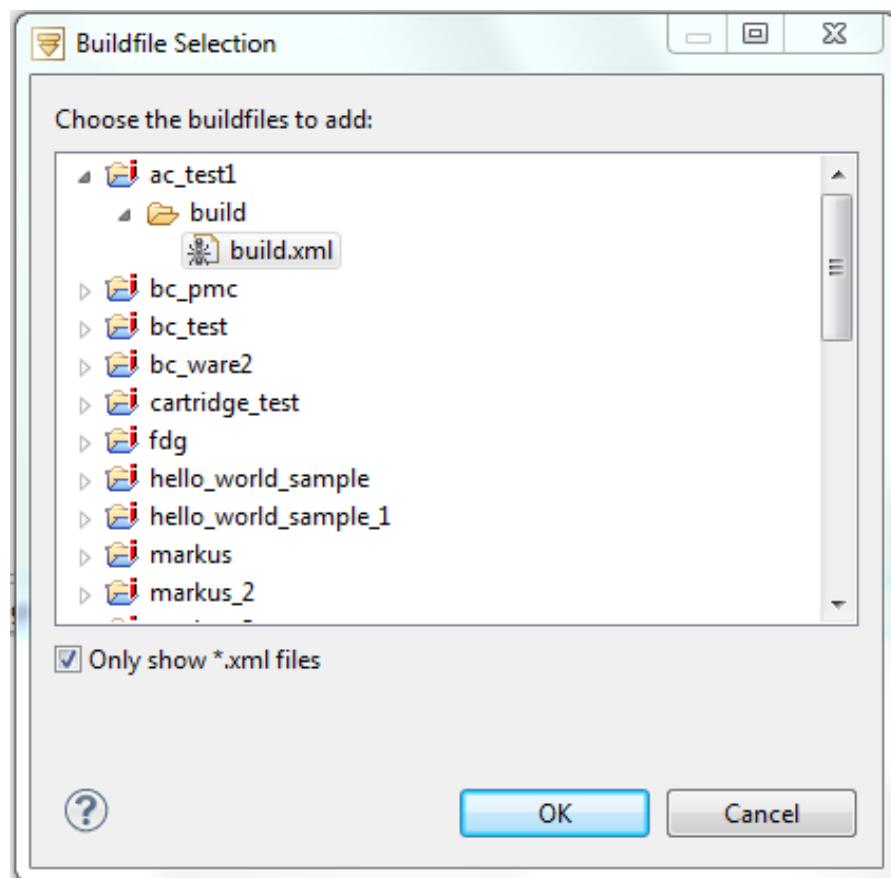
The ANT view provides a convenient environment to run selected ANT tasks.

- 1. Select Window | Show View | Other ... | ANT from the Workbench menu to open the ANT view.**
- 2. Add build files.**

Select the icon from Ant view toolbar to add build files. The Choose Location window is displayed.

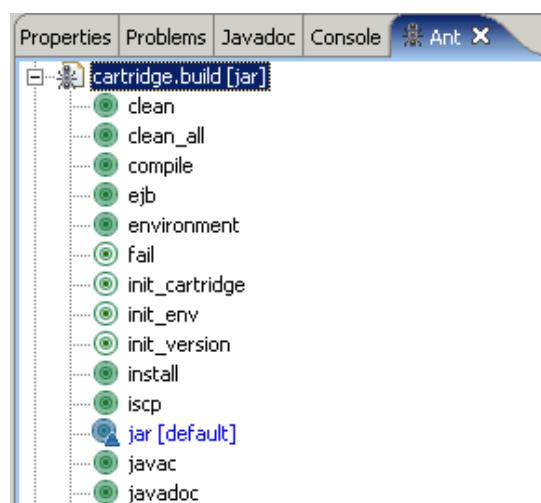
3. Expand the respective project and select the build directory to display available ANT targets.

Figure 33. Select Build Files



4. Select the build.xml file, then OK.
5. Double-click the ANT target to execute in the ANT view.

Figure 34. Available ANT targets



NOTE: The ANT view displays a project name as defined in the build.xml file of the project. The default is cartridge.build. When working with multiple projects in parallel, change the project names in the build.xml files to the cartridge name.

Run ANT Manually

Alternatively to running a build from within Intershop Studio, ANT can be executed from the command line as follows:

1. **Open a DOS command window or console and change to the \build directory inside the cartridge development directory.**

For example, change to <IS_SOURCE>\<cartridge_name>\build\ .

2. **Prepare the environment for ANT.**

Run the environment.bat file, located in <IS_HOME>\bin.

3. **Run the ant command.**

This automatically loads the build.xml file, starting the build process. To restrict the build process to particular targets, provide the respective target as parameters, e.g., ant javadoc. To get help for ANT, enter the following commands:

```
ant -help      //lists the general targets
ant -projecthelp //lists the special build targets
```

Localizing Templates During the Build Process

A special build file (localize.xml) is provided which makes it possible to create localized template sets for multiple languages automatically during the build process. The localization process is based on the template localization tool (tLoc), hence requires tLoc being installed.

The build file:

- identifies and tags the localizable strings in the template set of a cartridge
- creates a localized template set

The actions defined in the build file expect the following configuration files:

Table 8. Configuration files for template localization

File	Description
tloc/etc/prepares_config.xml	tLoc configuration file for the prepare-step (i.e., the process that tags localizable strings inside the templates).
tloc/etc/extracts_config.xml	tLoc configuration file for the extract-step (i.e., the process that computes the IDs for each localizable string inside the templates).
tloc/etc/<locale_ID>/merges_config.xml	tLoc configuration file for the merge step, (i.e., the process that generates the localized template set, using a pre-defined dictionary file containing the translations). You need to provide one configuration file per locale, located in individual subdirectories.

File	Description
IS_SOURCE/<cartridge_name>/staticfiles/tloc/dict (*_dict.*)	The dictionary file that is used to create the localized template set. If no dictionary can be found, the localized template set will use the strings from the original template set. Only CSV dictionaries are currently supported. Note that the translated templates are stored in IS_TARGET/<cartridge_name>/generated/tloc.

NOTE: Localizing the cartridge upon build requires a dictionary file <cartridge_name>_dict.csv (which may be empty) in <IS_SOURCE>/<cartridge_name>/staticfiles/tloc/dict.

For background information on the localization process, tLoc and tLoc configuration, see the [Template Localization Guide](#).

NOTE: The localize.xml target is not included with the set of default build targets. In order to localize templates, you have to select localize.xml as build target when starting the build.

Cartridge Editor

Intershop Studio features the Cartridge Editor which enables developers to manage all important cartridge properties in one place. The Cartridge Editor opens upon creating a new Intershop 7 cartridge, i.e., after completing the cartridge wizard (see [Creating New Cartridges](#)).

To open existing cartridges within the Cartridge Editor:

1. **Select Cartridge Explorer.**
2. **Rightclick the desired cartridge.**
Context menu opens.
3. **Click Open with | Other.**
Editor Selection dialog starts.
4. **Select Cartridge Editor.**
5. **Confirm dialog with OK.**

The Cartridge is open in Cartridge Editor.

The Cartridge Editor provides several tabs that group certain sets of cartridge details. These details tabs include:

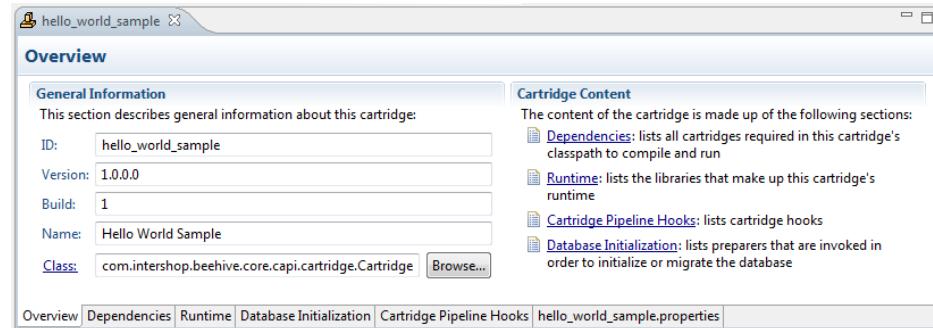
- Overview
- Dependencies
- Runtime
- Database Initialization
- Cartridge Pipeline Hooks
- Properties

The following sections describe the settings accessible on these tabs in detail.

Cartridge Editor: Overview

On the Overview tab, you can edit general properties of the cartridge, including ID, version, build, name, and cartridge controller class.

Figure 35. Cartridge Editor Overview tab



Cartridge Editor: Dependencies

Figure 36. Cartridge Editor's Dependencies tab

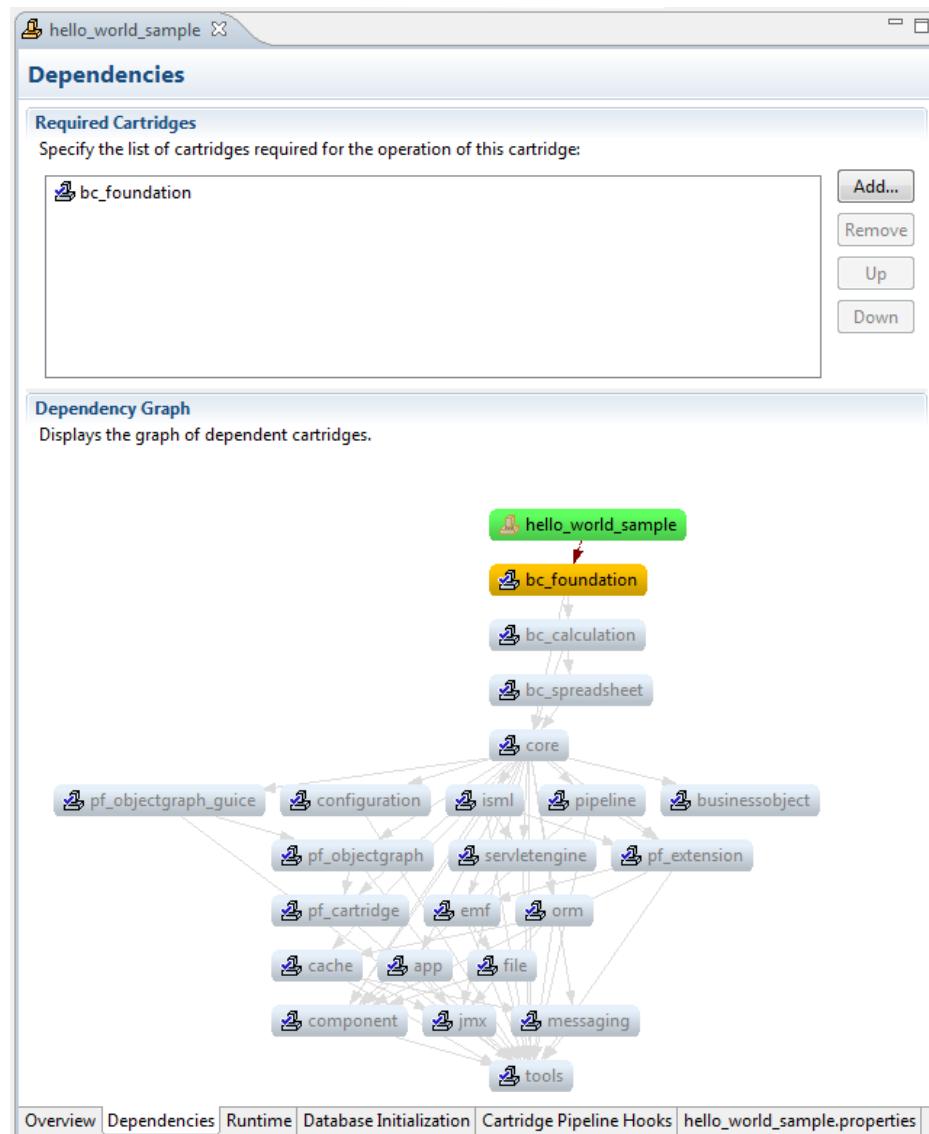


Table 9. Cartridge Editor's Dependencies tab

Option	Description
Add	Allows for adding a cartridge to the list of cartridges that are required for the current cartridge's operation. Clicking Add opens an according Select Cartridge dialog.
Remove	Allows for removing a cartridge from the list of required cartridges. To do so, select the cartridge to be deleted in the list, and click Remove.
Move Up/Move Down	Allows for re-sorting the list of required cartridges, i.e., changing the dependency hierarchy. To change a cartridge's ranking, select it in the list, and click Up or Down, accordingly.

NOTE: The changes made to the cartridge's dependencies are immediately visualized in the dependency graph below the list.

Cartridge Editor: Runtime

The Runtime tab is divided into two panels: Cartridge Lib Folder and Cartridge Project Class Path.

On this tab, you can manage additional libraries that the cartridge uses, or edit the cartridge class path, respectively.

Figure 37. Cartridge Editor's Runtime tab

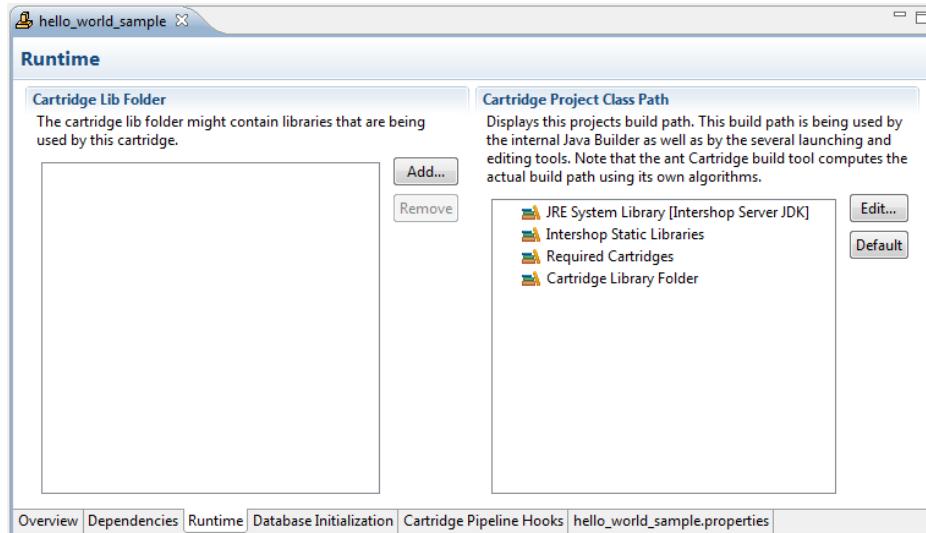


Table 10. Cartridge Editor's Runtime tab

Option	Description
Add	Allows for adding additional libraries required for the cartridge's operation. Clicking Add opens a dialog for selecting the resource from your file system (starting in <IS_HOME>).
Remove	Allows for removing a resource from the lib folder. To do so, select the library, archive, etc. in the list, and click Remove.
Edit	Clicking Edit opens the Java Build Path page of the cartridge's properties dialog, where you can specify: <ul style="list-style-type: none"> • source folders, • required projects, • JARs and class folders, • build class path order. For more information about the cartridge classpath, see <i>Managing Classpath Settings</i> .
Default	If, for any reason, cartridge project's class path settings are not working properly, you can reset the settings to the default values.

NOTE: For common projects, it is not necessary to modify the cartridge project's class path. Should it be necessary to make changes, it may be that the the settings do not seem to work not properly. In such a case it may be helpful to refresh the class path.

To refresh class path:

- 1. Click Project from menu bar.**
- 2. Select Refresh/Reset Cartridge Classpath.**

Refresh/Reset Cartridge Classpath dialog starts.

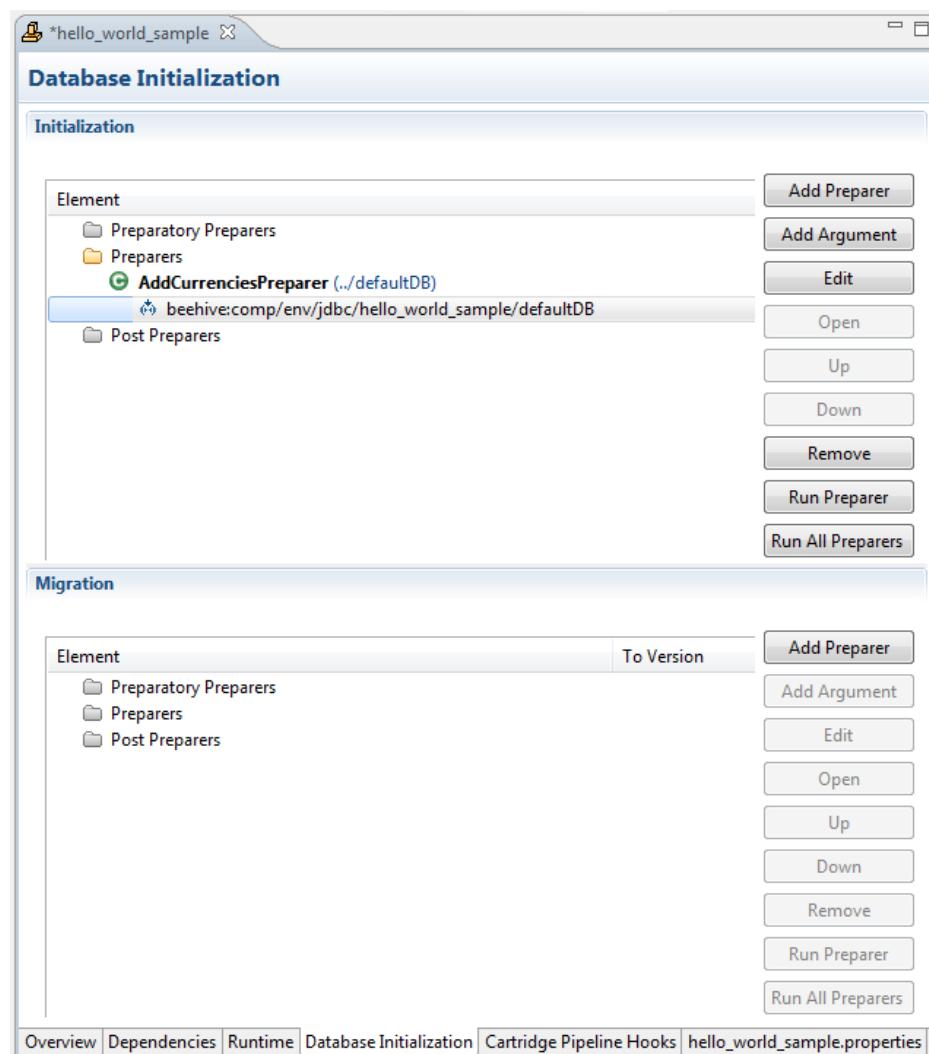
- 3. Select Refresh.**

Cartridge's class path is refreshed.

Cartridge Editor: Database Initialization

The Database Initialization tab is divided into two panels: Initialization and Migration. These panels list the preparers that the current cartridge requires to initialize or, respectively, migrate the database. Using this tab, you can manage the preparers to be invoked.

Figure 38. Cartridge Editor's Database Initialization tab



On both the Initialization and the Migration panel, the following options are available:

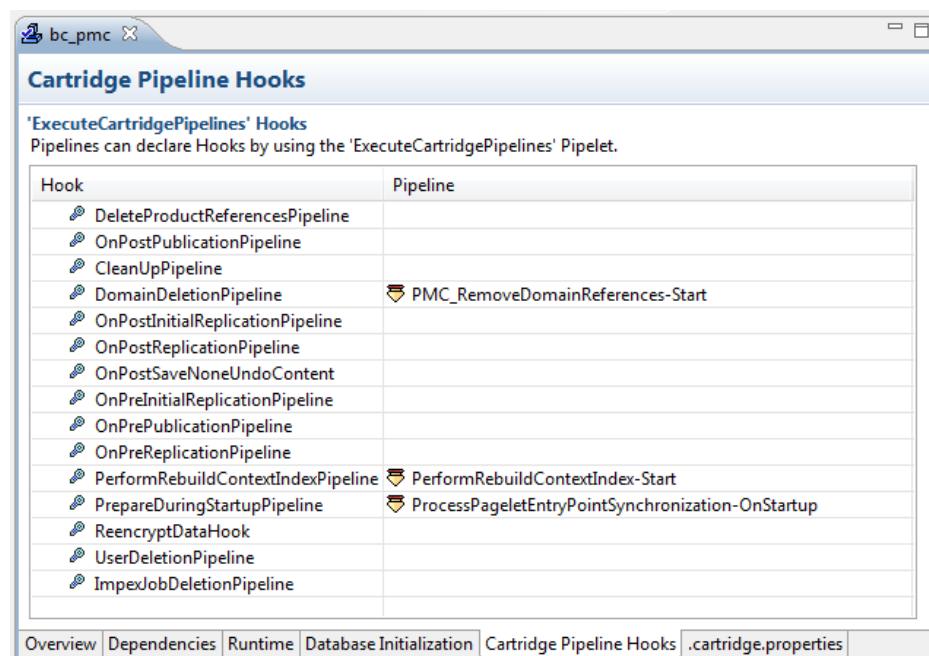
Table 11. Cartridge Editor's Database Initialization tab

Option	Description
Add Preparer	Allows for adding preparers, grouped by execution background (preparatory preparers, preparers and post preparers). To do so, select a preparer group in the list and click Add Preparer. This adds a property cell editor in the corresponding preparer group section, prompting you to directly enter a preparer name, to browse for an existing preparer or to create a new one.
Add Argument	Allows for adding arguments for the preparers, e.g., specific properties files. Click Add Argument opens a property cell editor in the corresponding preparer section that prompts you to directly enter a file name, to browse for an existing file or to create a new one.
Edit	Allows for editing the element that is currently selected in the list.
Open	Select a preparer's parameter and click Open. The parameter is opened within its according editor.
Move Up/Move Down	Allows for re-sorting the list of preparers, i.e., changing the execution order. To change a preparer's position, select it in the list, and click Up or Down, accordingly.
Remove	Allows for removing a preparer or an argument from the list, i.e., excluding it from execution. To do so, select the preparer or argument in the list, and click Remove.
Run Preparer	Allows for executing a single preparer from the list. To do so, select the intended preparer in the list, and click Run Preparer.
Run All Preparers	Allows for executing all preparers included in the list. To do so, simply click Run All Preparers.

NOTE: The execution of DBInit/DBMigrate preparers requires the cartridge to be listed in the property cartridges.dbinit in the Intershop 7 cartridgeList.properties file.

Cartridge Editor: Cartridge Pipeline Hooks

The Cartridge Pipeline Hooks tab lists the hooks available in your cartridge and, if any, their assigned pipeline start nodes.

Figure 39. Cartridge Editor's Pipeline Hooks tab

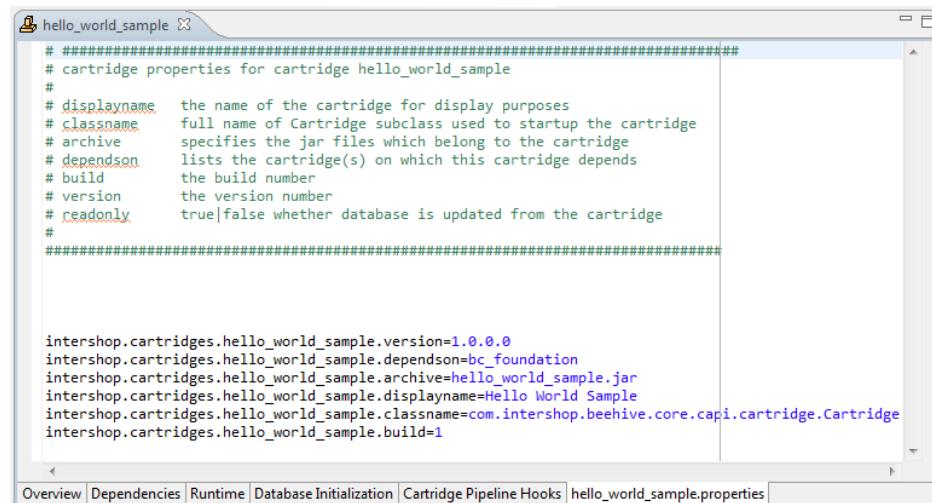
The following options are available:

Table 12. Cartridge Editor's Pipeline Hooks tab

Option	Description
Assigning a pipeline start node	To add a pipeline start node assignment, click the Pipeline column in the row of the hook to be edited. This opens a property cell editor, prompting you to directly enter a pipeline start node, to open a selection dialog or to browse for the intended pipeline.
Editing a pipeline start node	To edit a pipeline start node assignment, click the corresponding pipeline start node. This opens a property cell editor, prompting you to directly edit the selected pipeline start node, to open a selection dialog or to browse for another pipeline.
Accessing the pipeline	To access the pipeline represented by a hook on this tab, double-click the hook name. This opens the corresponding pipeline in the <i>Visual Pipeline Editor</i> , with the <i>ExecuteCartridgePipelines</i> pipelet selected.

Cartridge Editor: Properties

On the Properties tab, you can directly edit the contents of the cartridge's *.properties file.

Figure 40. Cartridge Editor's Properties tab

Development Directory

All cartridge development directories are located directly under a special folder which constitutes the cartridge development root directory. The cartridge development root directory is identified by the variable `IS_SOURCE` in `<IS_HOME>/intershop.properties`. The cartridge development root directory can reside anywhere on the system as long as `IS_SOURCE` is set correctly.

Note that some aspects of the development directory structure are subject to variation. A typical development directory structure is shown in *Table 13, “Cartridge development directories below <IS_SOURCE>”*. Depending on the cartridge functionality, certain sub-directories may be missing, or additional directories serving special purposes may exist.

Table 13. Cartridge development directories below <IS_SOURCE>

Directory Structure	Description
<code><cartridge_dev_root></code>	The directory that contains all cartridge development directories. Stored in environment variable <code>IS_SOURCE</code> of <code>intershop.properties</code> .
<code>..<cartridge_dev_dir></code>	A cartridge development directory
<code>..\\build</code>	Directory storing XML-based build files used by the ANT build tool.
<code>..\\javasource</code>	This directory contains all Java source files and related descriptor files for pipelets and persistent objects.
<code>..\\..\\com</code>	Contains all Java source code files for new pipelets, business objects, and manager classes. The files are stored inside a system of subfolders depending on the package structure of the application unit, such as <code>com/intershop/sample/...</code>
<code>..\\..\\resources</code>	Contains the naming manager configuration files for persistent objects as well as for manager, provider and service classes. Some

Directory Structure	Description
	of these configurations files (e.g. for persistent objects) are typically generated automatically during the build process, while others are set up manually.
..\\..\\test	Contains test cases and related resources for tests based on the Intershop 7 test framework.
..\\model	Contains the UML object model representation. The model is used to generate code for persistent objects, manager and other classes represented in the model using the Intershop Studio EDL model editor.
..\\staticfiles	This directory contains static files, such as pipelines, templates, or images that are part of your cartridge. These files are copied to different locations as defined in the <code>build.xml</code> file.
..\\..\\cartridge	Contains pipelines, templates and additional libraries provided by the new cartridge. Also contains the <code>dbinit.properties</code> file specifying the database preparer classes and the package where property files and scripts used by the preparers are located.
..\\..\\root	Contains files to be copied under the root directory of a Intershop 7 installation. Sub-folders are used to specify the exact location. For example, a file stored under <code>root/lib</code> will be copied to <code><IS_HOME>/lib</code> .
..\\..\\share	Contains files to be copied under the <code>share</code> directory of an Intershop 7 installation. Sub-folders are used to specify the exact location. For example, a file stored under <code>share/system</code> will be copied to <code><IS_HOME>/share/system</code> .

Build Directory

Like development directories, all cartridge build directories are aggregated under a special cartridge build root directory. The cartridge build root directory is identified by the variable `IS_TARGET` in `<IS_HOME>/intershop.properties`. The cartridge development build directory can reside anywhere on the system as long as `IS_TARGET` is set correctly. A typical build directory structure is detailed in *Table 14, "Cartridge build directories below <IS_TARGET>"*.

Note that the structure of the build directory may vary, depending on the components deployed with your cartridge.

Table 14. Cartridge build directories below <IS_TARGET>

Directory Structure	Description
<code><cartridge_build_root></code>	The directory that contains all cartridge build directories. Stored in environment variable <code>IS_TARGET</code> of <code>intershop.properties</code> .
<code>..\\<cartridge_build_dir></code>	A particular cartridge build directory.

Directory Structure	Description
..\\..\\generated	Directory to store classes generated automatically during the build process.
..\\..\\release	Main directory hosting all files generated during the build process.
..\\..\\bin	Contains binary files deployed with a cartridge, such as batch files or shell scripts.
..\\..\\docs	Contains the API documentation (JavaDocs) for public classes generated during the build process.
..\\..\\model	Contains the UML object model representation. The model is used to generate code for persistent objects, manager and other classes represented in the model using the Intershop Studio EDL model editor.
..\\..\\pipelines	Contains the pipelines deployed with the cartridge.
..\\..\\templates	Contains the templates deployed with a cartridge. Templates for import/export purposes are aggregated in a sub-folder / import . Moreover, templates are stored in separate sub-folders, depending on the localeID they are designed for.
..\\..\\static	Contains static files deployed with a cartridge, e.g., images. Images are stored in separate sub-folders, depending on the localeID they are designed for.
..\\..\\lib	Contains jar files deployed with a cartridge.
..\\..\\..\\com/...\\dbinit	Contains sql scripts used in database preparation.

Business Object Development

Business Objects

The Business Object Layer

The business object layer provides an explicit business-oriented domain model as a stable and object-oriented API on top of the data/persistence layer.

The business object layer API models the behavior of business domain objects that operate on data objects representing the state of these domain objects.

The business object layer API fulfills the following purposes:

- It defines entity objects and their relations with each other.
Entity objects are business objects that have an identity (as compared to simple value objects w/o an identity)
- It defines the mapping of entity objects, their attributes, and their relations to an underlying data/persistence layer.

It hereby hides the the underlying internal implementation, which can but doesn't have to be based on the existing Intershop 7 ORM model.

- It provides mechanisms to customize and extend the functionality of existing business objects.

A major advantage of the new business object layer's extension mechanism is, that the existing functionality that comes with the core product can be customized and extended w/o breaking the CAPI barrier and without access to internal implementation details.

Thus you can easily implement expressive and explicit customer-specific domain models on top of generic Intershop 7 business object types.

- It allows the behavior of business objects to be changed dynamically with respect to the application (channel) that uses the business objects.
- It provides a more object-oriented view on the available business functionality, making for example user interface development much easier.
- It permits easy navigation between business objects.

Business Object Layer Terms

This section introduces and defines important code artifacts that are part of the business object layer.

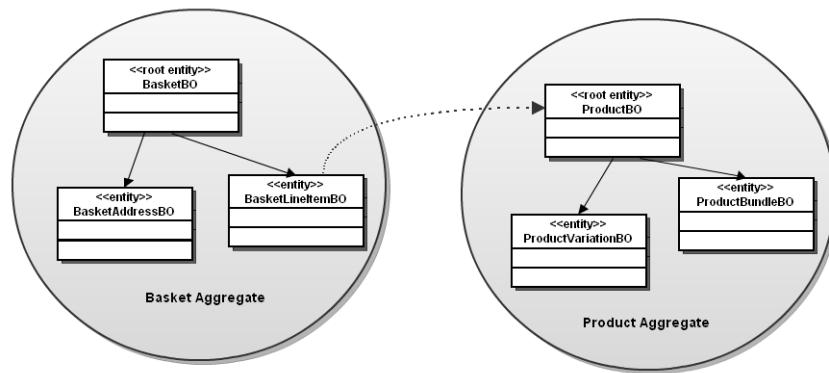
Aggregates

Aggregates encapsulate interrelated business functionality. The term aggregate can be seen as an equivalent for the term 'engine' that is used for technical features.

Aggregates consist of entity objects, value objects, BO repositories, BO extensions, and BO context objects.

In order to be usable in different kinds of applications, aggregates must be designed to be self-contained. Each aggregate is managed in their own cartridge.

Figure 41. Aggregates



Entity Objects

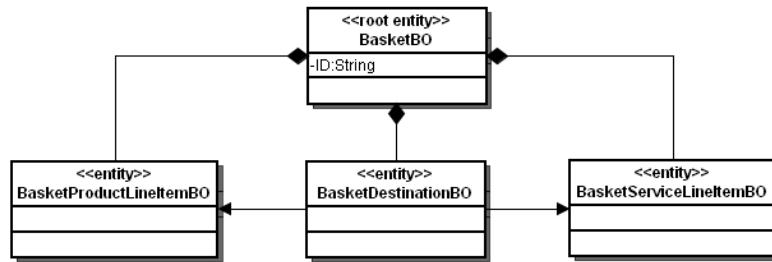
Entity objects are business objects that have their own identity.

Entity objects can have relations with other business objects either within the same aggregate or in foreign aggregates.

Entity objects combine state and behavior. Within entity objects state is represented by attributes and behavior by methods. The methods execute business operations, which usually manipulate the object graph. Often, entity objects are mapped to objects of an underlying persistence layer, so they can be seen as a wrapper around some delegate object.

Entity objects may serve as entry points to aggregates. These entity objects are called root entities.

The aggregate's other inner objects can only be accessed and managed via the root entity. For example, to create an object within an aggregate you use methods dedicated to this purpose either belonging to the root entity or to an entity that you access via a path starting from the root entity.

Figure 42. Entity Objects

Example:

```

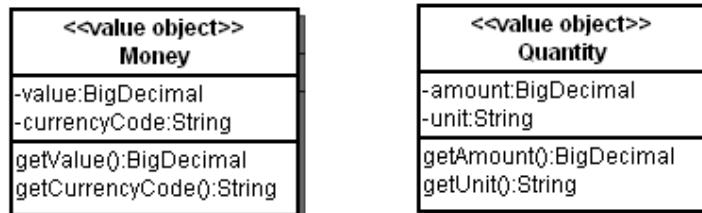
// BasketBO.java

package com.intershop.component.basket.capi;
import com.intershop.beehive.businessobject.capi.BusinessObject;
public interface BasketBO extends BusinessObject
{
    public BasketLineItemBO addProductToBasket(ProductBO product);
    public Collection<? extends BasketLineItemBO> getLineItems();
    public BigDecimal getTotal();
    public BasketBORespository getRepository();
}
  
```

Value Objects

In contrast to entity objects, value objects do not have an identity. Value objects are defined by the values of their attributes and they provide methods for accessing these attributes.

Usually, value objects are used to set the values of attributes of entity objects or for example to store intermediate results during calculations.

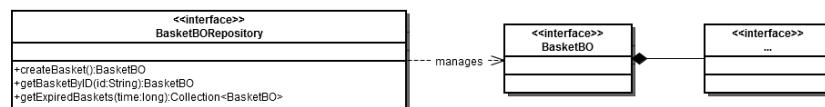
Figure 43. Value Objects

Repositories

A repository is responsible for the lifecycle management of business objects. It handles the creation of new objects, their persistence in the underlying persistence layer, their lookup by some object IDs or other search criteria, and their deletion.

There may be multiple different repository implementations for the same type of business object. For example, the `BasketBORepository` interface could be implemented by an implementation that holds baskets in memory only. Another implementation could store them in the database via the ORM engine. For each aggregate, there is only one repository which manages the root entity. All other entities can be accessed via the root entity methods only.

Figure 44. Repositories



Example:

```

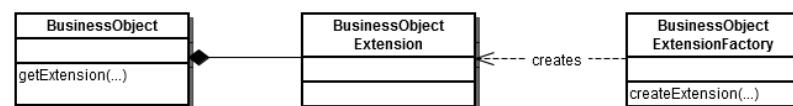
// BasketBORepository.java

package com.intershop.component.basket.capi;
import com.intershop.beehive.businessobject.capi.BusinessObjectRepository;
public interface BasketBORepository extends BusinessObjectRepository
{
    public BasketBO createBasket();
    public BasketBO getBasketByID(String id);
    public Collection<? extends BasketBO> getAllBaskets();
    public void deleteBasket(String id);
}
  
```

Extensions

In order to support the customization and extension of functionality of business objects, business objects can be enhanced by attaching an extension object to them. Extensions are instantiated by extension factories. An extension factory can decide whether it is applicable for a given business object and can create an extension instance for this object. The decision may include checks of the interface type, the implementation type or even the attributes (state) of the business object. The list of available extension factories is the key to achieve an app-specific behavior: by attaching different lists to different apps a business object can behave differently, depending from which app it is accessed.

Figure 45. Business Object Extensions



Example:

```
//RepositoryBOBasketExtensionImpl.java

public class RepositoryBOBasketExtensionFactory extends
    AbstractDomainRepositoryBOExtensionFactory ...
{
    /**
     * The ID of the created extensions which can be used to get them
     * from the business object later.
     */

    public static final String EXTENSION_ID = "BasketB0Repository";
    @Override
    public BusinessObjectExtension<RepositoryBO>
        createExtension(RepositoryBO repository) ...
    {
        return new RepositoryBOBasketExtensionImpl(EXTENSION_ID,
            repository);
    }
}
```

Business Object Context

Context objects are used to provide the runtime context for business objects and their methods. Therefore, every business object lives in a business object context. The context object holds the list of available extension factories. Additionally, it also handles the caching of business object instances.

Every entity implementation will get a reference to the context in which it was instantiated.

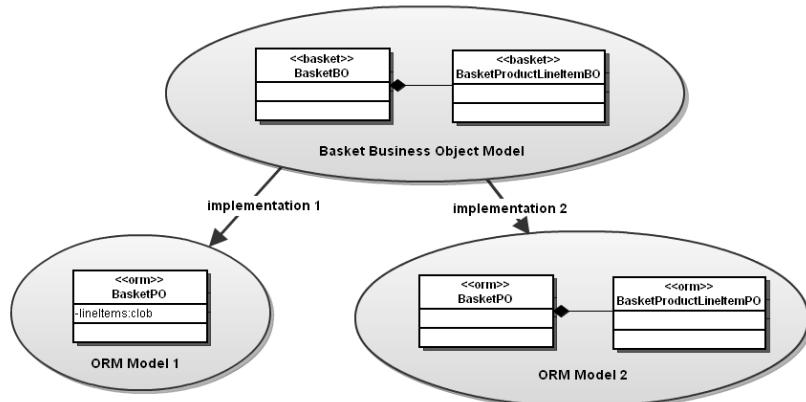
Persistence Layer Mapping

The business object layer is designed for optimum usability by application programmers. It comes with a nicely designed API which reflects all important domain concepts as Java interfaces, methods, etc.

The business object layer API serves as the primary API for programmers in customer projects or Intershop solutions and hides the internal implementation details (like the mapping to the persistence layer) from them. It has to remain stable for a long time.

In contrast, the underlying persistence layer for a business object is designed for maximum performance (read, write). The beauty and stability of the API is secondary, as it will not be exposed to any application programmers. The persistence layer implementation is subject to optimization and therefore to permanent change. It may be redesigned for a better performance without breaking the business object API.

The underlying persistence layer will therefore have a different object structure than the business object layer. Which structure should be chosen cannot be defined here and depends on the concrete requirements for the business object.

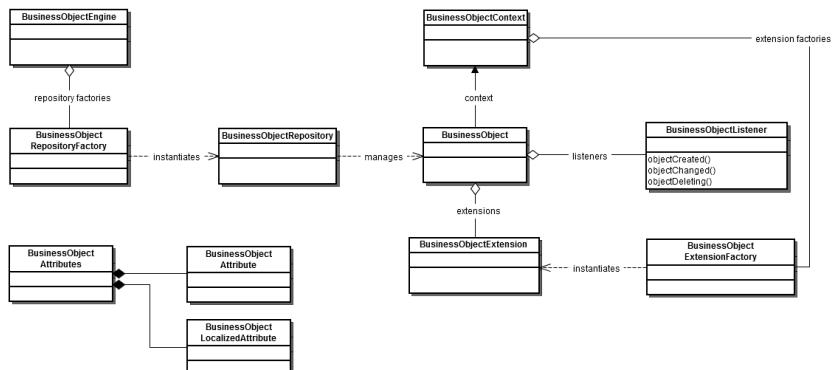
Figure 46. BO to PO Mapping

Business Object Layer Implementation

Business Object Framework

In order to support the development of business objects, Intershop 7 provides a number of super classes and interfaces, which must be subclassed or implemented. The framework classes are located in the *businessobject* cartridge.

The following diagram shows the conceptual structure of the framework:

Figure 47. Business Object Framework Classes

Repository Types

A business object repository can be implemented in several ways:

- The implementation could straight access the underlying persistence layer. This means, the repository provides all objects of a certain type without any additional partitioning. Such an implementation is needed as an entry point into

the business object graph. In order to get such a repository, a repository factory must be implemented that must be registered at the business object engine.

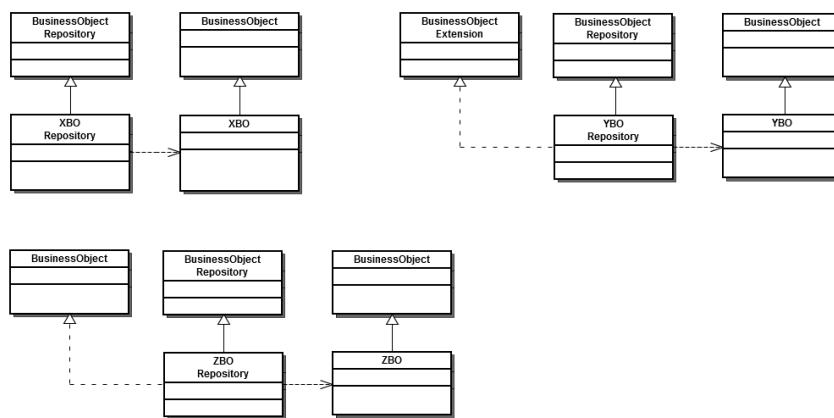
- The implementation could be represented by a business object, which internally serves a repository for other business objects.

With this, multiple repositories representing different partitions can be built.

For example, an address book can be seen as a repository for addresses.

- The implementation can be provided as an extension for another business object. For example, there can be a general RepositoryB0 that is internally mapped to a Domain, and a BasketB0Repository is implemented by an extension that can be attached to the RepositoryB0 and internally manages BasketP0s (which require a domain). A domain in Intershop 7 can be seen as a repository for something that is stored in the database, such as products, baskets, line items, etc.

Figure 48. Repository Types



Business Object Custom Attributes

It may be necessary for an extension to store custom data in the host object. For this purpose, the framework defines an API that can be used to set or get custom attributes and localized custom attributes at a business object.

An extension can be implemented that provides access to such custom attributes. How this API is mapped to an underlying implementation is left to the implementer of a business object. Intershop 7 provides an implementation of such an extension that is applicable to all business objects that internally delegate to an ExtensibleObject. The business object attributes API is mapped to the extensible object attribute values.

Business Object Lifecycle

We must distinguish between two different lifecycles:

- the lifecycle of the Java object representing a business object and
- the lifecycle of the persistent entity

The Java instance is bound to the context, so it is valid as long the context is in use. In Intershop 7, such contexts will have a lifetime of one request, so after the request the BO instance will become invalid and cannot be used anymore.

The persistent entity has a different lifecycle, since such objects can exist for a long time (e.g. as entries in the database).

Business Object Change Listener

Business objects can notify interested listeners about important lifecycle events, like object creation, object changes, and object deletion. Such events refer to the persistent lifecycle of the object, not the Java lifecycle.

Using this notification mechanism, extensions can register themselves at their host object in order to perform necessary cleanup or initialization actions of their underlying persistent state.

Transaction Handling

The business object layer does not define its own transaction handling. Insteads, it relies on the transaction handling of the underlying persistence layer (if applicable).

In Intershop 7, transactions are controlled by the pipeline. This will not change with a new business object layer. The business object implementation should not handle its own transactions, as it may be necessary to synchronize the commits of multiple different business objects with each other. At best, transactions will be handled by the pipeline engine and the underlying ORM engine, so the business object developer does not have to care about transactions at all.

Identifiers

Business objects (e.g. root entities) must have an ID that can be used to look them up later. This ID can be retrieved by the `getID()` method, and is generated by the repository creating the business object.

In the business object layer, such IDs are simply defined as attributes of type `String`. The IDs purpose is an internal one: they are used to lookup objects within a repository, where the repository can also be an entity within an aggregate object (see below). The possible values are not specified here, this is left to the repository implementation.

For repositories that map the objects to ORM objects, it is recommended to use the UUID of the mapped PO as ID.

Depending entities, e.g. objects that are reachable via the root entity only, may also have an ID. This ID can be used within the scope of the aggregate to identify the object. For example, such entities could be looked up by using some method in the root entity (or the entity being the repository of the BO in question). How the IDs of internal entities map to the underlying persistence layer is also left to the implementation. Some implementations may prefer to use UUIDs, some other implementations may rely on auto-generated sequence numbers in the datastore, some others may need to use compound keys, etc.

It is recommended that the ID of non-root BOs contains the ID of their parent BO. This allows looking them up from the outside repository. The ID should not leave Intershop 7, it is for internal purposes only (the only exception is the users browser, so the ID may take part in web forms).

Typically, business objects are also referred by a so-called 'semantic key'. This is the ID seen by the business side. It is used when communication outside of Intershop 7 happens, e.g. for exports and imports. There is no defined structure or naming for this kind of ID, it is up to each BO to define it. It must be documented in the documentation of the BO API. When it exists, it needs to contain the name of the repository the BO lives in, and must be globally unique.

The business ID can be provided by the repository, but it might also be set from the outside (e.g. when importing data). For aggregate objects, the 'inner' objects also need then a semantic key, which needs to contain also the semantic key of their parent object. If no such key can be derived from the existing business data, it needs to be created artificially.

Caching

Business objects are valid as long as their context is valid. Since the context is usually bound to a single request, the lifetime of an object is the time until the request is processed and finished. On the next request, a new context object will be created, so the business object must be retrieved from the repository again.

The repository may implement a caching strategy which returns the same business object instance again, if an object for a given ID is looked up multiple times with the same context.

Additionally, the underlying persistence layer (like the ORM engine) is free to implement its own caching strategy to allow caching across multiple requests or even threads.

Some methods in a business object may be very expensive, so their result should be cached. For example, operations like calculating the total of a basket or converting some XML CLOB from the persistent object back into a Java representation is very expensive. The result of such an operation can be set as an instance variable at the business object.

Since the business object depends on the context, and the context is different for each concurrent request, the transactional isolation between objects is ensured.

Cache Invalidation

If a business object holds its own internal cached state (like for performance reasons), it may be necessary to notify it about changes that affect the validness of such cached state. For example, if a basket caches the total of all its line items, this total becomes invalid if the quantity of a line item is changed.

There are two possible ways to implement such a cache invalidation:

- the business object can register at each important business object directly using the Business Object Listener mechanism and clear the cached stated if such an object is changed
- there can be a central cache invalidation handler that is passed around the whole aggregate (e.g. all business objects within the aggregate) and that notifies all registered objects

The invalidation might have two scopes:

- a business object changes some data, which invalidates other data of this object

- the changes to the business object might invalidate data of other objects related to the changing object

But since each of the implementation of the objects in an object graph may be exchanged independently, the first case cannot really happen - there might always be someone out there relying on the changed data. This means that, when exposed data is changed, all interested parties must be invalidated (when data which is never exposed changes, no other objects need to be notified, because they cannot rely on it).

For doing this, Intershop 7 provides a central `BusinessObjectInvalidationHandler`. This handler knows about all objects which need invalidation, and propagates the invalidation event to them.

This is a two step process:

1. **The root aggregate creates a new handler object on its own creation. This handler is then given to all child objects depending on this aggregate, which in turn can register themselves with the handler.**
2. **When an object changes exposed data, it tells the handler, which in turn notifies all subscribed objects (this includes the calling object, if it is registered).**

Members of an aggregate can pass the `InvalidationHandler` as a constructor argument. The handler is needed for:

- giving the instances to other, newly created instances
- register themselves as listener
- start an invalidation process

Note that the `invalidate()` call to the handler is sufficient: if the caller is registered, it will be called back from the handler. Also, it is not necessary that the caller invalidates its data directly before or after the call to the handler.

When registering as a listener, an anonymous inner class should be used. This avoids exposing the `Listener` interface on the business object itself, so these methods cannot be called from the outside.

Persistent Objects

Persistent Objects and Modeling

Functionality encoded by a cartridge typically requires persistent storage of information in the database. If no suitable database tables exist, new tables have to be created. In order to make these database tables accessible to application components such as pipelets, new persistent objects need to be created at the persistence layer. It may also be necessary to implement additional manager classes, which operate on the new persistent objects.

Creating a manager class on top of a new persistent object is particularly recommended for objects which do not consist of just one PO, but which are represented by a network of POs and helper classes.

API Structure and Interface Types

Intershop 7 exposes persistent objects (POs), managers and providers through well-defined CAPI interfaces which hide actual implementations from the user. This design approach has been chosen to ensure API stability across Intershop 7 releases.

When retrieving a persistent object using the associated manager class, you do not get a handle on the respective PO class implementing the persistent object. Instead, you get an interface (for example, `com.intershop.beehive.xcs.capi.product.Product`) exposing methods you can use to work on the persistent object.

Enfinity Definition Language (EDL)

Intershop 7 includes Enfinity Definition Language (EDL) a textual domain-specific language (DSL) for modeling persistent objects for Intershop 7. For a comprehensive introduction to EDL, see *EDL Reference*.

Persistent Objects

Persistent Object Classes

Persistent objects (POs) are Java objects used to store data persistently to the database. A PO consists of four files:

- **A PO class representing the persistent object**

The respective classes carry the suffix `PO`, such as `ProductPO.java`.

- **A descriptor file describing the object-relational mapping (OR mapping)**

The file extension of the descriptor file is `*.orm`, such as `ProductPO.orm`.

- **A factory class for lifecycle management**

The respective classes carry the suffix `POFactory`, such as `ProductPOFactory.java`.

- **A key class for identifying persistent objects**

The respective classes carry the suffix `POKey`, such as `ProductPOKey.java`.

These classes are part of the internal package of a cartridge and not exposed publicly.

As discussed in *CAPI Interfaces*, the PO class typically implements a interface (e.g., `Product.java`), through which the PO is made available to clients.

Object-Relational Mapping

- **Mapping Classes to Tables**

In Intershop 7, single POs are mapped to single tables or database views. Other possibilities like mapping classes to join tables or other structures are not used within Intershop 7. Each instance of the PO corresponds to a row. Attributes of the class correspond to columns. The primary key attribute of the class maps

onto the primary key of the table, which is the column (or the columns) used to uniquely identify a row.

■ **Mapping and Inheritance Relations**

Intershop 7 uses an approach according to which only leaf classes are mapped onto the database, whereas super classes must be abstract. Tables for leaf classes contain both the attributes inherited from the super class as well as their own attributes.

Managers - Access to Persistent Objects

Manager classes provide access to POs, as well as methods to perform simple operations on these POs. Manager classes are transient Java classes that have a public constructor. For each registered manager, there always exists only one instance.

Base Classes for Persistent Objects

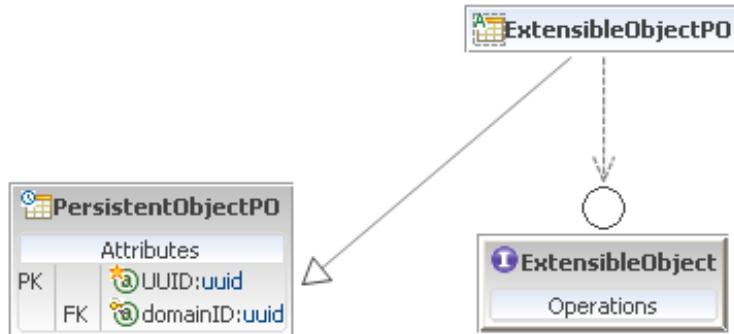
Many POs in Intershop 7 are derived from the general base classes `PersistentObjectPO` or `ExtensibleObjectPO`, which are both part of `com.intershop.beehive.core.capi.domain.PersistentObjectPO` automatically provides a range of attributes, ensuring these attributes are defined in a way compatible with Intershop 7 mechanisms and processes. Attributes automatically provided by `PersistentObjectPO` include:

- **A UUID as primary key**
- **An optimistic control attribute (oca)**
- **A domainID attribute**

This is a foreign key attribute used to reference the domain to which the persistent object belongs. Classes derived from `PersistentObjectPO` automatically inherit methods to get and set the `domainID`, as well as methods to get and set domain instances for a persistent object referenced by the `domainID`.

```
Domain getDomain()
void setDomain(Domain aDomain)
String getDomainID()
void setDomainID(String aDomainID)
```

In addition, `ExtensibleObjectPO` (a sub-class of `PersistentObjectPO`) provides functionality which allows developers to add custom attributes at runtime, as described in *Extensible Object Attributes*.

Figure 49. ExtensibleObjectPO and PersistentObjectPO base class

Modeling Example for Persistent Object Classes

The following EDL snippet defines a persistent object, including a relation respective a dependency to other objects.

```

namespace com.intershop.training.bc_warehouse.internal
{
    orm class WarehousePO extends ExtensibleObjectPO implements Warehouse
    {
        index(addressID);

        attribute name : string<256> required;
        attribute location : string<256>;
        attribute capacity : int;
        attribute description : string localized;
        attribute addressID : uuid;

        dependency address : Address handler "com.intershop.beehive.core. ...
            capi.profile.ProfileMgr"
        {
            foreign key(addressID);
        }

        relation stockItemPOs : StockPO[0..n] inverse warehousePO ...
            implements stockItems;
    }
}
  
```

Native Attributes

What Are Native Attributes?

Native attributes are persistent attributes which directly map onto columns in the database table represented by a PO. They are part of the object model of the PO.

Native attributes are stable, which means that they cannot be created nor can their definition be modified at runtime. Therefore, native attributes are primarily used to model attributes which every instance of a PO is supposed to possess. Note also

that native attributes are not localized. Attributes that are subject to localization have to be modeled as extensible object attributes.

For each native attribute, Intershop Studio creates a pair of attribute accessor methods: a method to read, and a method to set the attribute value (unless the attribute modifier `readonly` has been set to true, see below). These methods always have the signature:

```
AttributeType getAttribute() void setAttribute(AttributeType aValue)
```

where `Attribute` designates the name of the attribute in the object model, and `AttributeType` the attribute's data type.

Data Types and Type Mapping

In Intershop 7, various type conversions are performed for the attributes of persistent objects.

- The EDL model editor provides a number of elementary types that can be used to model attributes of POs. The types are either standard Java types like Java primitives, custom types contained in the model, or artificial types that are translated during the code generation process into a real Java type.
- The model types are translated into a Java type that is used in the API of the generated Java class for the PO, e.g. for the attribute's setter and getter methods.
- In addition, the model types are translated into a JDBC type that is used to store the attribute in the database. The JDBC type is declared in the ORM deployment descriptors.
- The ORM engine itself provides a mapping of the JDBC types into Oracle types.

The respective mappings for supported data types are listed in the following table:

Table 15. Attribute Type Mapping

Model Type	Java Type	JDBC Type	Oracle Type
<code>boolean</code>	<code>boolean</code>	<code>BOOLEAN</code>	<code>NUMBER(1)</code>
<code>byte</code>	<code>byte</code>	<code>TINYINT</code>	<code>NUMBER(4)</code>
<code>char</code>	<code>char</code>	<code>CHAR</code>	<code>VARCHAR2(3)</code>
<code>short</code>	<code>short</code>	<code>SMALLINT</code>	<code>NUMBER(6)</code>
<code>int</code>	<code>int</code>	<code>INTEGER</code>	<code>NUMBER(11)</code>
<code>integer</code>	<code>int</code>	<code>INTEGER</code>	<code>NUMBER(11)</code>
<code>long</code>	<code>long</code>	<code>BIGINT</code>	<code>NUMBER(21)</code>
<code>float</code>	<code>float</code>	<code>FLOAT</code>	<code>FLOAT(63)</code>
<code>double</code>	<code>double</code>	<code>DOUBLE</code>	<code>FLOAT(126)</code>
<code>decimal</code>	<code>java.math.BigDecimal</code>	<code>DECIMAL</code>	<code>NUMBER(38,6)</code>
<code>java.math. BigDecimal</code>	<code>java.math.BigDecimal</code>	<code>DECIMAL</code>	<code>NUMBER(38,6)</code>
<code>java.math. BigInteger</code>	<code>java.math.BigInteger</code>	<code>DECIMAL</code>	<code>NUMBER(38,6)</code>
<code>time</code>	<code>java.util.Date</code>	<code>TIMESTAMP</code>	<code>TIMESTAMP</code>
<code>datetime</code>	<code>java.util.Date</code>	<code>TIMESTAMP</code>	<code>TIMESTAMP</code>
<code>java.util.Date</code>	<code>java.util.Date</code>	<code>TIMESTAMP</code>	<code>TIMESTAMP</code>

Model Type	Java Type	JDBC Type	Oracle Type
char*	java.lang.String	VARCHAR0*	VARCHAR2()*
byte	byte	BLOB	BLOB
blob	java.sql.Blob	BLOB	BLOB
clob	java.sql.Clob	CLOB	CLOB
text	java.lang.String	CLOB	CLOB
java.io.Serializable	java.io.Serializable	BLOB	BLOB
Serializable	java.io.Serializable	BLOB	BLOB

(*) a length must be specified

In addition, there are some complex types that are mapped to artificial names in the object model. The complex types consist of one or more attributes that are mapped again according to their type.

Table 16. Mapping of Complex Types

Model Type	Java Type
rate	com.intershop.beehive.foundation.quantity.ExchangeRate
ExchangeRate	com.intershop.beehive.foundation.quantity.ExchangeRate
money	com.intershop.beehive.foundation.quantity.Money
Money	com.intershop.beehive.foundation.quantity.Money
quantity	com.intershop.beehive.foundation.quantity.Quantity
Quantity	com.intershop.beehive.foundation.quantity.Quantity
productref	com.intershop.beehive.xcs.common.ProductRef
ProductRef	com.intershop.beehive.xcs.common.ProductRef
User	com.intershop.beehive.core.capi.user.User

NOTE: The cartridge core provides a file types.edl, which lists primitive data types, known to the code generator and declares external data types. This file will be imported automatically in new EDL model files.

Extensible Object Attributes

What Are Extensible Object Attributes?

When modeling a PO, it is often impossible to predict which attributes each particular instance of the PO requires. Moreover, certain attributes may be relevant for a certain subset of PO instances only.

For example, with respect to the CAPI object Product, an attribute like Color may be relevant for clothing, but less relevant for books. Therefore, additional attributes may have to be added to a PO at runtime, depending on the exact usage of the object.

The base class ExtensibleObjectPO provides a wide range of methods to add extensible object attributes at runtime, and to read and modify these attributes and attribute values later on.

Consider the following two methods made available by ExtensibleObjectPO :

```
void putString(String aName, String aValue)
```

```
void putInteger(String aName, Integer aValue)
```

Calling the first method creates a custom attribute with the first parameter defining the name, e.g., Color, and the second parameter setting the attribute's value, e.g., red. The second method works similarly, except that the attribute is of a different data type.

For each put method, a corresponding get method exists which is used to retrieve the value of an extensible object attribute, passing the attribute name as parameter.

```
String getString(String aName)
Integer getInteger(String aName)
```

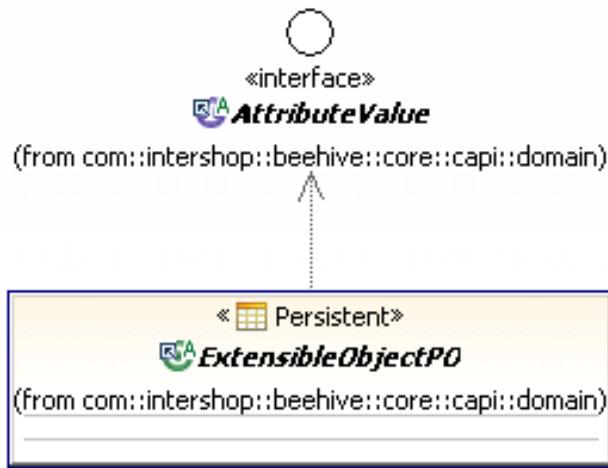
Besides using these general methods for attributes which are unknown at design time, it is possible to define extensible object attributes directly in the object model. For extensible object attributes defined in the object model, Intershop Studio generates the standard set of getter and setter methods, such as

```
String getDescription()
Integer setDescription(String aValue)
```

AttributeValue Tables

To store extensible object attributes, each PO based on ExtensibleObjectPO is associated with a special PO implementing com.intershop.beehive.core.capi.domain.AttributeValue.

Figure 50. AttributeValue tables



For example, the PO WarehousePO is associated with the PO WarehousePOAttributeValuePO, which implements AttributeValue. Note that the associated attribute-value PO is also a persistent object.

WarehousePOAttributeValuePO maps onto a special table storing all extensible object attributes defined for the WarehousePO class.

Each instance of a class derived from AttributeValue represents an attribute of a specific type with a specific value belonging to a specific instance of an ExtensibleObjectPO. Going by example, each custom attribute-value pair defined for a particular warehouse is stored in its own instance of WarehousePOAttributeValuePO, thus mapping on a particular row in the underlying

database table. The Intershop Studio automatically generates the necessary AttributeValue classes where necessary.

NOTE: Extensible object attributes are not stored in the database table underlying a particular persistent object. Instead, extensible object attributes are stored in a separate attribute-value table which is associated with the main table of the persistent object.

Localizable Extensible Object Attributes

Attribute values sometimes have to vary as a function of the locale used. Localizable extensible object attributes provide the necessary mechanism.

Like extensible object attributes, localizable extensible object attribute can be defined directly in the object model. As a consequence, Intershop Studio generates the following set methods (assuming a string attribute Description):

```
void setDescription(String aValue, LocaleInformation aLocale)  
void setDescription(String aValue)
```

The first method allows you to supply a locale along with a value for Description, tying the value to this particular locale. The second method automatically binds the attribute value to the current request locale (if defined) or to the lead locale (fallback). A similar logic applies to the get methods generated by Intershop Studio.

For extensible object attributes not specified in the model, a similar mechanism exists, based on the following methods:

```
void putString(String aName, String aValue, LocaleInformation aLocale)  
void putString(String aName, String aValue)
```

The first method creates an attribute with a particular name and a particular value specific to a particular locale. The second method creates an attribute-value pair bound to the lead locale.

Note that localizable extensible object attributes are also always represented as instances of AttributeValue classes.

Replicated Extensible Object Attributes

Replicated extensible object attributes combine the properties of native attributes and extensible object attributes. This mechanism creates attributes which are generated as part of the PO, and which are replicated inside the POs AttributeValue table. This setting is used for native attributes which should participate in the search mechanisms that operate on AttributeValue tables.

Data Types

Data types you can use for extensible object attributes include:

- String
- Character
- Int
- Double
- Long
- Boolean
- BigDecimal

- Date
- Quantity
- Money
- Text
- com.intershop.beehive.core.capi.domain.PersistentObject

Direct Custom Attributes

What Are Direct Custom Attributes?

Like the extensible object attribute mechanism, direct custom attributes (DCA) provide a way to enhance existing POs with additional attributes. With respect to object relational mapping, however, direct custom attributes behave much like native attributes. Direct custom attributes are mapped onto core table columns of the PO to which they are added. They are not stored in the associated AttributeValue table. This may increase performance, e.g., when using these attributes as conditions in database queries. Because the attributes are stored directly in the database table underlying a PO, executing a query using these attributes does not necessitate table join operations between the PO's core table and the associated AttributeValue table.

The direct custom attribute mechanism is supported by the Intershop 7 import/export framework. Note that the following restrictions apply to direct custom attributes:

- DCAs can only be added to subclasses of ExtensibleObject
- DCAs cannot be localized. They are always bound to the lead locale.
- DCAs cannot be added or modified at runtime. Adding DCAs requires calling an ALTER TABLE statement. The ALTER TABLE statement must consider staging environments.

The following example for an SQL block adds a column to table BASICPROFILE:

```
begin
staging_ddl.add_column('BASICPROFILE', 'MYCOLUMN', 'VARCHAR2(100CHAR)');
end;
/
```

The first parameter denotes the table name, the second parameter the column name, and the third parameter the column definition. The stored procedure checks internally whether a staging environment exists and calls the required commands accordingly. You can execute the SQL block either manually from an SQL console or as part of a script file that is called by the *SQLScriptPreparer*.

- DCAs for a particular PO must have unique names. Name conflicts lead to errors during server startup.
- There is no automatic Oracle context index creation for DCAs. This must be done manually when implementing a cartridge.

DCA Definition

Direct custom attributes are defined using simple XML-based descriptions. All attribute definitions contributed by a cartridge are declared in the

`directCustomAttributes.xml` descriptor file. The name of the descriptor file is fixed. During cartridge development, the file is located in `<IS_SOURCE>/<cartridge_name>/staticfiles/cartridge`. When the cartridge is deployed on a system, the file is installed in the directory `<IS_SHARE>/system/cartridges/<cartridge_name>/release`.

These are all elements involved in configuring direct custom attributes in a `directCustomAttributes.xml` file:

■ **<is_direct_att>**

The root element

■ **<entity>**

This element is used to encode all attributes to be added to a particular subclass of `ExtensibleObject`. Each entity element embeds `<projectname>`, `<classname>` and one or more `<attribute>` definitions.

■ **<projectname>**

This tag does not have a special meaning in the current context, but is required for compatibility reasons.

■ **<classname>**

Specifies the PO to which a custom attribute is to be added.

■ **<attribute>**

Encodes the properties of an attribute to be added to the class specified by the `<classname>` element of the owning `<entity>`. Note that the following properties are set implicitly:

- `primary-key = false`
- `required-attribute = false`
- `null-allowed = true`
- `optimistic-control = false`

■ **<attribute-name>**

Defines the name of the attribute.

■ **<column-name> (optional)**

Specifies the name of the database column onto which the attribute is mapped. If missing, the attribute name is assumed to be the column name.

■ **<type>**

Specifies the data type of the attribute. The data type is restricted to `INT`, `DOUBLE`, and `CHAR`, where `INT` is mapped onto the Java type `java.lang.Integer`, `DOUBLE` is mapped onto `java.lang.Double`, and `CHAR` is mapped onto `java.lang.String`.

■ **<type-info> (optional)**

Specifies the length of the data type `CHAR`.

DCA Access

Direct custom attributes can be accessed using the `get` and `put` methods made available by `ExtensibleObject`, such as:

```
void putString(String aName, String aValue)
```

```

void putInteger(String aName, Integer aValue)
String getString(String aName)
Integer getInteger(String aName)

```

Relationships Between Persistent Objects

Typically, your object model consists of a set of POs that do not stand alone, but are connected in various ways. In the object model, these connections between classes are modeled as relationships. In the context of POs, a relationship between two classes indicates a link between the database tables underlying the POs. How Intershop Studio maps relationships in the model on database constructs depends on relationship properties, i.e. role and dependency properties, including the relationship type, the multiplicity indicators on a relationship, and the way a relationship is named. The following sections describe all aspects of relationships in more detail.

Relationship Types: Associations and Weak Relation

Intershop 7 is based on two basic relationship types:

■ Association relationships

Association relationships express a uni- or bidirectional semantic connection between classes. In UML, there are several types of association relationships: association proper (as shown in the EDL snippet below), aggregation, and composition. To the EDL model editor, these distinctions are irrelevant and will therefore be ignored in what follows.

The following EDL snippets define a relation between the two objects `WarehousePO` and `StockPO`.

Relation definition at the `WarehousePO` side:

```
relation stockItemPOs : StockPO[0..n] inverse warehousePO implements stockItems;
```

Relation definition on the `StockPO` side:

```
relation warehousePO : WarehousePO[1..1] inverse stockItemPOs implements warehouse
{
    foreign key(warehouseID) -> (UUID);
```

■ Weak relations

Weak relations are specific to Intershop 7. Weak relations express a unidirectional relationship between classes. "Unidirectional" in this context means that the relationship is only navigable in one direction. Weak relations are modeled as dependency.

The following EDL snippet (from `WarehousePO`) models the weak relation connecting `WarehousePO` and `Address`. It expresses that we only navigate the relationship from `WarehousePO` to `Address`, never the other way around.

```
dependency address : Address handler "com.intershop.beehive.core. ...
    capi.profile.ProfileMgr"
{
    foreign key(addressID);
}
```

Compared to associations, weak relations are internally treated in a very different way. Weak relations are not registered in the ORM deployment descriptor file and they do not require to re-create the referenced class. Therefore, weak relations can be used to link custom PO classes to CAPI objects that Intershop 7 provides (such as Product).

Multiplicity of Relationships

In UML, relationships are characterized by multiplicity labels on classes. These labels express how many objects, i.e. instances of a class, participate in the relationship, indicating how many objects are linked to one another.

Note that UML uses a wide variety of different multiplicity labels, such as 1, 1..1, 1..n, 0..n, 3..25, etc.

Depending on the multiplicity types on each side of the relationship, three basic kinds of relationships are commonly distinguished:

- One-to-one
- One-to-many
- Many-to-many

Distinguishing these relationship types is crucial, since they are mapped onto database constructs in very different ways.

Relationship Names and Roles

Typically, relationships are named, the name expressing the semantic content of the relationship. There are two ways to name a relationship:

- You can provide a name for the dependency, i.e. the relationship itself. This approach is taken when modeling weak relations.

dependency address : Address ...

- You can provide names for the roles that the classes play that are connected via the relationship. This approach is taken when modeling association relationships.

relation stockItemPOs : StockPO[0..n] ...

One-to-Many Relationships

One-to-many relationships are common in relational database design, allowing the representation of complex data sets in an economic way. For example, each instance of `WarehousePO` points to the instances of `StockPO` that belong to it. Likewise, each `StockPO` points to its `WarehousePO`.

To relate warehouse and stock data as described, a special column is needed in the `StockPO` table, identifying the correct `WarehousePO` instance for each `StockPO` instance. This special column acts as the foreign key, and it commonly maps onto the primary key of the related table, in our example, the `WarehousePO` table.

Returning to the representation of relationships, the implication is that you must introduce a special attribute that serves as the foreign key and maps onto the primary key attribute of the associated PO class. In a one-to-many relationship, the

foreign key is defined within the class on the "many-side," e.g. StockPO, and it maps onto the primary key of the class on the "one-side," e.g. WarehousePO.

EDL snippet from the "many-side," e.g. StockPO:

```
index(warehouseID) ;

attribute warehouseID : uuid required readonly;

relation WarehousePO : WarehousePO[1..1] inverse stockItemPOs implements warehouse
{
    foreign key(warehouseID) -> (UUID);
}
```

Note that one-to-one relationships are organized in precisely the same way, with the exception that the foreign key can be assigned to either class.

For one-to-many relationships, Intershop Studio creates special methods for both classes involved, the class on the "many-side" and the class on the "one-side." Methods differ depending on which class is considered.

■ Many-Side Methods

The methods generated for the class on the "many-side," e.g. StockPO, allow you to access the role representing the class on the "one-side," e.g. WarehousePO.

```
public WarehousePO getWarehousePO()
```

The method is generated inside the code section `relationship accessor methods`.

■ One-Side Methods

For the class on the "one-side," e.g. WarehousePO, Intershop Studio generates a method to access associated the role representing the class on the "many-side," e.g. StockPO:

```
public Collection getStockItemPOs() public Iterator createStockItemPOsIterator()
```

In addition, Intershop Studio creates relationship wrapper methods. These methods check whether a particular element participates in the relationship:

```
public boolean isInStockItemPOs(StockPO anElement) public int getStockItemPOsCount()
```

Weak Relations

Weak relations are unidirectional relationships to classes which implement `PersistentObject`. Unidirectional means that they can be traversed only in one direction: from the source class to the target class. The target class always has a multiplicity of 1. In the EDL snippet from `WarehousePO` below, the weak relation expresses that each instance of `WarehousePO` is associated with exactly one instance of `Address`. In contrast to normal association relationships, the weak relation does not express the opposite statement, namely that each instance of `Address` can be associated with one or more instances of `WarehousePO`.

```
dependency address : Address handler "com.intershop.beehive.core.capi...
    profile.ProfileMgr"
{
    foreign key(addressID);
}
```

The code that Intershop Studio generates for weak relations only affects the source class, not the target class. Therefore, weak relations are particularly useful if you

create new POs coupled with existing objects (such as `Address`) which you are unable to recompile or change.

Another common use case is hiding the relation to the "One-Side-PO" of a PO located in the same cartridge, because it is not intended to make this relation "public" for this PO. In that case the handler must not be defined and the factory of the PO is called to locate the related instance by a primary key lookup.

Note that a foreign key is needed inside the source class, e.g. `Warehouse`, mapping onto the primary key of the target class, e.g. `Address`. The foreign key enables you to identify the particular target class instance, e.g. a particular product, to which a source class instance, e.g. a particular `WarehousePO`, is connected.

The following EDL snippet from `WarehousePO` declares the foreign key attribute `addressID`:

```
attribute addressID : uuid; index(addressID);
```

Weak Relations to CAPI Interfaces

Typically, a weak relation connects a custom PO with other persistent objects represented by a CAPI interface.

In our example, the weak relation connects the custom PO `WarehousePO` with the standard Intershop 7 CAPI interface `Address`. When modeling a weak relation like this, you have to provide the symbolic name of the manager which provides access to the object behind the CAPI interface.

Intershop Studio uses the provided manager to create weak relation accessor methods.

NOTE: The provided factory has to provide a `resolve<ClassName>FromID()` method. Moreover, the business object behind the interface has to provide a `getUUID()` method. Otherwise, compile errors may result. Going by example, when constructing a weak relation from a custom class to the CAPI interface `Address`, the respective manager would be `ProfileMgr` (which provides a `resolveAddressFromID()` method).

Generated Methods for Weak Relations

Intershop Studio generates a method which allows you to access the target class instance to which a particular source class is connected. For example, for the weak relation connecting `WarehousePO` with `Address`, the following method is generated (inside the code section `weak relation accessor methods`):

```
public Address getAddress()
{
    if (getAddressIDNull())
    {
        return null;
    }

    ProfileMgr factory = (ProfileMgr) NamingMgr.getInstance()...
        lookupManager(ProfileMgr.REGISTRY_NAME);

    return (Address) factory.resolveAddressFromID(getAddressID());
}
```

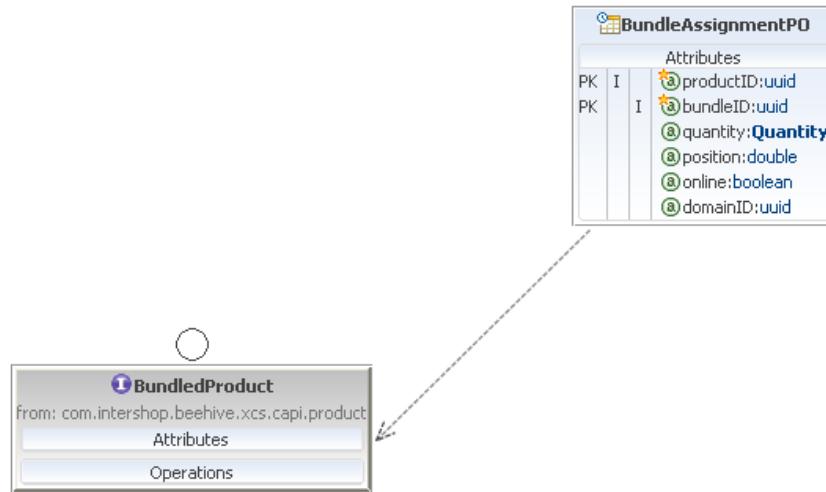
Notice that the method calls `resolveAddressFromID()` from `ProfileMgr`, which has been provided as factory.

Many-to-Many Relationships

Many-to-many relations are common in Intershop 7 applications. For example, consider the relation between product and supplier: Each supplier may deliver more than one product, while each product may be delivered by more than one supplier. Or, product bundles, which combine products to form a new product, e.g. a computer, mouse, and monitor may be bundled into a complete computer package. Here too, a many-to-many relationship exists, because each product bundle can contain many different products, and each product can be part of different product bundles. Note that this relationship is reflexive, because product bundles are also products.

Intershop 7 uses so-called assignment classes to simulate many-to-many relationships. Consider *Figure 51, “Assignment classes in the representation of product bundles”* which expresses the relationship between products and product bundles (which again are products). Via the assignment class `BundleAssignmentPO`, product bundles are paired with products by pairing values for `bundleID` (a foreign key attribute referencing UUIDs of product bundles) with values of `productID` (a foreign key attribute referencing products that participate in a bundle). Both foreign keys map to the primary key of `Product` (`UUID`).

Figure 51. Assignment classes in the representation of product bundles



Note that `BundleAssignmentPO` is a PO but is not derived from `ExtensibleObjectPO`. `BundleAssignmentPO` has a natural primary key, composed of the two foreign key attributes.

Managers for Persistent Objects

What Are Managers?

Manager classes provide access to POs, as well as methods to perform simple operations on these POs. Manager classes are transient Java classes that have a public constructor. For each registered manager, there always exists only one instance.

NOTE: It is not recommended to instantiate a manager class using the public constructor.

Managers are registered and looked up via the naming manager (see *Configure the Naming Manager* for details on manager registration). Manager implementation classes should carry the suffix `MgrImpl`, such as `WarehouseMgrImpl`.

Base Class

Manager implementation classes are derived from the base class `com.intershop.beehive.core.capi.common.ManagerImpl`.

Manager Methods

The set of methods implemented by a manager class will vary depending on the type and the complexity of the persistent objects it is responsible for. In fact, you are free to implement any method you think is needed. For example, methods provided by a manager may (but do not have to) include:

- Methods to create, get, and remove persistent objects
- Methods to manage relations between persistent objects

Modeling of Managers and the respective Implementations

Although possible from a technical point of view, Intershop itself does not model the Intershop 7 manager interfaces and their respective implementing classes any longer.

Instead Intershop recommends using the Intershop Studio Java editor to write the necessary manager interfaces and classes explicitly.

CAPI Interfaces

What Are CAPI Interfaces?

Intershop 7 exposes POs, managers and providers through well-defined CAPI interfaces which hide actual implementations from the user. This design approach has been chosen to ensure API stability across Intershop 7 releases.

When retrieving a persistent object using the associated manager class, you do not get a handle on the respective PO class implementing the persistent object. Instead, you get an interface (for example, `com.intershop.beehive.xcs.capi.product.Product`) exposing methods you can use to work on the object.

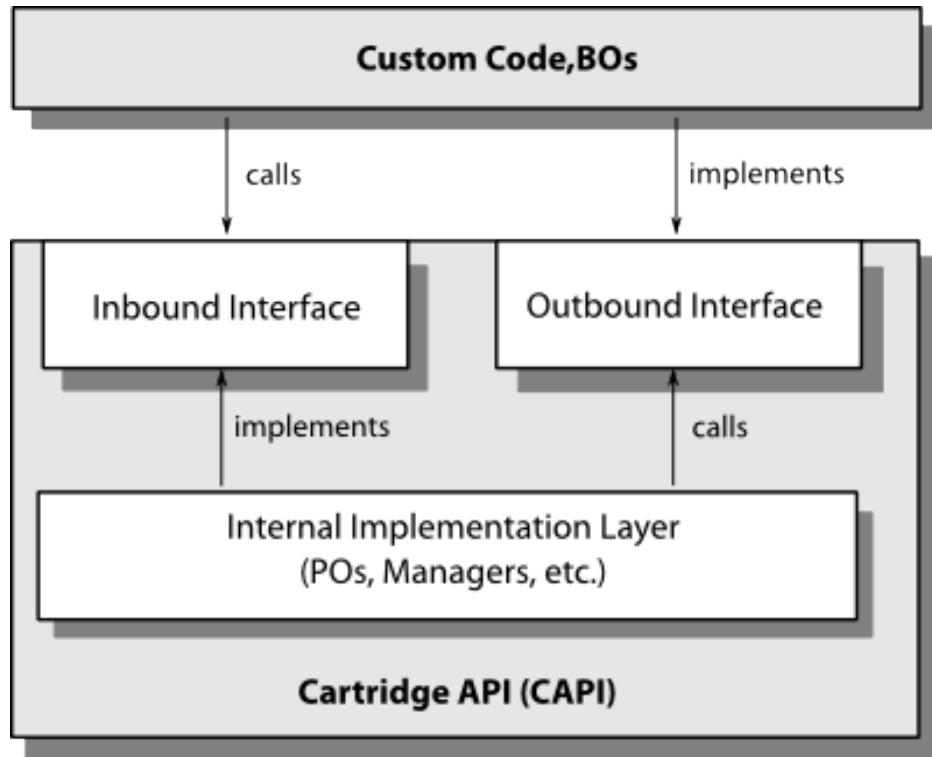
Intershop 7 stores public cartridge interfaces and internal implementations in distinct packages. For each cartridge, public interfaces are assembled in the sub-package `capi`. These interfaces constitute the Cartridge API (CAPI). Implementation classes are assembled in the sub-package `internal`. Classes in the sub-package `internal` are not exposed publicly and are not included during JavaDoc generation.

NOTE: It is recommended for custom projects to adopt the same API structure.

Inbound and Outbound CAPI Interfaces

Intershop 7 distinguishes two types of CAPI interfaces: inbound and outbound interfaces.

Figure 52. Inbound and outbound interfaces



Inbound Interfaces

Inbound interfaces provide access to functionality made available by Intershop 7. Inbound interfaces are called from within custom code (e.g. BOs, custom pipelets) to access persistent objects and managers, such as the product manager, products, or baskets. The actual implementation of inbound interfaces is encoded within the standard set of cartridges. As a developer, you don't need to worry about details of the implementation.

CAUTION: Do not extend or re-implement inbound interfaces through custom code. Inbound interfaces may be subject to change as part of platform updates, e.g. by adding new methods. Consequently, custom code implementing such an interface will not compile in the new API version. Such code therefore breaks the API.

Outbound Interfaces: Provider Classes

Outbound interfaces are represented by provider classes. They provide an easy mechanism to plug custom code into pre-defined cartridge logic provided by Intershop 7. Outbound interfaces are called when executing standard low level business processes. In contrast to inbound interfaces, outbound interfaces can be (and in fact, are designed to be) re-implemented by custom code.

CAPI Interfaces and Persistent Objects

An example for a set of CAPI interfaces modeled on top of custom POs is shown in the following EDL snippets:

```
import "enfinity:/demo/edl/com/intershop/demo/capi/Warehouse.edl";
namespace com.intershop.demo.internal
{
    orm class WarehousePO extends ExtensibleObjectPO implements Warehouse { ... }
}

import "enfinity:/demo/edl/com/intershop/demo/capi/Stock.edl";
namespace com.intershop.demo.internal
{
    oca orm class StockPO implements Stock { ... }
}
```

- The PO `WarehousePO` implements the interface `Warehouse`.
- The PO `StockPO` implements the interface `Stock`.

The interfaces are part of the `capi` package while the implementation classes are part of the `internal` package.

Note that POs and interfaces use different base classes:

- The PO `WarehousePO` inherits from the abstract class `com.intershop.beehive.core.capi.domain.ExtensibleObjectPO`.
- The interface `Warehouse` extends the interface `com.intershop.beehive.core.capi.domain.ExtensibleObject`.

CAPI Interfaces and Managers

Although possible from a technical point of view, Intershop itself does not model the Intershop 7 manager interfaces and their respective implementing classes any longer.

Instead Intershop recommends using the Intershop Studio Java editor to write the necessary manager interfaces and classes explicitly.

The separation of interfaces and implementing classes into separate packages (`capi / internal`) applies for managers as well.

Example:

- The CAPI interface extends `com.intershop.beehive.core.capi.common.Manager`.
- The manager implementation class `WarehouseMgrImpl` implements the interface `WarehouseMgr`.
- The manager implementation class inherits from `com.intershop.beehive.core.capi.common.ManagerImpl`.

Providers

Provider classes provide an easy mechanism to plug custom code into pre-defined cartridge logic provided by Intershop 7. Outbound interfaces are called when executing standard low level business processes. In contrast to inbound interfaces,

outbound interfaces can be (and in fact, are designed to be) re-implemented by custom code.

An example for an outbound interface is

`com.intershop.beehive.bts.api.shipping.ShippingCostProvider`. This interface declares methods used by the pipelet `CalculateShippingBasePrices` to calculate shipping costs for a subset of product line items. Intershop 7 provides a default implementation for this interface. However, clients may create custom implementation of this interface to enable additional shipping methods.

Each provider is associated with an abstract class (named

`Abstract<Interface_Name>Impl`) that implements the interface. Custom classes implementing an outbound interface should be derived from the abstract class. For example, a class to re-implement the outbound interface `ProductPriceProvider` should be derived from the abstract class `AbstractProductPriceProviderImpl`. To make sure the outbound interface actually uses the custom implementation, the respective implementation class must be registered with the provider interface at the naming manager.

Abstract base classes have been introduced as an additional layer to make sure that custom code based on outbound interfaces is not affected by API changes. For example, if an outbound interface is enhanced with additional methods as part of a platform update, a modified abstract class will be provided as well ensuring that the new interface methods have a default implementation.

Creating Models

Start the Model Wizard

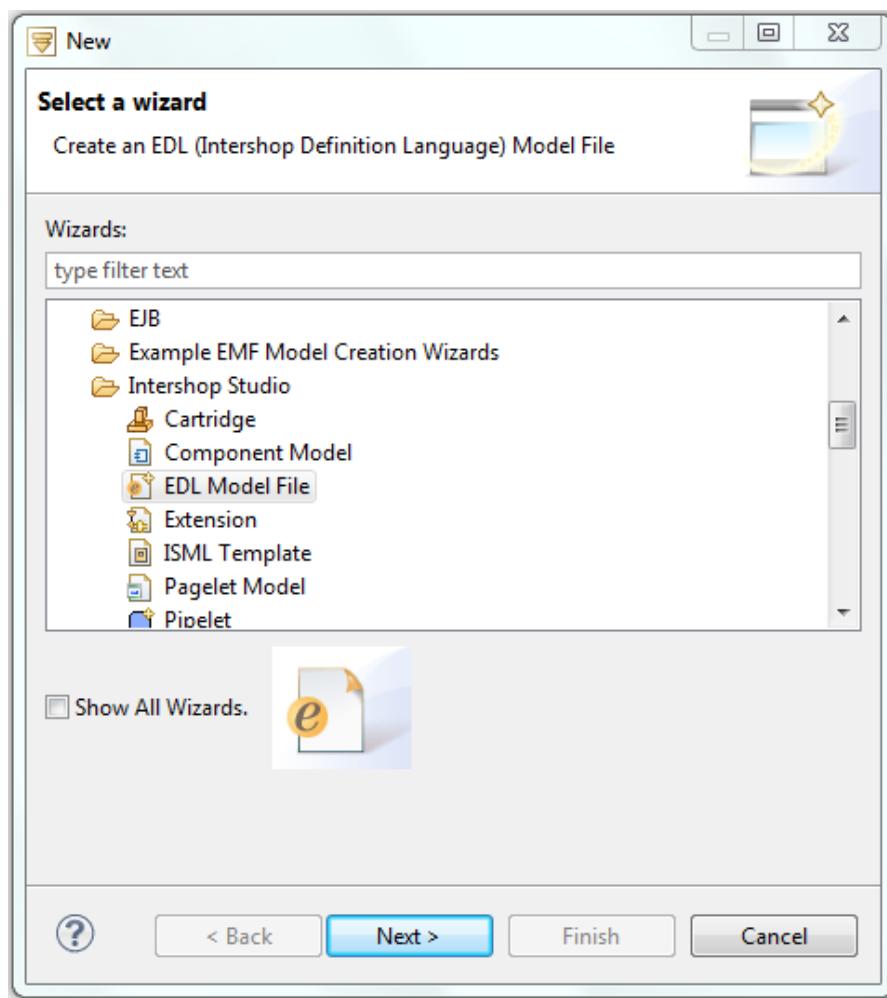
The EDL model file wizard helps you to set up new EDL models.

NOTE: It is recommended to create separate model files for each data object respectively interface.

To start the model wizard:

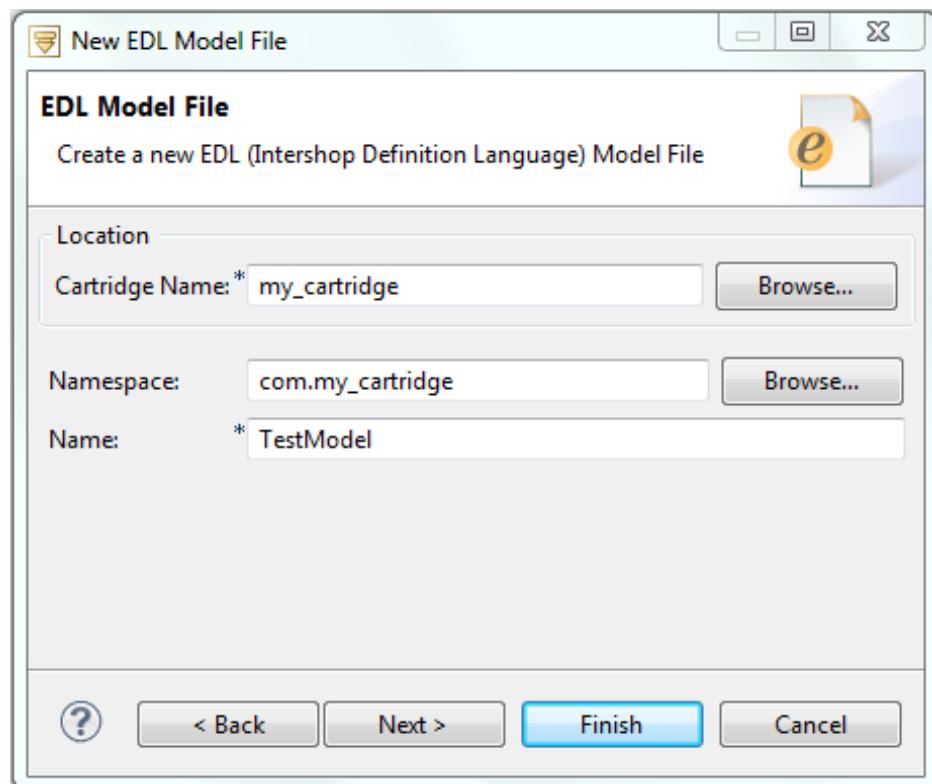
- 1. In the Cartridge or Package Explorer view, select the cartridge project.**
- 2. From the menu bar, select File | New | Other... | Intershop Studio | EDL Model File, then click Next.**

The model wizard is displayed. If you have not selected a cartridge before starting the model wizard, the template wizard first prompts you to select the cartridge for the model.

Figure 53. Selecting the Model Wizard

Set General Model Properties

The model wizard allows you to set general model properties as described in the table below:

Figure 54. General Model Properties**Table 17. General Model Properties**

Property	Description
Location	Defines the cartridge for the new model. System will store the file in the selected cartridge /edl/com/<cartridge_name>. If needed system itself will create the required folders.
Namespace	Enter the namespace in which the model should reside.
Name	The file name must be unique within the cartridge. An EDL file which is named the same as an existing EDL file in the same cartridge will overwrite the existing EDL file. A valid file name must have the extension edl.

Once all properties are defined, click Next.

Define the Initial Model Content

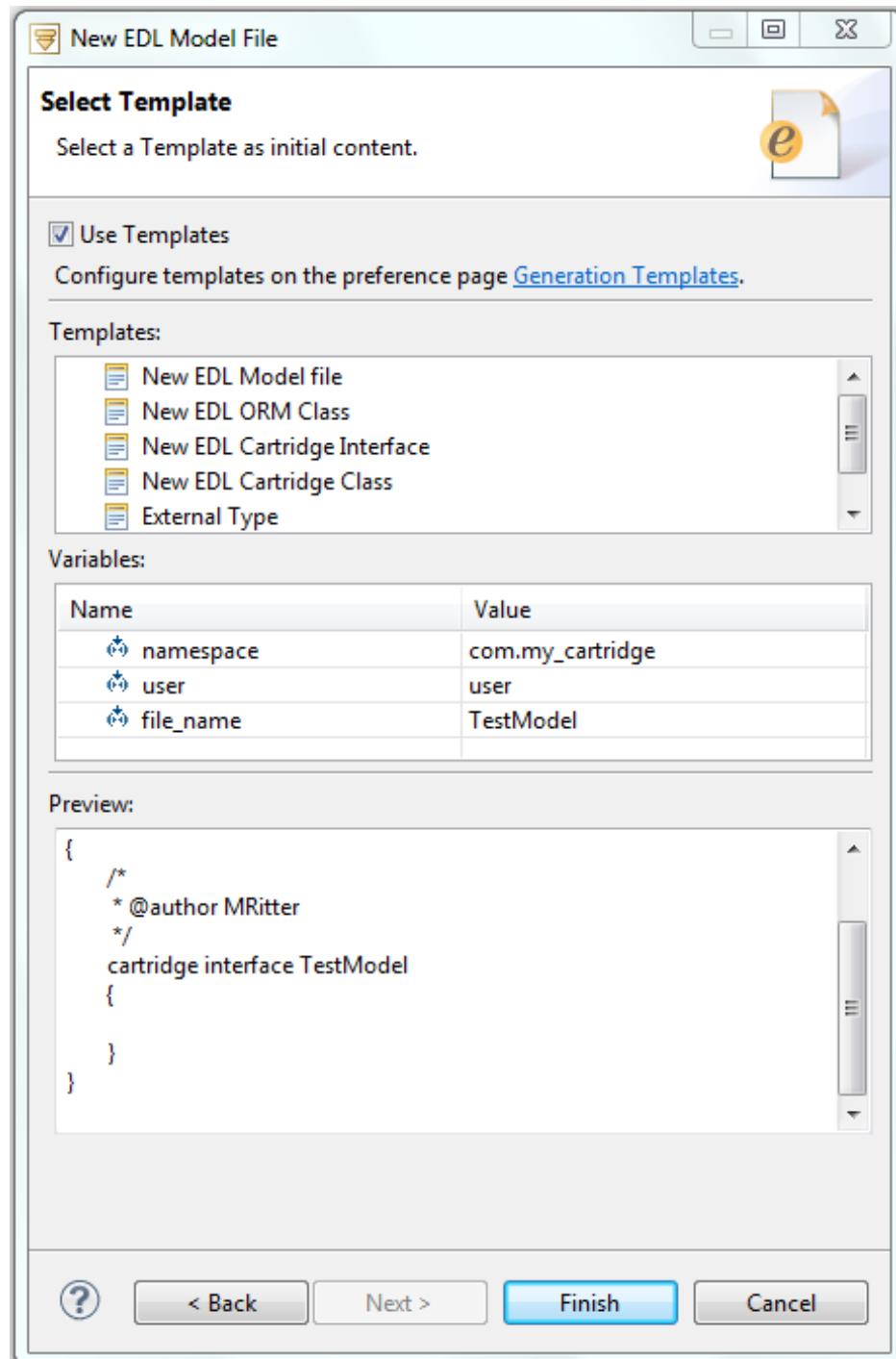
On the next wizard page, you can select a template to define the initial EDL structure to be generated.

You can choose from the templates:

- New EDL Model File
- New EDL ORM Class
- New EDL Cartridge Interface

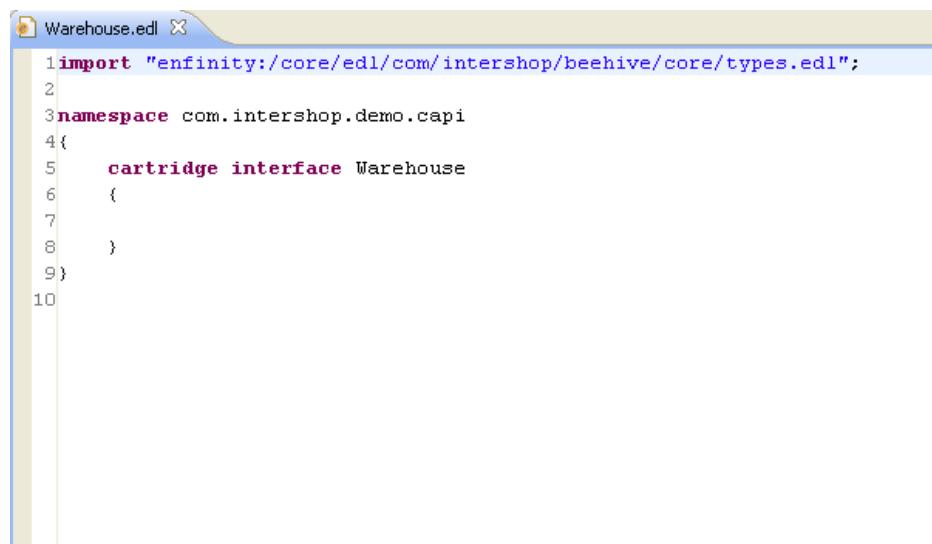
- New EDL Cartridge Class
- External Type

Figure 55. EDL Template Selection



Once all properties are defined, click Finish. This opens the model in the default EDL model editor.

NOTE: All descriptions below refer to the EDL Source Editor.

Figure 56. Initial EDL Model


```

1 import "enfinity:/core/edl/com/intershop/beehive/core/types.edl";
2
3 namespace com.intershop.demo.capi
4 {
5     cartridge interface Warehouse
6     {
7     }
8 }
9
10

```

Import additional EDL files

New EDL models import automatically the EDL file `types.edl`.

You can import additional EDL files or map external types into your model:

1. Use the content assist feature of the EDL Source Editor to insert a required EDL file `import` statement.

```
import "enfinity:/core/edl/com/intershop/beehive/core/capi/profile/Address.edl";
```

2. Use the content assist feature to insert an `external` statement, mapping an existing class as EDL type.

```
external Collection type "java.util.Collection";
```

NOTE: You can use the external statement also for Intershop 7 types that are not fully defined in EDL, like e.g. Manager.

Working with EDL Models

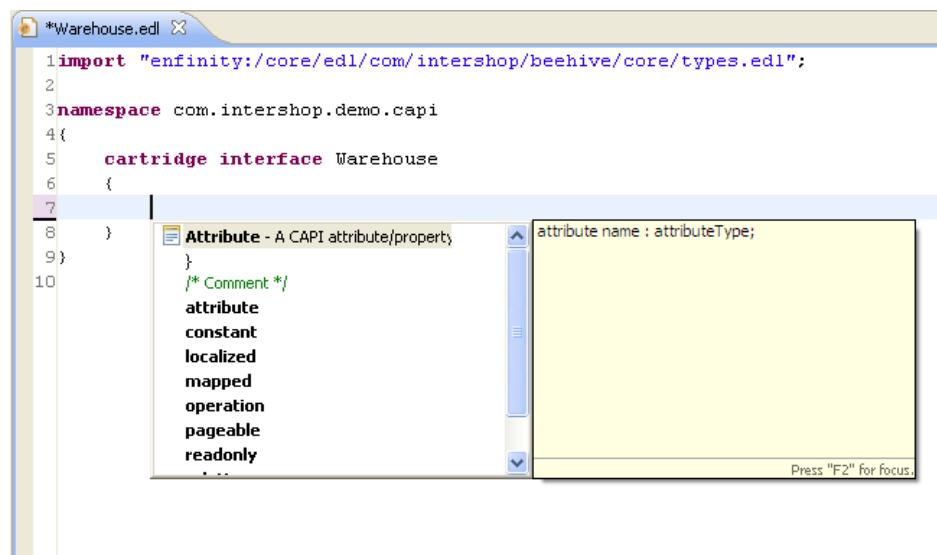
Add New Model Elements

To add new model elements, such as classes or interfaces, you should create the respective EDL files as described in *Creating Models*.

Add Attributes and Operations

To add attributes to a model element, open the respective element's EDL file. Insert your attribute or operation declaration.

NOTE: You can use content assist at various places when creating and modifying object models. To activate the content assist, press **Ctrl + Space**. The content assist displays a floating window with a list of possible options (such as classes or types) available at the current point.

Figure 57. Adding an Attribute

Specify additional settings (attribute modifiers, operation parameters and return values) as required.

Figure 58. Attribute Modifiers

Add Relationships and Dependencies

To add relationships and dependencies, insert and configure the required declaration statements.

Figure 59. Relationship Definition

```

1 import "enfinity:/core/edl/com/intershop/beehive/core/types.edl";
2
3 namespace com.intershop.demo.capi
4{
5     cartridge interface Warehouse
6     {
7         attribute name : string required ;
8
9         relation address: Add
10    }
11}
12
13

```

[0..1]
[0..n]
G AdditionalContent - com.intershop.component.foundation.capi.content
I AdditionalContentMgr - com.intershop.component.foundation.capi.content
I Address - com.intershop.beehive.core.capi.profile
I AddressBook - com.intershop.beehive.core.capi.profile

Delete Model Elements

To delete model elements (classes, interfaces) delete the respective model files itself.

To delete attributes, operations, relations and dependencies, delete the respective declarations in the model file.

Compare Data Object Models

Using the data object model comparer, you can compare an EDL model with either an earlier version of the same model, or with a different EDL model.

Compare Object Model Versions

To compare a object model with an earlier version:

- 1. In the Cartridge or Package Explorer, right-click the EDL model file you intend to compare with an earlier version.**

This opens the context menu.

- 2. From the context menu, select Compare With | Local History ...**

This opens the local history, ordered by revision time.

- 3. From the history, double-click the required entry.**

This opens the comparison window displaying the changes.

For details on the model comparison window, see *EDL Model Comparer*.

Compare Different Object Models

To compare different object models:

1. **In the Cartridge or Package Explorer, select the EDL model files you intend to compare.**
2. **With at least two EDL model files selected, right-click one of the model EDL files.**

This opens the context menu.

3. **From the context menu, select Compare With | Each Other.**

This opens the comparison window displaying the differences.

For details on the model comparison window, see *EDL Model Comparer*.

Modeling Persistent Objects for Intershop 7

Create Business Interfaces

Business interfaces (CAPI interfaces) are used to hide implementations from clients, making it possible to ensure API stability (see *What Are CAPI Interfaces?*).

To model a new CAPI interface, create a new EDL model file based on the template New EDL Cartridge Interface. The modifier `cartridge` will be assigned to this interface.

Specify the inheritance, e.g. `PersistentObject` or `ExtensibleObject`.

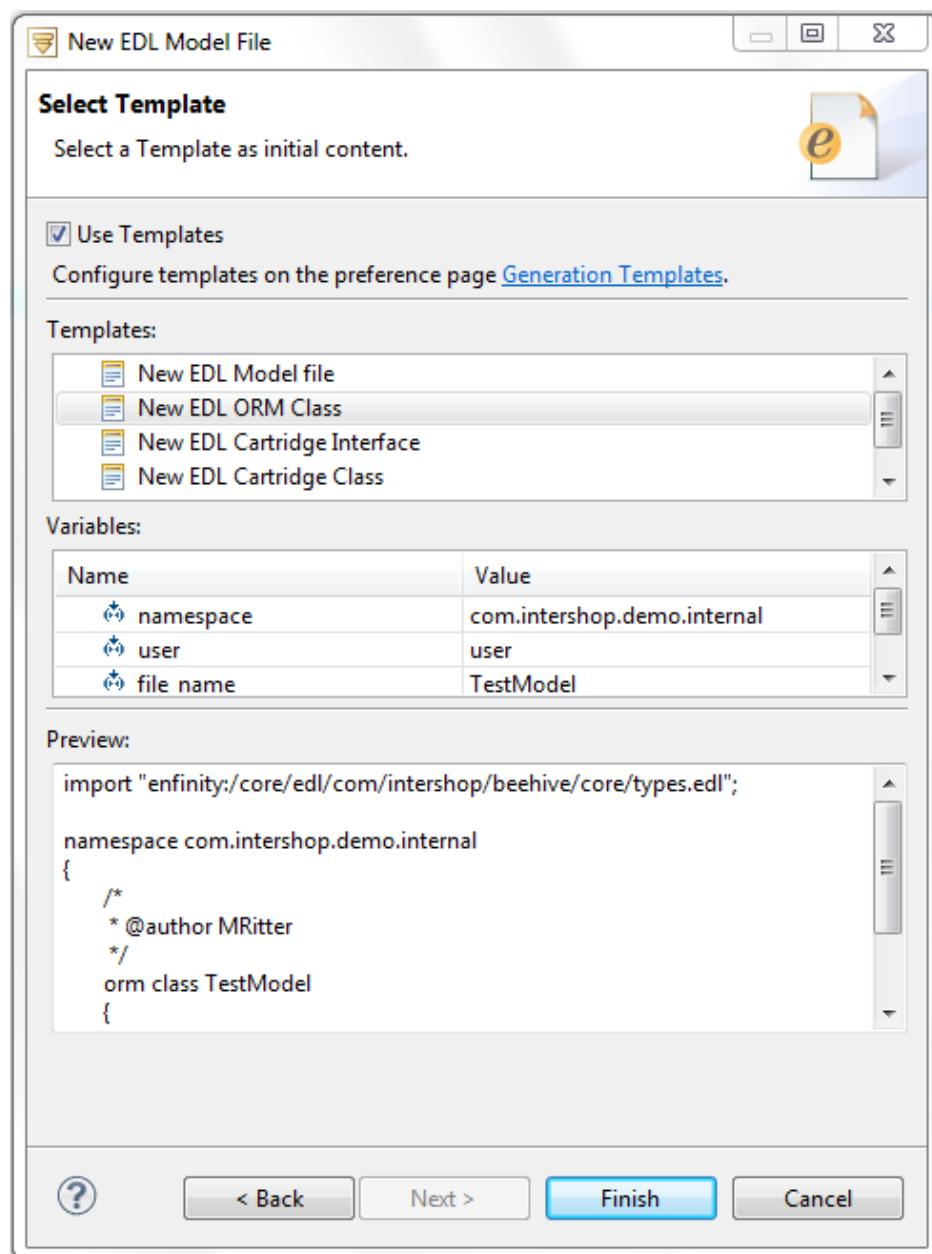
Insert all required attribute or operation declarations.

A sample business interface is shown below.

```
namespace com.intershop.demo.capi
{
    cartridge interface Warehouse extends ExtensibleObject
    {
        attribute name : string required;
        attribute location : string;
        attribute capacity : int;
        attribute description : string localized;
        relation stockItems : Stock[0..n] readonly;
        relation address : Address[0..1];
    }
}
```

Create a Persistent Object

To define a model element serving as persistent object create a new EDL model file based on the the template New EDL ORM Class . The modifier `orm` will be assigned to this class. This ensures that all classes required by the ORM engine are generated by Intershop Studio.

Figure 60. EDL Template New EDL ORM Class

Each persistent Intershop 7 object has to specify a primary key attribute and an optimistic control attribute (OCA).

There are actually 3 choices to ensure this:

■ **Inherit the new object from PersistentObjectPO**

Use this inheritance if your object does not require localized attributes or custom attribute creation at runtime.

The new object will inherit the attributes `uuid` (as primary key), `oca` as optimistic control attribute, `domainID` and `lastModified`.

```
orm class WarehousePO extends PersistentObjectPO
{
```

```
}
```

■ **Inherit the new object from ExtensibleObjectPO**

Use this inheritance if your object requires localized attributes or custom attribute creation at runtime.

The new object will inherit the attributes `uuid` (as primary key), `oca` as optimistic control attribute, `domainID` and `lastModified`.

```
orm class WarehousePO extends ExtensibleObjectPO
{
}
```

■ **Do not inherit from either PersistentObjectPO nor ExtensibleObjectPO; instead of that define explicitly primary key and OCA attributes for your new business object.**

Use the modifier `oca` to enforce the creation of an Optimistic Control Attribute.

Use the constraint `primary key` to define which attribute(s) are part of the primary key.

Use the constraint `index` to define an index on the specified attribute(s).

```
oca orm class StockPO implements Stock
{
    primary key(productID, warehouseID);
    index(productID);
    index(warehouseID);

    attribute count : int;
    attribute productID : uuid required readonly;
    attribute warehouseID : uuid required readonly;
    attribute shelfID : uuid;

    relation warehousePO : WarehousePO[1..1] inverse stockItemPOs ...
        implements warehouse
    {
        foreign key(warehouseID) -> (UUID);
    }
}
```

Create Native Attributes

Native attributes are persistent and directly map onto columns in the database table represented by a PO. For each native attribute, the EDL model editor creates a pair of attribute accessor methods: a method to read, and a method to set the attribute value (unless the attribute modifier `readonly` has been set to true).

```
// native attribute's declaration at the CAPI interface
attribute name : string required;

// native attribute's declaration at the ORM implementation
attribute name : string<256> required;
```

Create Extensible Object Attributes

Extensible object attributes are stored in the `AttributeValue` table associated with a PO. For example, extensible object attributes defined for `WarehousePO` are stored in `WarehousePOAttributeValuePO` (see *Extensible Object Attributes*).

For an attribute to be identified as extensible object attribute, you must assign either the modifier `localized` (for Localized Extensible Object Attribute) or `mapped` (for Extensible Object Attribute).

```
attribute description : string localized;
attribute description : string mapped;
```

Create Association Relationships

Association relationships express a uni- or bidirectional semantic connection between classes. In UML, there are several types of association relationships: association proper, aggregation, and composition. To the EDL model editor, these distinctions are irrelevant.

In order to create an association relationship between two classes, we will have to modify both respective EDL files. Let's have a look at the classes `WarehousePO` and `StockPO`, which are in a bidirectional one-to-many relation.

The `WarehousePO` object represents the "one-side" of the relationship, the specification is done like this:

```
relation stockItemPOs : StockPO[0..n] inverse warehousePO implements stockItems;
```

- `stockItemPOs`: name of the relation.
- `StockPO`: the related ORM class.
- `[0..n]`: the relation's cardinality.
- `inverse warehousePO`: the inverse relation in the target ORM class in case of a bidirectional relation.
- `implements stockItems`: this ORM relation represents the implementation of the specified CAPI relation.

The `StockPO` object represents the "many-side" of the relationship, the specification is done like this:

```
relation warehousePO : WarehousePO[1..1] inverse stockItemPOs implements warehouse
{
    foreign key(warehouseID) -> (UUID);
}
```

- `warehousePO`: name of the relation.
- `WarehousePO`: the related ORM class.
- `[1..1]`: the relation's cardinality.
- `inverse stockItemPOs`: the inverse relation in the target ORM class in case of a bidirectional relation.
- `implements warehouse`: this ORM relation represents the implementation of the specified CAPI relation.
- `foreign key(warehouseID) -> (UUID)`: The foreign key mapping of the relation. For bidirectional relationships, the foreign key mapping must be provided on only one side of the relationship, which is the class that defines the foreign key attributes.

NOTE: For more details please check the EDL reference at [here](#).

Create Weak Relations

Weak relations are unidirectional one-to-many relationships. Weak relations are typically used to connect a custom class with a standard Intershop 7 business interface (such as Address, see *Weak Relations*).

Weak relations are modeled as dependencies between two classes. In order to create a weak relation, modify the EDL file of the custom class. Let's have a look at the class WarehousePO, which is in a unidirectional one-to-many relationship with the standard CAPI interface Address.

```
dependency address : Address handler "com.intershop.beehive.core.capi.profile.ProfileMgr"
{
    foreign key(addressID);
}
```

- address: name of the dependency.
- Address: the related CAPI interface.
- handler "handlerclass": The Java class name of a dependency handler that can resolve the referenced objects by the foreign key attribute.
- foreign key(addressID): The foreign key mapping of the dependency. Currently, a dependency always maps to the UUID of the target type, therefore this attribute must always be a single local attribute of type uuid.

NOTE: For more details please check the EDL reference at here.

Create Manager Implementation Classes

Manager classes are transient Java classes that have a public constructor. For each registered manager, there always exists only one instance.

Manager implementation classes are derived from the base class `com.intershop.beehive.core.capi.common.ManagerImpl` (see *Managers for Persistent Objects*).

```
public class WarehouseMgrImpl extends ManagerImpl implements WarehouseMgr
{
    // here come the manager's methods
    public WarehouseMgrImpl()
    {
    }
}
```

Modeling of Managers and the respective Implementations

Although possible from a technical point of view, Intershop itself does not model the Intershop 7 manager interfaces and their respective implementing classes any longer.

Instead Intershop recommends using the Intershop Studio Java editor to write the necessary manager interfaces and classes explicitly.

Generating Code

Validate the Model

Before generating code, you should validate the model in order to detect possible problems which might prevent code generation. It is possible to validate the model representation of individual elements, or to validate all model elements in a package.

- To validate individual elements (e.g. a class or an interface), right-click into the open respective EDL file or right-click the EDL file in the Cartridge or Package Explorer view. From the context menu, select Code Generator | Validate.
- To validate all elements in a namespace, right-click the corresponding EDL folder in the Cartridge or Package Explorer View. From the context menu, select Code Generator | Validate.

Warning and error messages are displayed in the Problems View.

Generate Code

Code can be generated for single model elements, or for all model elements in a package.

- To generate code for individual elements (e.g. a class or an interface), right-click into the open respective EDL file or right-click the EDL file in the Cartridge or Package Explorer view. From the context menu, select Code Generator | Generate Code.
- To generate code for all elements in a namespace, right-click the corresponding EDL folder in the Cartridge or Package Explorer View. From the context menu, select Code Generator | Generate Code.

NOTE: Code is only generated from models that do not produce any error message during validation.

Generate Database Resource Files

To generate database indexes, database constraints and other database resources, right-click the the package (or the parent EDL folder) in the Cartridge or Package Explorer View. From the context menu, select Code Generator, followed by the required option (e.g. Generate Database Indexes).

How To

Handle Foreign Key Constraints

The EDL model editor can be used to define constraints for the foreign key attributes defined in your model. For example, for the foreign key attribute warehouseID which relates StockPO to WarehousePO (see *Create a Persistent Object*), the following constraint definition will be generated:

```
PROMPT /* Class com.intershop.demo.internal.StockPO */
EXEC staging_ddl.add_constraint('STOCK','STOCK_C0001','FOREIGN KEY ...
```

```
(WAREHOUSEID) REFERENCES WAREHOUSE (UUID) INITIALLY DEFERRED ...  
DEFERRABLE DISABLE NOVALIDATE');
```

If you declare foreign key attributes and want to use constraint definitions for these foreign key attributes in the database, proceed as follows:

■ Invoke Intershop Studio to create the index file dbconstraints.ddl.

This file contains the constraint definitions. The information contained in dbconstraints.ddl is used during the database initialization process.

Right-click the package (or the parent EDL folder) in the Cartridge or Package Explorer View. From the context menu, select JGen | Generate | Database Constraints to generate the dbconstraints.ddl file.

Intershop Studio stores the dbconstraints.ddl file in the dbinit/scripts directory of your cartridge, e.g., <IS_SOURCE>/<cartridge_id>/javasource/resources/<cartridge_id>/dbinit/scripts.

■ Configure the dbinit.properties file to include the dbconstraints.ddl file when creating constraints during the dbinit process.

The dbinit.properties file configures database preparer classes to execute during the dbinit process, and for each preparer class additional files that may be required as arguments. The dbinit.properties file is located in <IS_SOURCE>/staticfiles/cartridges.

Modify this file to contain the following configuration:

```
Class1 = com.intershop.beehive.core.dbinit.preparer.database. ....  
DatabaseConstraintsPreparerresources/<cartridge_id>/dbinit/ ....  
scripts/dbconstraints.ddl
```

The first entry invokes the preparer class (DatabaseConstraintsPreparer) necessary to create constraints in the database. The second entry provides the directory path to the dbconstraints.ddl file.

Note that the index for the keyword Class (1 in the example below) may vary, depending when the preparer should be executed relative to other preparers configured in dbinit.properties.

Handle Index Definitions

To improve database performance, you typically define database indexes on attributes.

An index is a common database device to facilitate search processes. Basically, an index defines a certain way to organize or sort data in the column to which the attribute refers. Just like the index of a book, which helps locate pages containing a certain piece of information, a database index helps identify the table row containing a certain attribute value, enabling you to quickly access other data contained in that row. Note that indexes can be simple or composite, depending on whether they are defined by a single attribute, or a set of attributes. Each attribute can participate in one or more indexes.

The EDL model editor supports the following index types:

■ Primary Index

Defines the primary key of the ORM class, consists of a comma-separated list of ORM attribute names. Due to restrictions in the ORM engine, a primary key can only be declared in the top-most superclasses.

■ Foreign Key Index

Foreign keys are used to represent dependencies between PO classes, reflecting dependencies between tables in the underlying database. A foreign key identifies instances of a dependent PO class. In other words, it identifies rows in a dependent database table. Technically, a foreign key consists of an attribute or set of attributes that are mapped onto the primary key attribute of the dependent table. A foreign key index is an index on one or more foreign key columns in a particular database table.

■ Semantic Key Index

Defines the semantic key (alternative key) of the ORM class, consists of a comma-separated list of ORM attribute names. Duplicate values in the indexed columns are not allowed.

■ Index

Defines an index on ORM attribute(s), consists of a comma-separated list of ORM attribute names. The index constraint can be modified as `unique`, which represents a unique index on the attribute(s)

To create an index on an attribute or set of attributes in the EDL model editor, two steps are involved:

- 1. The index has to be declared for the respective attribute(s).**
- 2. The index has to be generated.**

Declaring an Index

To declare an index on an attribute, use the corresponding ORM object constraints `primary key`, `semantic key`, `index`, or the ORM relation constraint `foreign key`.

```
oca orm class StockPO implements Stock
{
    primary key(productID, warehouseID);
    index(productID);
    index(warehouseID);
    relation warehousePO : WarehousePO[1..1] inverse stockItemPOs implements warehouse
    {
        foreign key(warehouseID) -> (UUID)
    }
}
```

Generating an Index

If you did not explicitly declare indexes for attributes in your model, Intershop Studio generates an index for all `FindBy` and `ForeignKey` attributes automatically and generates a warning message alerting you to the fact. If you declare indexes for attributes, this information must be made known to the database which involves the following basic procedures:

■ Invoke Intershop Studio to create the index file `dbindex.ddl`.

This file contains the SQL statements necessary to create the desired indexes in the database. The information contained in `dbindex.ddl` is used during the database initialization process.

Right-click the package (or the parent EDL folder) in the Cartridge or Package Explorer View. From the context menu, select JGen | Generate | Database Indexes to generate the dbindex.ddl file.

Intershop Studio stores the dbindex.ddl file in the dbinit/scripts directory of your cartridge, e.g. <IS_SOURCE>/<cartridge_id>/javasource/resources/<cartridge_id>/dbinit/scripts.

■ **Configure the dbinit.properties file to include the dbindex.ddl file when creating indexes during the dbinit process.**

The dbinit.properties file configures database preparer classes to execute during the dbinit process, and for each preparer class additional files that may be required as arguments. The dbinit.properties file is located in <IS_SOURCE>/staticfiles/cartridges.

Modify it to contain the following configuration:

```
Class1 = com.intershop.beehive.core.dbinit.preparer.database.DatabaseIndexesPreparer...
resources/<cartridge_id>/dbinit/scripts/dbindex.ddl
```

The first entry invokes the preparer class (DatabaseIndexesPreparer) necessary to create indexes in the database. The second entry provides the directory path to the dbindex.ddl file.

Note that the index for the keyword Class (1 in the example above) may vary, depending when the preparer should be executed relative to other preparers configured in dbinit.properties.

Configure the Naming Manager

The naming manager (com.intershop.beehive.core.capi.naming.NamingMgr) is a singleton that is able to handle manager and provider classes. In addition, the naming manager provides a convenience method to look up PO factories from the ORM engine.

Sample Configuration

The naming manager is configured using property files. There are separate property files for each type of object (managers.properties, services.properties, providers.properties). Each cartridge may provide such property files. The property files of all cartridges combined are loaded during server startup and used to initialize the manager, provider and service instances. The objects can then be looked up from the naming manager.

Each file contains mappings from symbolic names (the name used to look up the object from the naming manager) to the implementation class.

Cartridges can override the registrations of other cartridges by simply mapping the same symbolic name to a different implementation class. The order in which cartridges are loaded then decides which implementation class will actually be used. The order in which cartridges are loaded depends on the order of cartridges in the <IS_HOME>/share/system/cartridges/cartridgelist.properties file. The cartridge loaded last wins.

The property files must be located in each cartridge in the directory <IS_SOURCE>/<cartridge_id>/javasource/resources/<cartridge_id>/naming. The property

files will be packaged into the *.jar files of the respective cartridge during the build process.

The syntax of the property files follows the simple schema below:

```
symbolic_name=implementation_class_name
```

An example for configuring a provider class in the cartridge bc_marketing is

```
ProductLineItemUpdateProvider=com.intershop.component.requisition. ...
    capi.orderprocess.SimplifiedProductLineItemUpdateProviderImpl
```

With respect to manager classes, it is recommended to use the fully qualified name of the manager as symbolic name. An example (from bc_requisition) is shown below:

```
com.intershop.component.requisition.capi.RequisitionMgr=com. ...
    intershop.component.requisition.internal.RequisitionMgrImpl
```

When generating manager classes, Intershop Studio automatically inserts a constant (REGISTRY_NAME), with the fully qualified manager name as value, such as:

```
RequisitionMgr
{
    String REGISTRY_NAME = "com.intershop.component.requisition.capi.RequisitionMgr";
}
```

This constant is typically used to obtain the configured implementation class when working with a manager.

Fallback Mechanism

A special fallback mechanism is available to configure multiple implementation classes for providers. Multiple implementation classes are distinguished using a distinct suffix to qualify the symbolic name, as shown below:

```
ProductPriceProvider_Fallback1=com.intershop...ProductListPriceProviderImpl
ProductPriceProvider_Fallback2=com.intershop...ProductDiscountPriceProviderImpl
ProductPriceProvider_Fallback3=com.intershop...ProductScaledPriceProviderImpl
```

To work with a specific provider implementation, (for example, ProductListPriceProviderImpl), use the symbolic name, qualified by the appropriate suffix:

```
NamingMgr.getInstance().lookupProvider("ProductPriceProvider_Fallback1")
```

To obtain all available providers, use the symbolic name without the suffix:

```
NamingMgr.getInstance().lookupProviders("ProductPriceProvider")
```

Access Manager Classes and Business Interfaces

Using managers to access business objects ensures that you always work with the appropriate business interfaces which the business object exposes. The sample below illustrates how coupon objects are accessed by coupon code from within a pipelet, using the CouponMgr. The couponMgr is used in the pipelet's execute method to create a ResettableIterator storing the retrieved coupon instances.

```
CouponMgr couponMgr = (CouponMgr)
```

```
NamingMgr.getInstance().lookupManager(CouponMgr.REGISTRY_NAME);
ResettableIterator iter = Iterators.createResettableIterator(couponMgr...
getCouponInstances(couponCode.trim()));
```

Note that the requested manager is identified through a constant (`REGISTRY_NAME`) which is defined in each manager's business interface. The value of the constant serves as a key which is used to obtain the manager implementation class from the naming manager, as configured in the `managers.properties` file (see *Configure the Naming Manager*).

An more recent way to access a manager through the naming manager uses the following code:

```
CouponMgr couponMgr = (CouponMgr) NamingMgr.getManager(CouponMgr.class);
```

Access Factories and POs

If you need to get a handle on the POs directly and cannot work with the business interfaces, you can retrieve the POs via the PO's factory class. This is necessary, for example, when writing a manager implementation class.

Factory classes can be obtained in two different ways: using the naming manager, or using the ORM engine. The following snippet illustrates how to obtain the factory class for the PO `CouponInstancePO` via the naming manager:

```
private CouponInstancePOFactory getInstanceFactory() throws SystemException
{
    if (instanceFactory == null)
    {
        instanceFactory = (CouponInstancePOFactory)
            NamingMgr.getInstance().lookupFactory(CouponInstancePO.class);
    }
    return instanceFactory;
}
```

With the factory obtained, POs can be located, using the available finder methods. A simple example is shown below:

```
public Iterator get_couponInstances(String couponCode)
{
    return getInstanceFactory().getObjectsByCouponCode(couponCode).iterator();
}
```

Alternatively, the ORM Engine can be used to obtain the factory class. To get the ORM Engine:

```
ORMMgr mgr = (ORMMgr) NamingMgr.getInstance().lookupManager(ORMMgr.REGISTRY_NAME);
ORMEngine engine = mgr.getORMEngine();
```

Then obtain the factory class using the engine's `getFactory()` method:

```
instanceFactory = (CouponInstancePOFactory) ...
engine.getFactory(CouponInstancePO.class);
```

Use Finder Methods

Finder methods come in two types: as `getObjects`-method or as `findBy`-methods. The `getObjects`-methods are recommended to use whenever possible. When retrieving more than a single object, the return value of these methods is a

`java.util.Collection` object. The `findBy`-methods are deprecated. Typical finder methods made available by factory classes include:

- `getObjectByPrimaryKey()`

This method returns null if the object was not found.

`javax.ejb.FinderException` was replaced by a runtime exception (`com.intershop.beehive.core.capi.common.FinderException`). Example:

```
public DomainPO getDomain(String uuid)
{
    return domainFactory.getObjectByPrimaryKey(new DomainPOKey(uuid));
}
```

- `getObjectsByAttribute()`

This is an additional possibility offered by the Business Object Modeler: Attributes inserted into the PO model can be marked to be used for additional finder methods inside the factory class. For example, suppose you define an attribute `couponCode` to assign code numbers to coupons. Setting the property `GenerateQueryByValue` to true on the IS-ORM profile of the attribute will cause Intershop Studio to generate a method that enables you to locate POs depending on the coupon code.

- `Collection getAllObjects() Collection getObjectsBySQLWhere(String condition)`

```
public Iterator getDomains(String uuid)
{
    return domainFactory.getObjectsBySQLWhere("uuid=?", new Object{uuid}).iterator();
}
```

- `getObjectsByDomainID()`

This method returns all POs belonging to a certain domain. Intershop Studio automatically creates this method for all objects derived from `ExtensibleObject`.

- `getObjectsBySQLWhere()`

This method can be used to locate POs by submitting SQL statements.

- `getObjectsBySQLJoin()`

The methods allow to specify a FROM clause and a WHERE clause. The source table always has the alias "this", for example `select ... from domain_av this, <from> where <where>`. Example:

```
public Iterator getDomainAVs(String uuid)
{
    return domainAVFactory.getObjectsBySQLJoin(
        "domain d",
        "this.ownerid=d.uuid and this.ownerid=?",
        new Object{uuid}).iterator();
}
```

Use Bind Variables

ORM factories support bind variables:

```
Collection getObjectsBySQLWhere(String where, Object args);
Collection getObjectsBySQLJoin(String from, String where, Object args);
```

The placeholder for a variable in the SQL condition is "?".

NOTE: Do not quote the placeholder. For example, do not use "name=?!", but "name=?"

Bind variables are applicable to all attribute types, e.g. `String`, `Integer`, and `Date`.

Please note also the following:

- All WHERE and JOIN queries should use bind variables!
- Improved performance, due to less SQL parsing in the database)
- Improved security, due to encoding of parameters
- Leave the correct handling of Date values to the ORM engine.

Java Source Code Conventions

Since many programmers work on the same source files it is a good practice to define and use a common standard of formatting rules and other conventions when writing programming code.

Sticking to these conventions helps to keep a consistent code style that is easier to read and maintain by everyone involved in the development process.

This section describes these conventions as recommended by Intershop for writing Java source code including JavaDoc.

Source Code Formatting

Editor Settings

Change your editor settings as follows:

■ line length

Set the line length to a maximum of 120 characters in order to avoid the need for horizontal scrolling.

■ for indentation use spaces instead of tabs

As a general rule, use spaces instead of tabs when indenting lines. Each indentation step should be four spaces.

In case you edit old code that is still indented using tabs, replace these tabs with spaces.

Spaces are preferred over tabs to make the source code look identical across different editors and editor configurations.

■ encode files using UTF-8

All Java source code files should be encoded in UTF-8.

■ disable automatic code beautifiers

If your preferred editor includes functionality to automatically format code (code beautifiers), disable this feature. Especially, do not let your editor modify the formatting of existing code, otherwise the merging of code or the checking for code differences may become impossible.

For example, when using Intershop Studio, disable "save actions".

When organizing imports the usage of code beautifiers is allowed.

Language

Use English for all names of classes, interfaces, methods and variables. Write all comments in English.

Use only ASCII characters in Java source files. Especially, do not use special characters, such as German Umlauts (ä, ö, ü).

Line Breaks

Code blocks (everything between curly brackets) must begin and end on a new line.

Correct:

```
if (condition)
{
    doSomeStuff();
}
```

Wrong:

```
if (condition) {doSomeStuff();}
```

Nesting Levels

To avoid deep nesting, use early returns, continue and break:

Correct:

```
public void method(Object a)
{
    if (a==null) return;
    ...
}
```

Wrong:

```
public void method(Object a)
{
    if (a!=null)
    {
        ...
    }
}
```

Method Declarations

Please observe the following rules when declaring Java methods:

- Always specify the method visibility (public, protected or private).
- In case of multiple operators, specify the order by using brackets.
- Avoid using the `this.` keyword when possible.
- Instance and class variables must be written at the beginning of a class, they are followed by the class initializers, constructors, public methods and private methods of the class (in this order).
- Order parameters hierarchically. Begin with objects to be found or updated, followed by additional attributes.

Correct:

```
UserGroup getUserGroup(Domain domain, String id);
```

```
void setUserGroupName(UserGroup userGroup, String name);
```

Import Statements

Import statements: Add each class you want to import in a single statement. Do not import a whole package. Exceptions are JDK classes.

Correct:

```
import foo.bar.a;
import foo.bar.c;
```

Wrong:

```
import foo.bar.*;
```

If-Statements

The if and else branch should be always put into curly brackets e.g.

Correct:

```
if (condition)
{
    do_some_stuff
}
else
{
    do_some_more_stuff
}
```

Wrong:

```
if (condition) doSomeStuff();
```

Switch Statements

Switch statements should look as follows:

```
switch(condition)
{
    case ABC:
        statements;

    case DEF:
        statements;
        break;

    case XYZ:
    {
        int x=1;
        statements;
    }
    break;

    default:
        statements;
        break;
}
```

If the cases perform only trivial actions, line breaks can be omitted:

```
switch(x)
{
```

```

        case A:      return "A";
        case B:      return "B";
        case C:      return "C";
        case default: return "N/A";
    }

```

Blank Spaces

A keyword followed by a parenthesis should not be separated by a space:

Correct:

```

while (i > 0)
{
}

for (int i = 0; i < 100; i++)
{
}

if (a)
{
}
else if (b)
{
}

```

Please mind the following notes:

- If multiple lines contain similar statements, align spaces so that equal parts are in the same column.
- No space between a method name and the parenthesis "(" starting its parameter list.
- A blank space should appear after commas in argument lists.
- All binary operators except "." should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++") and decrement ("--) from their operands.

Correct:

```

a += c + d;
a = (a + b) / (c * d);

while(n < 5)
{
    n++;
}
printSize("size is " + foo + "\n");

```

- Casts should not be followed by a blank space.

Correct:

```

myMethod((byte)aNum, (Object)x);
myMethod((int)(cp + 5), ((int)(i + 3)) + 1);

```

Wrapping Lines

If an expression does not fit on a single line, break it according to the following rules:

- Break after a comma.

- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.

If these rules lead to confusing code or to code that is squashed up against the right margin, just indent 8 spaces instead.

Here are some examples for breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                             longExpression3));
```

Below are two examples for breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // AVOID
```

The following examples illustrate indenting method declarations. The first example is the conventional case. The second example would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
// Conventional indentation
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother)
{
}

// Indent 8 spaces to avoid very deep indents
private static synchronized workingLongMethodName(int anArg,
                                                 Object anotherArg, String yetAnotherArg,
                                                 Object andStillAnother)
throws Exception
{}
```

The following example shows how to break an if statements:

```
if (    (condition1 && condition2)
      || (condition3 && condition4)
      || !(condition5 && condition6))
{
    doSomethingAboutIt();
}
```

These are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                               : gamma;

alpha = (aLongBooleanExpression)
```

```
? beta
: gamma;
```

Source Code Naming

General Rules

As a general rule:

- use full American-English descriptors
- use mixed case names to improve readability
- avoid long names (not longer than 30 characters)
- do not use the underscore character ('_') in names, except for constants

Class Names

All class and interface names start with an uppercase letter, `FirstLetterOfAllWordsCapitalized`, as in:

```
public class Customer
{
}

public class SavingsAccount
{
}
```

Use the following abbreviations as suffixes for class names:

Table 18. Class names

Suffix	Description
*Impl	Implementation for a default implementation of an interface.
*Mgr	Manager
*Ctnr	Container
*Ctrlr	Controller
*Ctx	Context

Method Names

Method names start with an active verb describing some action and a lower-case first letter (`firstLetterInLowerCase`):

```
openFile();
createAccount();
```

When possible, use fully qualified names. If they are too long, use an abbreviation which consists of the first letters in upper case.

```
// complete word
getName();

// multiple complete words, each with upper case first letter
getLastName();
```

```
// abbreviation for "stock keeping unit", so only use the first
// letters in upper case
getSKU();

// abbreviation for "identifier", so only use the first letters
// in upper case
getID();

// short for "radio detection and ranging", but it's not an
// abbreviation, it's an acronym that is pronounced as a word,
// therefore treat it like a complete word
getRadar();
```

Please mind the following rules for naming customary type of methods:

■ **Getter Methods**

Getter methods, which are used to retrieve a value from an object, always start with the prefix "get":

```
getFirstName()
getWarpSpeed()
```

■ **Getter Methods returning a Boolean**

The preferred prefix for getter methods with a boolean return value is "is". Alternatively, the prefixes "can" or "has" can be used:

```
isString()
isPersistent()
=canPrint()
hasDepends()
```

■ **Setter Methods**

Setter methods, which are used to store a value in an object, should always start with the prefix "set":

```
setFirstName()
setWarpSpeed()
setPriceForCone()
```

Note, that there doesn't have to be a set-method for a read-only attribute.

■ **Collection Methods**

Methods that affect the adding or removal of something to a collection or a "to-many"-relation have the prefix "add" or "remove". Removing the object does not necessarily destroy it (e.g. delete it), but only removes it from the collection.

```
addUser(User user);
removeUser(User user);
```

■ **Lifecycle Methods**

Methods that manage the lifecycle of an object have the prefix "create" or "delete".

```
createProduct();
deleteProduct();
```

Package Names

Package names are formed from reverse Internet domain names:

```
com.intershop.*
```

The following package names are used for Intershop platform and solution development:

- `com.intershop.beehive.*`
Is used for all Intershop Intershop 7 platform components.
- `com.intershop.component.*`
Is used for all Intershop business components.
- `com.intershop.<solution name>.*`
Is used for all Intershop business solutions.
- `com.intershop.tools.*`
Is used for all Intershop tools.

Loop Counters

The local variable names `i`, `j`, `k`, `l` and so on should be used as loop counters.

Arguments/Parameters, Attributes/Properties, and Local Variables

The first letter should be in lower case. Common name suffixes include "ID" and "UUID":

```
customer
value
domainID
firstName
warpSpeed
customerAddress
numberOfCustomers
```

Local variables should not be named similar to Instance variables in the same scope.

Constant Names

Constants should be given a name in uppercase and with underscores to separate words. A constant must also be declared as a "static final" class variable. Examples are:

```
MINIMUM_PAYMENT
STATE_TAX
```

If there is a group of constants belonging together, keep the changing part at the end.

```
AUCTION_STATE_ONLINE
AUCTION_STATE_OFFLINE
```

Source Code Documentation

Non-JavaDoc Comments

Use single line comments to document the programming logic in the code, to structure code sections, or to document local variables.

Intershop Studio can automatically generate task entries that are clickable by simply marking a code line with a "TODO" tag.

Correct:

```
// comments  
// TODO comment here what to do
```

Use C-style comments for temporary changes or for comments that should not appear in the JavaDoc:

```
/* not for Java documentation purpose */
```

JavaDoc Comments

When documenting code artefacts such as classes and interfaces, methods, instances and class variables, write JavaDoc comments whenever possible and applicable:

Correct:

```
/**  
 * here follows documentation which will be processed by javadoc  
 */
```

JavaDoc supports comment tags to mark certain types of information. You should enrich your comments using these tags whenever possible:

- **@param**

Use in documentation comments for *method* and *constructor* declarations. They should consist of the name of the parameter followed by a short description.

- **@return**

Use in documentation comments when declaring methods whose result type is not void. @return paragraphs usually consist of a short description of the returned value.

- **@exception**

Use in documentation comments for *method* and *constructor* declarations. The description should consist of the name of an exception class (which may be a simple name or a qualified name) followed by a short description of the circumstances that cause the exception to be thrown. The documentation of the `java.rmi.RemoteException` can be omitted.

- **@see**

Use in any documentation comment to indicate a cross-reference to a class, interface, method, constructor, field, or URL.

- **@deprecated**

Use to document how to avoid using deprecated classes and methods and explain why an artifact has been deprecated.

Within JavaDoc documentation sections of classes, methods, and attributes, use JavaDoc tags in the following order:

- 1. @param**
- 2. @return**
- 3. @exception**
- 4. @see**
- 5. @deprecated**

The following are examples for writing "@see" references:

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
@see package
```

JavaDoc comments may contain HTML tags. Useful tags include <p>,
, <pre>, <code>, , = and .

When documenting your code you should always write complete sentences including descriptions of parameters, return values, and exceptions. For constant values of a discrete value set, document the actual value:

Correct:

```
/**
 * Type code for a RequisitionAWFDefinition for department approval.
 * The value is '<code>1</code>'.
 */
public static int DEFINITIONTYPE_DEPARTMENT = 1;
```

JavaDoc Examples

The following list contains hints and examples for documenting common code artifacts using JavaDoc:

■ Classes and Interfaces

Describe the purpose of the class, guaranteed invariants, usage instructions, and/or usage of examples.

```
/**
 * First Sentence gives a short description about the API.
 * ...here follows detailed documentation...
 *
 * @see BaseClassName
 */
```

```
class ClassName extends BaseClassName
{
}
```

■ Methods

Describe the nature, the purpose, any preconditions and any effects of the method. Give notes to the algorithms, provide usage instructions, and reminders. Also describe all scenarios where the method is related to other methods in the class so that a developer clearly knows how to access and use the functionality provided by the method.

```
/**
 * Description of the function of the method.
 *
 * @param    paramName    description
 * @return   description
 * @exception nameOfException circumstances which cause the exception
 * @see     referenceToOtherStuff
 */
public returnType doStuff(parameter)
    throws XYZException
{}
```

■ Class Variables

Describe the nature, the purpose, any constraints and the usage of instances and static variables.

```
/**
 * The current number of elements.
 * Must be non-negative, and less than or equal to capacity.
 */
protected int count;
```

Java Editor

The Java editor is used to write pipelets and other Java classes. This editor contains powerful features that help you to efficiently accomplish your development tasks, including:

■ Syntax Highlighting

The editor supports syntax highlighting where keywords, comments, strings, and other information are displayed in different colors. This makes it easier for you to read and understand your code. You can customize the colors assigned to the various syntax elements in the Workbench Preferences dialog.

■ Automatic Code Validation

When you modify Java classes, the Java editor validates the code automatically and identifies potential problems. Erroneous code is marked in the file and a warning or error symbol is placed on the left margin of the editor view. The code validation feature is especially useful because it ensures that your pipelet code is consistent with the pipelet descriptor. Any discrepancies become visible immediately. See *Automatic Code Validation* for more information.

■ Content Assist

The editor offers hints and auto completion while you type. See *Content Assist* for more information. This feature can be enabled/disabled and customized through the Workbench Preferences dialog under Java | Editor.

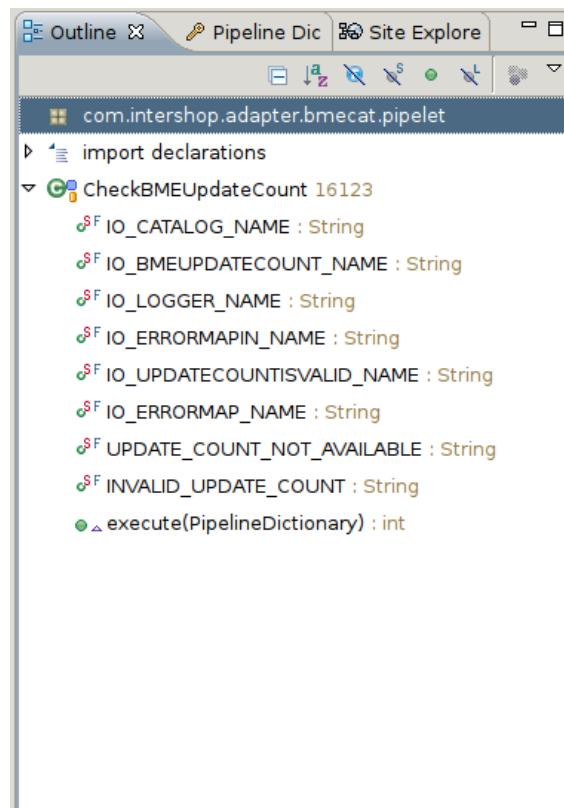
■ Pipelet Assist

The Pipelet Assist automatically performs certain pipelet development tasks and helps you to keep the pipelet configuration consistent with your code. For example, you can use the Pipelet Assist to add access statements for pipelet properties to your code. If the accessed property does not exist, the Pipelet Assist automatically adds it to the pipelet configuration. Using the Pipelet Assist, you can also copy properties from other pipelets, or define new ones. See *Pipelet Assist* for more information.

Outline View for Java Editor

In the context of the Java editor, the Outline View provides a structural view on the Java class (such as a Pipelet) that is currently active in the editor, including a view on its methods and variables and making available standard toolbar options.

Figure 61. Outline View for Java editor



Automatic Code Validation

Intershop Studio continuously validates your code while you write in the editor. Incorrect code is underlined and a warning or error symbol placed on the left margin of the editor view.

An appropriate task is also added to the list in the Problems View.

Intershop Studio can perform the automatic check only for a resource itself and its children. If the modifications in a resource result in errors in one of its superior

resources, the error cannot be detected. To solve this problem, you need to run the check manually:

- 1. In the Cartridge Explorer, right-click the resource you want to validate.**

The context menu is displayed.

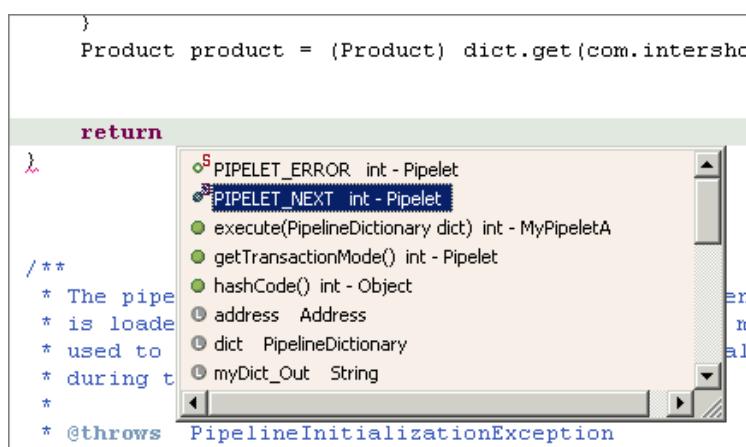
- 2. From the context menu, select Check.**

Intershop Studio performs a check on the selected element and all its child-elements. Any found problems are marked with appropriate symbols.

Content Assist

You can use content assist (also called code assist) when writing Java code or JavaDoc comments. To activate the content assist, press **Ctrl + Space**. The content assist displays a floating window with a list of possible completions for the code at the current cursor position. Select a suggested element to insert in the file.

Figure 62. Content Assist



For additional information on using the content assist, see the section [Getting Started | Basic Tutorial | Editing Java Elements | Using Content Assist](#) in the Java Development User Guide of the Eclipse online help.

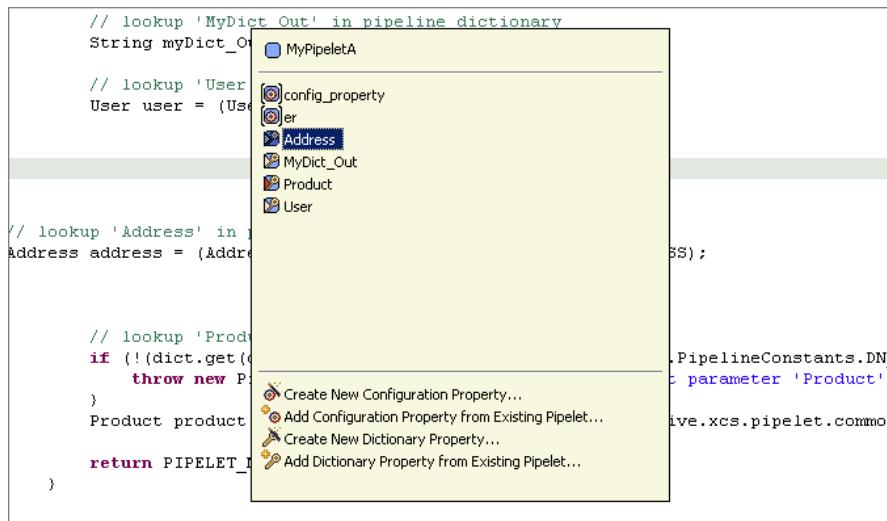
Pipelet Assist

The Pipelet Assist allows you to add access statements for pipelet properties to your code, or to define new pipelet properties.

To open the Pipelet Assist and insert an access statement:

- 1. Place the cursor at the position where you want to insert the access statement.**
- 2. Press **Ctrl-Shift + D** to open the pipelet assist. Alternatively, select **Edit | Pipelet | Insert Access Statement** from the Workbench menu.**

The pipelet assist displays a floating window at the cursor position.

Figure 63. Pipelet Assist

3. Select a property from the list and press Enter.

The pipelet assist inserts the new code at the cursor position and updates the pipelet descriptor files.

You can also use the Pipelet Assist to create new properties before inserting the access statement.

4. Press **Ctrl-Shift + D** to open the pipelet assist.

5. Click one of the commands on the bottom of the pipelet assist window.

See the following table for details:

Table 19. Defining Pipelet Properties

Command	Description
create new configuration property	Opens the New Pipelet Property dialog where you can define the property.
add configuration property from existing pipelet	If a property has been already defined for another pipelet, you can just copy it. The command opens the Pipelet Property dialog where you can browse for already defined properties.

When you have created the new property, it is added to the list of available properties in the Pipelet Assist. If you copied a dictionary property from an existing pipelet, the property inherits the access type and description from the original. The current access type is indicated by triangles that appear either on the left (in), the right (out), or both edges (bidirectional) of the icon assigned to the property. The color of the triangle indicates whether the property is optional (yellow) or required/guaranteed (red). Both attributes can still be changed.

Highlight the property in the list, then:

- Press Space to cycle through the possible access types (in, out, and bidirectional). The position of the triangle changes accordingly.
- Press Shift + Space to define whether a property is optional or required/guaranteed. The color of the triangle changes accordingly.

EDL Model Editor

User Interface

The EDL model editor provides a convenient user interface for viewing, creating, and maintaining object models.

The EDL model editor comprises:

■ Model Editor

A source code-based model editor is available to display and maintain class diagrams.

■ Outline View

The Outline View provides access to elements of an EDL model.

■ Properties View

The Properties View is used to set properties of model elements.

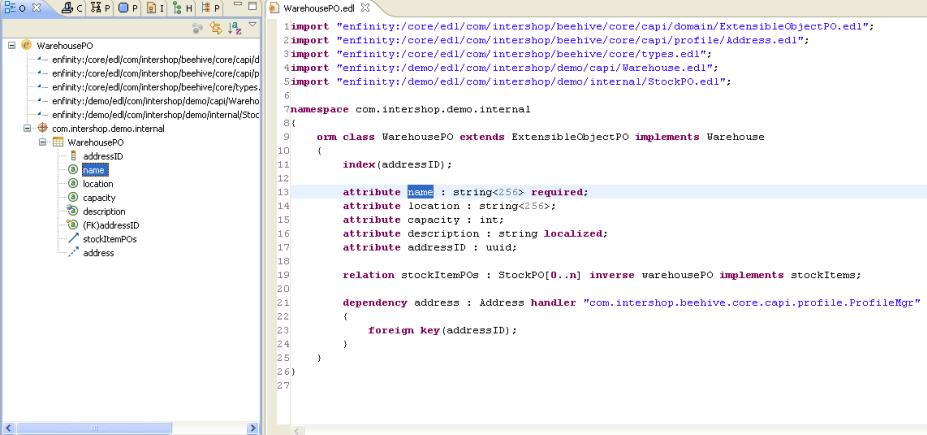
■ Code Generator

Using the code generator, you can automatically generate the complete set of classes from individual model elements, packages, or the entire model.

NOTE: The actually available functionality depends on the used model editor.

All descriptions below refer to the EDL Source Editor.

Figure 64. EDL Model Editor



The screenshot shows the EDL Model Editor interface. On the left is the Outline View, which displays a tree structure of EDL model elements under the package 'WarehousePO'. Elements shown include 'WarehousePO' (with attributes 'name', 'location', 'capacity', 'description', 'addressID', 'stockItemPOs', and 'address'), and 'com.intershop.demo.internal' (with a nested 'WarehousePO' element). On the right is the EDL Source Editor, showing the EDL code for the 'WarehousePO' class. The code includes imports for various EDL packages, a namespace declaration for 'com.intershop.demo.internal', and the class definition itself with its attributes and relationships.

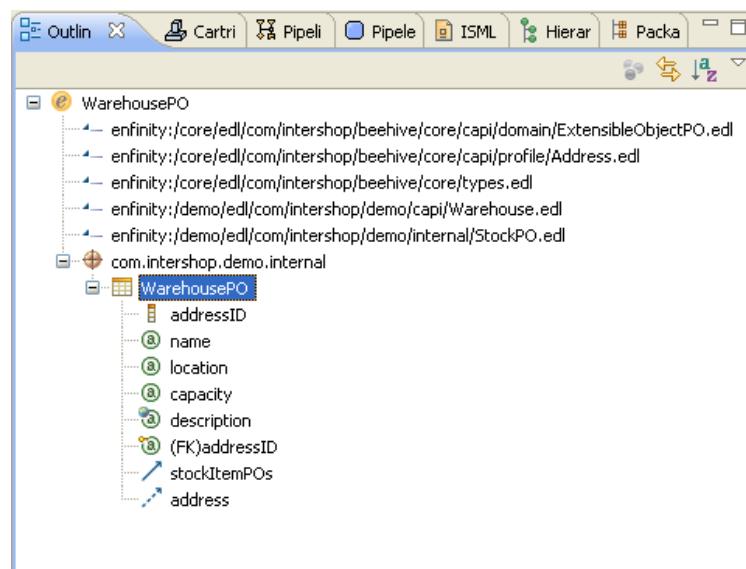
```

1 import "enfinity:/core/edl/com/intershop/beehive/core/capi/domain/ExtensibleObjectPO.edl";
2 import "enfinity:/core/edl/com/intershop/beehive/core/capi/profile/Address.edl";
3 import "enfinity:/core/edl/com/intershop/beehive/core/types.edl";
4 import "enfinity:/demo/edl/com/intershop/demo/capi/Warehouse.edl";
5 import "enfinity:/demo/edl/com/intershop/demo/internal/StockPO.edl";
6
7 namespace com.intershop.demo.internal
8 {
9     class WarehousePO extends ExtensibleObjectPO implements Warehouse
10    {
11        index(addressID);
12
13        attribute name : string<256> required;
14        attribute location : string<256>;
15        attribute capacity : int;
16        attribute description : string localized;
17        attribute addressID : uid;
18
19        relation stockItemPOs : StockPO[0..n] inverse warehousePO implements stockItems;
20
21        dependency address : Address handler "com.intershop.beehive.core.capi.profile.ProfileMgr"
22        {
23            foreign key(addressID);
24        }
25
26    }
27

```

Outline View for EDL Model Editor

In the context of the EDL Model Editor, the Outline View provides access to elements and resources contained in the model.

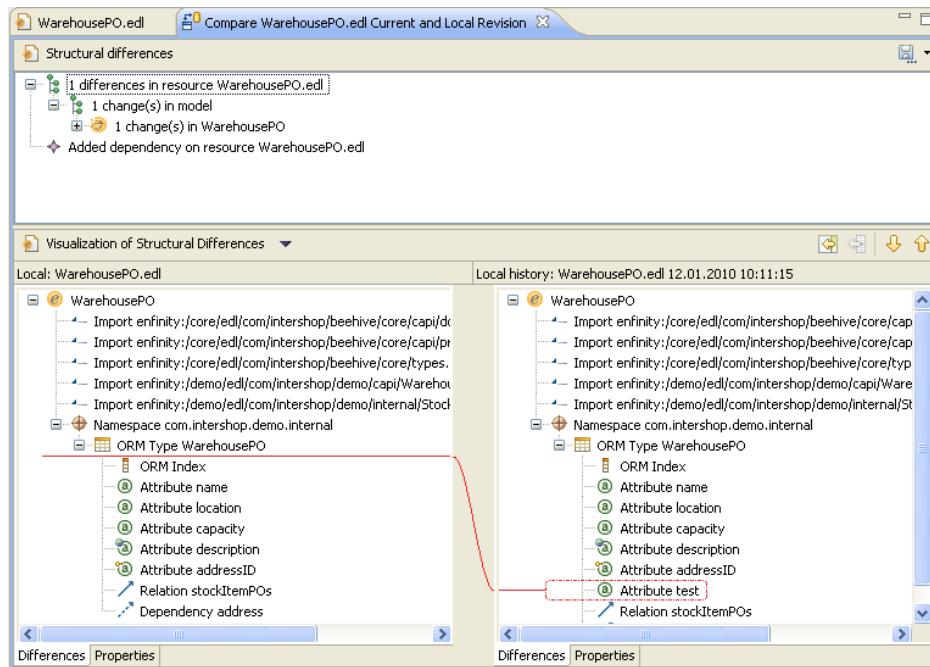
Figure 65. Outline View for EDL Source Editor

Content Assist

You can use content assist at various places when creating and modifying object models. To activate the content assist, press **Ctrl + Space**. The content assist displays a floating window with a list of possible options (such as classes or types) available at the current point.

EDL Model Comparer

The EDL model comparer can be used for two purposes. You can either compare different revisions of the same EDL model, or compare two independent models.

Figure 66. Comparing EDL models

The comparison window comprises two panes. The upper pane lists the elements that were added or removed. The lower pane visualizes the structural differences between the two compared models (or model versions), where the newer model (or model version) is displayed on the right.

The following symbols are used to visualize structural model differences:

Table 20. EDL model comparer symbols

Symbol	Description
	The "Plus" symbol indicates elements that have been added to the model.
	The "Minus" symbol indicates elements that have been removed from the model.
	The green line indicates elements that are present in the newer model or model version and missing in the older model or model version.
	The red line indicates elements that are missing in the newer model or model version and present in the older model or model version.

Object Path Expressions

Object Path Expressions: Overview

Object path expressions provide a simple way to navigate through persistent objects and relations between them, and to resolve the values of their fields for a given "root" object and a path.

The Object Path Expression Language is a simple language that was initially designed to access persistent objects from within ISML templates without having to write Java code. The concept was later extended to make object path expressions available to other domains as well, such as pipelines, pipelets, and queries.

Object Path Syntax

Object Path Elements

An object path consists of elements separated by colons:

```
This:is:an:object:path
```

The elements of the object path can be simple identifiers, consisting of alphanumeric characters, or they can be special expressions like parameter elements, null values or literals.

Parameters

Object path expressions can contain elements with parameters. Such sub-expressions are used like arguments for Java method calls. Parameter values are resolved as object path expressions as well.

Parameters directly follow their parent element (which represents a "method call") and are marked by round brackets. Multiple parameters can be separated by a comma.

```
This:is:an:object:path(This:is:an:argument:path)
This:is:an:object:path("This is a constant")
This:is:an:object:path(This:is:an:object:path("This is a constant"))
A:B(C:D(E)):F(G, H:I(J))
```

An object path consists of at least one element, and can be of arbitrary length.

Literals

Literals are marked by double-quotes at the beginning and the end of the expression.

```
"This is a constant string value."
```

A literal can also be passed as a parameter:

```
This:is:an:object:path("This is a constant parameter")
```

If the literal represents a value other than a string, it may contain a class cast which is enclosed by round brackets within the quoted expression.

```
"(Integer) 123"
"(BigDecimal) 123.45"
"(java.text.DecimalFormat) #,###"
```

The type for the cast must be a Java class that can take a String as an argument in its constructor. If the type is not located in one of the default packages `java.lang`, `java.math`, `java.text` or `com.intershop.beehive.foundation.quantity`, the full class name (including package) must be specified.

If the cast type constructor takes more than one string argument, multiple arguments can be separated using square brackets. Each argument can be a literal type itself.

```
"(Money) [USD] [(BigDecimal)123.45]"
```

NOTE: Java primitives such as "(int) 123" are not supported. The cast type must be an object type.

The object path expression language supports literals for Java classes using the syntax "(Class)<any Java class name>". For example, the following expressions are allowed:

```
OrderBO:Extension  
  ("(Class)com.intershop.sellside.appbase.b2c.capi.order.OrderBOPaymentExtension")  
ServiceConfigurationRepository:Services  
  ("(Class)com.intershop.component.orderimpex.capi.export.order.OrderExportService")
```

Null Values

The expression "null" or "NULL" (case-insensitive) is a reserved word representing a null value.

Mapping of Object Paths to Java Objects

Lookup Methods

The individual elements of object paths are mapped to invocations on Java objects or to lookups from the pipeline or session dictionaries. Which mappings are actually available depends on the position of the object path element inside the object path (see *Lookup Strategies*).

When resolving an object path, the root object is determined first, and then each subsequent element of the object path is used to navigate from the current object to the next object, starting from the initial root object. The object resolved last is then the final result.

The following types of lookup operations for a single object path element are available:

■ Pipeline Dictionary Lookup

The object path element is used as key for the pipeline dictionary. This lookup is normally used to determine the root object.

■ Session Dictionary Lookup

The object path element is used as key for the session dictionary. This lookup is normally used to determine the root object.

■ Template Loop Stack Lookup

The object path element is used as key for the current iterator in the template loop stack. This lookup is only used within an ISML <isloop> construct.

■ Extensible Object Lookup

The object path element is used as name for an extensible object attribute. The attribute is looked up for the current locale for the current user. Alternatively, multiple attributes can be looked up from the extensible object (e.g. in a loop context).

■ Map Lookup

The object path element is used as key for a hashmap-structure. The object to which the lookup method is applied must have a "get" method that takes a string (or an object) as argument.

■ Java Reflection Lookup for "get" Methods

The object path element is used as suffix for a get-method, e.g., the full method name is "get" + element name. If the object path element has parameters, the parameter values are resolved and passed as arguments to the underlying "get" method.

■ Java Reflection Lookup for "is" Methods

The object path element is used as suffix for an is-method, e.g., the full method name is "is" + element name. Such methods typically return a boolean value, e.g., `isValid()`.

If the object path element has parameters, the parameter values are resolved and passed as arguments to the "is" method.

■ Java Reflection Lookup for "has" Methods

The object path element is used as suffix for an has-method, e.g., the full method name is "has" + element name. If the object path element has parameters, the parameter values are resolved and passed as arguments to the "has" method.

■ Java Reflection Lookup for "createIterator" Methods

The object path element is used as part of the name of a createIterator-method, e.g., the full method name is "create" + element name + "Iterator".

If the object path element has parameters, the parameter values are resolved and passed as arguments to the "createIterator" method.

■ Collection Lookup

Java-collection classes like `java.util.Map` and `java.util.Collection` do not follow the JavaBean standard. Therefore, their properties are not accessible using `ObjectPath`. For this, an extra lookup method has been implemented that supports some special mappings for Maps, Collections and Arrays. The following path names can be resolved and are mapped to their underlying Java implementation method:

Table 21. Collection Lookup

Map	Collection	Array
Values	Iterator	Size
EntrySet	Size	
KeySet		
Size		

Lookup Strategies

The available mappings of individual object path elements are combined to form so-called lookup strategies. A lookup strategy resolves a particular object path element or a complete object path by selecting and applying well-defined lookup methods. Lookup strategies distinguish two different dimensions: a fallback

lookup strategy and a lookup strategy depending on the position of an object path element inside the object path.

■ Fallback Lookup Strategy

The object path framework is based on so-called "fallbacks". In order to resolve an object path element, mappings are tried in the order determined by the fallback priority until a mapping was found that produced a valid result for the current path element.

This fallback mechanism is implemented by a fallback lookup strategy. The fallback strategy defines the applicable lookup methods and the order in which they must be tried.

■ Position Lookup Strategy

When resolving the values for elements of an object path, the position of the element within the path is important, because it has an impact on the lookup methods that are applicable for resolving the element.

The position lookup strategy distinguishes three positions in a path. For each area, a fallback strategy (consisting of multiple lookup methods) is configured.

- **the head: this is the first element of an object path**
- **the body: this is the middle part (consisting of 0..n elements)**
- **the tail: this is the last element of the object path**

For example, in the expression

`This:is:an:object:path`

"This" is the head, "is:an:object" is the body, and "path" is the tail.

The head position is particularly important to resolve the initial root object from where the rest of the path can be resolved. Therefore, the pipeline dictionary lookup and the session dictionary lookup methods are typical fallbacks for the head position.

In a loop context (e.g. ISML loops), the tail position is of special interest, as it must return a value that can be looped over (e.g. iterated). Therefore, it contains the "createlteator" lookup method as a possible fallback. Such a lookup method does not make sense in the body part of a path, which is why it is missing there.

Object Paths in ISML

Note that literals can only occur as parameters in object paths, such as in `Product:Price("EUR")`. They cannot serve as root expressions, such as "This is a constant".

ISML Expressions

This lookup configuration applies to all "simple" ISML expressions (except `<isloop>`), such as `<isprint>`, `<isset>`, etc.

Table 22. Fallbacks to resolve object path elements in ISML expressions

Fallback Priority	Head Position	Body Position	Tail Position
I	Literal Value (full path)	Reflection "get"	Reflection "get"

Fallback Priority	Head Position	Body Position	Tail Position
2	Template Loop Stack	Extensible Object single attribute	Extensible Object single attribute
3	Pipeline Dictionary Lookup	Reflection "is"	Reflection "is"
4	Session Dictionary Lookup (with prefix "T_")	Map lookup	Map lookup
5		Collection lookup	Collection lookup

NOTE: Content assist in Intershop Studio provides convenient access to object path expressions in templates. See here for details.

ISML Loops

This lookup configuration applies to ISML <ISLOOP> expressions only. Note that the loop node requires a result object that is iterable, such as a collection, an array or an iterator. Therefore somewhat different fallbacks apply.

Table 23. Fallbacks to resolve object path elements in ISML loops

Fallback Priority	Head Position	Body Position	Tail Position
1	Literal Value (full path)	Reflection "get"	Reflection "createIterator"
2	Template Loop Stack	Map lookup	Extensible Object multiple attributes
3	Pipeline Dictionary Lookup	Collection Lookup	Reflection "get"
4	Session Dictionary Lookup (with prefix "T_")		Map lookup
5			Collection Lookup

Object Paths in Pipelines

Pipelet Input Aliases

Object paths can be used to define aliases for input parameters of pipelets. Pipelets generally support three kinds of fallback for the whole object path:

1. **Check for the null value.**
2. **Resolve a literal value (if the expression is enclosed by double-quotes).**
3. **Resolve the object path element by element using the position-dependent fallbacks listed in the following table.**

Parameter values are resolved recursively, again starting with fallback priority 1.

Table 24. Fallbacks to resolve object path elements for pipelet input aliases

Fallback Priority	Head Position	Body Position	Tail Position
1	Pipeline Dictionary Lookup	Reflection "get"	Reflection "get"

Fallback Priority	Head Position	Body Position	Tail Position
2		Extensible Object single attribute	Extensible Object single attribute
3		Reflection "is"	Reflection "is"
4		Map lookup	Reflection "createlterminator"
5			Map lookup
6			Extensible Object multiple attributes

NOTE: Content assist in Intershop Studio provides convenient access to object path expressions in pipelines. See here for details.

Pipeline Decision Nodes

For pipeline decision nodes, the values to be compared can be obtained using object path expressions. Generally, the same fallbacks as for pipelet aliases apply, except that literals and null cannot be used directly as decision value.

Pipeline Loop Nodes

Pipeline loop nodes support three general fallbacks for the whole object path:

1. **Check for the null value.**
2. **Resolve literal value (if the expression starts and ends with double-quotes).**
3. **Resolve the object path element by element using the position-dependent fallbacks listed in the next table.**

Parameter values are resolved recursively, again starting with fallback priority 1.

Note that the loop node requires a result object that is iterable, such as a collection, an array or an iterator. Therefore somewhat different fallbacks apply.

Table 25. Fallbacks to resolve object path elements for pipeline loop nodes

Fallback Priority	Head Position	Body Position	Tail Position
1	Pipeline Dictionary Lookup	Reflection "get"	Reflection "createlterminator"
2		Map lookup	Extensible Object multiple attributes
3			Reflection "get"
4			Map lookup

Web Services

The return value of a web service can have an alias that works similar to a pipelet alias. Therefore, the same fallback strategies apply here. See *Pipelet Input Aliases* for details.

Object Paths in Intershop Studio

Object path expressions can be used in the Intershop Studio pipeline debugger to inspect the values of objects that are currently stored in the pipeline dictionary. For this, open the "Watch" view and enter an object path. The same syntax and capabilities applicable to pipelet aliases can be used here. See *Object Paths in Pipelines*.

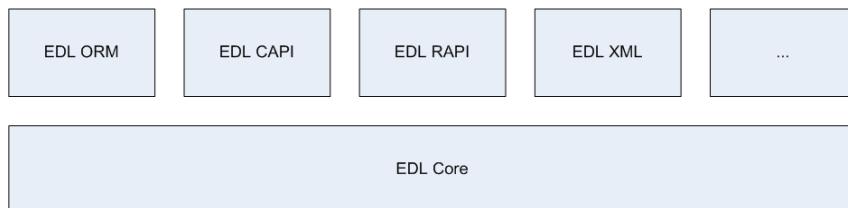
EDL Reference

The Enfinity Definition Language (EDL) as a textual domain-specific language (DSL) for modeling Java objects for Intershop 7.

EDL consists of a core language, which is extended by multiple sub-domain languages. The core language provides common concepts like namespaces, primitive types etc. that are shared by all sub-languages, while the sub-languages provide additional language features for expressing a certain Intershop 7 subsystem.

For example, the ORM sublanguage provides means to model Intershop 7 ORM classes with EDL. The sub-languages rely on the core type system and they can depend on each other.

Figure 67. EDL Language Structure



EDL Files

EDL models are stored in text files with the ending *.edl. They can be edited with a simple text editor or with a specialized EDL text editor in Intershop Studio that provides additional features such as syntax highlighting, code completion etc.

An EDL file consists of:

- imports of other EDL files that contain referenced types
- primitive type declarations
- external type declarations
- namespace declarations
- complex type declarations

Typically, each complex type (such as an ORM class) is defined in its own EDL file, which should have the same name as the contained type (similar to Java conventions). However, this is not a restriction of the language but a design recommendation. Basically, an EDL file can have any name and it would be possible to define all types of an Intershop 7 system in a single EDL file.

NOTE: EDL is a modeling language, not a programming language. Therefore, it only consists of declarations, but does not define how they are actually implemented.

EDL is intended to be used in combination with a code generator that can create the implementation code from the EDL declarations. Intershop Studio includes such a code generator.

Core Language

Comments

EDL defines 3 types of comments: single-line comments, multi-line comments and parser-agnostic comments.

```
// This is a single line comment. It ends with the line break.

/*
 * This is a multi-line comment.
 */

#
# This is a parser-agnostic comment. It can be contained in an EDL file,
# but will be ignored by the parser, so it will not be contained in the
# resulting EDL model and cannot be processed further.
#
```

Any language element can have an associated description, which is reflected by a multi-line comment in front of it.

Imports

All references to types that are defined in a different EDL file must be explicitly declared using an import statement. The import consists of a URI to the EDL file containing the wished type. The URI points to a file, not a model element, so it must include the full path to the file and the file ending.

```
import "enfinity:/core/edl/com/intershop/beehive/core/capi/domain/ ...
ExtensibleObject.edl";
```

Primitive Types

All primitive types that are used in an EDL model must be declared. There are no built-in primitive types.

```
primitive boolean;
primitive int;
primitive string;
```

Basically, any primitive type can be declared. However, there are restrictions which primitive types the code generator actually understands and is able to handle.

External Types

External types represent types that are used in EDL, but are not fully defined in EDL. They form an alias for an existing Java type, so that the type can be used in the model.

```
external Collection type "java.util.Collection";
```

Namespaces

A namespace can be compared with a Java package. It groups complex types into logical units. Namespaces can be nested. Alternatively, the name of the namespace can reflect the nesting by using a Java-like syntax. An EDL file can contain multiple namespaces.

```
namespace com
{
    namespace intershop
    {
    }
}

namespace com.intershop
{
    // this is equivalent to the namespace declaration above
}
```

Primitive and external types have no namespace, so they must be declared outside of the namespaces (e.g. in front of them). On the other hand, complex types such as ORM classes always must have a namespace, so they can only be declared within a namespace.

Complex Types

Namespaces contain the definitions of complex types. Currently, EDL supports ORM types, CAPI types, RAPI types and XML types.

All complex types in the EDL language follow roughly the same language structure; which features are available, depends on the specific type.

```
(<modifier>)* <sublanguage name> "class" | "interface" (<modifier> | ...
    <property>)*
"{"
    (<constraint> | <member>)*
"}"
```

Complex types typically have members that can be attributes, constants or relationships, depending on the sublanguage.

Example:

```
primitive uuid;
primitive string;

namespace com.intershop.example
{
    abstract orm class BusinessObject oca timestamp
    {
        primary key(UUID);
        unique index(name);
        attribute UUID : uuid;
        attribute name: string<256> type "CLOB" required;
    }
}
```

Attributes

An attribute is a member that has a name, a type, and some sublanguage-specific additional features. The common structure is:

```
(<modifier>)* "attribute" <name> ":" <type> ("=" <literal>)? ...
```

```
(<modifier> | <property>)* ";
```

The type of an attribute is usually a primitive type, or an external type.

Example

Please check the lines that declare attribute members.

```
primitive uuid;
primitive string;

namespace com.intershop.example
{
    abstract orm class BusinessObject oca timestamp
    {
        primary key(UUID);
        unique index(name);

        attribute UUID : uuid;
        attribute name: string<256> type "CLOB" required;
    }
}
```

Relationships

A relationship is a member whose type is another complex type. Depending on the sublanguage, different features of relationships are supported. The common structure is:

```
(<modifier>)* "relation" <name> ":" <type> <cardinality> ...
(<modifier> | <property>)* ("{" <foreign key> "}") | ";"
```

Example

```
// declaration on the one-side of a relation, e.g. WarehousePO
relation stockItemPOs : StockPO[0..n] inverse warehousePO implements stockItems;

// declaration on the many-side of a relation, e.g. StockPO
relation warehousePO : WarehousePO[1..1] inverse stockItemPOs implements warehouse
{
    foreign key(warehouseID) -> (UUID);
}
```

Modifiers

A modifier is a single keyword that specifies the behavior of a complex type, attribute or relationship. It can be placed in front of the element (e.g. "abstract"), or behind the element. Which modifiers are supported depends on the sublanguage.

Typical modifiers are:

- `abstract` for classes
- `readonly, required, localized` for attributes
- `pageable` for relationships

Properties

A property consists of a single keyword and a value, which is usually a literal, or sometimes a pre-defined keyword. It can be placed behind the element definition.

Properties and modifiers can be mixed. Which properties are supported depends on the sublanguage.

Typical properties are:

- extends SomeSuperClass
- implements SomeSuperInterface1, SomeSuperInterface2
- table "..."
- column "..."
- cached soft

Constraints

Some complex types or members (ORM relationships, dependencies) can have a body with additional constraints. A body is contained within curly brackets "{" and "}". The constraint provides additional definitions that refer on the members of the element.

Example:

```
primary key(...);
foreign key(...) -> (...);
unique index(...);
```

Literals

There are a number of literals that can be used in EDL, e.g. for expressing default values or other values. Currently, EDL supports float literals (a floating point number), integer literals, string literals ("..."), boolean literals (true or false) and the null literal (null).

Reserved Keywords

EDL comes with a number of reserved keywords that cannot be used as identifiers for classes, members and so on. If such an identifier collides with an existing keyword, the identifier must be escaped. Xtext supports the "^" character for escaping, i.e., the identifier must get "^" as prefix.

This is the complete list of reserved keywords:

```
abstract, attribute, binary, block, cached, cartridge, class, column,
constant, containment, default, delete, dependency, element, extends,
external, false, foreign, handler, implements, import, index,
interface, inverse, key, localized, mapped, namespace, none, null,
observable, oca, operation, orm, pageable, pattern, primary,
primitive, propagate, readonly, relation, remote, remove, replicated,
required, searchable, semantic, soap, soft, strong, table, tag, text,
throws, timestamp, true, type, unique, weak, xml
```

Example:

```
namespace com.intershop.^orm
{
    ...
}
```

ORM Models

ORM Classes

An ORM class describes a type for the ORM object-relational-mapping layer of Intershop 7. Its Java implementation can be generated by the Intershop 7 code generator. An ORM class declaration has the structure:

```
(<modifier>)* "orm" "class" <name> (<modifier> | <property>)*
{
    (<constraint> | <member>)*
}
```

An ORM class must be defined within a namespace.

Table 26. ORM Class Modifiers

Modifier	Description
abstract	Declares an ORM class to be abstract, e.g. to serve as superclass for other ORM classes.
oca	The code generator will create an additional OCA attribute, which is used for transaction control. This modifier can only be used on top-most superclasses.
timestamp	The code generator will create an additional timestamp attribute which contains the last modification time of the object. This modifier can only be used on top-most superclasses.

Table 27. ORM Class Properties

Property	Description
extends <superclass>	Defines the superclass from which the ORM class inherits. Must be another ORM class.
implements <interfaces>	A comma-separated list of interfaces, which the ORM class implements. The interfaces can be CAPI interfaces or external types.
table "tablename"	The name of the database table to which the ORM class is mapped. Only non-abstract ORM classes are mapped to a database table. The default value is the class name of the ORM class (without namespace names).
cached <referencetype>	The reference type that is used for caching instances of the ORM class in the ORM cache. The type can be one of strong , soft , weak or none . Default is soft .

Table 28. ORM Class Constraints

Constraint	Description
primary key(<orm attributes>);	Defines the primary key of the ORM class, consists of a comma-separated list of ORM attribute names. Due to restrictions in the ORM engine, a primary key can only be declared in the top-most superclasses.

Constraint	Description
<code>semantic key(<orm attributes>);</code>	Defines the semantic key (alternative key) of the ORM class, consists of a comma-separated list of ORM attribute names.
<code>index(<orm attributes>);</code>	Defines an index on ORM attributes, consists of a comma-separated list of ORM attribute names. The index constraint can have the unique modifier, which represents a unique index on the attributes.

ORM Attributes

An ORM attribute is a member of an ORM class. It has the structure:

```
(<modifier>)* "attribute" <name> ":" <type> ("<" <length> ">")? ...  
 (= <defaultvalue>)? (<modifier> | <property>)* ";"
```

The type of an ORM attribute can be either a primitive or an external type.

The optional length of the attribute defines the length of the underlying database column. It must be an integer. If not specified, the default length of the code generator applies.

The attribute can have a default value, which is given as a literal.

Table 29. ORM Attribute Modifiers

Modifier	Description
<code>localized</code>	The attribute is localized, e.g. it can have multiple values, one for each locale. Due to restrictions in the code generator, localized attributes are currently only supported for ORM classes that inherit from ExtensibleObjectPO.
<code>mapped</code>	The attribute is not stored in this object directly, but it is mapped to an extensible object attribute (as non-localized attribute). Due to restrictions in the code generator, mapped attributes are currently only supported for ORM classes that inherit from ExtensibleObjectPO.
<code>observable</code>	The code generator will create onChangeHooks for the attribute, which can be implemented to trigger additional actions when the attribute has been changed.
<code>readonly</code>	The attribute can only be read, there will be no setter method.
<code>replicated</code>	Holds the attribute twice, as a "normal" column and as an extensible object attribute. Due to restrictions in the code generator, localized attributes are currently only supported for ORM classes that inherit from ExtensibleObjectPO.
<code>required</code>	The attribute must have a non-null value. This will be reflected by a database NOT NULL constraint.

Modifier	Description
searchable	The ORM objects can be searched by this attribute, i.e., the generated ORM factory will get an additional <code>getObjectsByAttribute</code> method.

Table 30. ORM Attribute Properties

Property	Description
column "columnname"	The name of the database column to which this attribute is mapped.
type "columntype"	The type of the database column to which this attribute is mapped.
cached <referencetype>	The reference type that is used for caching instances of the ORM attribute in the ORM cache. The type can be one of <code>strong</code> , <code>soft</code> , <code>weak</code> or <code>none</code> . Default is <code>strong</code> .
handler "handlername"	The Java class name of an ORM attribute handler implementation that can read / write the attribute from / to JDBC.

ORM Relations

An ORM relation is a member of an ORM class. It has the following structure:

```
(<modifier>)* "relation" <name> ":" <type> "[" <cardinality> "]" ...  
<modifier> | <property>)* ("{" <foreign key> "}") | ";"
```

An ORM relation always points to an ORM class, i.e., the type is an ORM class. Its cardinality can be one of `0..1`, `1..1`, or `0..n`.

Table 31. ORM Relation Modifiers

Modifier	Description
readonly	The relation is readonly, i.e., there are no setter methods.
pageable	The relation is pageable, i.e., there are special getter methods that return an <code>PageableIterator</code> .

Table 32. ORM Relation Properties

Property	Description
inverse <orm relation>	Specifies the relationship in the target ORM class that represents the inverse relation of this relation. This can be used to implement bidirectional relations. The inverse relation must have this relation as inverse relation.
implements <capi relation>	If set, this ORM relation represents the implementation of the specified CAPI relation. Special adapter methods will be generated.
cached <referencetype>	The reference type that is used for caching instances of the ORM relation in the ORM cache. The type can be one of <code>strong</code> , <code>soft</code> , <code>weak</code> , or <code>none</code> . Default is <code>soft</code> .

Property	Description
<code>delete <deleteaction></code>	Specifies the delete action of the relation, if the parent object is removed. Can be one of <code>default</code> , <code>remove</code> , <code>block</code> , or <code>propagate</code> . <code>Remove</code> ; sets the foreign key to null. <code>Block</code> allows delete operations on a source object only if there is no associated object with a foreign key pointing to this instance. <code>Propagate</code> causes all objects containing a foreign key to be deleted as soon as the source object is deleted. <code>Default</code> activates Block, Remove, or Propagate depending on the multiplicity label of the role for which <code><deleteaction></code> is specified, and also depending on the multiplicity of the inverse role.

Table 33. ORM Relation Constraints

Constraint	Description
<code>foreign key (<orm source attributes>) -> (<orm target attributes>);</code>	The foreign key mapping of the relation. It specifies the mapping of one or multiple source attributes (which must be defined in this class) to one or more target attributes (which must be the primary key of the target class). For bidirectional relationships, the foreign key mapping must be provided on only one side of the relationship, which is the class that defines the foreign key attributes.

NOTE: Due to restrictions in the code generator, ORM relations must currently always be bidirectional.

ORM Dependencies

ORM dependencies represent so-called weak relations for ORM classes. They have the following structure:

```
"dependency" <name> ":" <type> (<properties>)*
{
    <foreign key constraint>
}
```

The type of an ORM dependency can either be an ORM class, or a CAPI interface. Currently, there are no modifiers. The cardinality is always `0..1`.

Table 34. ORM Dependency Properties

Property	Description
<code>handler <handlerclass></code>	The Java class name of a dependency handler that can resolve the referenced objects by the foreign key attribute.

Table 35. ORM Dependency Constraints

Constraint	Description
<code>foreign key (<orm source attribute>);</code>	The foreign key mapping of the dependency. Currently, a dependency always maps to the UUID of the target type, therefore this attribute

Constraint	Description
	must always be a single local attribute of type uuid.

ORM Example

```

namespace com.intershop.demo.internal
{
    orm class WarehousePO extends ExtensibleObjectPO implements Warehouse
    {
        index(addressID);

        attribute name : string<256> required;
        attribute location : string<256>;
        attribute capacity : int;
        attribute description : string localized;
        attribute addressID : uuid;

        relation stockItemPOs : StockPO[0..n] inverse warehousePO      ...
            implements stockItems;

        dependency address : Address handler "com.intershop.beehive. ...
            core.capi.profile.ProfileMgr"
        {
            foreign key(addressID);
        }
    }
}

```

CAPI Models

CAPI Interfaces

The CAPI models define the Intershop 7 cartridge API (CAPI). Typically, this is just a thin interface layer that is implemented by ORM classes or other Java classes.

A CAPI interface has the following structure:

```

cartridge" "interface" <name> (<property>)*
"{
    (<member>)*
}"

```

Members can be CAPI attributes, constants, relations or operations.

Table 36. CAPI Interface Properties

Property	Description
extends <superinterfaces>	A comma separated list of CAPI interfaces from which this interface inherits.

CAPI Attributes

A CAPI attribute is a member of a CAPI interface. It has the following structure:

```
(<modifier>)* "attribute" <name> >:> <type> (<dimension>)* (<modifier>)* ";"
```

The type of a CAPI attribute can be a primitive type or an external type.

The optional dimension defines the dimensions for arrays of the type. For each dimension, a pair of brackets must be specified, e.g. `[]` stands for a two-dimensional array.

Table 37. CAPI Attribute Modifiers

Modifier	Description
localized	The attribute is localized, e.g. it can have multiple values, one for each locale.
mapped	The attribute is not stored in this object directly, but it is mapped to an extensible object attribute (as non-localized attribute).
readonly	The attribute can only be read, there will be no setter method.
required	The attribute must have a non-null value.

CAPI Constants

A CAPI constant is a member of a CAPI interface. It has the following structure:

```
"constant" <name> ":" <type> &=& <default value> ";"
```

It defines a public static final attribute. The type can be a primitive type or an external type. The value must be provided as a literal.

CAPI Relations

A CAPI relation is a member of a CAPI interface. It has the following structure:

```
(<modifier>)* "relation" <name> ":" <type> "[" <cardinality> "]" ...  
<modifier>)* ";"
```

The type of the relation must be another CAPI interface. The cardinality can be `0..n` or `0..1`.

Table 38. CAPI Relation Modifiers

Modifier	Description
readonly	If the relation can only be read, i.e., there is no setter method.
pageable	The relation is pageable, i.e., there are special getter methods that return an <code>PageableIterator</code> instance.

CAPI Operations

A CAPI operation is a member of a CAPI interface. It has the following structure:

```
"operation" <name> "(" (<parameter>)* ")" ":" <return type> ...  
<dimension>)* (<property>)* ";"
```

The optional parameters (as comma-separated list) are defined as:

```
<name> ":" <type> (<dimension>)*
```

The parameter types and the return type of an operation can be a primitive type, a CAPI interface or an external type.

The optional dimension defines the dimensions for arrays of the parameter or return type. For each dimension, a pair of brackets must be specified, e.g. stands for a two-dimensional array.

Table 39. CAPI Operation Properties

Property	Description
throws <exceptions>	A comma-separated list of exceptions that the method throws. Exceptions are always external types, as they can currently not be defined in EDL.

CAPI Example

```
namespace com.intershop.demo.capi
{
    cartridge interface Warehouse extends ExtensibleObject
    {
        attribute name : string required;
        attribute location : string;
        attribute capacity : int;
        attribute description : string localized;

        relation stockItems : Stock[0..n] readonly;
        relation address : Address[0..1];
    }
}
```

XML Models

XML Classes

XML classes define Java data structures that can be converted into / from an XML representation by a generated encoder / decoder. An XML class can be a value type, or an object type, depending on its inheritance from the DValue or DObject superclasses.

```
(<modifier>)* "xml" "class" <name> (<modifier>)*
"{
    (<member>)*
}"
```

The members can be XML attributes or XML containments.

Table 40. XML Class Modifiers

Modifier	Description
abstract	The class represents an abstract superclass.

Table 41. XML Class Properties

Property	Description
extends <xml superclass>	The XML super class.
implements <capi interfaces>	A comma-separated list of CAPI interfaces that the type implements.

XML Attributes

An XML attribute is a member of an XML class. It has the following structure:

```
attribute" <name> ":" <type> (<dimension>)* ("=" <default value>)? ...  
<property>)* ";"
```

The type can be a primitive type or an external type.

The optional dimension defines the dimensions for arrays of the attribute type. For each dimension, a pair of brackets must be specified, e.g. stands for a two-dimensional array.

The default value can be a literal.

Table 42. XML Attribute Properties

Property	Description
tag	The name of the attribute in the XML document.
type	Defines how the attribute is mapped to XML, one of element, attribute or text.
pattern	A formatter pattern to express an XML string conversion for some types, e.g., for date values.

XML Containments

An XML containment is a member of an XML class that represents a sub-tree in an XML document. It has the following structure:

```
"containment" <name> ":" <type> "[" <cardinality> "]" (<property>)* ";"
```

The type must be an XML class. The cardinality can be 0..1 or 0..n.

Table 43. XML Containment Properties

Property	Description
tag	The name of the XML tag to which the containment is mapped. Default is the containment name.
type	Defines how the containment is mapped to XML, one of element or text.

XML Example

```
import "enfinity:/core/edl/com/intershop/beehive/core/common/DObject.edl";
primitive string;
namespace com.intershop.example
{
    xml class DProduct extends DObject
    {
        attribute name : string tag "product-name" type element;
        containment bundledProducts : DProduct[0..n] tag "bundled-products" type text;
    }
}
```

R API Models

R API Interfaces

The R API model defines the Intershop 7 remote API. It consists of R API interfaces that represent a single remote service each. They have the following structure:

```
(<modifier>)* "remote" "interface" <name> (<modifier> | <property>)*
"{
    (<member>)*
}"
}
```

Members can be RAPI operations.

Table 44. RAPI Interface Modifiers

Modifier	Description
abstract	If the generated implementation represents an abstract superclass from which other services inherit.
soap	If present, an additional SOAP stub / skeleton will be generated for the interface.
binary	If present, the contents of SOAP messages will be encoded in binary form. Default is an XML encoding.

Table 45. RAPI Interface Properties

Property	Description
extends <rapi superinterface>	The super class from which this remote interface inherits.

RAPI Operations

RAPI operations represent method declarations for remote interfaces. They have the following structure:

```
"operation" <name> "(" (<parameter>)* ")" ":" <return type> ...
    (<dimension>)* (<property>)* ";"
```

The optional parameters (comma-separated list) are defined as:

```
<name> ":" <type> (<dimension>)*
```

The parameter types and the return type of an operation can be a primitive type, an XML type, a CAPI interface or an external type.

The optional dimension defines the dimensions for arrays of the parameters or the return type. For each dimension, a pair of brackets must be specified, e.g. stands for a two-dimensional array.

Table 46. RAPI Operation Properties

Property	Description
throws <exceptions>	A comma-separated list of exceptions that the method throws. Exceptions are always external types, as they can currently not be defined in EDL.

Pipelet Development

Pipelets

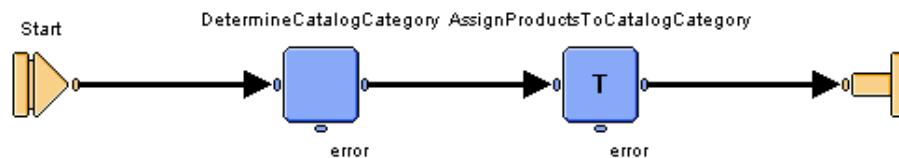
What Are Pipelets?

Pipelets are small, re-usable units or building blocks that perform the business logic in Intershop 7. Each pipelet models a discrete business function.

For example, the `AssignProduct` pipeline (see below) models the process of assigning a product to a catalog category. It breaks down this process into the following processes, each being modeled by a pipelet:

- Determine the catalog category
- Assign the product to the catalog category

Figure 68. The AssignProduct pipeline



Pipelets interact with the business object layer by using business objects to access information persistently stored in the database. They access managers of the business object layer to obtain business objects and to operate on them.

Pipelets also interact with pipelines/the pipeline processor. Pipelines contain pipelets (and other components), each pipelet representing a different processing step within the pipeline. Pipelines represent a model for complex business processes where each pipelet within the pipeline fulfills only one, discrete business function.

Pipelets exchange data with other pipelets via the pipeline dictionary. Pipelets use the pipeline dictionary to read and/or write dynamic data produced during pipeline execution. Pipelets may also use and modify data from the current session object, and read configuration data.

Pipelet Class

Each pipelet consists of a Java class file (Pipelet class) and two or more XML files (Pipelet descriptor files). Pipelet classes have the following general characteristics:

- A pipelet class extends the abstract base class `com.intershop.beehive.core.capi.pipeline.Pipelet`. The base class defines several methods used by the pipeline processor when executing a pipelet node.
- A pipelet class must implement the `execute()` method. The `execute()` method, called by the pipeline processor when executing a pipelet node, encodes all operations performed by the pipelet. The `execute()` method receives the pipeline dictionary as parameter.
- The `execute()` method returns a status code back to the pipeline processor. The default return value is `PIPELET_NEXT`, indicating that pipelet execution was successful and that the next node in the default execution path of the pipeline should be targeted. The return value `PIPELET_ERROR` can be used to indicate that pipeline execution should continue via the pipelet's modeled error exit, so allowing the pipelet to influence the flow of pipeline execution. This return value can be used if the pipeline defines an error exit, e.g., a special branch for error handling, for the respective pipelet.
- If pipelet execution fails, for instance if a required input parameter is not available, a `PipeletExecutionException` should be thrown. Whereas returning `PIPELET_ERROR` indicates a "soft" error used to influence pipeline execution (e.g. the user forgot to enter his password), throwing the exception implies a "hard" error indicating a bug in the template or pipeline design (e.g. the name of the authentication server is not available in the pipeline dictionary as required).

The basic structure of a pipelet class (with default return constant) is shown below:

```
public class MyPipelet extends Pipelet
{
    public int execute(PipelineDictionary dict) throws PipeletExecutionException
    {
        // do something here
        ...
        // set return value
        return PIPELET_NEXT;
    }
}
```

Pipelet Descriptor Files

Each pipelet is associated with two or more XML-based descriptor files: a parameter file and a description file (one per locale).

The *parameter file* stores the pipelet parameters and properties including the pipelet name, class name and package path, transaction properties, configuration data, as well as information about data the pipelet reads from or writes to the pipeline dictionary and the current session object.

The *description file* stores localizable information, such as the general pipelet description and descriptions associated with particular pipelet properties. See *Set General Pipelet Properties* and *Set Pipelet Properties* for information on pipelet properties and parameters, and how to set them in Intershop Studio.

Creating Pipelets

The pipelet wizard allows you to add a new pipelet to a cartridge project. The wizard collects the required information, including pipelet name, pipelet description, and pipelet properties.

Start the Pipelet Wizard

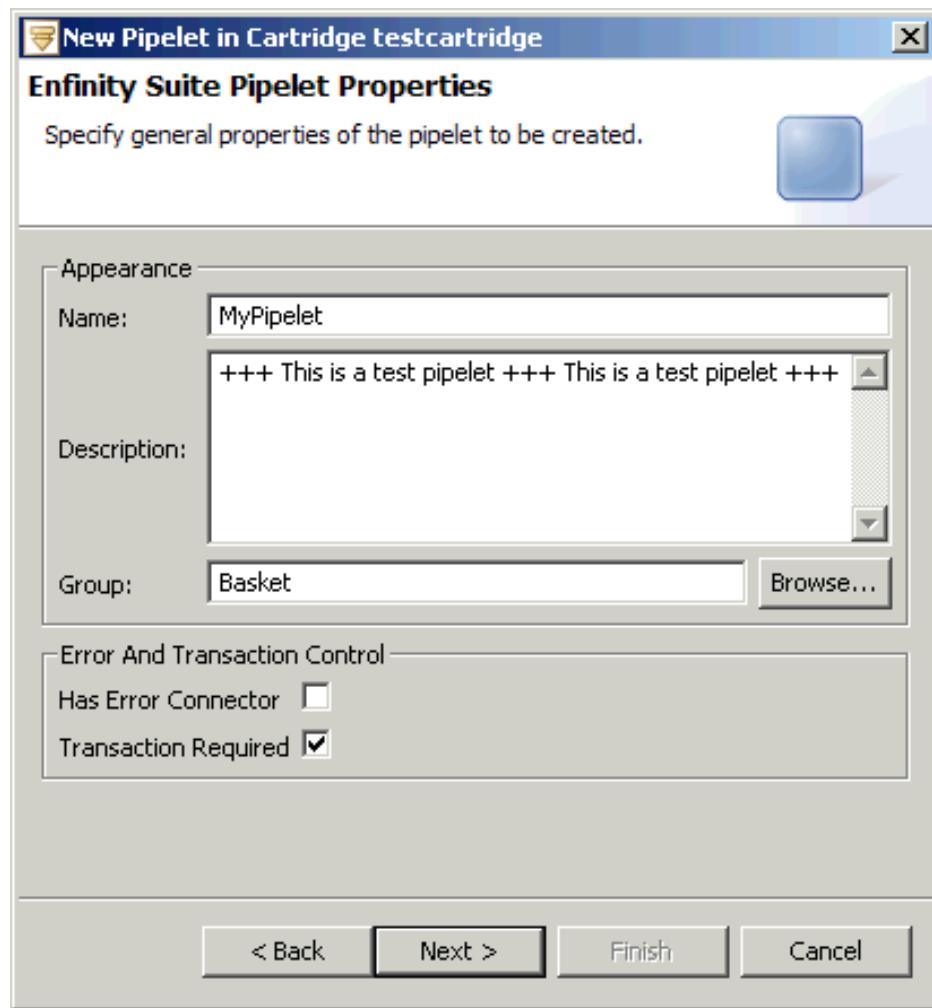
To start the pipelet wizard:

- 1. In the Cartridge Explorer, right-click the cartridge project for the new pipelet to open the context menu.**
- 2. From the context menu, select New | Pipelet.**

The pipelet wizard is displayed. If you have not selected a cartridge before starting the pipelet wizard, the pipelet wizard first prompts you to select the cartridge for the pipelet.

Set General Pipelet Properties

The first wizard page allows you to set general pipelet properties as described in the table below:

Figure 69. General Pipelet Properties**Table 47. General Pipelet Properties**

Property	Description
Name	Defines the pipelet name used for display purposes, for example, in the pipeline editor and the pipelet view.
Description	Specifies description information for a pipelet. The pipelet description can be localized.
Group	Assigns the pipelet to a pipelet group. Pipelet groups are used in the pipeline editor and the pipelet view to facilitate pipelet handling.
Has Error Connector	Specifies whether the pipelet has an error connector. The error connector defines a special pipeline flow if the pipelet cannot be successfully executed.
Transaction Required	Requiring a transaction for a pipelet ensures that all transaction-sensitive methods of the pipelet are executed in the scope of the same transaction.

Once all properties are defined, click Next to proceed.

Define Class Name and Package Structure

On the second page you can define the class name and the package structure for the pipelet. See the table below for details:

Figure 70. Class Properties

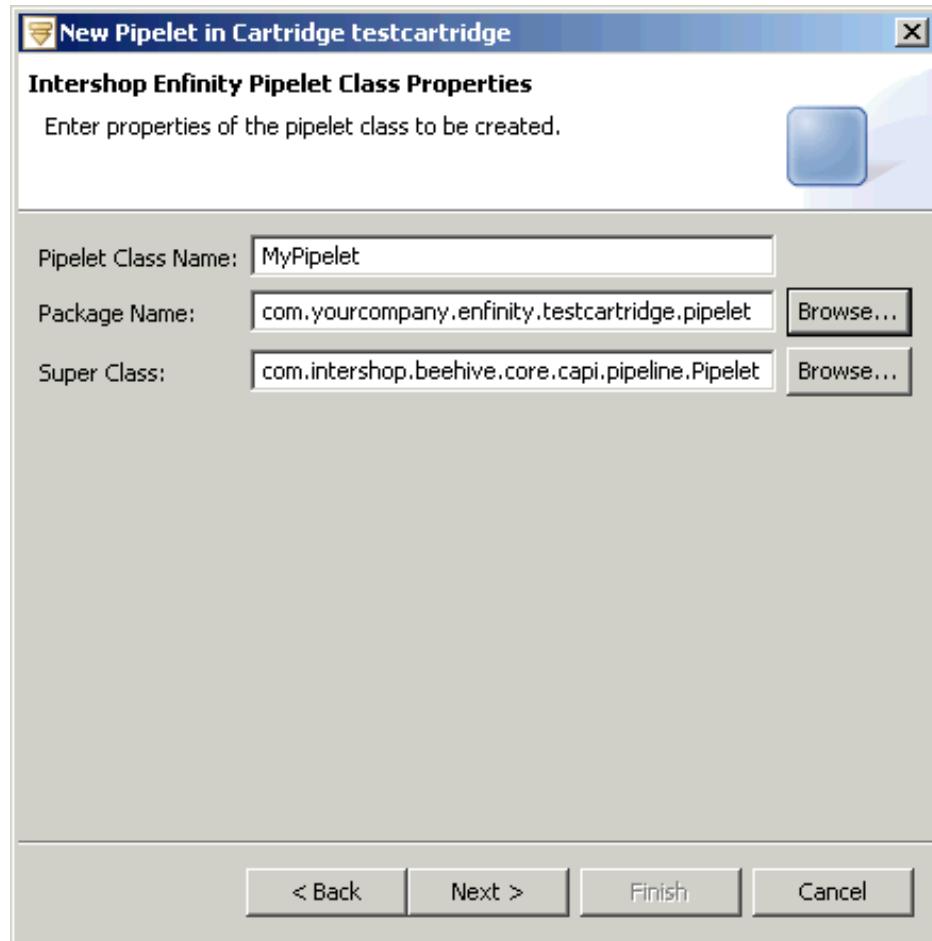


Table 48. Pipelet Class Name and Package Structure

Property	Description
Pipelet Class Name	Defines the name of the pipelet's Java class.
Package Name	Defines the package containing the pipelet. Clicking the Browse button next to the field opens a window showing all available packages within the current structure of the cartridge development directory.
Super Class	Sets the super class for the pipelet. Intershop Studio uses the standard Intershop 7 class com.intershop.beehive.core.capi.pipeline.Pipelet as default. Clicking the Browse button next to the field opens a list with all available pipelet classes in the cartridges to which a direct or indirect dependency exists. Choose a class from this list to set a super class that is different from the default.

After all properties are defined, click Next to proceed.

Set Pipelet Properties

On the third page of the wizard you can set configuration, and dictionary properties of the pipelet. See the table below for details:

Figure 71. Pipelet Properties

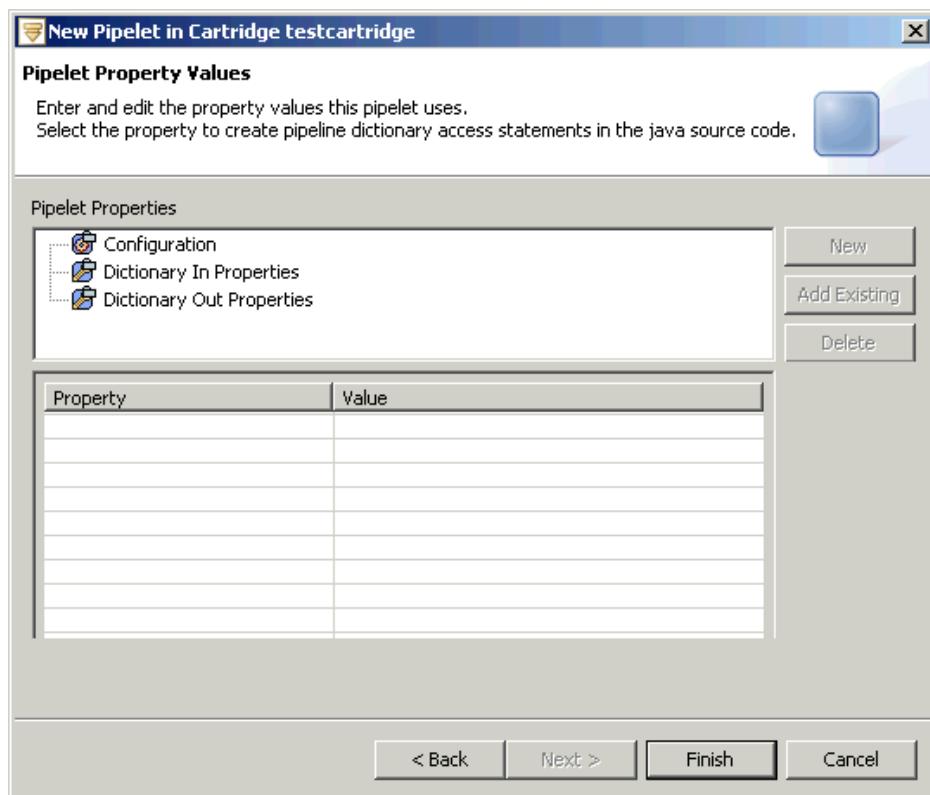


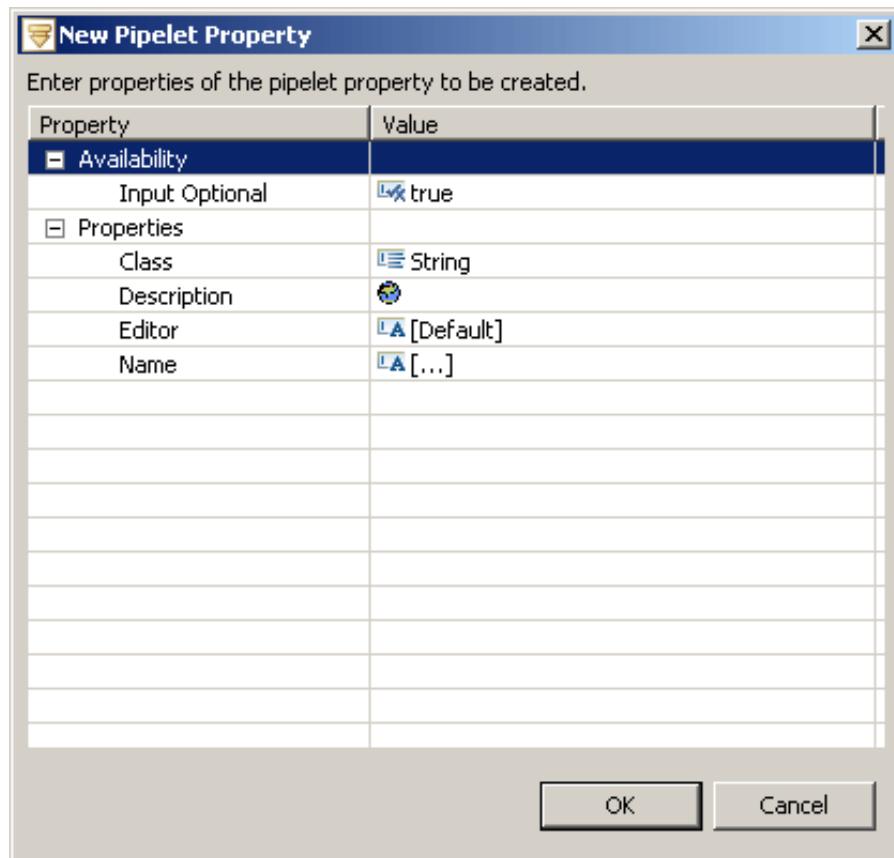
Table 49. Pipelet Properties

Property Group	Description
Configuration Properties	Configuration properties are read from the pipelet configuration. They are handled within the init() method of the pipelet.
Dictionary Properties	Dictionary properties are read from (dictionary-in) or written to the pipeline dictionary (dictionary-out). They are available during the runtime of the associated pipeline.

Using the wizard, you can define new properties or reuse properties which have already been defined for other pipelets. To define a configuration or dictionary property, select the respective category in the upper pane and do one of the following:

- **Define new pipelet properties**

To define a new pipelet property that currently does not exist in the system, click the New button. This opens the New Property dialog.

Figure 72. Define New Pipelet Property

The following fields are available to define a new pipelet parameter:

Table 50. Pipelet Properties

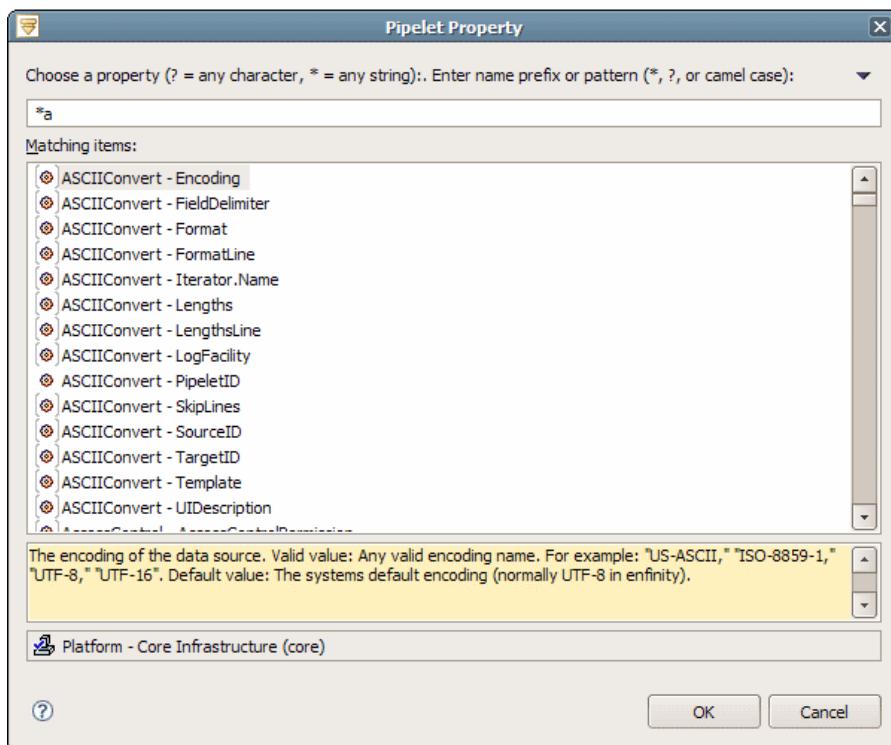
Property	Description
Required	Determines whether the parameter is required (true) or not (false) for successful pipelet execution.
Name	Name of the parameter; used for display purposes.
Description	Localizable description for the pipelet, used in the pipeline editor.
Class	Defines the class to which the parameter belongs. Click the value field to open a list of all classes that are available on your system.
Group	This option is available for dictionary properties. The option determines whether the defined property serves as input parameter (<code>dictionary_in</code>), output parameter (<code>dictionary_out</code>), or both input and output parameter.
Editor	This option is available for configuration properties. Choose the editor type (Regular Editor or Selection Editor) to be used to edit the configuration parameter. See

Property	Description
	<i>Working with Pipelet Property Editors</i> for more information.
Create Java Access Statement	Use this option (and the sub-options it contains) to automatically add constant and variable definitions and accessor methods for a property to the pipelet class, such as methods to read a dictionary input property from the pipeline dictionary and store it in a local variable, to put a dictionary output property into the pipeline dictionary, or to read and store a pipelet configuration property.

■ Add existing pipelet properties

If a property has been already defined for another pipelet, you can re-use it by just copying it. In order to assign an existing property to the pipelet properties list, click the Add Existing button. This opens the Pipelet Property dialog.

Figure 73. Add Existing Property



To locate the intended property, use the search options offered in this dialog:

- Specify a search string, using, if required, camel-case expressions or using the content assist.
- Use the filtering options, i.e., specify a working set or a cartridge context.

Select the property you want to add to your pipelet from the search result list, then click OK.

After you defined all pipelet properties, click Finish to start the actual pipelet generation process.

Working with Pipelets

Edit General Pipelet Properties

General pipelet properties can be edited in the Properties View. To edit pipelet properties, select the pipelet in the Cartridge Explorer and edit properties in the Properties View as needed.

NOTE: Configuration and dictionary properties are edited via the Pipelet Properties Wizard. See Add Additional Pipelet Properties for details.

Add Additional Pipelet Properties

To add additional properties to a pipelet:

1. Right-click the pipelet in the Cartridge Explorer to open the context menu.
2. From the context menu, select Pipelet | Edit Pipelet Properties.

This opens the Pipelet Properties Wizard.

3. Add or modify properties as needed.

See *Set Pipelet Properties* for additional information.

NOTE: Pipelet properties can also be added via the pipelet's properties view in the Cartridge Explorer. To add a property, select the respective row (e.g., Configuration Properties) and click. To delete a property, select the respective property in the Properties view and click.

Working with Pipelet Property Editors

Values for pipelet configuration properties are typically provided in the pipeline editor, when incorporating the pipelet in a specific pipeline. To avoid pipelet configuration errors, it is possible to pre-define possible values for the configuration parameter, or to impose restrictions on the values which a pipelet can accept. The following pipelet property editors are available to pre-define configuration values:

■ String Choice Editor

This editor is used to define a set of possible values for a configuration parameter. These values are then displayed in a drop-down list to the user configuring the pipelet in the pipeline editor. It is also possible sort the values in the drop-down list.

■ Regular Expression Editor

This editor is used to impose general restrictions on possible configuration values. When a configuration value is provided, it is checked against the regular expression defined in this editor. An error message can be defined to be displayed in case the check fails.

The editor supports editing regular expressions using the content assist, and provides a dialog that allows to match the regular expression against an input string.

■ Java Type Name Editor

This editor is used to define a fully qualified name of a Java type as the property value.

■ Element Reference Editor

This editor is used to define a reference to a cartridge element (pipelines, templates, queries, static files, etc.) as the property value.

■ Multiline Text Editor

This editor is used to define property values intended to hold multi-line texts.

To define pipelet properties using, for example, the String Choice editor, proceed as follows:

- 1. Open the pipelet in Pipelet Model Editor.**
- 2. Expand the tree data structure of the pipelet.**
- 3. Select the desired configuration .**
- 4. Navigate to Value Editor tab of the Properties view.**
- 5. Select the desired value editor from the drop-down list.**

It may be necessary to re-select the configuration within data tree structure to refresh the display of the Value Editor.

- 6. Click the arrow icon .**
Value editor's settings are displayed.
- 7. Fill in the desired values into the appropriate field.**
- 8. Save the pipelet file to save changes.**

NOTE: Similarly, regular expressions, Java types, cartridge element references or multi-line texts are configured as property values using the corresponding editor.

Explore Pipelet References

Intershop Studio offers extensive search functions which allow for exploring the multi-use of a pipelet very easy.

- 1. Open Pipelets View.**
- 2. Select the desired pipelet.**
- 3. Right-click and select Cartridge References from context menu.**
- 4. Set filter for your search.**

You may be search across the whole workspace or set filter for search in workspace projects only or to the current cartridge or a specific other cartridge or working sets.

The Search view lists all the places where the pipelet is referenced. At a glance it can be seen in what cartridges and in which pipelines the pipelet is used.

Using the Pipelet Assist

The Pipelet Assist automatically performs certain pipelet development tasks and helps you to keep the pipelet configuration consistent with your code. For example, you can use the Pipelet Assist to add access statements for pipelet properties to your code. If the accessed property does not exist, the Pipelet Assist automatically

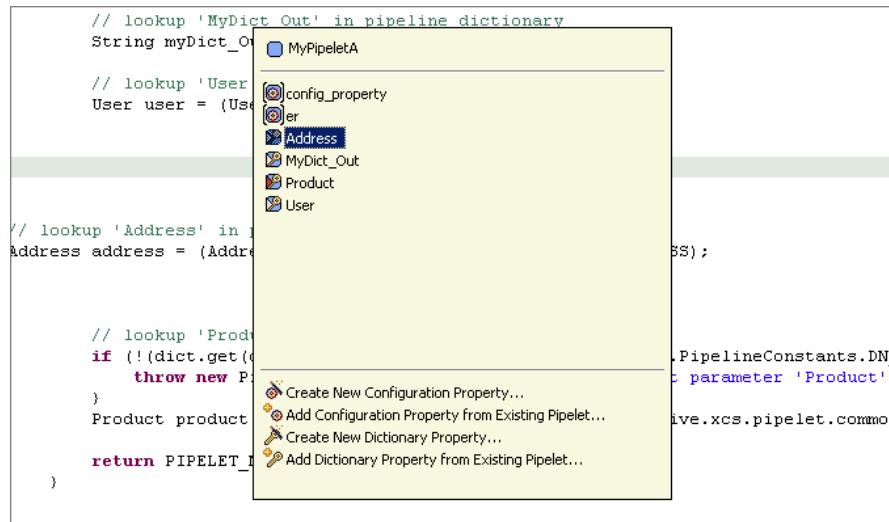
adds it to the pipelet configuration. Using the Pipelet Assist, you can also copy properties from other pipelets, or define new ones.

To open the Pipelet Assist and insert an access statement:

- 1. Place the cursor at the position where you want to insert the access statement.**
- 2. Press **Ctrl-Shift + D** to open the pipelet assist. Alternatively, select **Edit | Pipelet | Insert Access Statement** from the Workbench menu.**

The pipelet assist displays a floating window at the cursor position.

Figure 74. Pipelet Assist



3. Select a property from the list and press Enter.

The pipelet assist inserts the new code at the cursor position and updates the pipelet descriptor files.

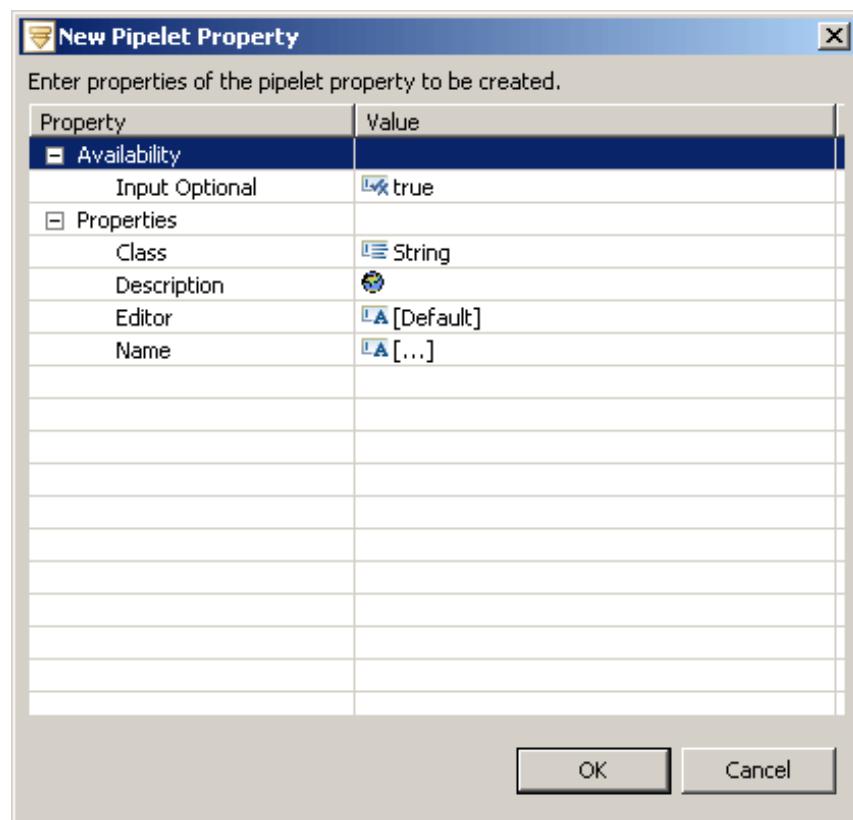
You can also use the Pipelet Assist to create new properties before inserting the access statement:

- 1. Press **Ctrl-Shift + D** to open the pipelet assist.**
- 2. Click one of the commands on the bottom of the pipelet assist window.**

- Create new property

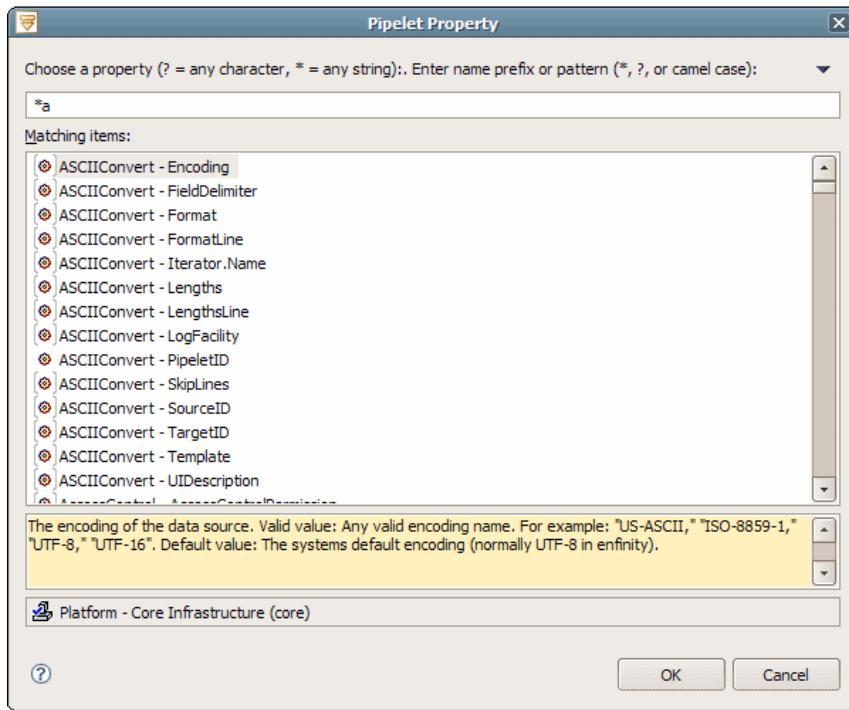
Opens the New Pipelet Property dialog where you can define the property.

Set the property type, name, and description and define whether the property is required/guaranteed or optional. For localizable attributes, e.g. the description, click the **[...]** button to enter values for the different locales.

Figure 75. New Pipelet Property dialog

- Add property from existing pipelet

If a property has been already defined for another pipelet, you can just copy it. The command opens the Pipelet Property dialog where you can search for already defined properties.

Figure 76. Add existing property

To locate the intended property, use the search options offered in this dialog:

- Specify a search string, using, if required, camel-case expressions or using the content assist.
 - Use the filtering options, i.e., specify a working set or a cartridge context.
- Select the property you want to add to your pipelet from the search result list, then click OK.

When you have created the new property, it is added to the list of available properties in the Pipelet Assist. If you copied a dictionary property from an existing pipelet, the property inherits the access type and description from the original. The current access type is indicated by triangles that appear either on the left (in), the right (out), or both edges (bidirectional) of the icon assigned to the property. The color of the triangle indicates whether the property is optional (yellow) or required/guaranteed (red). Both attributes can still be changed. Highlight the property in the list, then:

- Press Space to cycle through the possible access types (in, out, and bidirectional). The position of the triangle changes accordingly.
- Press Shift + Space to define whether a property is optional or required/guaranteed. The color of the triangle changes accordingly.

Using the Quick Fix Function

Use the Quick Fix function to automatically solve frequently occurring problems. For example, if your pipelet code contains an access statement to an undefined

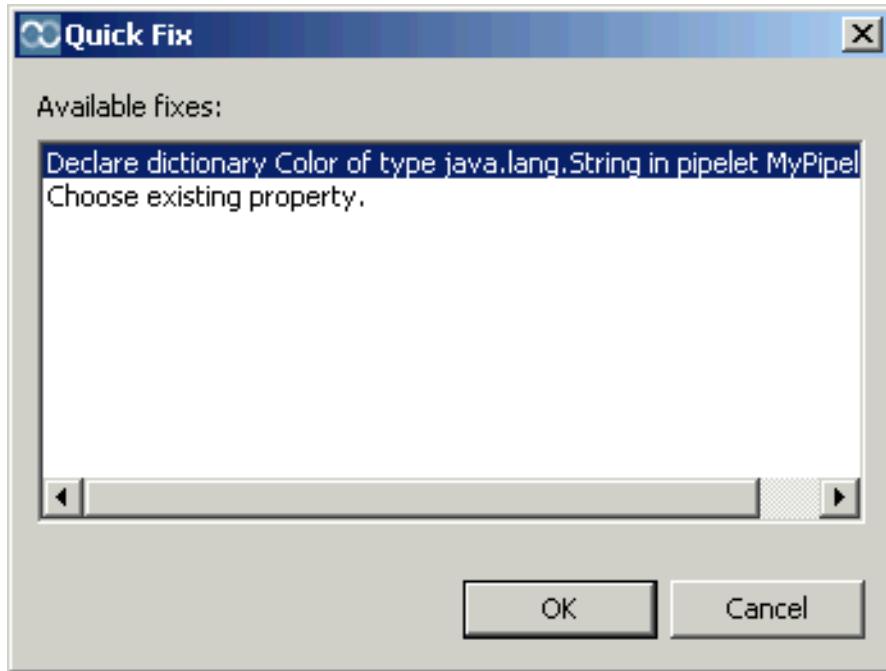
pipelet property, an appropriate task will be added to the task list. You can then select a quick fix that will add the missing property to the pipelet configuration.

To apply quick fixes to task list entries:

- 1. Right-click on a warning or error in the task list.**
- 2. From the context menu, select Quick Fix ...**

The Quick Fix dialog opens.

Figure 77. Quick Fix Dialog



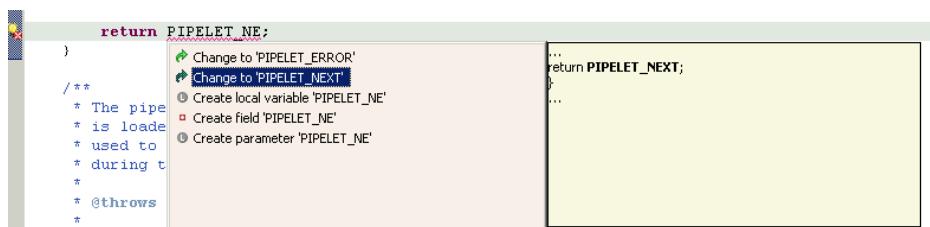
NOTE: If the command in the context menu is grayed out, no quick fix is available.

- 3. Choose one of the options to fix the problem and click OK.**

The fix is carried out and the task removed from the list.

The quick fix function can also be invoked directly from the Java editor. An icon on the scroll bar appears in case a quick fix is available for an error or a warning. Click on the icon to open the list of available quick fixes, including a preview of the proposed solution.

Figure 78. Quick fix in Java Editor



NOTE: If the automatic build is disabled, Intershop Studio does not automatically update warning and error markers in the code (including proposed quick fixes). To update these markers after modifying your code, build your project manually as described in here. To update these markers for

a single resource only, right-click the element in the Cartridge Explorer to open the context menu and select Check Element.

Pipelet Reload

New or modified pipelets deployed on the development system need to be loaded before they can be executed by the Intershop 7 application server. This is done by reloading all pipelets that belong to the cartridge which contains the new or modified pipeline.

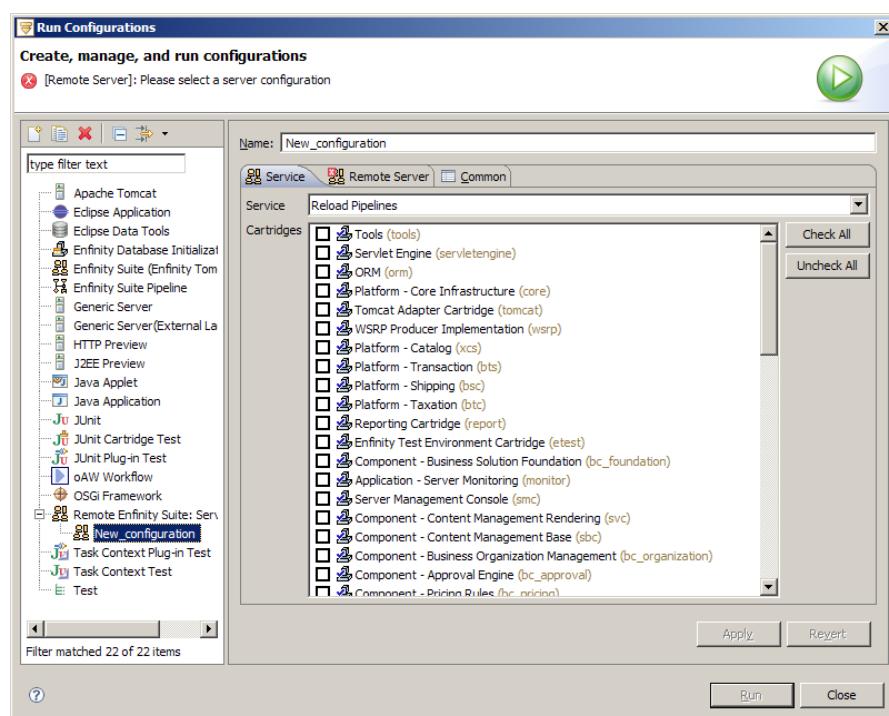
NOTE: The cartridge containing the new or modified pipelets must be registered on the development system for the pipelet reload to work. For more information, see here.

- From the context menu, select Run | Run Configurations.

The page to create, manage and run configurations is displayed.

- On the configurations panel, select a Remote Intershop 7 Service configuration. If necessary, right-click Remote Intershop 7 Service to create a new configuration, or right-click the configuration to clone it.
- On the Service tab, select Reload Pipelets from the Service drop-down list. Select the check boxes for the cartridges whose pipelets need to be reloaded.

Figure 79. Pipelet Reload



- On the Remote Server tab, select the server connection parameters to use. If necessary, configure a new server connection. See *Remote Server Configuration* for configuration details.
- Select Run to start the pipelet reload.

To run the same configuration again, select Run | Run History from the workbench menu.

NOTE: The Intershop 7 application server can be configured to reload pipelines and pipelets automatically. If the property intershop.pipelines.CheckSource in the appserver.properties file is set to "true", the server checks for new pipeline or pipelet files before executing a request.

Editing Pipelets on Remote Servers

Intershop Studio provides the possibility to directly edit pipelets on remote servers. This features (also referred to as "hot edit") enables you, for example, to modify pipeline or pipelet parameters on a non-development system (e.g. test or production system).

NOTE: Remote editing should be used to diagnose problems and fix smaller issues. When dealing with larger issues, consider using the remote development feature instead (for details, see *Remote Server Configuration*).

For remote editing, Intershop Studio provides a specialized perspective, the Intershop 7 Remote Editing perspective. With respect to the explorers, views and editors, the Intershop 7 Remote Editing perspective is similar to the Cartridge Development perspective.

To edit a pipelet on a remote system:

- 1. Configure a remote server connection**

See *Remote Server Preferences* for details.

- 2. In the workbench, click Window | Open Perspective | Other. Select the Intershop 7 Remote Server Perspective from the list.**

Having established the connection to the remote server, the cartridges are displayed in the Cartridge Explorer.

Styleguide

Quick Overview

This section contains a quick overview of conventions and rules that are outlined and discussed in more detail in the remaining sections of this document. Use this checklist whenever developing / reworking pipelets.

The following things, you *should* do:

- Make sure you always use Intershop Studio for editing pipelet descriptors to keep source code and descriptor in sync.
- Make sure your pipelet name fits the pipelet-naming scheme.
- Make sure you assigned the pipelet to a meaningful pipelet group.
- Make sure you configured the correct pipelet transaction mode.
- Make sure you configured whether or not the pipelet uses an error exit.
- Make sure your pipelet does not transform strings into object types (i.e. integers, doubles).
- Make sure you configured required parameters as such in the pipelet descriptor.
- Make sure your pipelet correctly handles localizable custom attributes.
- Make sure you provided a meaningful pipelet JavaDoc.

- Make sure you documented the individual I/O parameters.
- Make sure you documented when the error exit is used (possible exception are the standard pipelets Create..., Get...).

Things you *should not* do:

- Never transform strings into object representations (use formatter utility pipelets instead).
- Do not pass parameters as String, but use the original type (e.g. int).
- Never access form data in the pipeline dictionary (use form data utility pipelets instead).
- Never change the pipelet I/O parameter set of existing pipelets in a way that is not backward compatible (see API migration section for rules).
- Never introduce local pipelet variables that are not accessed in a read only manner (exceptions are caches).
- Avoid subclassing of pipelets since this introduces unnecessary dependencies (use utility helper classes to extract code shared across different pipelets).
- Using the error exit is no reason to log an error as it is part of the normal behavior for the pipelet.

Pipelet Modeling

To ensure synchronization between pipelet descriptor and implementation (i.e. Java source code), the use of tools is required. Manual pipelet descriptor changes should be avoided at all costs. The recommended tool for pipelet modeling and editing is Intershop Studio.

Pipelet Naming

Pipelet Class Naming

It is recommended to use a consistent naming scheme for pipelets that indicates the purpose of a pipelet based on its name.

The naming conventions outlined below are by no means complete but apply to about 80% of all cases.

The pipelets used most frequently are those managing the life cycle of persistent objects. The following naming scheme is recommended for the following pipelets:

■ `Create<Object>`

Creates a new instance of the specified business object based on some sort of identification criteria, i.e., primary key attributes, retrieved from the pipeline dictionary. This pipelet should focus on object creation with the primary key attributes and other mandatory attributes. The pipelet should not take care of completely configuring the business object with all optional parameters. This should be done by a subsequent update pipelet. The pipelet should always provide an error connector that is used whenever the business object could not be created.

- `Update<Object>`

Updates a business object instance (available in the pipeline dictionary) with some new attributes (available in the pipeline dictionary). This pipelet should not have an error connector and should gracefully ignore errors that might come up when updating the object attributes. Update pipelets will usually not care about custom attributes. These are to be handled by a generic pipelet provided by the platform.

- `Remove<Object>`

Removes a business object based on some sort of identification criteria which is usually retrieved from the pipeline dictionary. The pipelet should not have an error connector and should gracefully ignore errors during the removal process, e.g., a missing object.

Once business objects have been created, it is essential to have a powerful means to identify them and look them up in other business processes. The following pipelets should be provided to accomplish this.

- `Get<Object>ByUUID`

Tries to identify a single object instance based on its primary key (i.e. the UUID for all subclasses of `PersistentObject`).

The UUID is retrieved from the pipeline dictionary. The pipelet uses the error exit in case no object with the specified UUID could be found.

- `Get<Object>By<Attr>`

Tries to identify a single object instance based on some attribute(s) (that is not the primary key). The identifying attribute(s) are retrieved from the pipeline dictionary. Get pipelets always provide an error connector that is used whenever the requested business object could not be found.

- `Get<Object>By<Object>`

Refers to the same concepts as the `Get<Object>By<Attr>` pipelet. The only difference is that here another, usually associated, object is used for the lookup.

- `Get<Object>sBy<Attr>`

Tries to identify a single object or a collection of objects based on an attribute (which is not the primary key). The identifying attributes are retrieved from the pipeline dictionary. This pipelet is important whenever lists of business objects need to be processed, e.g., all departments of an organization. The pipelet does not provide an error connector. Instead, empty iterators are stored in the pipeline dictionary if errors occurred or empty result sets were returned.

- `Get<Object>[s]By<Object>`

Refers to the same concepts as the `Get<Object>sBy<Attr>` pipelet. The only difference is that here another, usually associated, object is used for the lookup.

Beside standard life cycle and lookup pipelets, pipelets are required that do some actual processing on the business objects of the pipeline dictionary. The naming

scheme for these pipelets depends on the type of processing done by the pipelet. Below are some examples and recommendations that might be used as guidance:

■ `Calculate<Object>`

Is used whenever some sort of calculation is done on the object. This usually leads to updates on the object attributes. This pipelet requires that the object in question was identified by an appropriate lookup pipelet before.

■ `Process<Object>`

With this pipelet, some sort of processing takes place on the object that was identified previously by a lookup pipelet.

Pipelet Group Naming

Pipelet groups do not have a real function when it comes to pipelet or pipeline processing. They are purely declarative and are used by the Visual Pipeline Manager for an alternative grouping of the whole pipelet set. The feature is designed to help developers locate all pipelets that do something with certain business objects without looking at their name.

The VPM pipelet grouping only works successfully when all pipelets are categorized within a small but meaningful set of pipelet groups. Too many groups are less helpful than too few groups.

- There should be at least one pipelet group for each business component used in a solution. One group is sufficient for smaller components that do not define many new business objects. For example, it is probably sufficient to have one pipelet group for the approval component. The group contains all approval process-related pipelets.
- For larger components with lots of business objects, there should be more than one group. For such components, groups' names should be chosen that relate to the business objects being processed. There should, at least, be groups for pipelets that deal with departments, cost centers, budgets and organizations.
- If pipelets extend platform functionality, e.g., additional user management functions, use the name of the pipelet groups that are used in the platform. This ensures that the pipelets show up in the appropriate section in the VPM even though they come from a different cartridge.
- There should be exactly one pipelet group per project that holds all project-specific pipelets. No business pipelets should be assigned to this group. Usually, only a few presentation-related pipelets fall into that group.
- Assign obsolete pipelets to a special pipelet group named "Deprecated" to indicate that support for the pipelet is not going to be continued for the next product releases.
- Avoid having generic pipelet group names such as `Common`, `Misc` or `Util`.

Note that pipelet group names should start with a capital letter and consist of a single word only.

Pipelet Parameter Naming

Pipelet parameter names and types are an essential part of the pipelet API because they basically make up the pipelet signature. Whenever a pipelet name does

not allow developers to infer the pipelet's meaning at first glance, the pipelet parameters will. Choosing an appropriate set of pipelet parameters is essential for pipelet reuse.

Pipelet parameter names need to make sense for the pipelet that uses the parameter. When choosing a name, do not focus on the pipeline the pipelet might be used in. Due to the dictionary aliasing feature of Intershop 7, there is no need to synchronize the pipelet parameter names of all pipelets that are used in a single pipeline. This gives more freedom to the developer for choosing semantically meaningful names. Below are some general parameter naming conventions that are mandatory:

- Pipelet parameter names should always start with a capital letter. The general naming scheme follows the one for Java classes. Consequently, User, UserGroup, Basket are valid while usergroup, userGroup and BASKET are invalid.
- Pipelet parameter names are usually named like the business object type the parameter refers to. For example, a parameter that reads a Basket instance from the dictionary should be named like the class: Basket.
- Pipelet parameters for reading identification information should be named with the class name of the identified business object with an "ID" suffix. For example, a pipelet for looking up baskets by domain should have a parameter DomainID.

As a general rule, pipelet parameter sets should be designed in a way that good interoperability between the different pipelet groups, e.g., life-cycle, lookup, processing, is ensured. There is, for example, no point in having ID attributes in processing pipelets. Instead, processing pipelets should read business object instances from the pipeline dictionary that have to be provided by the lookup pipelets. The benefit is that you do not need to replicate the code required for retrieving the instances of business objects throughout all pipelets.

Although pipelet parameter naming is done with a pipelet focus only, there are parameters that are shared by pipelets and pipelines. To ease development, you should access and store standard objects with standard parameter names.

The following parameter names should be used for standard platform objects:

- Domain
- User
- UserGroup
- Permission

When accessing platform parameters, it is always a good idea to take a look at the naming conventions used in the platform. Although the conventions are not as strict as suggested here, they might still provide some guidance.

Pipelet Parameter Types

Like parameter names, parameter types are part of the pipelet's API. Hence, correct and consistent usage of parameter types is important. The following rules apply:

- Pipelet dictionary I/O parameters should refer to CAPI interfaces only. A pipelet should never declare an object located in the internal packages as input or output type.

- Pipelet dictionary I/O parameter representing primitive types should be read from the pipeline dictionary using their Java object representation (i.e. Integer, Double, String, Money, Quantity, BigDecimal,...).
- Pipelet configuration parameters always need to be declared as primitive types (int, double, string).

In case a pipelet consumes numeric data (e.g. numeric type codes, dates, money values, ...) it must use the appropriate Java object types. A pipelet must not read numeric information as string and convert them internally into the corresponding object type (e.g. by using `Integer.valueOf()`). There are two very important reasons for this rule:

■ Localization

Using standard string -> object conversion methods such as `Integer.valueOf()` fails as soon as the application supports multiple locales. In such a scenario, users will provide numeric input based on their locale setting (e.g. "20.50" for English and "20,50" for German). Pipelets that use the standard methods conversion will fail with a `NumberFormatException`.

■ Reusability

Declaring numeric input as strings reduces reusability of the pipelets in other environments such as Web services and jobs. Retrieving string based numeric data from an HTTP/HTML based application is just one (very common) environment for a pipelet. However, suppose you want to use the same pipelet for implementing Web services. There is no point in transforming an integer retrieved from a Web service call into a string, which will then again be transformed to an integer within the pipelet.

If numeric data is posted as string content from a template (e.g. an HTTP POST of an update form) it needs to be converted using one of the following formatter utility pipelets before it can be used:

- VerifyBoolean
- VerifyInteger
- VerifyDouble
- VerifyDate
- VerifyMoney
- VerifyQuantity

This rule ensures that localization support can be centralized in the formatter pipelets instead of being replicated into all business pipelets.

As a special case, we often have to implement pipelets which need to support multiple types for a given input parameter. Consider the following wrong (but commonly used) example:

Wrong:

```
UpdateCustomAttribute
| <IN> Object : ExtensibleObject
| <IN> Locale : LocaleInformation
| <IN>AttributeName : String
| <IN>AttributeType : String
| <IN>AttributeValue : String
```

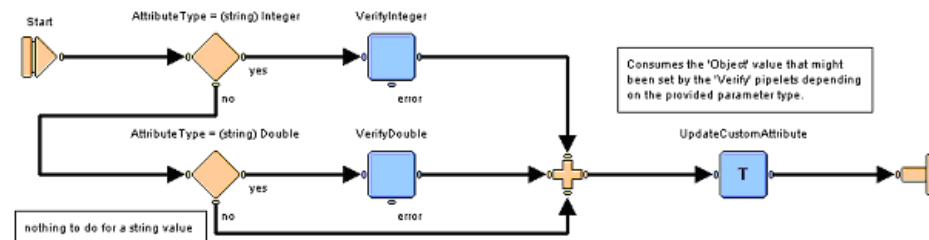
In this example, the attribute value needs to support `string`, `integer` and `double`. The (wrong) pipelet implementation reads those values as `string` and internally converts them by evaluating a numeric type code (that is also provided as a `string`). This is very bad practice, due to the dangerous internal conversion of strings (see the discussion on localization issues above) and the limited reusability of the pipelet. A better API version of this example pipelet would look like this:

Correct:

```
UpdateCustomAttribute
| <IN> Object : ExtensibleObject
| <IN> Locale : LocaleInformation
| <IN>AttributeName : String
| <IN>AttributeValue : Object
```

Note that the pipelet does not read a numeric type code any more. Instead, the pipelet reads the attribute value as an object. The implementation will use the `instanceof` operation to check for supported types (i.e. `String`, `Integer`, `Double`) and call the appropriate `ExtensibleObject` method. In an HTTP/HTML environment where the values are always posted as `strings`, an additional decision node should be used to evaluate the submitted type code in order to decide which conversion utility pipelet should be used. The following simplified pipeline view summarizes this approach.

Figure 80. Working with pipelet parameter types



Note that the type checks in this sample pipeline are just examples. In real pipelines, the type codes are likely read from somewhere else (i.e. from a web form or a form record).

Pipelet Parameter Fallbacks

Pipelets are supposed to be reusable across pipelines and even different projects. It is therefore important to design pipelets to be more generic than it might be required in the context of a certain use case. One important feature of generic pipelets is the support for statically configured configuration parameters that are used as fallbacks in case a declared I/O parameter is not available in the pipeline dictionary.

This issue is best discussed with an example. Consider a pipelet that can be used to remove a specified custom attribute for a provided `ExtensibleObject` instance.

Wrong:

```
RemoveCustomAttribute
| <IN> Object : ExtensibleObject
| <IN> Locale : LocaleInformation
| <IN>AttributeName : String
```

For the original use case, the pipelet received the name of the attribute to be removed from the pipeline dictionary (i.e. the user selected the attribute to

be removed explicitly). However, it is also perfectly possible that the pipeline designer needs to remove a custom attribute independently of the user interaction. Therefore, the attribute name should alternatively be configurable in the pipelet configuration.

Correct:

```
RemoveCustomAttribute
| <CONFIG> DefaultAttributeName : String
| <IN>     Object : ExtensibleObject
| <IN>     Locale : LocaleInformation
| <IN>     AttributeName : String
```

The pipelet implementation would select the attribute name to be used such that the pipeline dictionary value always takes precedence:

```
public void init()
    throws PipelineInitializationException
{
    cfg_defaultAttributeName = (String)getConfigurations() ...
        .get("DefaultAttributeName");
    ...
}

public int execute(PipelineDictionary dict)
    throws PipeletExecutionException
{
    // lookup AttributeName in pipeline dictionary
    String attributeName = (String)dict.get(IO_ATTRIBUTENAME_NAME);

    //{{ execute

    String attrName = attributeName != null
        ? attributeName
        : cfg_defaultAttributeName;
    if(attrName == null)
    {
        throw new PipeletExecutionException(
            ...
            "Mandatory input parameter 'AttributeName' not available.");
    }
    ...
}
```

This is a pattern that can be applied to many pipelets. All pipelet input parameters for which a static configuration at the pipeline level does make sense should complement the dictionary input parameters with appropriate 'Default' configuration parameters.

The following specific rules apply:

- The name of configuration parameter is derived from the dictionary input parameter by adding the prefix 'Default'.
- Parameter types of the input and configuration parameter need to match (i.e. String-->String, Integer-->int, Double-->double).
- The dictionary input parameter always takes precedence over the configuration parameter.
- Both parameters need to be declared as optional. For required parameters the existence check is implemented along with the parameter selection (see code sample above).

- The parameter fallbacks only make sense for primitive input types that can be provided by the pipeline designer (i.e. numeric type codes, string IDs but no UUIDs or dates).
- To enforce that a default value is used it must be ensured that the dictionary input returns null. Use the alias 'NULL' for the input parameter to insure the input value is null.

Pipelet Form Data Access

Pipelets *must not* make any assumptions about the environment they are used in. To be reusable across different projects, a pipelet must not make any assumptions about the UI that triggers the pipelet's execution. To be reusable in job- or web service-driven implementations, a pipelet must not even assume that there is an HTTP request triggering the pipelet.

It is therefore forbidden for pipelets to directly access HTTP form data stored in the pipeline dictionary. The following pipeline dictionary API calls should not be used:

```
Iterator createFormKeyIterator()
Iterator createFormPrefixedKeyIterator()
String getFormValue(String)
String getFormValues(String)
void removeFormValues(String)
void setFormValue(String, String)
void setFormValues(String, String)
```

When implementing pipelines for an HTTP/HTML based application, the following form handling utility pipelets should be used to access the form data and provide it in a controlled way for all other pipelets.

```
GetFormSelection
GetFormRecord
CreateWebForm
GetWebForm
UpdateWebForm
ValidateWebForm
UpdateClipboard
GetClipboard
```

Pipelet Error Handling

A consistent handling of errors across different pipelets is another important aspect of our pipelet API. The following basic rules apply:

- Pipelets are supposed to throw a `PipeletExecutionException` in case a required input parameter is not available in the pipeline dictionary.
- Pipelets are supposed to throw a `PipelineInitializationException` in case a required configuration parameter is not available.
- Pipelets should not catch `RemoteException`, `RuntimeExceptions` and `Throwable` in their execution body.
- Pipelets should deal with `CreateException`, `FinderException`, `RemoveException` and any other custom business exceptions (in case these exceptions are declared at the manager level) by using the error exit (if the pipelet declares one).

To decide whether a certain pipelet should declare an error exit at all, refer to the discussion in the pipelet naming scheme section (*Pipelet Naming*).

Even though pipelets often need to indicate detailed error information by storing certain error codes in the pipeline dictionary, this style guide does not provide a naming scheme for error codes, since this is an issue that needs to be addressed and better supported by the platform. In the meantime, error codes should be avoided (if possible) within pipelets. Instead use utility pipelets like `SetDictionaryValue` that should be connected to the error exit of pipelets to record a custom error code. Note that this only works for pipelets that can experience only one error situation. If it is required to return an error code from the pipelet, you should declare an output parameter `ErrorCode` and return a human-readable error name.

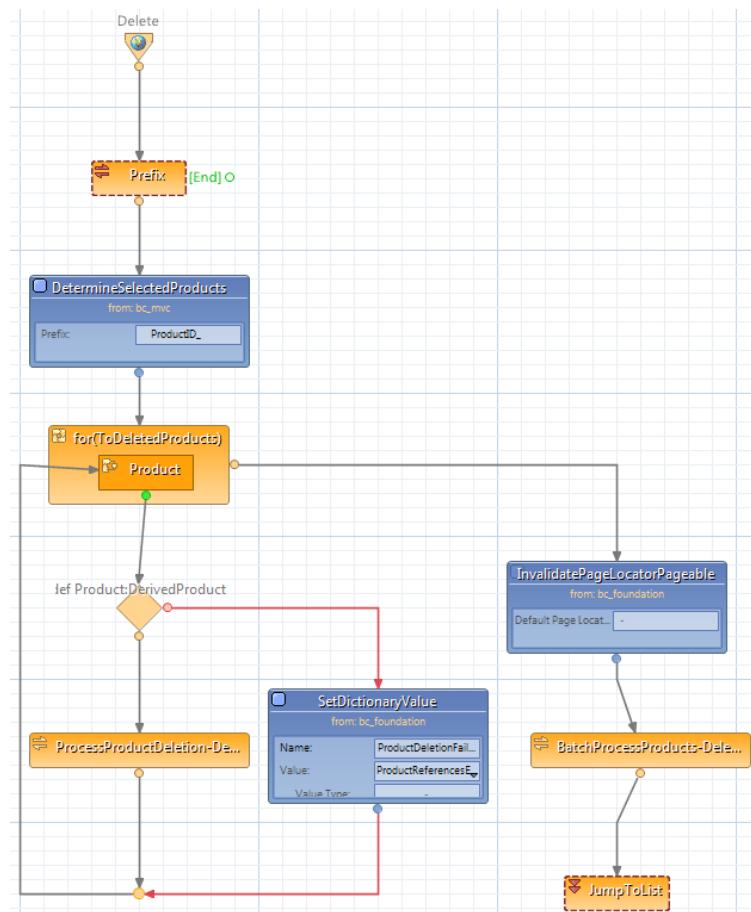
Pipeline Development

Pipelines

What Are Pipelines?

One of the most important Intershop 7 concepts is the concept of a *pipeline*. Pipelines connect pipelets to model complex business processes, and they call appropriate templates to turn the results of a business process into a response that is sent back to the client.

Figure 81. A Sample Pipeline



Every Intershop 7 storefront or Intershop 7 back office request triggers the execution of a pipeline. The requested URL identifies which pipeline needs to be called to serve the request. For example, the URL

```
http://<host>/INTERSHOP/web/WFS/<Site>/en/-/USD/ ...
    AddToBasket-Add?UserID=123&ProductID=456
```

will trigger the pipeline AddToBasket-Add.

Pipeline Elements

A pipeline consists of several different elements:

■ Pipelet Nodes

Trigger the execution of the referenced pipelet by calling the `execute()` method of the associated pipelet class.

■ Control Nodes

Modify the execution path of a pipeline, for example, in creating loops, in calling other pipelines, or in creating alternative pipeline branches whose execution depends on certain conditions. In addition, control nodes include the start nodes (indicating where the execution path starts) and the end nodes.

■ Interaction Nodes

Generate responses to requests or interact with the client. The interaction nodes call templates to generate the response sent back to the client.

■ Transitions

Connect pipelet, control, and interaction nodes.

Each pipeline defines a set of nodes that are processed once the pipeline is called. Start node and transitions determine the execution path of a pipeline, i.e. the order in which the pipeline nodes are processed.

A pipeline may define multiple alternative execution paths, each beginning with a start node that has a distinct name.

Public and Private Pipelines

Intershop 7 distinguishes public and private pipelines, depending on how a pipeline can be triggered.

- Public pipelines can be triggered by an external source, using a HTTP request. For example, presentation pipelines (see *Pipeline Types*) are typically public. Note that pipelines triggered from a template via the ISML functions `url()` or `urlex()` also need to be public. Intershop Studio marks the start node of public pipelines using a little globe (🌐).
- Private pipelines are triggered by an internal source, i.e. by other pipelines using a call node or a jump node. For example, processing pipelines (see *Pipeline Types*) are typically private.

This distinction is defined via the call mode property on the pipeline start node.

Pipeline Dictionary

What Is the Pipeline Dictionary?

The exchange of data between pipeline elements and between pipelines is enabled by the pipeline dictionary. The pipeline dictionary is a dynamic list of key-value pairs of data. Pipeline elements can access the data contained in the pipeline dictionary, modify this data, or put additional data into the pipeline dictionary.

- Pipelets typically read data stored in the pipeline dictionary under a certain key, perform some operations using the data, and finally write the result of the operation back into the pipeline dictionary. The dictionary properties of a pipelet determine which data the pipelet reads from the pipelet dictionary, and which data the pipeline writes back. See *Set Pipelet Properties* for details.
- Control nodes of a pipeline (e.g., decision nodes) often use data stored in the pipeline dictionary to decide on the execution path of a pipeline.
- If an interaction node is reached, the templates that are referenced by the interaction node can use the data in the pipeline dictionary to prepare the response sent back to the client.

What Does the Pipeline Dictionary Contain?

The pipeline dictionary set up for public pipelines initially contains:

- Parameter data passed with the HTTP request, e.g., as part of the URL or via method POST.
- General data put into the pipeline dictionary by the request handler, including `CurrentDomain`, `CurrentRequest`, `CurrentSession`, and `CurrentUser`.

The content of the pipeline dictionary set up for private pipelines, i.e., sub-pipelines called via a call or jump node, depends on whether the called sub-pipeline declares a pipeline API (see *Pipeline Parameters and Pipeline API*).

- If the called pipeline does not declare a pipeline API, only a single pipeline dictionary instance is used which is shared from the calling pipeline to the called pipeline.
- If the called pipeline declares a pipeline API, a new, local dictionary instance is set up for the called pipeline, including the general data provided by the request handler, and all dictionary objects defined as parameter on the start node of the called pipeline.

Pipeline Types

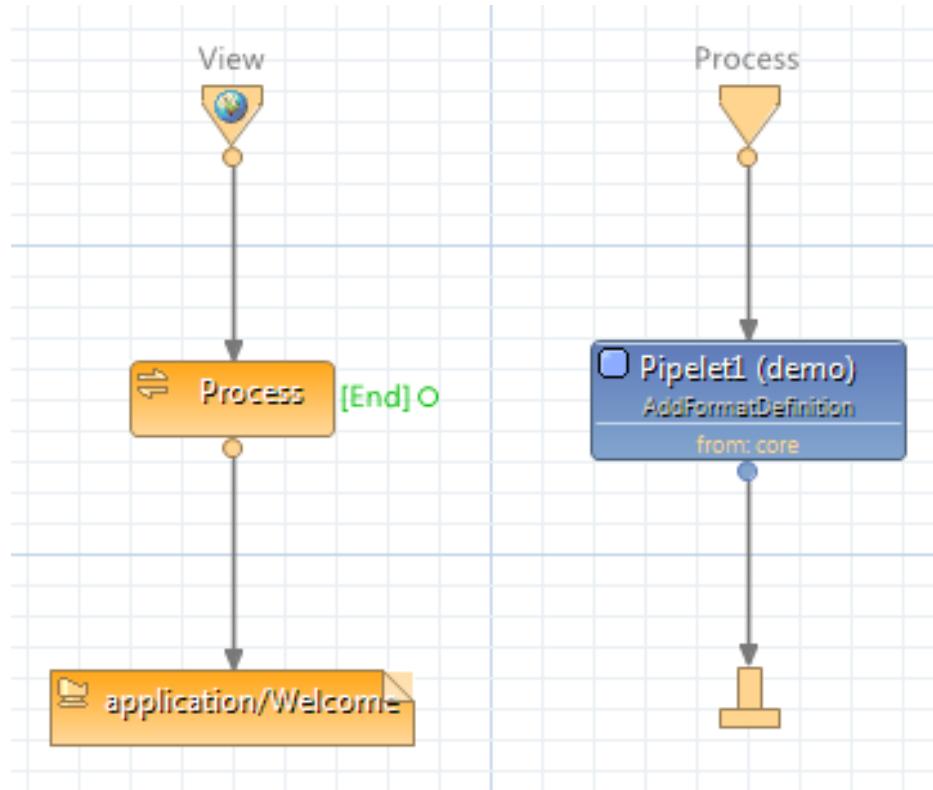
Intershop 7 distinguishes three major pipeline types: view pipelines, processing pipelines, and backoffice pipelines.

View pipelines typically start with a start node, which may be public. They are responsible for the presentation component of a request.

Process pipelines typically start with a private start node. They perform the actual business functions, such as reading data, performing calculations on these data, and updating the data source.

Storefront requests typically target a view pipeline which then calls one or more process pipelines using call nodes. Once the process pipelines are finished, they return to the view pipeline which eventually processes an appropriate template to present results generated by the process sub-pipelines.

Figure 82. View and Process Pipelines



In contrast to view pipelines and process pipelines, *backoffice pipelines* perform administrative functionality. For example, backoffice pipelines perform import and export and initiate processes at a scheduled time or as a recurring event. Additional pipeline types distinguished by Intershop Studio are webservice pipelines and job pipelines.

Depending on the chosen type of pipeline a pipeline has different settings. For example, a process pipeline does not support public start nodes or interaction nodes.

Pipeline Execution

Pipeline execution is initiated by the request handler servlet. The request handler servlet parses the URL, identifies which pipeline needs to be executed to serve the request and sets up the pipeline processor.

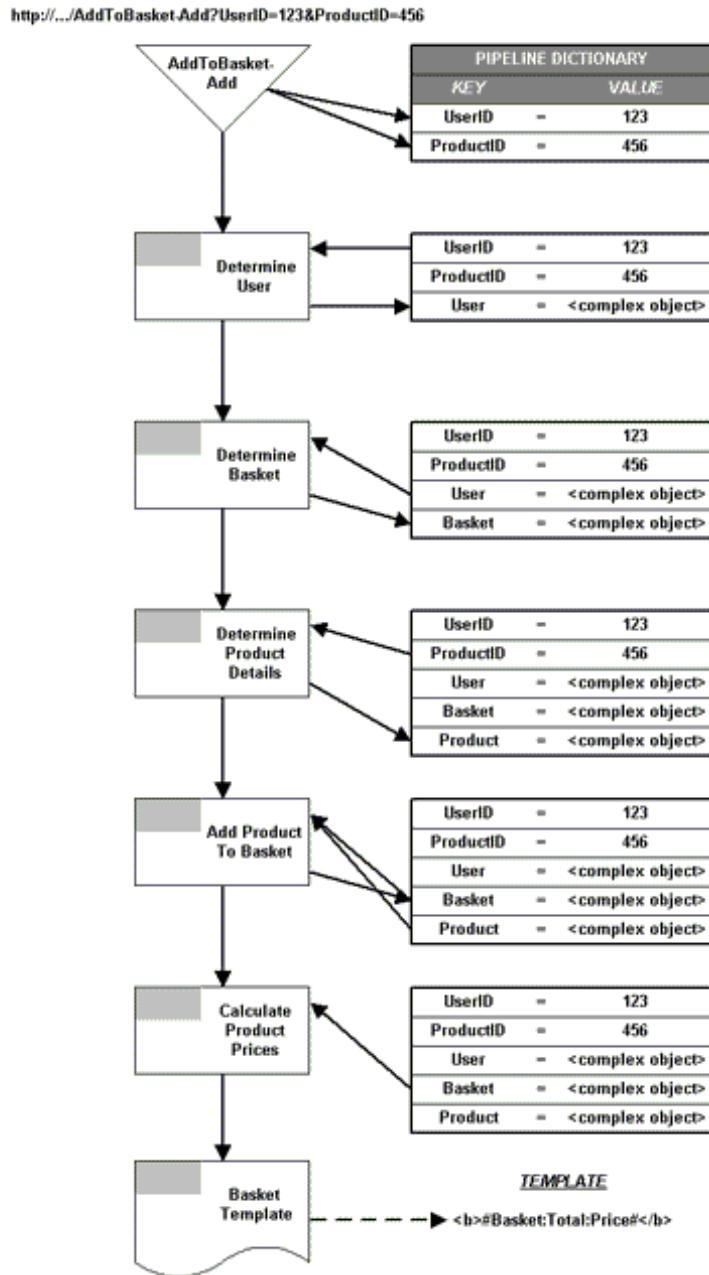
The pipeline processor checks whether the requested pipeline is loaded, loading it if necessary, and it prepares the environment for pipeline execution. Before initiating the execution of individual pipeline nodes, the pipeline processor sets up the pipeline dictionary. Initially, the pipeline dictionary stores data provided with the request, for example form data submitted through a POST method.

The pipeline processor manages the flow of the pipeline elements through the pipeline. The processor manages the transitions and returns the response back

to the request handler, which in turn relays the response to the Web adapter. The Web adapter, using the Web server, returns the response to the user.

To understand the way a pipeline is processed, consider the figure below, which illustrates the steps involved in processing a hypothetical AddToBasket pipeline, highlighting the exchange of data between pipelets and the pipeline dictionary.

Figure 83. Processing a Pipeline



The URL triggering this pipeline is:

```
http://<host>/INTERSHOP/web/WFS/<Site>/en/-/USD/ ...
AddToBasket-Add?UserID=123&ProductID=456
```

Assuming that the request handler servlet has appropriately analyzed the request (identifying which pipeline on which server needs to be triggered) the essential components of pipeline behavior, as shown in the figure above, are as follows:

■ **Pipeline Processor Checks Whether the Pipeline is Loaded**

Note that a pipeline is loaded only once, even if concurrently executed by multiple clients.

■ **Pipeline Processor Initializes the Pipeline Dictionary**

Any parameters passed to the start of the pipeline by the request are stored as key-value pairs in the pipeline dictionary. The parameter name is the "key" and has an associated "value". The data are maintained in the dictionary until the pipeline ends and clears the dictionary.

■ **Pipeline Processor Executes Each Pipelet and Flow Control Action in Sequential Order**

Pipelets are executed by calling the `execute()` method of each pipelet class, passing the pipeline dictionary as argument. Each pipelet may require a key-value pair (or pairs) to be available in the pipeline dictionary. The pipelet checks the pipeline dictionary; if it finds the necessary key(s), it executes its business function. As a result of executing, the pipelet may add other key-value pairs to the pipeline dictionary. The dictionary cumulatively stores all key-value pairs from all preceding pipelet activity until it reaches an end-and-clear point. The dictionary is passed to any sub-pipeline called, and is returned to the calling pipeline after the sub-pipeline has been executed.

■ **Pipeline Processes the Interaction End Node, Calling a Template**

Using data stored in the pipeline dictionary, a response is generated from the template (an HTML page) to be sent back to the client. Serving a request may include more than one pipeline. Due to the separation of presentation and processing pipelines, it typically includes sub-pipelines which are called into a process with a call node. Note also that many pipelines, in particular processing pipelines, do not terminate in an interaction node.

Pipeline Loading

Before a pipeline can be executed, the pipeline needs to be loaded by the pipeline processor (see *Pipeline Execution*). Whenever pipelines are loaded by the pipeline processor, one pipelet instance is created for each pipelet occurrence in the pipeline. If the same pipelet is added multiple times to the pipeline, multiple instances are created. Therefore, once a pipeline is loaded, there are as many pipelet instances as there are pipelet nodes in the pipeline. However, when the system receives multiple concurrent request to execute one and the same pipeline, it does not reload the pipeline and create even more pipelet instances. A pipeline is loaded only once.

NOTE: The pipeline loading mechanism has important consequences for variable handling in pipelets. See here for details.

In order to increase performance in production system, it is possible to preload all pipelines and pipelets on application server startup. Pipeline and pipelet preloading can be triggered for individual cartridges and sites, or for all cartridges and sites in a system. In order to preload pipelines and pipelets, adjust the settings for the properties below within the `is_home/share/system/config/cluster/`

`appserver.properties` file. With the settings below, pipelines and pipelets from all cartridges and sites are preloaded on application server startup.

```
intershop.pipelines.PreloadFromCartridges=all  
intershop.pipelines.PreloadFromSites=all  
intershop.pipelets.PreloadFromCartridges=all
```

In case a pipelet cannot be instantiated properly when preloading a pipeline, for example, because the pipelet descriptor does not match up with the pipelet, or because the pipelet is not found, the system can react in two different ways:

- A placeholder pipelet is added to replace the missing pipelet. Depending on the function of the missing pipelet, the pipeline may still work. This may be useful for development purposes or in controlled environments. Note that missing pipelets may lead to erroneous and unpredictable pipeline behavior.
- A `PipelineInitializationException` is thrown when trying to load the pipeline, and the pipeline cannot be executed. The exception is written to the error log file. Internally, the system marks the pipeline as "incompletely loaded" and tries to reload the pipeline when the pipeline is requested again.

Adjust the following property within the `is_home/share/system/config/cluster/appserver.properties` file to determine whether a placeholder should be inserted (value "PlaceHolder") or an exception be thrown (value "Exception"):

```
intershop.pipelets.OnLoadError
```

Internal Pipeline Representation

Internally, a pipeline is represented by a set of XML files which describe the pipeline parameters and store localizable descriptions, respectively. These files are:

■ Pipeline Parameter File

The pipeline parameter file lists all pipelet, control, and interaction nodes, and the transitions connecting pipeline nodes. When calling a pipeline, the pipeline processor parses this file to determine the nodes that have to be processed and their relative order.

■ Pipeline Descriptor File

The pipeline description is stored in a separate XML file in order to facilitate localization. There is one pipeline description file for each locale.

As a developer, you do not have to worry about the specifics of these files. When developing new pipelines in the Intershop Studio Pipeline Editor, these files are automatically maintained, without any intervention necessary.

Pipeline Overloading and Pipeline Inheritance

Overloading vs. Inheritance

Custom projects typically need to modify Intershop 7 standard pipelines. Sometimes, these modifications affect only a single start node of the pipeline. For example, a custom project may simply need to add a pipelet to an existing pipeline performing some additional operations. In other cases, the modifications might be

more complex, or even require to replace the current pipeline components with entirely different ones.

Intershop 7 provides two alternative means to customize existing pipelines, depending on how extensive the customizations are:

■ Pipeline Overloading

A custom project can provide a new version of a pipeline which completely replaces the existing pipeline. The custom pipeline effectively re-defines all start nodes of the existing pipelines. Pipeline overloading should be used in case of extensive modifications to a pipeline.

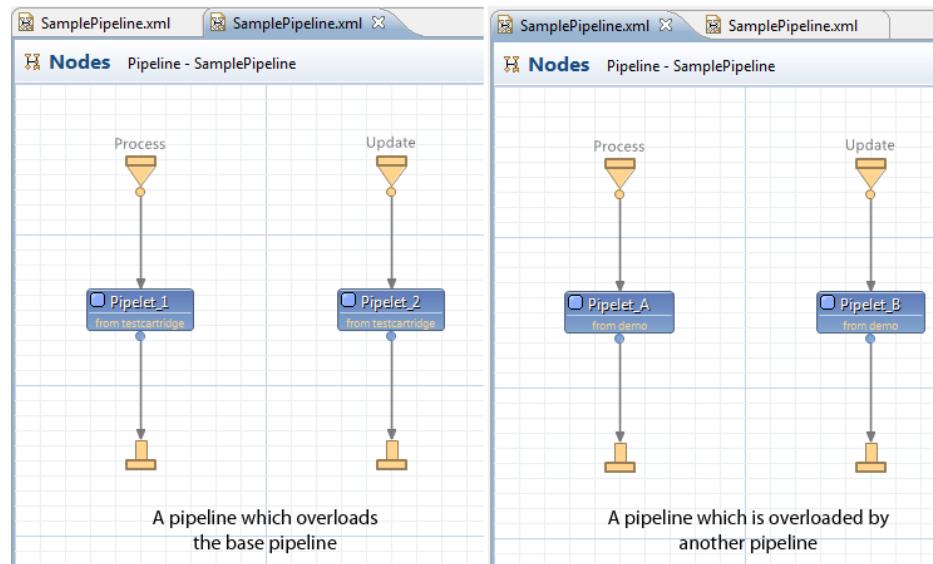
■ Pipeline Inheritance

In this case, a custom pipeline replaces only individual start nodes of an existing pipeline, while all other start nodes are inherited from the existing pipeline (also referred to as parent pipeline). Pipeline inheritance should be used in case the modifications are minor and affect individual start nodes only.

NOTE: Both pipeline overloading and pipeline inheritance are always dependent on the context of a site, i.e. dependent on the cartridges which are bound to a site, and the hierarchy of cartridges in the context of a site. For a custom pipeline to overload or inherit from an existing pipeline, the custom cartridge containing the new pipeline has to be ranked higher in the cartridge hierarchy than the cartridge containing the existing pipeline. This also implies that a custom pipeline may overload or inherit from an existing pipeline in the context of site A, but not in the context of site B. See also here on this topic.

To illustrate the difference, consider the two simple examples below. The first figure illustrates the effect of pipeline overloading. Both pipelines have the same name and identical start node sets. The pipeline shown left overloads the pipeline shown right, hence this is the pipeline executed when the `SamplePipeline` is called.

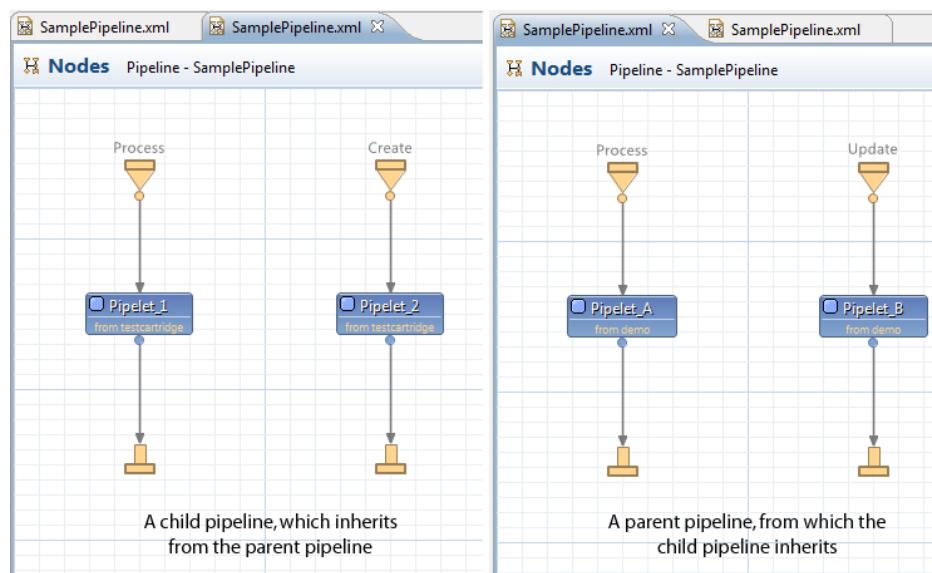
Figure 84. Pipeline overloading



In the second sample, the pipeline shown right is the parent pipeline. It serves as basis for inheritance. The pipeline shown left inherits from this pipeline. Note that the start node sets of both pipelines differ. The child node overloads the sub-pipeline with start node `Create`. It provides a new start node `Create`, and inherits

start node Update from its parent pipeline. The only start node which never gets executed is the parent pipeline's Process start node.

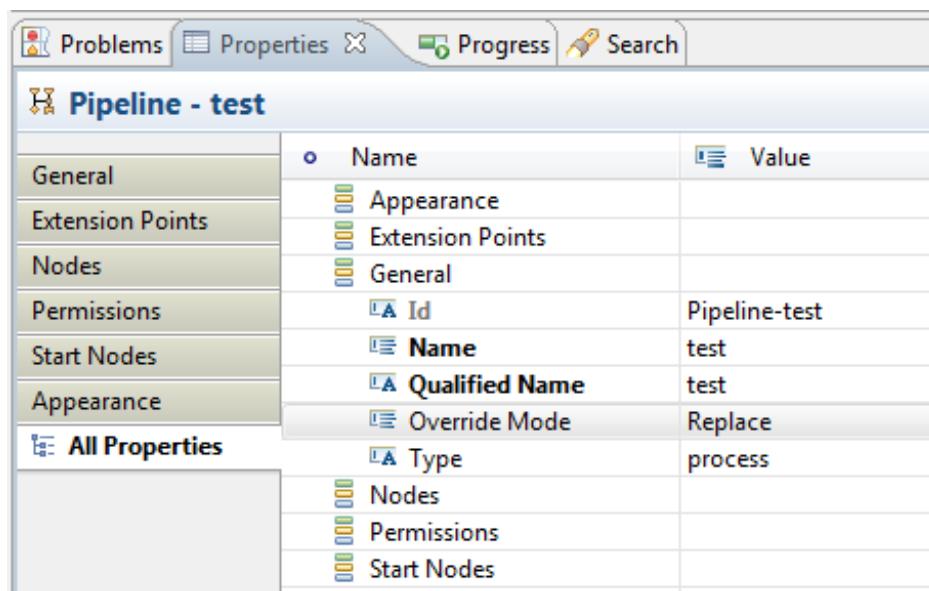
Figure 85. Pipeline inheritance



Defining an Overload Relation

For a custom pipeline to be able to overload an existing pipeline, the following requirements have to be met:

- The custom pipeline and all its start nodes must have the same name.
- To get started, it is recommended to copy the entire pipeline into the custom cartridge and then modify the start nodes as required. This makes sure that the same start nodes are used, and that the custom cartridge provides the same set of start node compared to the existing cartridge.
- The pipeline property "Override Mode" has to be set to Replace (which is the default value when creating pipelines).

Figure 86. Defining pipeline overload

Pipeline Overloading: Basic Properties

The overloading pipeline can define additional start nodes not present in the overloaded base pipeline. Likewise, the overloading pipeline does not necessarily have to re-implement all start nodes. Note, however, that start nodes which are not re-implemented are not visible anymore. Inconsistencies arising from missing start nodes are not detected automatically. Hence, it is up to the developer to take care of this issues.

A reimplemented start node can have a different call mode (public or private, see *Pipeline Types*) compared to the corresponding start node of the base pipeline.

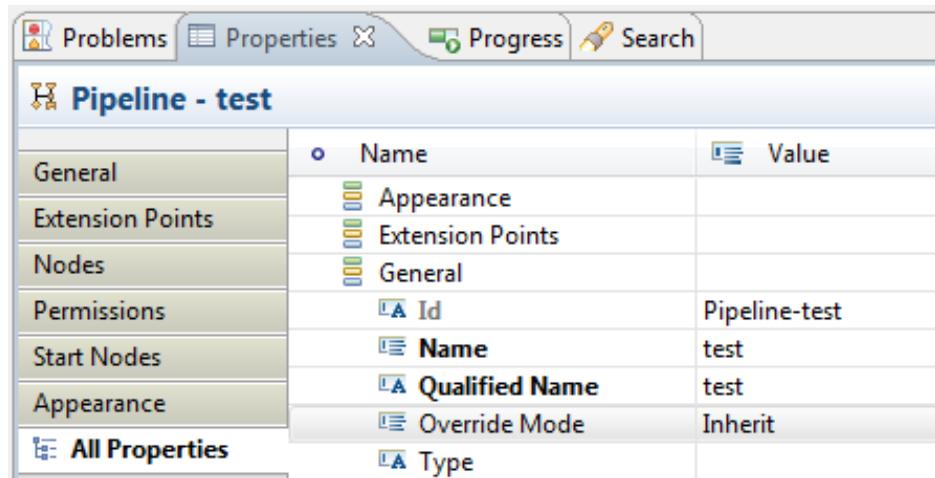
A reimplemented start node can terminate differently (e.g., end node vs. interaction end node) compared to the corresponding start node of the base pipeline.

A reimplemented start node can declare a different pipeline API (see *Pipeline Parameters and Pipeline API*). It is up to the developer to make sure that no inconsistencies arise.

Defining a Pipeline Inheritance Relation

For a custom pipeline (child pipeline) to inherit from an existing pipeline (parent pipeline), the following requirements have to be met:

- The child pipeline must have the same name.
- The start nodes to replace (i.e. overload) must have the same name as the respective start nodes in the parent pipeline.
- The pipeline property "Override Mode" has to be set to Inherit.

Figure 87. Defining pipeline inheritance

Pipeline Inheritance: Basic Properties

- A pipeline can have only a single parent pipeline from which it inherits. Inheritance from multiple pipelines in parallel is not possible.
- Pipeline inheritance relations are transitive. If, for example, a pipeline ViewProduct in a cartridge project inherits from the ViewProduct pipeline in the cartridge sub, which itself inherits from the ViewProduct pipeline in the cartridge base, then the pipeline ViewProduct in the cartridge project inherits from the ViewProduct pipeline in the cartridge base as well.
- A pipeline inherits all start nodes from its parent pipelines.
- A start node can reimplement any start node with the same name that exists in the inheritance path. It is not limited to start nodes of the direct parent. To overwrite a start node, simply create a start node with the same name in the custom pipeline. It is not necessary to declare explicitly at a start node that it is derived from another start node.
- A reimplemented start node can have a different call mode (public or private, see *Pipeline Types*) compared to the corresponding start node of the parent pipeline.
- A reimplemented start node can terminate differently (e.g., end node vs. interaction end node) compared to the corresponding start node of the parent pipeline.
- A reimplemented start node can declare a different pipeline API (see *Pipeline Parameters and Pipeline API*). It is up to the developer to make sure that no inconsistencies arise.

Pipeline Inheritance and Call/Jump Node Resolution

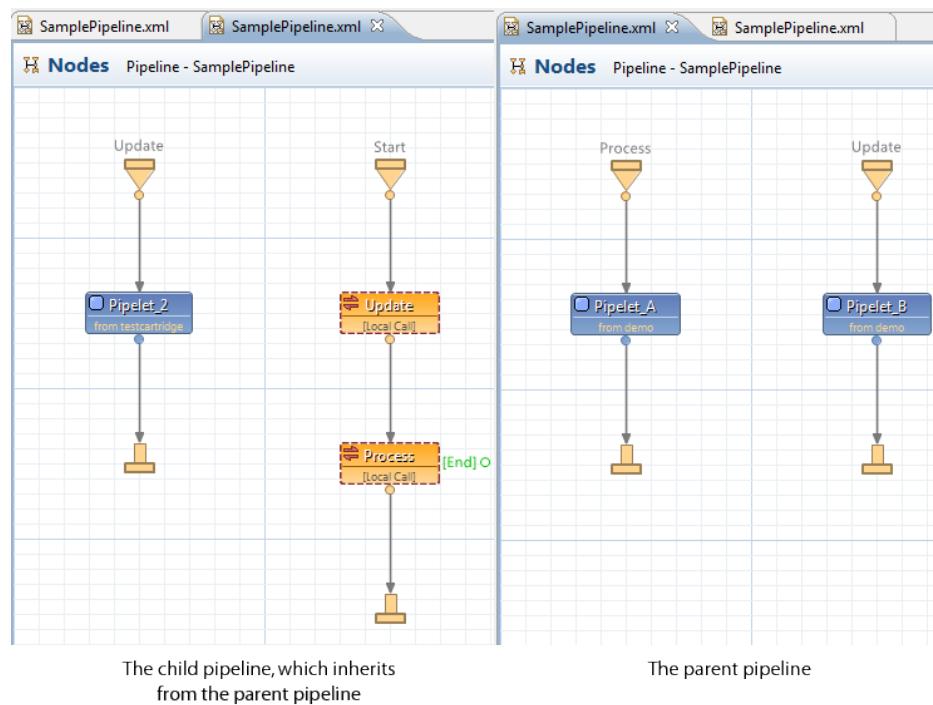
In complex projects, pipelines are typically interconnected via jump and call nodes (see *Ending Pipelines With a Jump* and *Calling a Sub-Pipeline*). When working with pipeline inheritance, special attention must be paid to the lookup path via which

the start node referenced by a call or jump node is resolved. Intershop 7 uses the following principles to resolve start nodes:

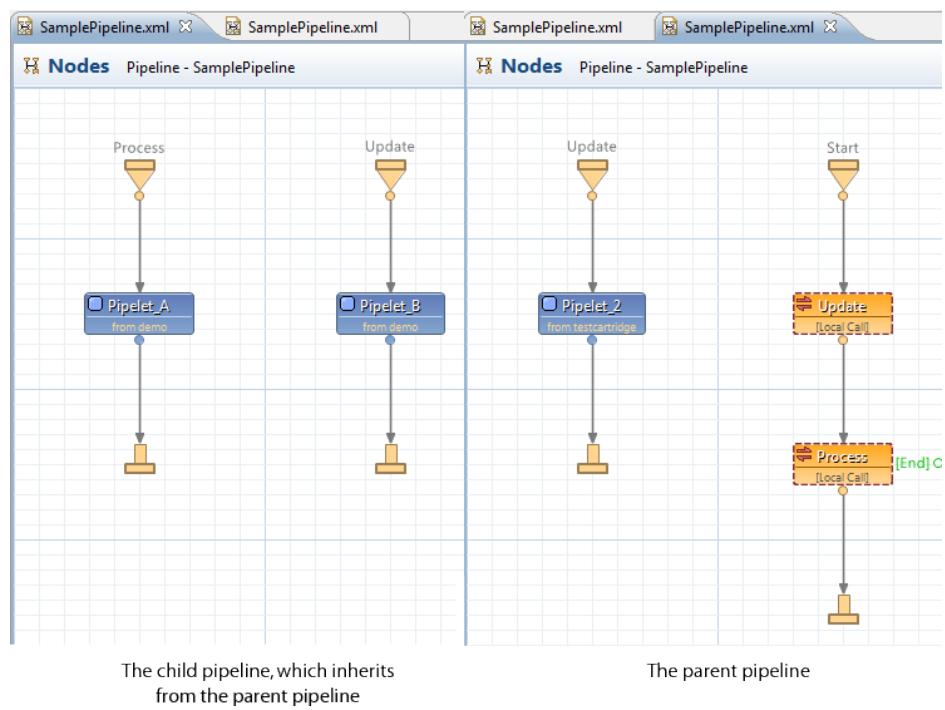
- When resolving a start node, Intershop 7 first checks whether the respective pipeline which is highest in the hierarchy.
- If this pipeline does not contain the start node in question, Intershop 7 checks whether the pipeline defines an inheritance relation. If it does, the referenced parent pipeline is checked for the start node.
- This continues until the start node has been found. If no start node is accessible, an exception is thrown.

Consider the simple example below. The child pipeline (left) reimplements the start node "Update" of the parent pipeline. The start node "Start", defined within the child pipeline, invokes two call nodes. The first call node references start node "Update", which resolves to the respective start node defined in the child pipeline. On the other hand, start node "Process", which is referenced by the second call node, resolves via the parent pipeline.

Figure 88. A child pipeline calling start nodes partially resolved by its parent pipeline

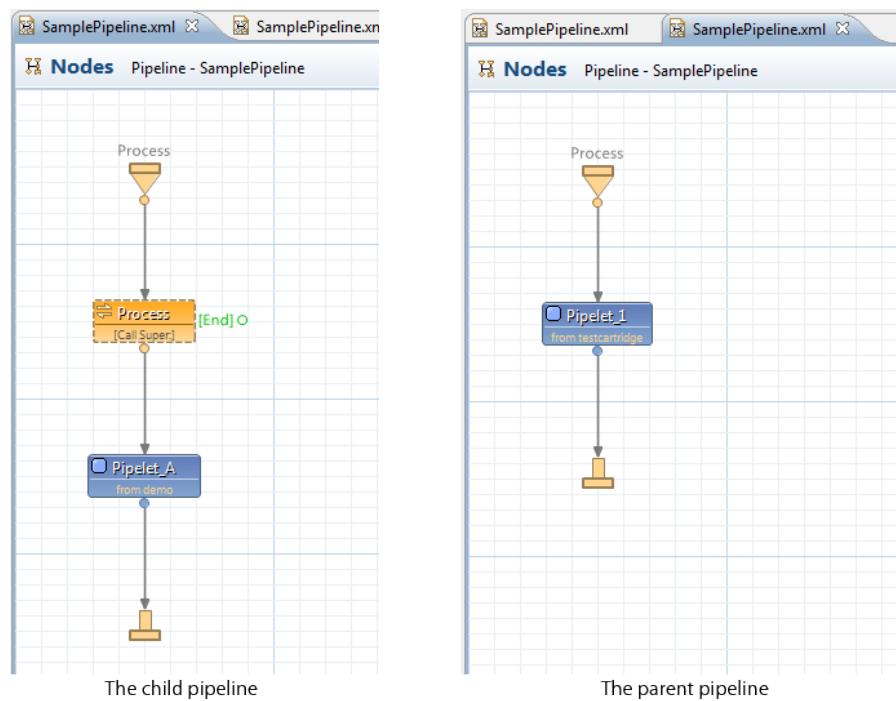


In the next example (see figure below), the scenario has been switched. It is now the parent pipeline (right) which contains a start node ("Start") invoking two call nodes. The important thing to note here is that both call nodes reference start nodes ("Update", "Process") which are resolved not by the parent pipeline, but by the child pipeline.

Figure 89. A parent pipeline calling start nodes resolved by a child pipeline

The "Super:" Operator

The "Super:" operator is used to point to a start node of the current pipeline's parent pipeline. Typical use cases are scenarios in which a standard pipeline needs to be extended with additional operations to be performed. In the example below, the child pipeline reimplements a start node of the parent pipeline ("Process"). The reimplemented start node executes the original start node (i.e., the "Process" start node defined by the parent pipeline), and then executes an additional pipelet.

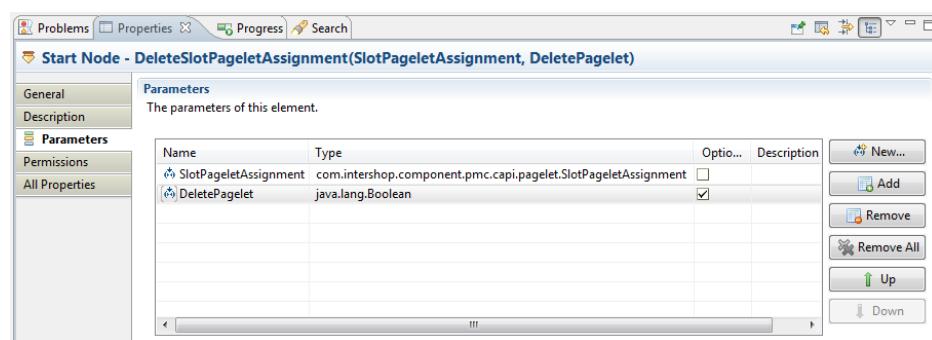
Figure 90. A child pipeline calling a start node from the parent pipeline

Pipeline Parameters and Pipeline API

What Is the Pipeline API?

Each pipeline typically expects certain parameters to be in the pipeline dictionary which it needs for successful execution. Likewise, processing pipelines, terminating with an end node, can be expected to return certain data. The term pipeline API refers to the parameter definitions which describe which data the pipeline expects, and which data the pipeline returns.

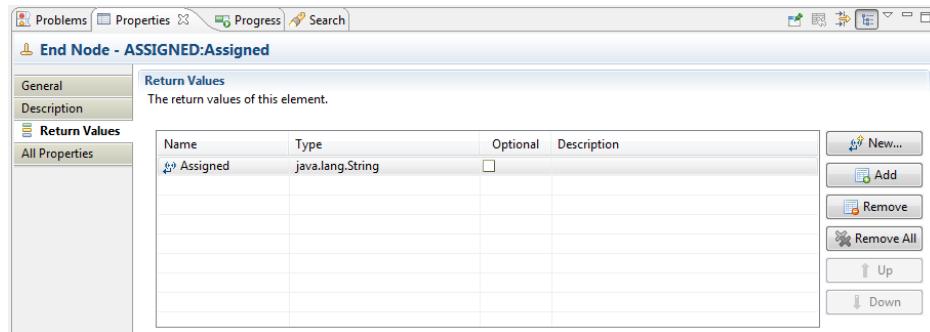
Parameters which the pipeline expects to find in the pipeline dictionary are defined on the pipeline's start node. For example, the start node DeleteSlotPageletAssignment of pipeline ProcessPagelet defines that the pipeline expects the parameters SlotPageletAssignment and DeletePagelet. For each parameter, the definition states the dictionary key (name), the data type and whether the parameter is optional or mandatory.

Figure 91. Pipeline parameter definition on the start node

If a pipeline contains an end node, the end node defines which parameters the pipeline returns to the pipeline dictionary of the calling pipeline. For example, the end node of pipeline parameters which the pipeline provides are defined on the end node of a pipeline ProcessPagelet defines that the pipeline provides the parameter Assigned. For each parameter, the definition states the dictionary key (name), the data type and whether the parameter is guaranteed by the pipeline (i.e. always returned) or whether it is optional.

NOTE: If a pipeline defines multiple end nodes, make sure that each end node returns the same set of parameters.

Figure 92. Pipeline parameter definition on the end node



NOTE: The pipeline API takes effect for sub-pipelines only, i.e. for pipelines called by other pipelines. The pipeline API and associated mechanisms (e.g. for pipeline dictionary processing and aliasing, see below) do not take effect for pipelines triggered by an HTTP request, regardless of whether the property Strict on the respective start node has been set to true or not.

Pipeline API and Pipeline Dictionary Processing

For a pipeline with pipeline API defined (i.e., with property Strict set to true on the start node), a new, local pipeline dictionary instance is created. The parameters defined on the pipeline start node determine which dictionary objects will be copied from the dictionary of the calling pipeline to the local dictionary of the target pipeline. The local pipeline dictionary contains:

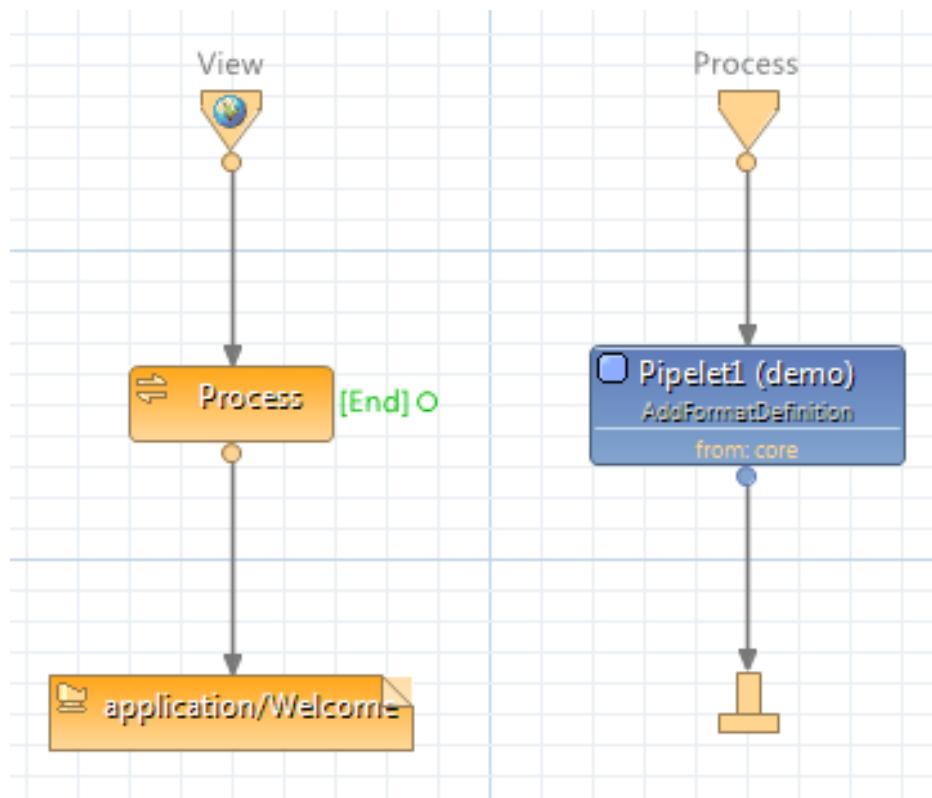
- the dictionary objects declared at the pipeline start node
- generic dictionary objects put into the pipeline dictionary by the RequestMgr, including CurrentDomain, CurrentRequest, CurrentSession, and CurrentUser.

If the calling pipeline does not provide the dictionary objects defined by the pipeline API of the called sub-pipeline, the request is aborted. Note that this is in contrast to pipeline behavior in earlier releases of Intershop 7, in which a single pipeline dictionary instance has been used by a pipeline and all its sub-pipelines.

NOTE: Keep in mind that evaluation of start parameters when setting up the pipeline dictionary is effective only for sub-pipelines. A pipeline triggered via HTTP request starts executing with a dictionary containing all parameters passed through the HTTP request, ignoring any parameter definitions on the pipeline start node.

In a similar manner, the return values defined on the end node of a sub-pipeline determine which objects will be copied back from the pipeline dictionary of the sub-pipeline into the pipeline dictionary of the calling pipeline.

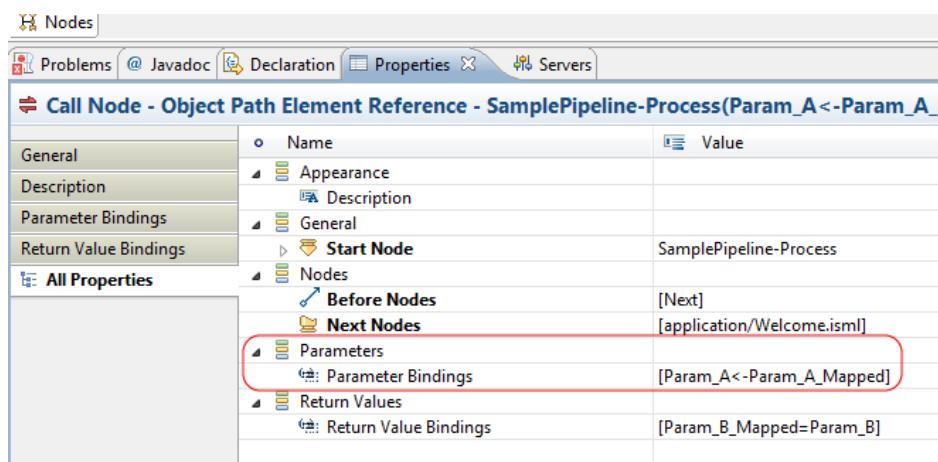
As an example, consider the simple pipeline below.

Figure 93. A simple test pipeline

Assume that the API of sub-pipeline TestPipeline-Process defines parameter Param_A on its start node, and the parameter Param_B on its end node. Hence, when executing TestPipeline-Process, the pipeline dictionary will contain Param_A (but no other dictionary object which TestPipeline-Process may produce). When TestPipeline-Process has finished, Param_B will be copied back into the dictionary of the calling pipeline TestPipeline-View.

Aliasing

On call or jump nodes, alias definitions can be defined which are evaluated when setting up the local pipeline dictionary for the called pipeline, or when returning to the pipeline dictionary of the calling pipeline. Aliasing makes it possible to change the key of a dictionary object when copying it to the pipeline dictionary of a sub-pipeline, or back to the pipeline dictionary of the calling pipeline. For example, if TestPipeline-View provides Param_A for TestPipeline-Process, but the start node of TestPipeline-Process expects this object under key Param_A_Mapped, the simple alias definition shown below will do.

Figure 94. Call node definitions: aliasing

NOTE: A pipeline which itself does not expose a pipeline API may nevertheless provide alias definitions, to be evaluated in case a sub-pipeline has a pipeline API defined.

Overloading the Pipeline API

As discussed in *Pipeline Overloading and Pipeline Inheritance*, existing Intershop 7 cartridges can be replaced by customized pipeline versions using the overload mechanism.

For example, the Intershop 7 demo cartridge provides the pipeline `IncludeProductDetails` which overloads a pipeline with the same name contained in cartridge `sld_ch_consumer_app`. Since the demo cartridge is ranked higher than `sld_ch_consumer_app` in the hierarchy of cartridges bound to the site `PrimeTech-PrimeTechSpecials-Site`, requests invoking pipeline `IncludeProductDetails` in the context of site `PrimeTech-PrimeTechSpecials-Site` will always trigger the project-specific pipeline version (see also *Adding Cartridges To Apps*).

When overloading a pipeline which has a pipeline API defined, make sure that your custom pipeline does not break the API of the existing pipeline. For example, if the API of the custom pipeline defines additional parameters, errors may occur because the pipeline which you overload may be called by other pipelines as well which do not provide these additional parameters.

Using Pipelines With and Without Pipeline API

The pipeline API feature has been introduced with a previous version of Intershop 7. While it is recommended to use the pipeline API with future projects, it is still possible to run legacy pipelines without pipeline API defined. To determine whether a pipeline uses the pipeline API or not, the system simply checks whether the property `Strict` on the respective start node is set to true or false.

- If property `Strict` is set to true on the start node, the system assumes that a pipeline API has been defined. Hence the system evaluates start parameters, creates a local pipeline dictionary containing only the dictionary objects defined as start parameters, and resolves alias definitions. Note that in this case, it is crucial for the pipeline to terminate in an end node on which the property `Strict` is set to true as well. If property `Strict` is set to false on the end node, the pipeline dictionary will be discarded and not returned to the calling pipeline.

- If property `Strict` is set to `false` on the start node, the system assumes that the pipeline does not define a pipeline API. Start parameters will not be checked, the entire pipeline dictionary is passed to the sub-pipeline, and alias definitions will be ignored. Note that in this case, the setting for property `Strict` on the end node of the pipeline is irrelevant. The calling pipeline will receive the complete pipeline dictionary, regardless of whether return values are defined on the end node or not.

Pipeline Debugger

The pipeline debugger allows the developer to track the execution of a pipeline step-by-step in Intershop Studio in order to locate errors precisely.

Using the Pipeline Debugger, developers can easily check the storefront behavior of pipelines, track specific pipeline sections, or monitor the status of the pipeline dictionary at each step using a special watch window. The pipeline debugger, unlike standard programming language debugger, operates on the pipeline rather than the source code level.

Most often, the pipeline under scrutiny is triggered by an action in the storefront. It is also possible to trigger a pipeline directly, using the browser URL input box. However, such an isolated action may leave out certain dictionary keys that the pipeline expects and is not the recommended method.

The debugging process is fairly straightforward: you set up the storefront situation you want to check, open the appropriate pipeline in the Visual Pipeline Manager, set breakpoints, trigger an action, and then step through the pipeline in Intershop Studio observing its behavior. The process allows you to check both the path taken by the pipeline (which jumps, calls, and error branches are used, for example) and to monitor the status of the pipeline dictionary in the debugger watch window.

Creating Pipelines

Pipelines are designed in the Pipeline Editor. When creating a new pipeline, a simple wizard helps to collect required information such as the pipeline name and the cartridge to which the pipeline belongs.

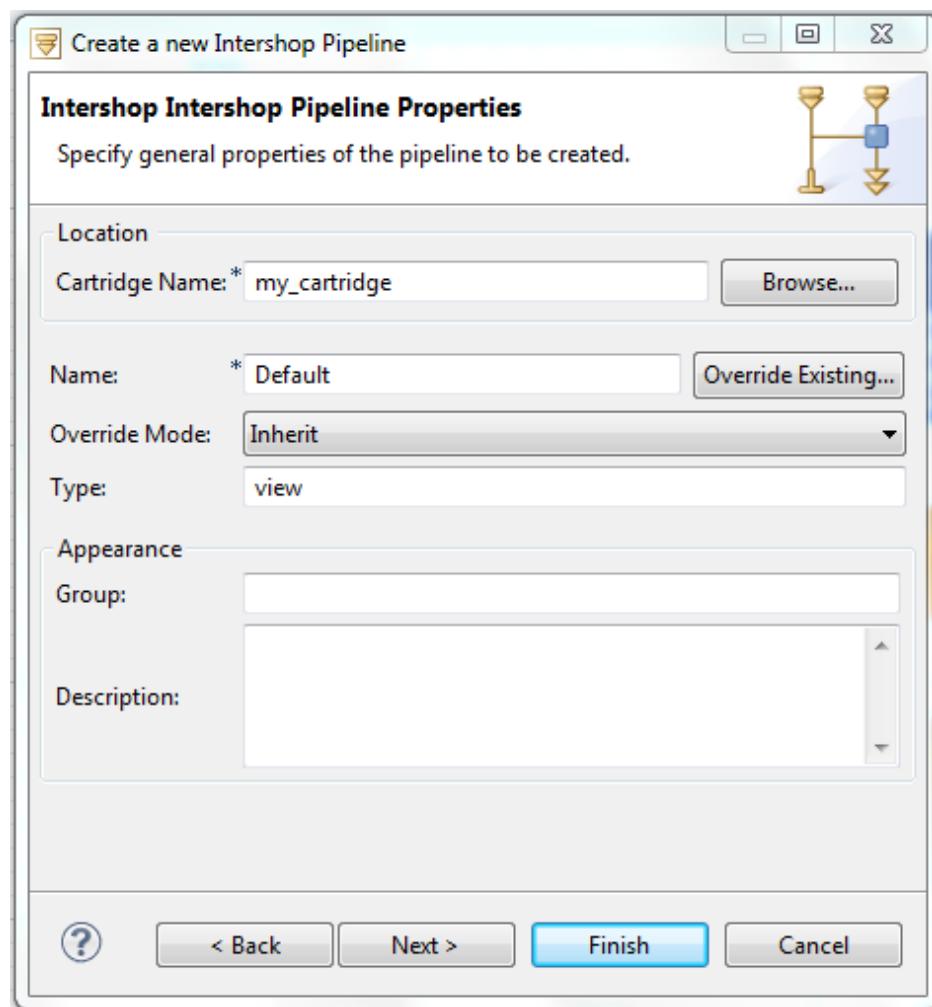
Start the Pipeline Wizard

To start the pipeline wizard:

1. **In the Cartridge Explorer, right-click the cartridge project for the new pipeline to open the context menu.**
2. **From the context menu, select New | Other.**
Select a Wizard dialog is displayed.
3. **Select Pipeline and confirm with Next.**
The pipeline wizard is displayed.

Set General Pipeline Properties

The pipeline wizard allows you to set general pipeline properties as described in the table below:

Figure 95. General Pipeline Properties**Table 51. General Pipeline Properties**

Property	Description
Cartridge Name	This field displays the preselected cartridge. If the pipeline is to be created in another cartridge, the cartridge can be specified here.
Name	Defines the pipeline name used for display purposes, for example, in the Pipeline Editor and the Pipeline View.
Description	Specifies description information for a pipeline. The pipelet description can be localized.
Group	Assigns the pipeline to a pipeline group. Pipelet groups are used in the Pipeline Editor and the Pipeline View to facilitate pipeline handling.
Pipeline Type	Assigns the pipeline to a pipeline type (view, process, backoffice, job, extension, include, monitor, pagelet, pageletslot or webservice). Intershop Studio uses this information when checking a pipeline for possible problems. For example, a view pipeline should have a public start node and end in an interaction end node. A process pipeline, on the other hand, should have a private start node and return to the calling pipeline. The Pipeline

Property	Description
	Editor uses different background colors to indicate process or presentation pipelines.
Override Mode	Defines the override mode in super pipeline relations. If Inherit is selected, the current pipeline is inherited from the hierarchically preceding pipeline. If Replace is selected, the preceding pipeline is overloaded.

Once all properties are defined, click Finish. This opens a new Pipeline Editor window for the new pipeline, with the pipeline properties displayed in the Properties View. Moreover, the new pipeline is now accessible both in the Cartridge Explorer and the Pipeline View.

Add Pipelets and Flow Control Elements

To add pipelets to a pipeline:

1. **Open the Pipelet View.**
2. **In the Pipelet View, select the pipelet to be added to the cartridge.**
3. **Drag the pipelet into the Pipeline Editor window.**

To add flow control elements:

1. **Select the flow control element in the Pipeline Component palette of the Pipeline Editor.**
2. **Toggle the flow control element into the Pipeline Editor palette.**
3. **Click within the graphical editor area to drop the element.**

Drag and drop pipelets and flow control elements freely to arrange them as required.

See *Pipeline Component Reference* on available flow control elements.

Add Transitions

If you place a node or pipelet elsewhere on the workspace, and later drag it into the pipeline, you must draw the transition yourself by clicking on the connector button of one node, and dragging toward the next node. Transitions always connect output with an input connector. The input connector is represented by the pipeline element itself.

To draw transitions connecting pipelets and flow control elements:

1. **Toggle Transition in the Visual Pipeline Editor's palette.**

This attaches a connector sign to the mouse arrow. An additional sign is displayed signaling that the current mouse position is illegal.

Figure 96. Illegal Connector Position



2. Move the mouse arrow to an output connector of a pipelet or flow control element.

A legal connector position has been reached if the illegal sign disappears.

Figure 97. Legal Connector Position



3. Click the left mouse button to fix the position for the output connector.

4. Drag the mouse arrow to an input connector of a pipelet or flow control element. Once a legal connector position has been reached, click the left mouse button to fix the position for the ingoing connector.

The transition is now marked by an arrow connecting the outgoing with the ingoing connector.

5. Drag the mouse arrow to a empty space and select a pipelet or flow control element from context menu.

New pipelet or flow control element is created and the transition is marked by an arrow connecting the outgoing with the ingoing connector.

Manage Pipeline Layout

Use the following features to manage the pipeline layout in the Pipeline Editor:

■ **Orientation of pipeline elements**

To change the orientation of a pipeline element, right-click the element to open the context menu. From the context menu, select Change Orientation.

■ **Add bend points to transitions**

By default, the Pipeline Editor draws a straight diagonal to mark a transition connecting two nodes. To manually convert transitions from diagonals to right-angles, left-click a line to create a bend point, place the cursor on the bend point, and drag the bend point to a chosen location. To apply bend points automatically, right-click the transition and select Layout Connections from the context menu.

■ **Remove bend points from transition**

To remove all bend points from a transition, thereby converting the transition to a straight line, right-click the transition and select Remove Bend Points from the context menu.

Manage Node and Transition Properties

Flow control elements and transitions require or offer the possibility to set certain properties. To set or modify properties for a node, select the node in the Pipeline Editor window and edit the properties in the Properties View.

- **If the value of a property is a non-localized string, enter the value in the respective row and column of the Properties View.**
- **If the value of a property is localized, click  to enter localized versions of the value.**
- **If the property has a fixed range of values, a drop-down menu appears from which the value can be selected.**

See *Pipeline Component Reference* for details on the properties to be set for each node.

Manage Pipeline Parameters

Many pipelines require a set of start parameters. Start parameters identify data that must be present in the pipeline dictionary for the pipeline to execute successfully. When calling the pipeline via an HTTP request, start parameters are encoded as part of the URL.

Start parameters are configured individually for each start node. To configure start parameters:

Pipeline parameters can be added via the pipeline's properties view. To add a pipeline parameter, select start nodes Parameters tab and click New. To delete a parameter, select the respective parameter and click Remove.

Manage Pipeline Access Control

For each pipeline, access control lists define the permissions a user needs to execute this pipeline in general, or a specific start node of this pipeline. Every cartridge that has pipelines provides a `pipelines-acl.properties` file. The `pipelines-acl.properties` file includes the access control list (ACL) for every pipeline of the cartridge whose access should be restricted. These access control lists are loaded into the pipeline cache upon server startup or pipeline reload. Intershop Studio enables you to configure access permissions for pipelines which are then added to the respective `pipelines-acl.properties` file.

You can configure:

- **Access permissions for individual start nodes**

These permissions are configured on start nodes and apply to these start nodes only.

- **Default access permissions for entire pipelines.**

These permissions are configured on pipelines. They act as derived permissions which apply to all start nodes of a pipeline.

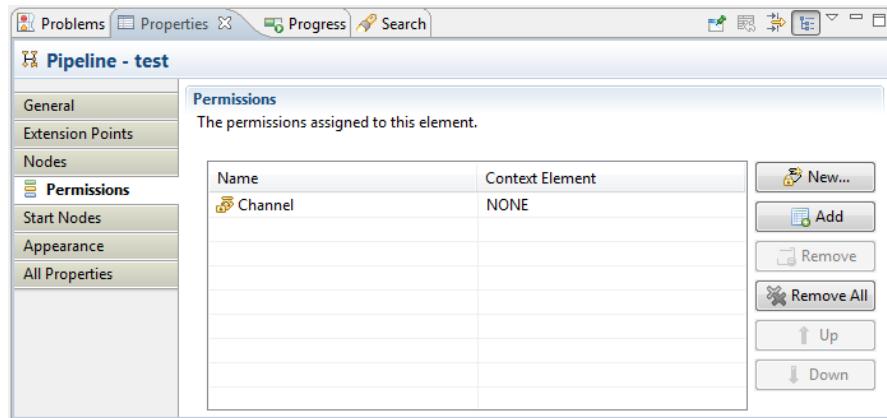
To configure default permissions:

1. **Select start node in the Visual Pipeline Editor or**

select the pipeline by clicking empty space in the Visual Pipeline Editor.

2. Select Permission tab.

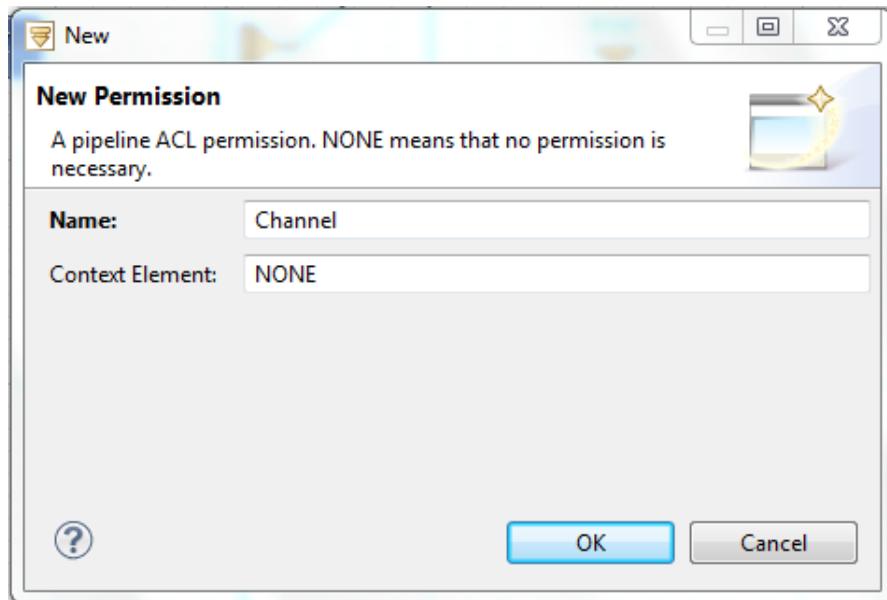
Figure 98. Pipeline Access Permissions



3. Click New.

4. Enter context object and permission ID.

Figure 99. New Permission Dialog



Access permissions configured for a pipeline can be browsed in the Cartridge Explorer. To modify context object or permission ID:

■ In the Cartridge Explorer:

Select the access permission

Access permissions can be modified through the Properties View:

Select Permissions tab in Properties View.

NOTE: Access permissions can also be added, deleted or sorted via the pipeline's properties view in the Cartridge Explorer. To add a permission, select the Permissions row and click Add. To delete a permission, select the respective permission and click Remove. To sort permission list, select a permission and click Up or Down to move the permission within the list.

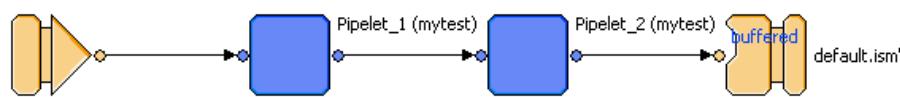
Defining Basic Pipeline Patterns

This section presents various samples illustrating basic pipeline patterns. Samples used range from very simple pipelines to sets of connected pipelines with complex interactions and execution flows.

A Simple Pipeline

The figure below shows a pipeline that begins with a start node, executes two pipelets, and ends with an interaction node. Upon processing the interaction end node, the pipeline dictionary is cleared.

Figure 100. A Simple Pipeline

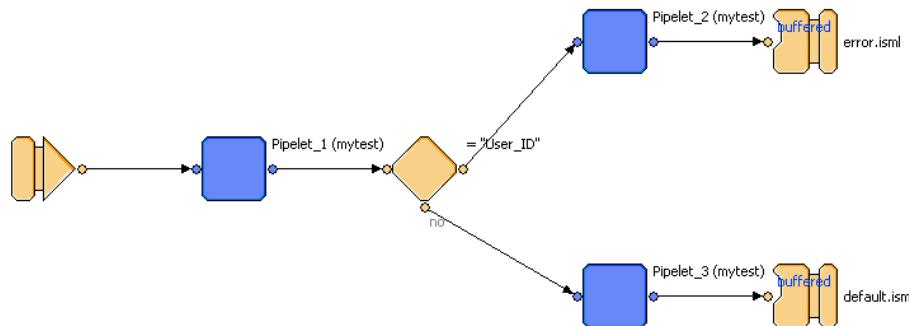


Changing the Pipeline Execution Flow

The pipeline in this example does not progress directly from start to end. The decision node determines the execution flow based on the evaluation of the condition defined in the decision node. See *Decision Node* on the properties to be set for decision nodes.

In the example pipeline, when the evaluation of the condition is positive, the "yes" branch is taken and the first template displays. When the evaluation is negative, the "no" branch is taken, and a different interaction node is reached.

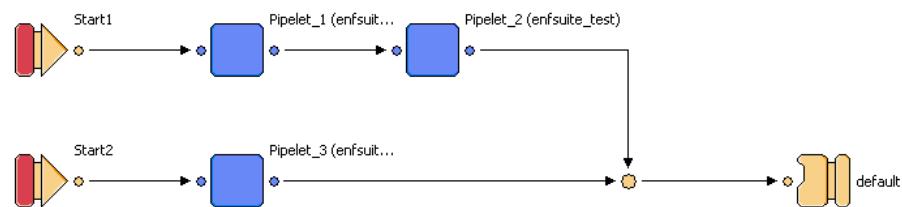
Figure 101. A Complex Execution Flow



Using Multiple Start Nodes

A pipeline can have multiple start nodes, each named uniquely. In the figure below, the execution path starting with Start1 does not execute Pipelet3.

See *Start Node* on the properties to be set for start nodes.

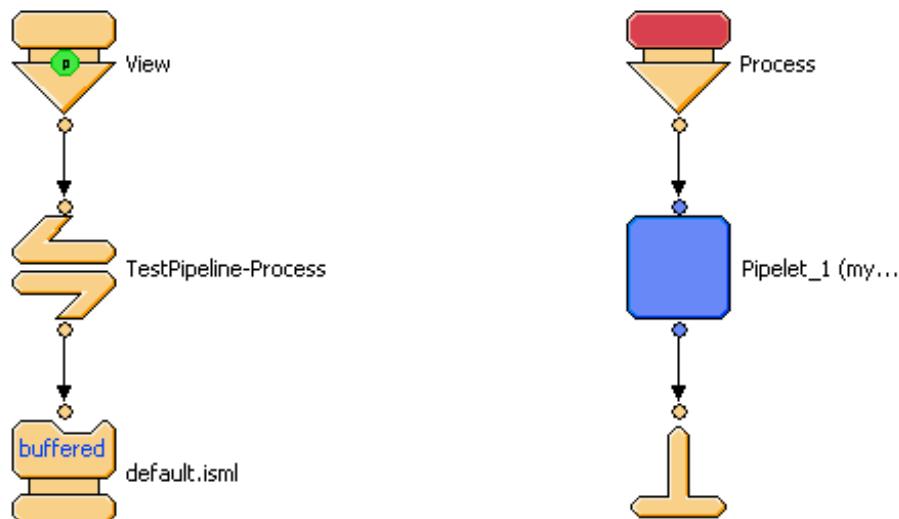
Figure 102. Using Multiple Start Nodes

Calling a Sub-Pipeline

This sample shows a pipeline calling another pipeline via a call node. The sub-pipeline is a normal pipeline with all types of nodes, pipelets, interactions, decisions, calls, etc. Called pipelines execute like subroutines. Control returns to the next node in the original pipeline after the called pipeline finishes.

See *Call Node* on the properties to be set for call nodes.

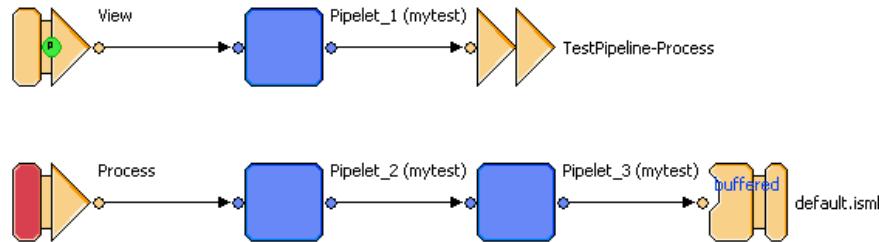
When designing sub-pipelines, remember not to end them with interaction nodes, which clear the pipeline dictionary. Instead, use an end node. After the end node of the sub-pipeline, the execution returns to the calling pipeline with the pipeline dictionary intact.

Figure 103. Using Call Nodes

Ending Pipelines With a Jump

This example shows how to end a pipeline with a jump node which enables you to jump to a new pipeline once the original pipeline has finished. Remember that although the dictionary remains active after a jump, the execution flow does not return to the original pipeline.

See *Jump Node* on the properties to be set for jump nodes.

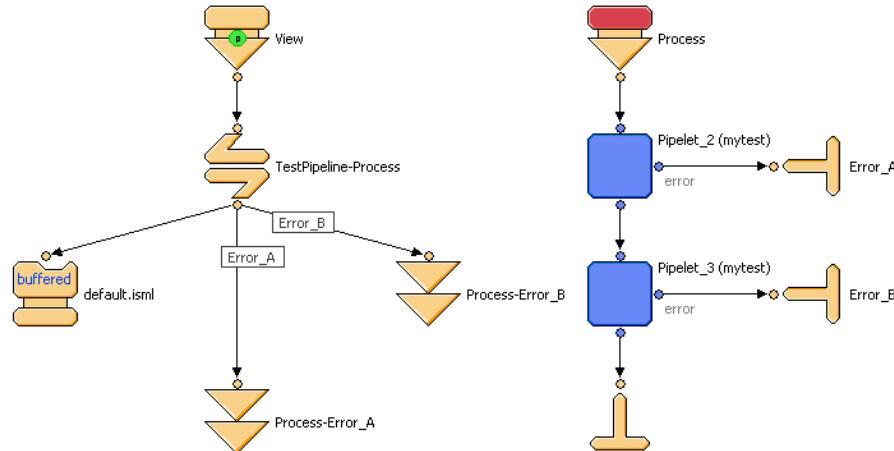
Figure 104. Using Jump Nodes

Using Named End Nodes and Multiple Call Node Exits

End nodes of a pipeline can be named. In conjunction with the capability of call nodes to continue depending on which end node a called sub-pipeline has reached, named end nodes can be used to precisely track and control the execution path of pipelines.

In the example below, named end nodes are used to differentiate errors resulting from the execution of Pipelet_2 and Pipelet_3 inside the sub-pipeline starting with start node Process. This sub-pipeline is triggered by a call node of a higher-order pipeline. If the sub-pipeline executes smoothly, an interaction end node is processed following the call node. However, if the sub-pipeline reaches one of the two named end nodes indicating a pipelet execution problem, jump nodes follow on the call node, each targeting a particular pipeline.

See *End Node* on the properties to be set for end nodes and *Call Node* on the properties to be set for call nodes.

Figure 105. Using Named End Nodes

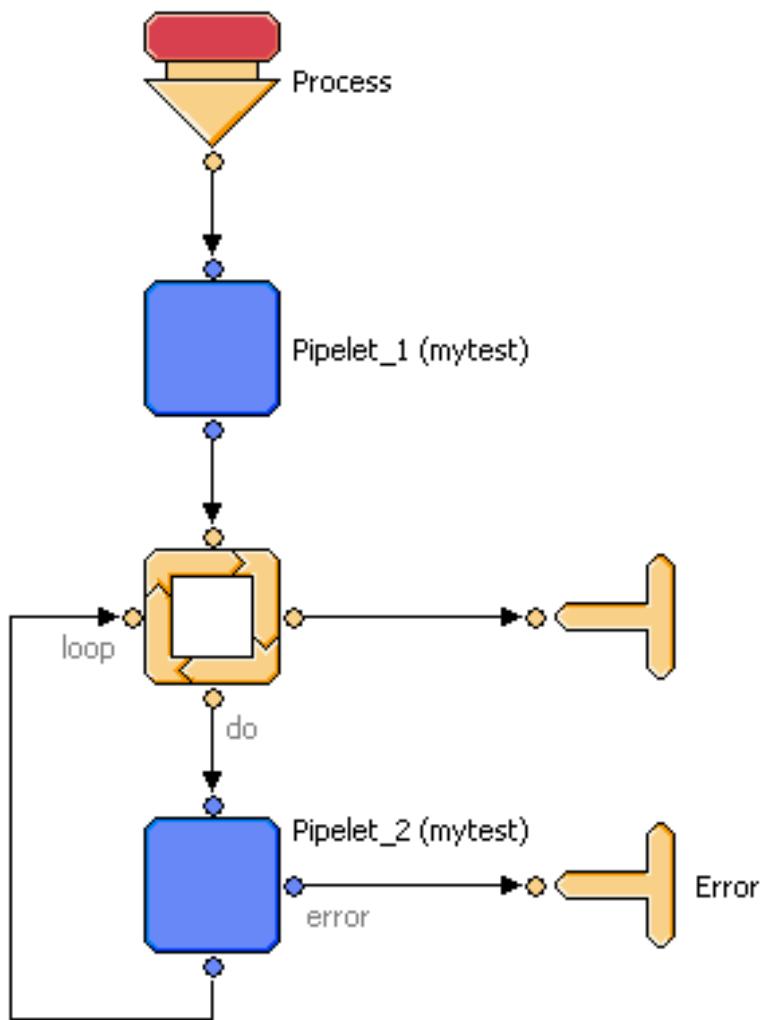
Using Loop Nodes

This example shows logic that is implemented with a loop node. An iterator object is first placed in the dictionary. When the path of execution enters the loop node through the IN transition, the first element in the iterator is removed and placed in the Element dictionary key. The path of execution then proceeds along the DO transition. Any pipelets along this path are executed, until a transition directs

execution to the loop node by means of the LOOP transition. Each time this occurs, the loop node checks the iterator to see if any elements remain to be iterated through. If so, the next element from the iterator is stored in the Element dictionary key and the DO transition is once more followed. If not, the iterator is reset to the first element and the NEXT transition is followed, thus ending the loop.

See *Loop Node* on the properties to be set for jump nodes.

Figure 106. Using Loop Nodes



Working with Pipelines

Check Pipeline and Data Flow

Use the Pipeline Dictionary View to conveniently track the data flow in the pipeline dictionary. Using the dictionary view, you can track for each pipeline node which data are read from the dictionary to process the node and which data the node puts into the pipeline dictionary. See *The Pipeline Dictionary View* for details.

Intershop Studio provides the possibility to check a pipeline and to indicate errors and warnings in the Problems View. The pipeline check can be performed automatically while editing or can be started manually.

To enable automatic pipeline check:

1. **From the menu bar, select Window | Preferences to open the Workbench Preferences dialog.**
2. **In the tree pane on the left, select Intershop Studio, then Problem Annotation.**

The Problem Annotation panel is displayed.

3. **Select and deselect checkboxes to define which types of problems should be indicated during automatic or manual checks.**

See *Problem Annotation Preferences* for details.

To enable automatic pipeline checks:

1. **From the menu bar, select Window | Preferences to open the Workbench Preferences dialog.**
2. **In the tree pane on the left, select Intershop Studio, the Problem Annotation.**

The Problem Annotation panel is displayed.

3. **Select the checkbox "Check Element on Resource Modification and While Editing."**

To start the pipeline check manually:

1. **In the Cartridge Explorer, select the pipeline to be checked.**
2. **Right-click the pipeline to open the context menu.**
3. **From the context menu, select Check Element.**

Possible errors and warnings are displayed in the Problems View. See *Using the Quick Fix Function* for details on the icons used and on using the quick fix function to solve pipeline problems.

Pipeline Reload

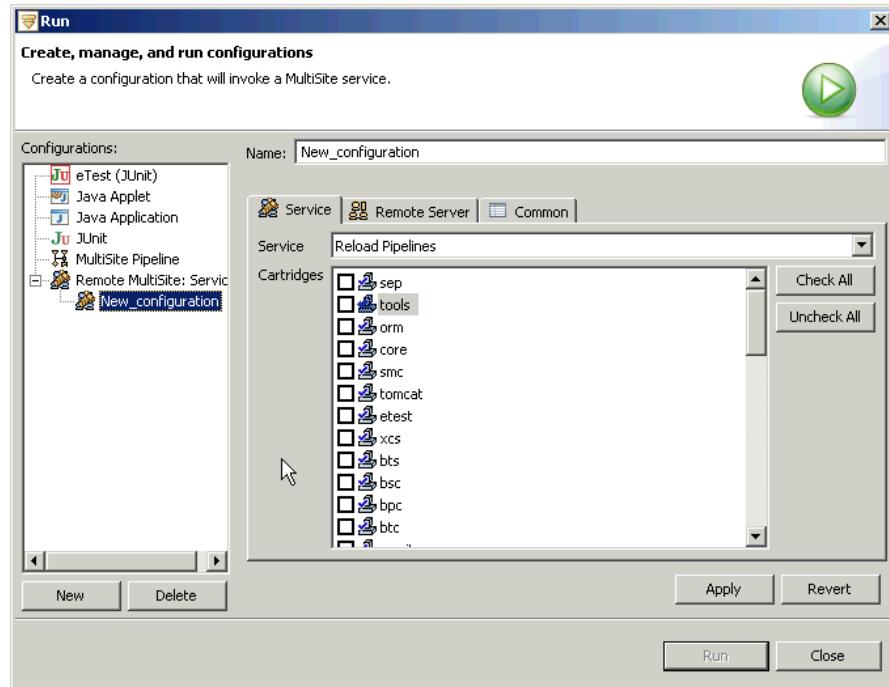
New or modified pipelines deployed on the development system need to be loaded before they can be executed by the Intershop 7 application server. This is done by reloading all pipelines that belong to the cartridge which contains the new or modified pipeline.

NOTE: The cartridge containing the new or modified pipelines must be registered on the development system for the pipeline reload to work. For more information, see here.

1. **Select Run | Run... from the Workbench menu.**

The page to create, manage and run configurations is displayed.

- 2. On the Configurations panel, select a Remote MultiSite Service configuration. If necessary, right-click Remote MultiSite Service to create a new configuration, or right-click the configuration to clone it.**
- 3. On the Service tab, select Reload Pipelines from the Service drop-down list. Select the check boxes for the cartridges whose pipelines need to be reloaded.**

Figure 107. Reload Pipelines

4. On the Remote Server tab, select the server connection parameters

to use. If necessary, configure a new server connection. See Reload Pipelines from the Service drop-down list. See *Remote Server Configuration* for configuration details.

- 1. On the Common tab, select the Pipeline Development perspective as perspective to switch to when running the pipeline reload.**
- 2. Select Run to start the pipeline reload.**

To run the same configuration again, select Run | Run History from the workbench menu.

NOTE: The Intershop 7 application server can be configured to reload pipelines and pipelets automatically. If the property intershop.pipelines.CheckSource in the appserver.properties file is set to "true", the server checks for new pipeline or pipelet files before executing a request.

Compare Pipelines

Intershop Studio provides an easy-to-use mechanism to compare two pipelines, pipeline A (displayed in the left pane of the comparer) and pipeline B (displayed in the right pane of the comparer). Using the pipeline comparer, it is possible to

- compare a pipeline with an earlier version of this pipeline (pipeline history).

- compare a pipeline with an overriding pipeline (pipeline B overrides pipeline A).
- compare a pipeline with an entirely different pipeline.

Starting from a high-level structure compare view (Pipeline Structure View or XML Compare View, see below), you can zoom into the pipelines for detailed comparisons. Two options for detailed comparison are available: the graphical Pipeline Compare window and the Text Compare window.

■ Pipeline Structure View

The pipeline structure viewer compares pipelines based on pipeline and pipeline node properties as well as the pipeline graph extension. It supports the following comparison modes:

- Determine Element Equality using Node ID - Instead of the node name, this mode takes the node ID for the comparison. This prevents renamed nodes from being qualified as additional or removed nodes.
- Ignore Position Changes - This mode ignores pipeline layout changes, i.e., node position and orientation as well as transition ways and bend points.
- Ignore Localization Changes - This mode can be used to ignore localized properties. In case there is no localized information available, e.g., when comparing versions of one pipeline, this mode is automatically enabled.

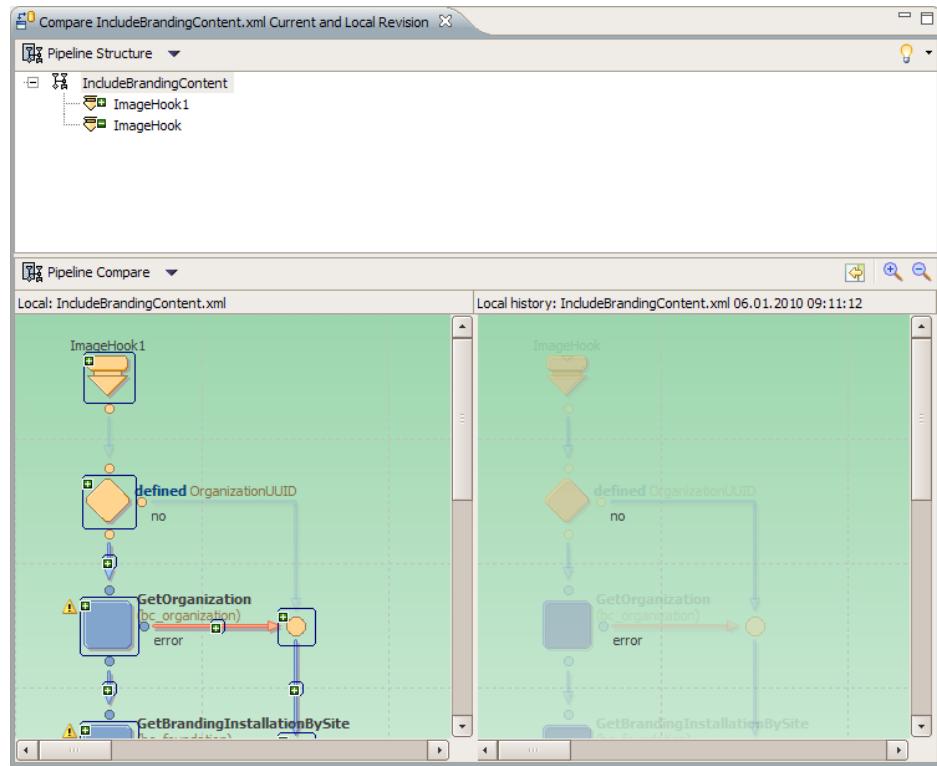
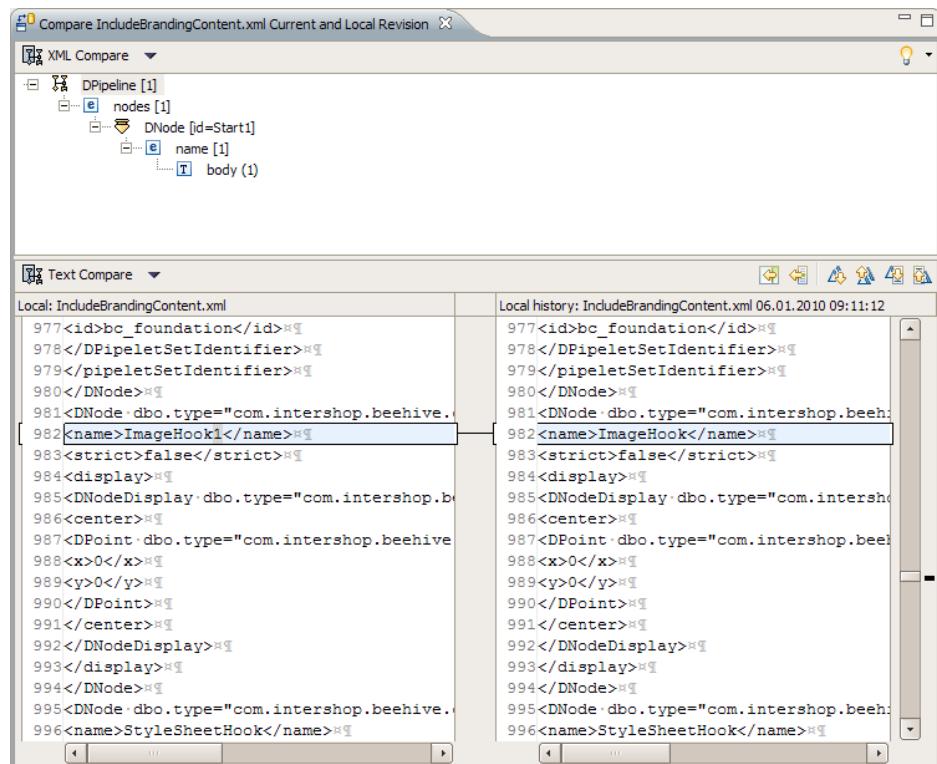
■ XML Compare View

Using the XML structure comparer, the pipelines are treated as XML files. Mapping preferences (which are accessible via Preferences|General|Compare/Patch|XML Compare) define the available elements. Two pre-configured mapping definitions for pipelines are available as comparison options:

- Pipeline/Pipelet XML - This is the default mode for comparing Intershop 7 pipeline/pipeline XML files.
- Pipeline (Ignore Display) - This mode ignores the display tags in Intershop 7 pipeline/pipeline XML files, which helps to better identify changes to the pipeline behavior.

The other mapping definitions may be used with XML files other than Intershop 7 pipelines.

In addition to the structure comparers, Intershop Studio 2.7 includes two comparison editors, which visualize pipeline differences either graphically ("Pipeline Compare") or textually ("Text Compare"), and allow for directly editing the pipelines.

Figure 108. Visually comparing pipeline structures**Figure 109. Comparing pipeline XML files**

The pipeline comparer indicates removed, new or modified nodes, transitions and pipelets. The pipeline comparer not only detects differences at pipeline element

level (such as a new or missing jump node), but also differences at property level (such as modified or changed pipelet parameters).

NOTE: The pipeline comparer takes pipeline start nodes as the basis for comparing sub-pipelines. For example, a sub-pipeline with start node UpdateAddress is compared to the respective pipeline with the same start node. If start node names have changed, both pipelines are treated as independent.

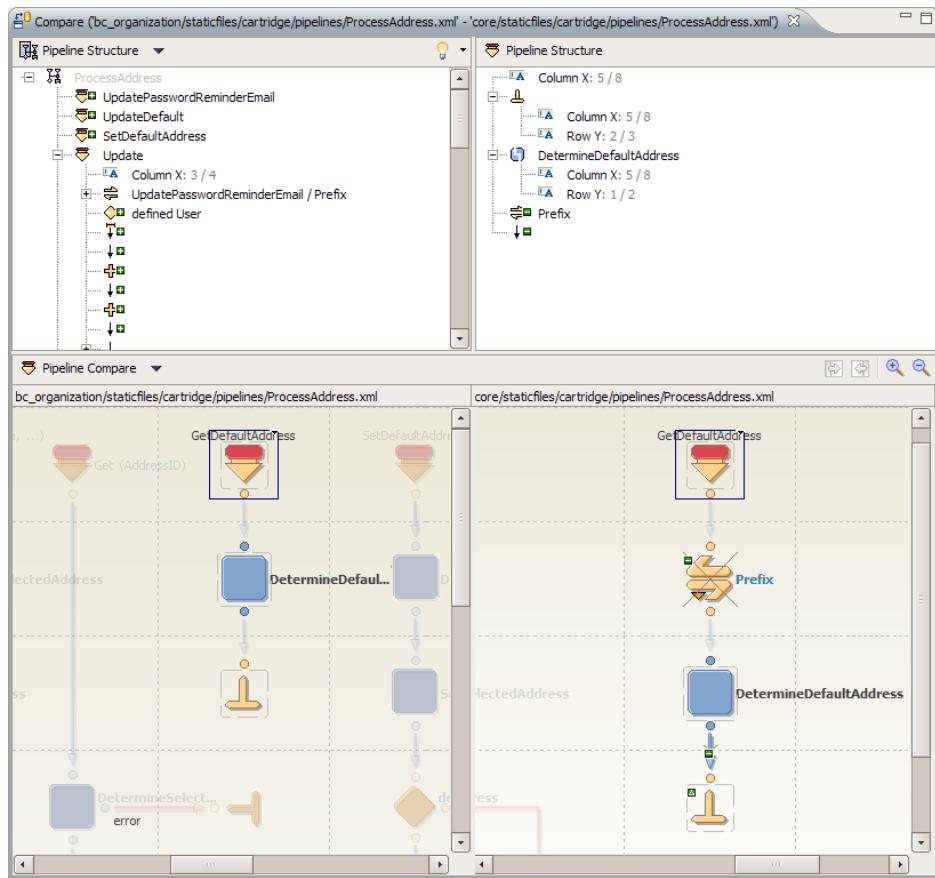
When working with the pipeline comparer, you can:

- double-click an element in the difference structure to directly jump to the respective pipeline element,
- use the Zoom In or Zoom Out icons to change the size of the pipeline section to view,
- open the options drop-down menu to select an appropriate compare mode.

For example, in the figure below, the pipeline comparer contrasts the pipeline ProcessAddress (pipeline A) from the core cartridge with its overriding pipeline ProcessAddress (pipeline B) from the cartridge bc_organization. The difference structure reveals that pipeline A has a sub-pipeline with start node Prefix which is missing in pipeline B. This is indicated by the symbol  on the respective start node. On the other hand, pipeline B has two sub-pipelines (with start nodes SetDefaultAddress and UpdateDefault) which are missing in pipeline A. This is indicated by the symbol  on the start nodes. In addition, these nodes are crossed out in the right pane of the pipeline detail view (showing pipeline B).

Looking in more detail at sub-pipeline ProcessAddress-GetDefaultAddress (which is present in pipeline A and pipeline B), the difference structure indicates that the sub-pipeline has two elements (a transition and a jump node) which are missing in pipeline B. When double-clicking on these elements, you jump to these elements in the left pane of the pipeline editor where you can investigate the exact position of these elements in the pipeline.

See *Pipeline Comparer* for a description of the symbols used to visualize differences between pipelines and pipeline elements.

Figure 110. Pipeline Comparison

In order to compare a pipeline with a different version of the same pipeline from the pipeline history:

- 1. Select the pipeline in the Cartridge Explorer.**
- 2. Right-click to open the context menu.**
- 3. From the context menu, select Compare With | Local History.**

This opens a new workbench in the pipeline editor, displaying the pipeline comparison, a list of all pipelines available in the pipeline history, and a structural outline of the pipeline to which the current version is compared.

In order to compare two different pipelines:

- 1. Select the pipelines to compare in the Cartridge Explorer.**
- 2. Right-click to open the context menu.**
- 3. From the context menu, select Compare With | Each Other.**

This opens a new workbench in the pipeline editor, displaying the pipeline comparison, and a structural outline of the pipeline to which the current version is compared.

In order to compare a pipeline with its overriding pipeline:

- 1. Select the overridden pipeline in the Cartridge Explorer or pipeline view.**
- 2. Right-click to open the context menu.**
- 3. From the context menu, select Compare With | Overriding Pipeline.**

This opens a new workbench in the pipeline editor, displaying the pipeline comparison, and a structural outline of the pipeline to which the current version is compared.

Pipeline References and Dependencies

Intershop Studio offers extensive search functions which allow for exploring dependencies and relevancies of pipelines.

To explore pipelines which are able to call the current pipeline:

- 1. Open Pipeline View.**
- 2. Select the desired pipeline.**
- 3. Right-click and select Cartridge References from context menu.**
- 4. Set filter for your search.**

You may be search across the whole workspace or set filter for search in workspace projects only or to the current cartridge or a specific other cartridge or working sets.

The Search view lists all the places where the pipeline is referenced. At a glance it can be seen in which cartridges and subsequent pipelines the current pipeline is used. So it is easy to recognize, for which parent pipelines the current pipeline is relevant.

To explore a pipeline's dependencies:

- 1. Open Pipeline View.**
- 2. Select the desired pipeline.**
- 3. Right-click and select Elements Used from context menu.**
- 4. Set filter for your search.**

You may be search across the whole workspace or set filter for search in workspace projects only or to the current cartridge or a specific other cartridge or working sets.

The Search view lists all relevant artifacts of the current pipeline (pipelets, nodes, templates and queries). Thus, for example all pipelets used in the current pipeline are displayed. In addition, all reachable nodes are listed, which can be reached by the current pipeline. So it is easy to see on which child pipelines current pipeline depends.

Using the Content Assist

You can use content assist at various places when creating and modifying pipelines. To activate the content assist, press **Ctrl + Space**. The content assist displays a floating window with a list of possible options (such as dictionary values or

templates) available at the current point. Content assist is signaled by the icon  when clicking on the item for which assistance is required.

See *Content Assist* for a description of the different contexts in which content assist is available.

Using the Quick Fix Function

Use the Quick Fix function to automatically solve frequently occurring problems. Available quick fixes can be checked and selected directly from the Template Editor, or by selecting the respective problem in the Properties view. To apply quick fixes to entries in the Properties view:

- 1. Right-click on a warning or error in the task list.**
- 2. From the context menu, select Quick Fix ...**

The Quick Fix dialog opens.

NOTE: If the command in the context menu is grayed out, no quick fix is available.

- 3. Choose one of the options to fix the problem and click OK.**

The fix is carried out and the task removed from the list.

The quick fix function can also be invoked directly from the Pipeline Editor. Select the element that contains the error and press **Ctrl + 1** to open the list of available quick fixes, including a preview of the proposed solution.

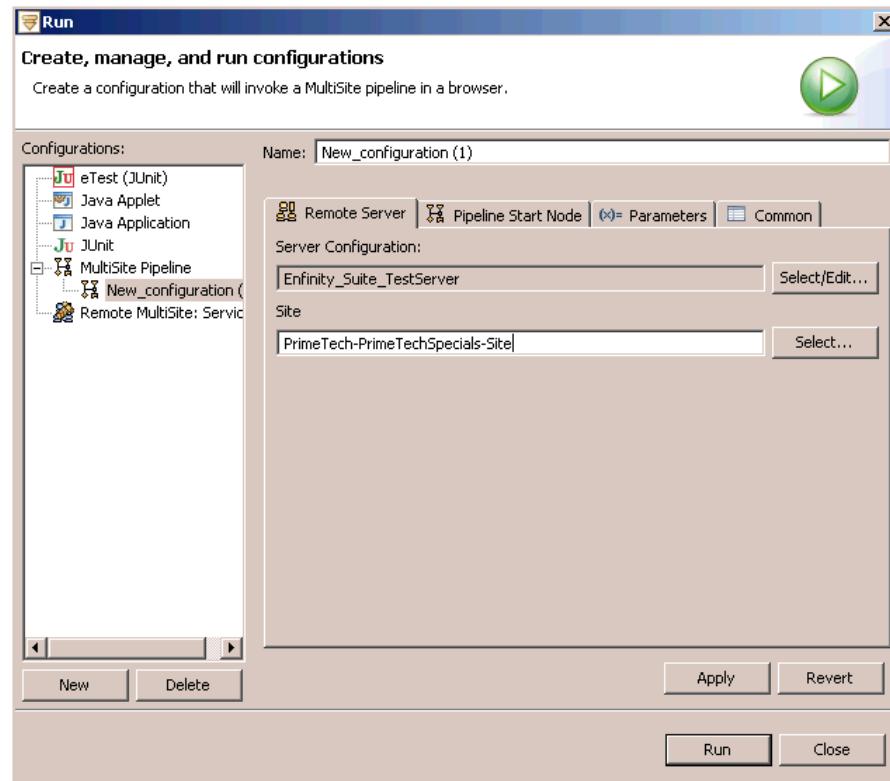
NOTE: If the automatic build is disabled, Intershop Studio does not automatically update warning and error markers in the code (including proposed quick fixes). To update these markers after modifying your code, build your project manually as described in here. To update these markers for a single resource only, right-click the element in the Cartridge Explorer to open the context menu and select Check Element.

Run Pipeline From Intershop Studio

To test or debug a pipeline, you can run the pipeline directly from within Intershop Studio. To run a pipeline:

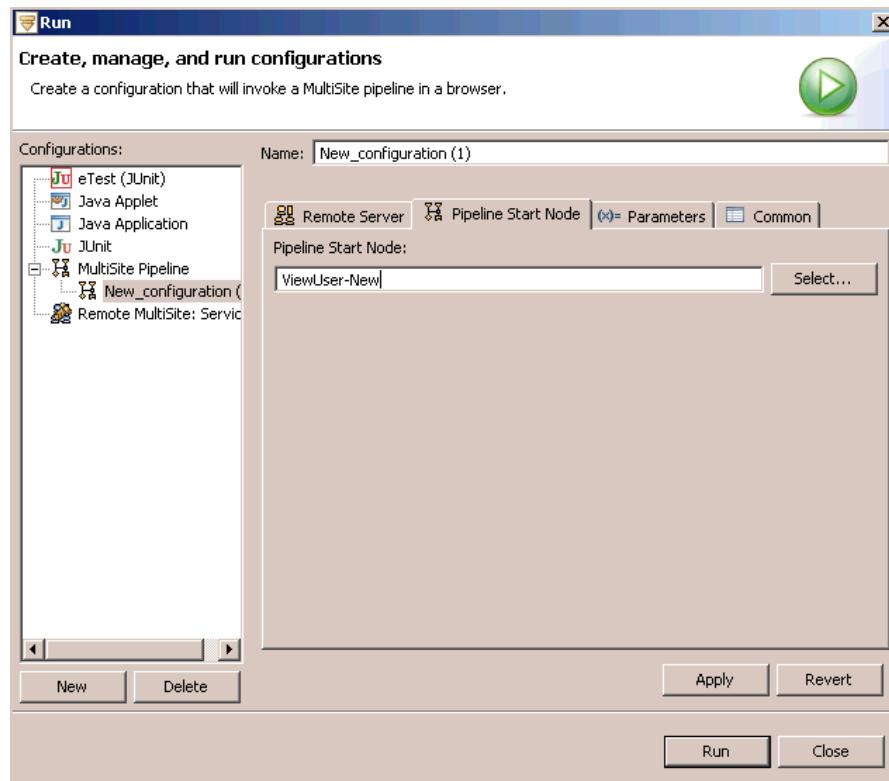
- 1. Select Run | Run ... from the Workbench menu.**
This opens the wizard to create, manage and run configurations.
- 2. In the Configurations panel, select "MultiSite Pipeline" and click New.**
- 3. On the Remote Server page, select a remote server connection. Specify the site in whose context the pipeline to debug runs.**

Click Select to retrieve a list of all sites available on the remote server.

Figure 111. Launching a pipeline from Intershop Studio

4. On the Pipeline Start Node page, specify the pipeline and the pipeline start node to run.

Click Select to retrieve a list of all pipelines and start nodes available to the site provided on the Remote Server page.

Figure 112. Launching a pipeline from Intershop Studio

5. **On the Parameters page, provide parameters and parameter values which the pipeline needs for execution.**
6. **Click Run.**

This opens a new editor window displaying the generated page or any other information the pipeline might return.

Editing Pipelines on Remote Servers

Intershop Studio provides the possibility to directly edit pipelines on remote servers. This features (also referred to as "hot edit") enables you, for example, to modify pipeline or pipelet parameters on a non-development system (e.g. test or production system).

NOTE: Remote editing should be used to diagnose problems and fix smaller issues. When dealing with larger issues, consider using the remote development feature instead (for details, see *Remote Server Configuration*).

For remote editing, Intershop Studio provides a specialized perspective, the Intershop 7 Remote Editing perspective. With respect to the explorers, views and editors, the Intershop 7 Remote Editing perspective is similar to the Cartridge Development perspective.

To edit a pipeline on a remote system:

1. **Configure a remote server connection**

See *Remote Server Preferences* for details.

2. In the workbench, click **Window | Open Perspective | Other**. Select the **Intershop 7 Remote Server Perspective** from the list.

Having established the connection to the remote server, the cartridges are displayed in the Cartridge Explorer.

Debugging Pipelines

Preparing to Debug

To use the pipeline debugger, you must know what behavior you want to check and know the pipeline name(s) associated with the behavior. Note that server name, pipeline name, and start node are displayed in the address URL box of the storefront browser, if needed. To determine the pipeline to be debugged:

1. **Access your storefront.**

The default address is:

```
http://<server>/INTERSHOP/web/WFS/<site name>
```

2. **Trigger some basic functionality by clicking on a link.**

3. **Determine which pipeline and which start node is called.**

Use the Pipeline View to locate the pipeline and open it in the Pipeline Editor.

To debug the pipeline you want to check you can:

- **Trigger the pipeline in the storefront**

You can trigger the pipeline by clicking the respective link, or by submitting the respective URL directly in the browser.

- **Run the pipeline from within Intershop Studio.**

This makes it possible to debug pipelines which are not accessible through the storefront.

Set Breakpoints

The pipeline debugger is activated by setting breakpoints in a pipeline and then triggering the action. When the pipeline encounters the breakpoint, it stops and brings the Pipeline Editor forward. Unlimited breakpoints can be set. To set breakpoints:

1. **Select a node in the chosen pipeline.**

2. **Select Run | Add/Remove Pipeline Node Breakpoints.**

This sets a breakpoint on the chosen node. Conversely, if the node already has a breakpoint assigned, the breakpoint is removed. Breakpoints are marked by a green rectangle.

Figure 113. Start Node with Breakpoint (Inactive)



NOTE: Note that the breakpoint in the figure above is inactive, as indicated by its transparent fill color. It will become active once a debug connection and site context have been defined.

Breakpoints can be managed in the Breakpoints View, which is a standard Eclipse view. The Breakpoints View displays all breakpoints from all pipelines currently open in the Pipeline Editor. Using the Breakpoints View, you can filter out the breakpoint of the pipeline that is currently active, remove selected breakpoints, or remove all breakpoints at once.

Define Connection and Site Context

In order to debug a pipeline, a connection must be defined and established between Intershop Studio and the system on which the pipeline is deployed. In addition, you have to specify the site in whose context the pipeline is going to run.

To configure and establish a connection and specify the site context:

1. **In the workbench menu, select Run | Debug to open the Debug connection page.**

Select a configuration from the Configurations pane. Otherwise, select New and proceed as described below.

2. **On the Remote Server page, click Select/Edit to select a server connection or define new connection parameters.**

See *Server Connection Parameters* for details on the connection parameters to be set.

3. **On the Remote Server page, enter a site ID to reference the site to access. Alternatively, click on the Select button.**

This will prompt Intershop Studio to connect to the remote server and retrieve a list of all accessible sites.

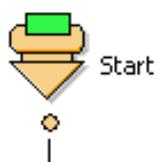
4. **Click Debug to open the connection.**

If the connection has not already been established, Intershop Studio now attempts to connect to the Intershop 7 server using the specified parameters.

You can provide a name under which the connection_parameters are saved. The parameters can then be reused for future debugging sessions.

Once the debug connection has been established, all breakpoints set for pipelines that are in the scope of the site context will become active. Active breakpoints have a non-transparent fill color, as shown below:

Figure 114. Start Node with Breakpoint (Active)



Launch Debug Session

To launch a debug session, you have to trigger the pipeline to debug, either by calling it in the storefront or by running the pipeline directly from Intershop Studio.

See [Run Pipeline From Intershop Studio](#) for details on how to run the pipeline directly.

To trigger the pipeline in the storefront:

- 1. Click the link in the storefront page which will trigger the pipeline you wish to debug.**

Alternatively, you can submit the URL directly via the browser. However, make sure to pass all parameters required by the pipeline to be debugged.

- 2. Since a breakpoint has been set on the pipeline, the Debug perspective is brought forward.**

The pipeline is halted at the first breakpoint encountered.

- 3. Continue with stepping through the pipeline in debug mode.**

Step Through a Pipeline in Debug Mode

Once the pipeline debugger has begun, you can step through the pipeline in various ways. See the table below for the possible step-through actions. The actions can be triggered using the respective icons in the Debug View or by selecting them in the Run menu of the workbench.

At each step, the Debug View displays the current pipeline node.

Figure 115. Current Node in Debug View

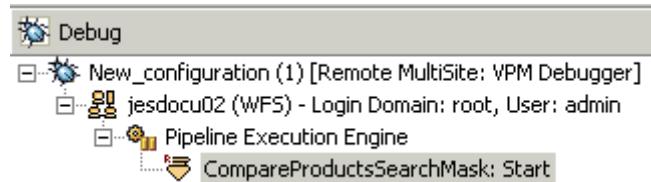


Table 52. Debug Actions

Action	Icon	Description
Step Into		Steps from the beginning of one node to the beginning of the next one.
Step Over		Steps forward one node and steps over call or jump nodes (i.e. executes them as a single step without displaying them in the editor).
Step Return		Terminates the debug process.

NOTE: When debugging, the pipeline execution tree in the Outline View provides a high-level overview of the current debug position and the respective pipeline context.

Working With Debug Views

For each node, information on current variables and expressions in the pipeline dictionary are displayed in the Variables View. To focus on selected expressions, it is possible to mark variables and expressions of interest as watch expressions, which adds them to a separate view, the Expressions View.

To enter an expression to the list of watch expressions:

1. **Mark the expression of interest in the Variables View.**
2. **From the workbench menu bar, select Run | Add Watch Expression.**
3. **Change to the Expressions View to inspect the expression.**

Pipeline Security

Since pipeline execution can be triggered from external sources via an URL, special care is necessary to secure pipelines, making sure that a pipeline responds only to requests from authorized clients.

Public and Private Start Nodes

The configuration parameters for a start node include the call mode parameter. The call mode determines whether a pipeline can be triggered from an external source using HTTP requests, or from an internal source only using call or jump nodes. The call mode parameter for start nodes can take the following values:

■ **Public**

For access from an external source such as the storefront, using (for example) HTTP requests.

■ **Private**

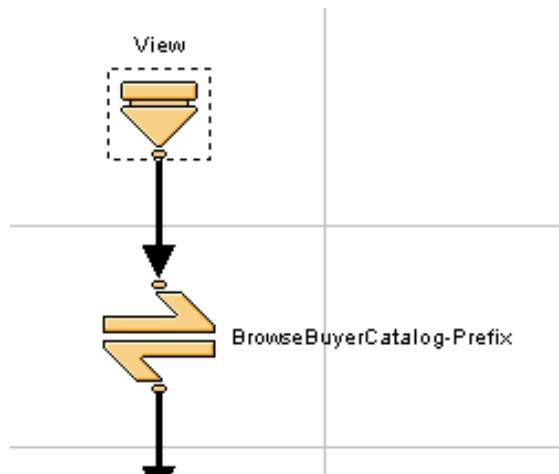
For access from an internal source such as another pipeline, using call or jump nodes.

For example, many viewing pipelines are triggered by external sources via HTTP requests, hence need to have a public start node. On the other hand, processing pipelines are typically called from viewing pipelines. Therefore, processing pipelines typically have private start nodes.

Prefix Pipelines

Prefix pipelines are generic pipelines which perform certain checks before the pipeline that has actually been requested is executed. Prefix pipelines therefore provide a powerful mechanism to prevent unauthorized access to public pipelines.

Prefix pipelines can be configured individually for each pipeline using a call node that triggers the prefix pipeline right after the start node, as shown below.

Figure 116. Simple prefix pipeline

As an alternative to inserting a call node after every start node, Intershop 7 supports so-called site prefix pipelines. The site preference value `SitePrefixPipeline` specifies the name of a pipeline which is automatically called before any public pipeline in the site is executed. The default value is set to the `CorePrefix` pipeline. See the figure below for details on this pipeline.

Figure 117. Site prefix pipeline

Properties		Problems	Javadoc	Ant
PrcBuyer				
Property	Value			
+ Product Type				
+ Proxy Unit				
+ Requisition Approval Workflow				
+ SAP Connection Preferences				
+ Security				
+ Seller				
- Site				
Pagecache Keyword Processing	Disabled			
Personalization Mechanism	PersonalizationGroupProvider			
Site Prefix Pipeline Name	Prefix			
Web Service Overview Page	Enabled			

The pipelet processor checks the `SitePrefixPipeline` preference value only once, the first time a pipeline in the site is executed. The `SitePrefixPipeline` preference value is reread only if the pipelines of the site or all pipelines in the system are reloaded.

If a pipeline is executed and the site's `SitePrefixPipeline` preference value contains a prefix pipeline name, the prefix pipeline is looked up in the normal manner and is then executed. This also means that a prefix pipeline can be overloaded in the usual way.

Different versions of the prefix pipeline can be defined using the start nodes `process`, `view`, or `backoffice`. These different versions are executed depending on the type of the pipeline (`Process`, `View`, or `Backoffice`) triggering the prefix pipeline. For example, if the original pipeline being executed is of type `Process`, a start node `process` is searched for in the prefix pipeline. If the system cannot find one, the pipelet processor looks for a default start node named `Start`. The name

of the originally called pipeline and the start node name are put into the prefix pipeline dictionary using the `CurrentPipelineName` and `CurrentStartNodeName` keys.

NOTE: The pipeline type is stored in the pipeline's XML descriptor file. By default, the pipeline types are stored in lower case letters (e.g., `<type>process</type>`). Since the pipeline lookup mechanism is case sensitive, the respective prefix pipeline start nodes must be lower case as well.

The originally called pipeline is not executed after the site prefix pipeline if the following happens:

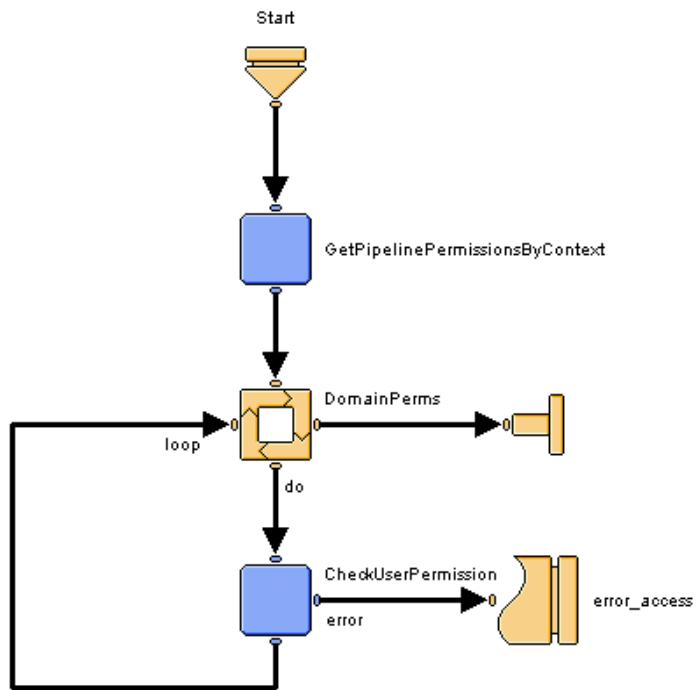
- The site prefix pipeline ends with an interaction
- The site prefix pipeline ends with an exception, in which case the error handler of the site prefix pipeline is looked up and executed

If the site prefix pipeline ends with a named end node, the originally called public pipeline is executed. The `SitePrefixEndNodeName` value of the site prefix pipeline is passed to the dictionary. The values of the site prefix pipeline dictionary are available to the original pipeline as well.

CorePrefix Pipeline

The `CorePrefix` pipeline is a prefix pipeline which is provided as a default. The pipeline checks if a user has the required permission to run a called pipeline. This is, basically, a two-step process:

- First, the `CorePrefix` pipeline determines the permissions that are required to execute the called pipeline at the specified start node in the given site context.
- Then, it checks whether the current user's permissions match the previously determined permissions. If so, the called pipeline will be executed; if not, the execution is denied and an error page will be displayed.

Figure 118. CorePrefix pipeline

The permissions themselves are configured in the `pipelines-acl.properties` file, located in the pipeline directory of every cartridge, and pre-loaded upon the pipeline cache initialization.

Pipeline Access Control Lists

For each pipeline, access control lists define the permissions a user needs to execute this pipeline in general, or a specific start node of this pipeline. Every cartridge or site that has pipelines provides a `pipelines-acl.properties` file, located in

- `<IS_HOME>\share\system\cartridges\<cartridge_name>\release\pipelines`
for cartridge pipelines, or
- `<IS_HOME>\share\system\sites\<site_name>\1\pipelines`
for pipelines directly bound to a specific site.

The `pipelines-acl.properties` file includes the access control list (ACL) for every pipeline of the cartridge or site whose access should be restricted. These access control lists are loaded into the pipeline cache upon server startup or pipeline reload.

ACL Lookup Process

As pipelines are always executed within a site context, the common pipeline lookup and overload mechanism must apply to the ACL lookup as well. The following steps are performed:

- 1. First, all available sites in the system are determined.**
- 2. The site specific ACLs are read from the `pipeline-acl.properties` located in the site's pipeline directory and stored into the site pipeline cache.**
- 3. The cartridges that are currently loaded in the given site are determined.**

Note that the order within the list of cartridges bound to a site is relevant. The system sequentially searches for cartridge resources in a top-down manner, starting with the first cartridge in the list.

- 4. For each cartridge in the list, the pipeline ACLs are loaded sequentially.**

Since a cartridge can be loaded in several sites, the cache prevents from loading the ACLs of the same cartridge again and again.

- 5. Each loaded entry of the cartridge's ACL file is checked against the site pipeline cache.**

Only those entries that have not already been added to the cache are added. Thus, ACLs can be overloaded like pipelines by adding the access control list to a cartridge which is on top of the lookup list or to the ACL list of the site.

The access control list lookup also considers cartridges and sites that do not define any pipelines. In this case, the cartridge's or site's pipeline directory only contains the file `pipelines-acl.properties`. This allows developers to overload the ACLs of pipelines by their project specific cartridges or sites.

ACL Syntax

The `pipelines-acl.properties` files and the actual access control lists must match a specific syntax. As all properties files, they are made of key-value pairs. `pipelines-acl.properties` files define the required permission for each pipeline-start node combination.

The key is `PipelineName-StartNodeName`. The value is a semicolon separated list of `CONTEXT:PERMISSIONID` pairs. In addition, you can define a default access control list for each pipeline without a start node, or an empty access control list for a pipeline.

The following lines illustrate the ACL syntax:

```
ViewCatalog-Edit=Organization:VIEW_CATALOG; Catalog:MANAGE_CATALOG
ViewCatalog-Dispatch=
ViewCatalog=Organization:VIEW_CATALOG
ViewHelp=
```

The entries for `ViewCatalog-Edit`, `ViewCatalog-Dispatch` and `ViewCatalog` are treated as independent entries and are not merged during the cache initialization. A call for `ViewCatalog-Edit` returns `Organization:VIEW_CATALOG; Catalog:MANAGE_CATALOG`, a call for `ViewCatalog-Dispatch` returns an empty list, a call for `ViewCatalog-Search` returns `Organization:VIEW_CATALOG` as fallback option, and a call for `ViewHelp-Index` returns an empty list.

Note that permissions are always checked against an `AuthorizationObject`, i.e., a context, and that one or more permissions need to be checked before a

pipeline is executed upon a user's request. This behavior is represented in the CONTEXT:PERMISSIONID values. CONTEXT defines the authorization object used to check the permission specified in PERMISSIONID.

For example, the ACL

```
ViewCatalog-Edit=Organization:VIEW_CATALOG; Catalog:MANAGE_CATALOG
```

means that if a user wants to start the pipeline ViewCatalog-Edit, the permission VIEW_CATALOG must be checked against the authorization object Organization, and the permission MANAGE_CATALOG against the authorization object Catalog. The number of CONTEXT:PERMISSIONID pairs per access control list is not limited, but a CONTEXT:PERMISSIONID pair should not appear more than one time in an access control list.

Styleguide

Quick Overview

This section contains a quick overview of conventions and rules that are outlined and discussed in more detail in the remaining sections of this document. Use this checklist whenever developing or reworking pipelines.

Things you *should* do:

- Make sure your pipeline name fits the pipeline-naming scheme.
- Make sure you assigned the pipeline to a meaningful pipeline group.
- Make sure you assigned the pipeline to the correct pipeline type (view, process).
- Make sure your pipeline is secure.
- Make sure you documented the I/O parameters of the pipeline (on the start node).
- Make sure you set the 'NULL' alias for unused pipelet input and output parameters.
- Make sure you are using the common user interaction patterns.
- Make sure you used the pipeline transaction handling.
- Make sure your pipeline is readable. Divide pipelines into smaller sub-pipelines if they become too complex.

Things you *should not* do:

- Avoid using common names for dictionary aliases in processing pipelines.
- Never transform strings into object representations. Use formatter utility pipelets instead.
- Do not use non-ASCII characters in the source code.

Pipeline Security

General

The following general rules apply:

- All start nodes that are not to be called from external users have to be marked as private. This includes start nodes of scheduled jobs and start nodes called during dbinit.
- Start nodes marked as public have to be secured depending on the context in which they are used:
 - If they are accessible for anonymous users, no further checks are required.
 - Make appropriate entries in the `pipeline-acl.properties` file (see *Pipeline Access Control Lists*).
 - For buying organization front end and supplier front end, call the `CheckApplicationContext` pipeline first.

Pipeline Access Control Lists

Pipeline access control lists provide in conjunction with the prefix-pipeline a way to define for each pipeline which permissions a user needs to execute this pipeline.

The general syntax of entries in the `pipeline-acl.properties` file looks like this:

```
<Pipeline Name>[-<Start Node Name>]=[<Context>:<Permission ID>]
[;<Context>:<Permission ID>]*
```

Here are some examples:

```
// An empty list - no permission required
ViewCatalog-Dispatch=

// More than one entry - all permissions required
ViewCatalog-Edit=Organization:VIEW_CATALOG;Catalog:MANAGE_CATALOG

// No start node given - those entries are used for all start nodes
// that are not explicitly defined
ViewCatalog=Organization:VIEW_CATALOG

// Valid Contexts are defined by the application,
// e.g. the Organization and Channel are provided
ViewHelp=
```

Process Pipelines

Separating View and Process Pipelines

Usually viewing pipelines model the page flow of an application whereas process pipelines provide access to the Java business logic at pipeline level. This separation ensures that:

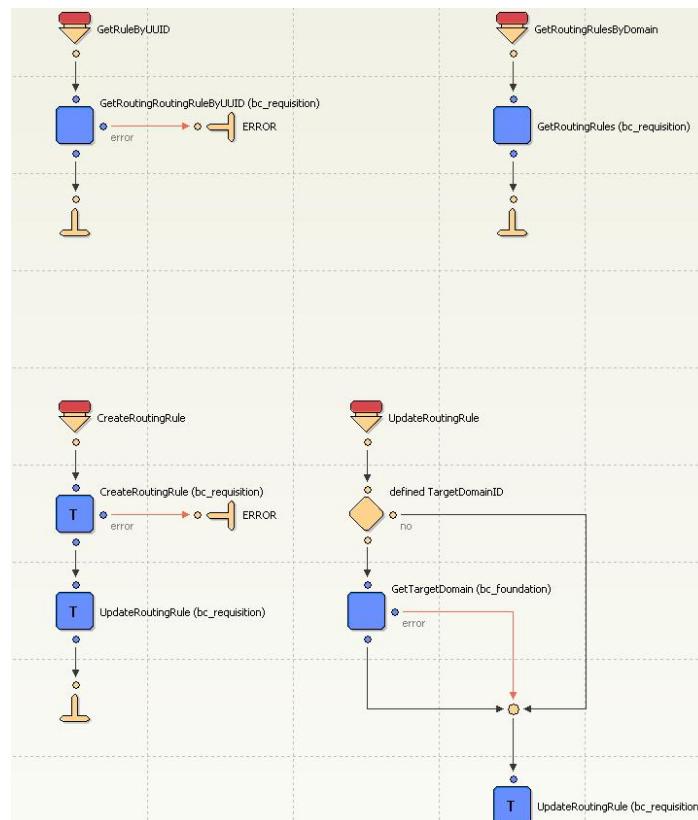
- The page flow can be changed on project base without re-implementing the business logic.
- Pipelines for business logic can be reused and view pipelines can be kept simple.

The following rules help to ensure high quality standards for process pipelines:

- Move pipeline logic (pipelets) from view- to process pipelines.
- View pipelines should not contain pipelets. They should only contain calls to generic process pipelines.
- Process pipelines should not use Web Forms.
- Avoid process pipelines with dynamic call nodes. Often dynamic call nodes are used to create customizable process pipelines. However it is very hard for an "outsider" to figure out what such a pipeline does. Therefore, they should be avoided.
- If possible locate process pipelines in business components, remove them from the back office application cartridge and from channel plugins.
- Avoid nested process pipelines. Try to move the call nodes into the view pipeline.

This is an example of a well-defined process pipeline:

Figure 119. Well-Defined Process Pipeline



Input Parameters

In regards to input parameters to process pipelines mind the following notes:

■ parameter naming

Objects that are expected as input by a process pipeline should not have a name starting with the prefix "Current..." .

If "Current" is part of the dictionary key it can cause overlappings with other objects in the dictionary. Using the Keymapper-Pipelet is not an option since it adds to the complexity of the pipeline.

If a process pipelines expects an UUID, the dictionary key must be named "UUID". There is one exception to this rule: the domainID must be named "DomainID" although it is actually an UUID.

■ **object vs. object reference**

Decide very carefully if it is necessary to pass the whole object or its ID (i.e. its UUID) to a process pipeline (and vice versa).

In some cases (i.e. removing an object) it is sufficient to pass only an ID to the process pipeline.

Whenever an object needs to be updated it should be passed to the process pipeline together with the data. This provides more flexibility in customer projects, prevents redundant code, extended runtime of the pipeline, and decreases the complexity of the pipelines.

In some cases pipelines are required that can handle the whole object as well as the object's semantic or internal ID. In these cases the pipeline needs to have two separate start nodes. Do not use a single start node for both cases since this would require additional pipeline logic afterwards to tell the different cases apart.

■ **convert form parameter into integer, boolean, etc.**

In many cases it is necessary to instantiate form parameters (Strings) into their real representation (Integer, Boolean, Date, Double).

It is okay to put the Integer, Boolean, Date or Double object into the pipeline dictionary under the same name as the form parameter. This reduces the number of keys in the pipeline dictionary.

However, use a different key for Integer, Boolean, Date or Double if the form parameter should be displayed again and no web form can be used (i.e. in an error case).

Process Pipelines in Business Components vs. in Application Cartridges

Process pipelines must be generic enough to be located in a business component.

There might be cases where a certain application requires a certain implementation of a process pipeline. For example, a process pipeline to validate the user input is specific in a way that it can only be used in a sellside back office application. As an exception such process pipelines can be located in the back office application cartridge (`sld_enterprise_app` or `sld_ch_base`).

Pipelines that manipulate real business objects should be analyzed very carefully to decompose it into more generic process pipelines if possible. By decomposing the pipelines and building a stable pipeline API the process pipelines become more reusable resulting in reduced development effort.

Templates should not be located in a business component cartridge. Exception: Templates used by the XML-export to generate the XML data or templates to convert import data from a proprietary format into Intershop 7 XML. Such templates may be located in business component cartridges.

Migration of Process Pipelines

The business logic of existing process pipelines should not be changed to ease migration.

If parts of a process pipeline need modification do not create a new process pipeline. Instead create a new Start Node in the same pipeline and copy the pipeline logic. Mark the "outdated" start node deprecated and name the new start node to be used in the future.

Pipeline Naming Guidelines

Presentation Pipelines

This section provides recommendations regarding the naming of presentation pipelines. All presentation pipelines should start with the View prefix. The prefix is followed by the name of the business object to be viewed.

View<Object>List

This type of pipeline is used whenever a pageable/non-pageable list of business objects is to be viewed. It is also used to perform an action on a set of selected business objects, e.g., delete. This pipeline usually displays the entry page of an HTML back office module.

Examples: `ViewDepartmentList`, `ViewUserList`, `ViewBuyingOrganizationList`

Table 53. Pipeline start node naming conventions

Start Node	Description
ListAll	Determines all business objects to be shown.
Search	Performs a search to determine a set of business objects to be shown. Use this start node name in case you do not distinguish between simple and advanced search.
SimpleSearch	Performs a simple search to determine a set of business objects to be shown.
AdvancedSearch	Performs an advanced search to determine a set of business objects to be shown.
Paging	Determines the next page of business objects to be shown.
Delete	Deletes a selected set of business objects.
Refresh	Re-displays the currently active page in case of a pageable list or simply all business objects (jump to node <code>ListAll</code>). This start node should be called whenever we jump back to the list, i.e., after update. <i>Hint:</i> Use page locator pipelets in <code>bc_foundation</code> to store the current pageable as session object.

View<Object>

This type of pipeline is used whenever a single business object is to be created/edited/deleted. It is also used to perform an action on a single business object, e.g., set as default, copy.

Examples: `ViewDepartment`, `ViewUser`, `ViewBuyingOrganization`

Table 54. Pipeline start node naming conventions

Start Node	Description
New	This start node prepares the creation of a new business object. In most cases it simply creates a new Web-form and sets the creation template. Hint: To reduce template complexity, do not use the same template for create and update.
Edit	This start node prepares the update of a new business object. In most cases it determines the business object to be edited, creates a new Web-form and sets the update template.
Show	This start node is used to show a new business object in read only mode. In most cases it determines the business object to be shown, prepares data and sets the update template.
Create	This start node creates a new business object. If the object could not be created successfully, the creation template is shown again (there should be an error message shown somewhere on the template).
Update	This start node updates an existing business object. If the object could not be updated successfully, the update template is shown again (there should be an error message shown somewhere on the template).
Delete	Deletes a given single business object. If the object deletion succeeded, the corresponding list template is shown again.

View<Object><AssociatedObject>List

This type of pipeline is used whenever associated objects or object relations of a single business object are to be listed. In most cases these pipelines are used to display a "tab" in an object detail view.

Example: ViewDepartmentUserList, ViewProductCustomAttributeList, ViewProductAttachmentList, ViewProductBundledProductList

Table 55. Pipeline start node naming conventions

Start Node	Description
ListAll	Determines all associated business objects to be shown.
Search	Performs a search to determine a set of associated business objects to be shown. Use this start node name if you do not distinguish between simple and advanced search.
SimpleSearch	Performs a simple search to determine a set of associated business objects to be shown.
AdvancedSearch	Performs an advanced search to determine a set of associated business objects to be shown.
Paging	Determines the next page of associated business objects to be shown.
New	This start node prepares the creation of a new associated business object. In most cases, it simply creates a new Web-form and sets the creation template. If the creation/update process is very complex or uses a wizard-approach, you may consider using a separate pipeline, i.e., ViewDepartmentUser, ViewProductCustomAttribute. Hint: To reduce template complexity, do not use the same template for create and update.
Edit	This start node prepares the update of an associated business object. In most cases it determines the associated business object to be edited, creates a new Web-form and sets the update template.

Start Node	Description
Create	This start node creates a new associated business object. If the object could not be created successfully, the creation template is shown again (there should be an error message shown somewhere on the template).
Update	This start node updates an existing associated business object. If the object could not be updated successfully, the update template is shown again (there should be an error message shown somewhere on the template).

View<Object><AssociatedObject>

This type of pipeline is used whenever an associated object or object relation of a single business object is to be created/edited/deleted. This functionality can also be included into the corresponding list pipeline in case of a simple creation/update/deletion process.

Example: ViewDepartmentUser, ViewProductCustomAttribute, ViewProductAttachment

Table 56. Pipeline start node naming conventions

Start Node	Description
New	This start node prepares the creation of a new associated business object. In most cases, it simply creates a new Web-form and sets the creation template. Hint: To reduce template complexity, do not use the same template for create and update.
Edit	This start node prepares the update of an associated business object. In most cases, it determines the associated business object to be edited, creates a new Web-form and sets the update template.
Create	This start node creates a new associated business object. If the object could not be created successfully, the creation template is shown again (there should be an error message shown somewhere on the template).
Update	This start node updates an existing associated business object. If the object could not be updated successfully, the update template is shown again (there should be an error message shown somewhere on the template).
Select<Object>	Use this start node name in case of a wizard-like creation/update process. Example: SelectRole, SelectSupplier

Process Pipelines

To facilitate the reuse of process pipelines, they should be named according to the task they perform, and not after their position in a workflow or the business process they represent. Thus process pipelines can be called from different places.

Examples: CreateBasket, LoginUser

Different entry points to the pipeline or special result handling methods should be reflected by defining separate start nodes.

Example: ProcessPayment-AuthorizationFailed

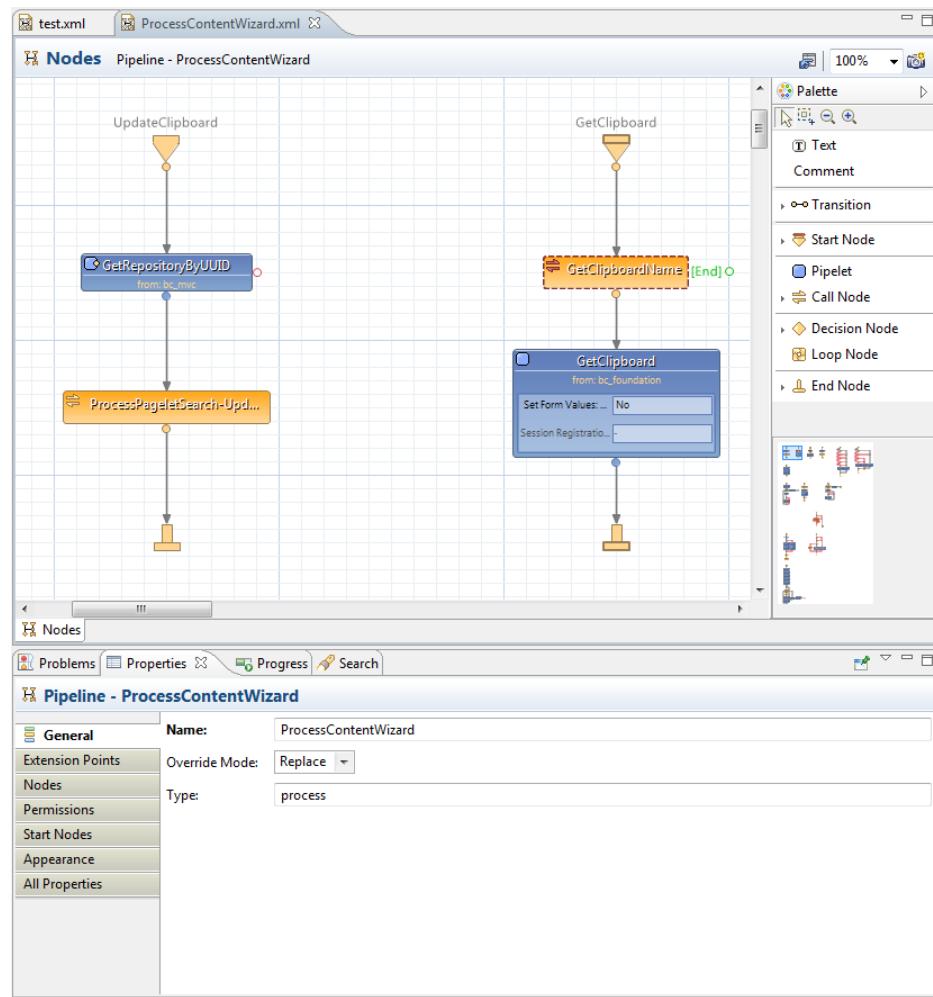
Visual Pipeline Editor

User Interface

The Visual Pipeline Editor provides a convenient graphical user interface for viewing, creating, and maintaining pipelines. In the editor, pipelines are visualized like flow charts describing the steps of a business process. When pipelines are executed, they begin with a start node and are executed step-by-step until they reach a stopping point, symbolized by an end node, a jump node, a stop node, or a template symbol (called an interaction node). Between the start and the end are the business logic contained in the pipelets, and the flow control logic that dictates whether, for example, a yes/no branch in the pipeline is needed, another pipeline must be called, or a template should be displayed.

When working with pipelines, you drag and drop icons, or nodes, within the workspace panel of the editor window. Transitions between nodes are drawn, either automatically or manually, where they attach at connectors. Click on any pipeline element to see a description, input and output parameters, and other information in the Properties View.

Node informations can be displayed in the graphical editor area via tooltip. Furthermore, pipeline nodes and transitions are type-specific colors and shapes.

Figure 120. Visual Pipeline Manager**Table 57. Toolbar Options**

Option	Description
	Export diagramm as an image.
	Expand palette.
	Collapse palette.
	Standard tool to select individual pipeline elements. To mark more than one element, press the Ctrl key and click on the elements to be selected.
	Marquee tool to select all elements within a certain area.
	Toggle buttons to zoom in the pipeline or zoom out the pipeline.
	Text tool to include comments to be displayed in the Pipeline Editor.

Palette

Depending on the pipeline type the palette supports different pipeline elements.

- 1. Toggle the desired pipeline element within the palette.**
- 2. Click in the graphical editor area to drop the element.**

Index Map

In the lower part of the palette there is an index map for easier navigation. The blue rectangle represents the current window of graphical editor. Drag and drop the rectangle for navigation.

Push Action

As described above, you can drag and drop single pipeline elements. The editor allows you to store various elements on a point above the other. Thus, the presentation is very confusing. To move several elements simultaneously, a push-action can be used.

- 1. Right-click the node.**

Context menu opens.

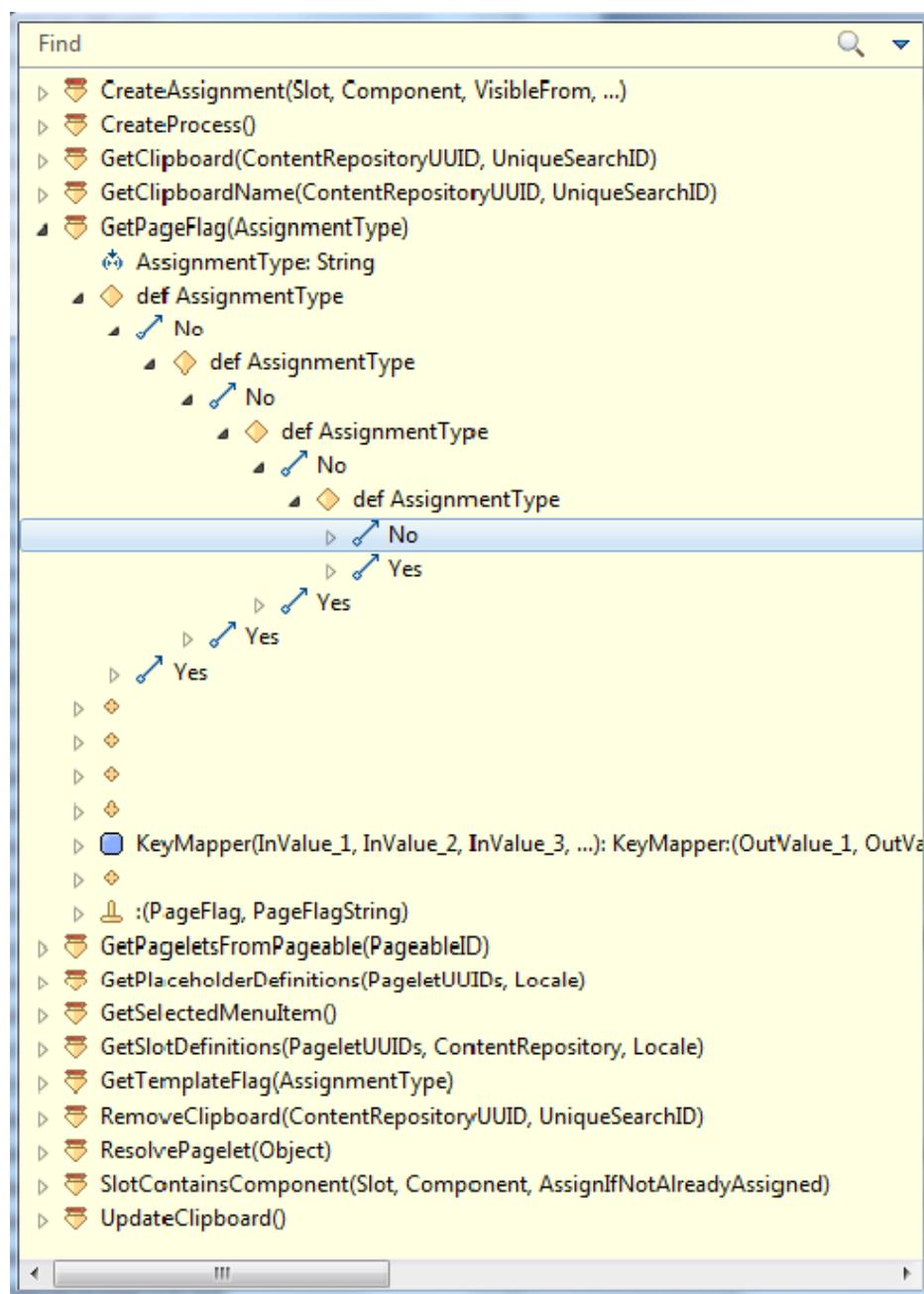
- 2. Select Push Node.**

Depending on the orientation of the pipeline a number of nodes is moved.

Quick Outline View

The Visual Pipeline Editor offers Quick Outline View for a pipeline. Press **Ctrl + 0** to see pipelines quick outline.

Press **Esc** or click outside the window to end the view.

Figure 121. Quick Outline

Quick outline offers a list of all start nodes of a pipeline in alphabetical order within a resizable pop-up window. In addition, quick outline provides a search field for start nodes. This search is useful for pipelines with many start nodes.

Quick outline offers a complete flip-tree data structure of the pipeline. You can expand or collapse data tree for navigation purposes. Double-click a node to end quick outline, but focus on the desired node within Visual Pipeline Editor.

Customize the Pipeline Editor

The look and feel of the Pipeline Editor can be customized in the following ways:

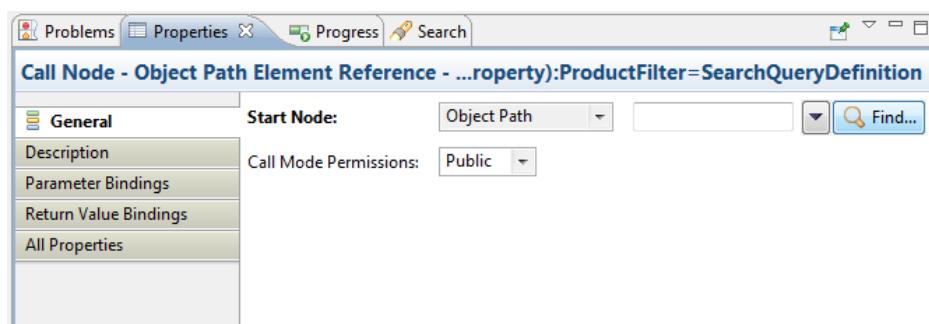
- Use the *Pipeline Editor Preferences* page to define grid size, join node usage and preferences for transitions.
- You can also modify the icon size for pipeline components as displayed on the palette, as well as hide or display descriptions for each node. To modify these settings, right-click on the palette displaying the pipeline components and select Settings from the context menu.

Content Assist

You can use content assist at various places when creating and modifying pipelines. To activate the content assist, press **Ctrl + Space**. The content assist displays a floating window with a list of possible options (such as dictionary values or templates) available at the current point. Content assist is signaled by the icon  when clicking on the item for which assistance is required.

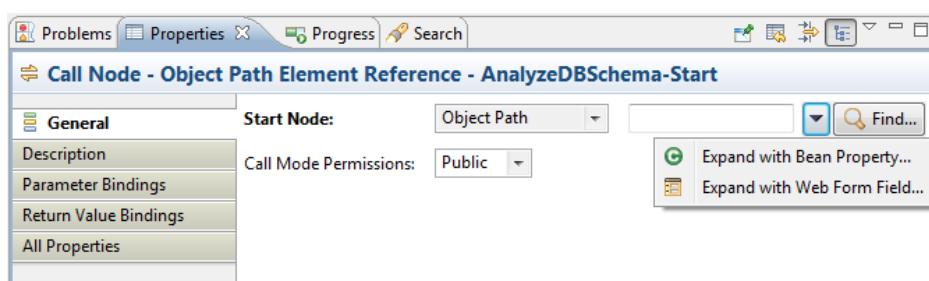
Search dialog is available to access available dictionary input parameter, including constants and object path expressions.

Figure 122. Search for Access Object Path Expressions

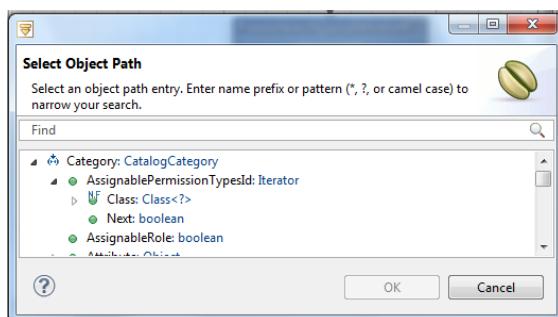


You can easily extend object path expressions with property from beans or web forms.

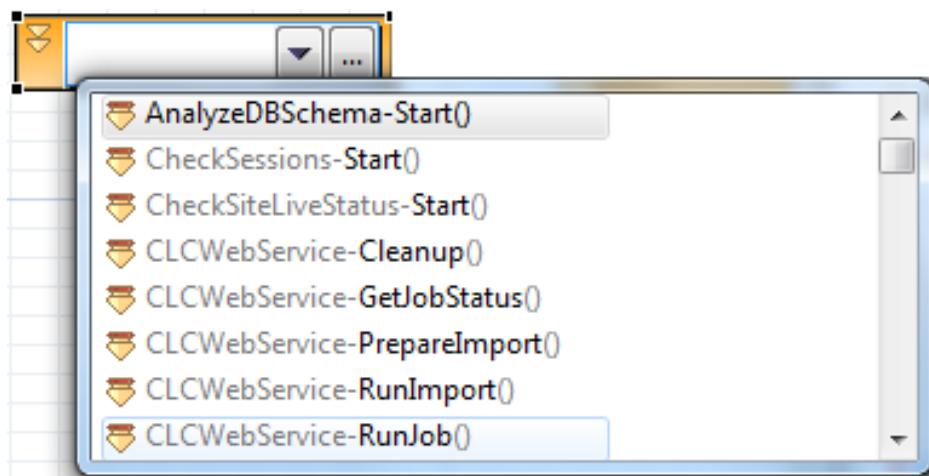
Figure 123. Expand Object Path



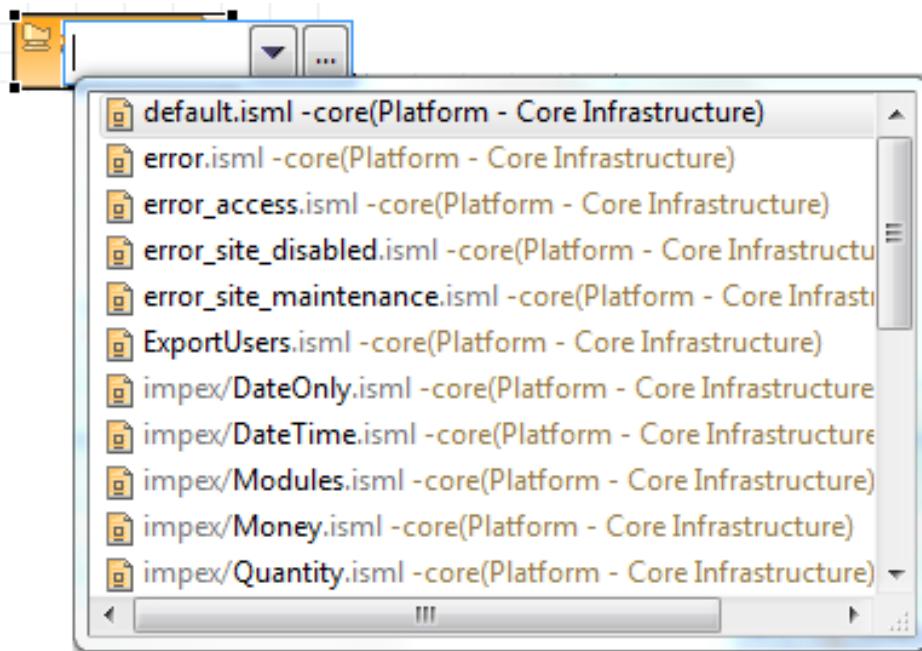
When selecting object path expressions, you can browse through the object path until the appropriate expression has been found.

Figure 124. Select Object Path Dialog

Content assist is available when editing call or jump nodes, indicating the start nodes of pipelines that can be accessed at this point.

Figure 125. Content Assist: Jump and Call Nodes

Content assist is available when editing interaction end nodes, indicating the templates that can be accessed at this point.

Figure 126. Content Assist: Interaction End Nodes

Pipeline Comparer

The pipeline comparer is used to compare two pipelines, pipeline A (displayed in the left pane) and pipeline B (displayed in the right pane). For example, you can compare a pipeline

- with an earlier version of the same pipeline, stored in the pipeline history,
- with a pipeline that overrides the current pipeline (overriding pipeline),
- with an entirely different pipeline.

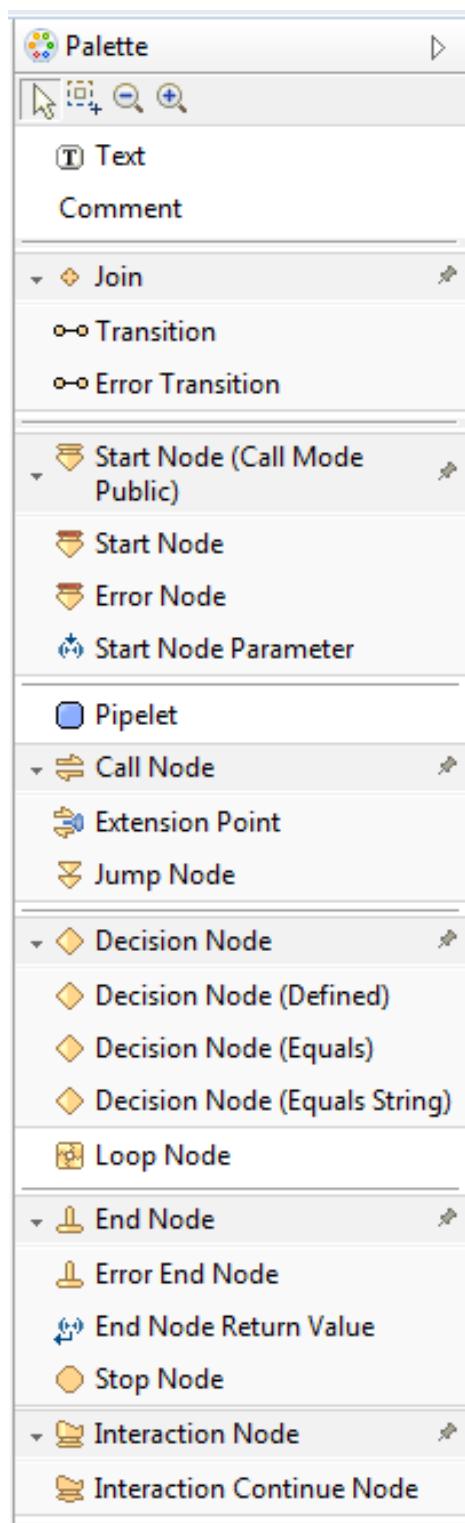
The following symbols are used to visualize pipeline differences:

Table 58. Pipeline Comparer Symbols

Symbol	Description
	The "plus" symbol is displayed on nodes, pipelets or transitions in pipeline A which are missing in pipeline B.
	This symbol is displayed on elements in pipeline A which have modified properties in pipeline B. For example, displayed on a transition in pipeline A which has a different transaction property set in pipeline B.
	Nodes or pipelets of pipeline B that are missing in pipeline A are crossed out and marked with a "minus" symbol.
	The discontinuous, light square is displayed on nodes or pipelets that have a corresponding element in the other pipeline.
	The continuous, dark square is displayed on nodes or pipelets of pipeline A that are missing in pipeline B.

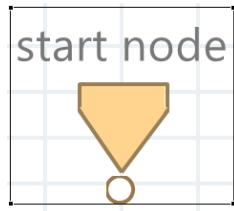
Pipeline Component Reference

Pipelines are composed of pipelet nodes and flow control nodes. While the pipelets perform the business logic, the flow control nodes are the decisive marking points in the pipeline flow. All available pipeline elements are displayed in the Pipeline Components palette of the Pipeline Editor. All pipeline nodes are linked by transitions, symbolized by arrows leading into and out of the nodes. The point at which a transition touches a node is symbolized by a small circle, called a connector.

Figure 127. Pipeline Components Palette

Start Node

Start nodes begin pipelines.

Figure 128. Start Node

Start nodes have the following behavior and qualities:

- A start node is required to launch any pipeline.
 - A given pipeline may also have multiple start nodes.
 - A start node begins the pipeline dictionary, or adds to an existing pipeline dictionary of sub-pipelines.
- A sub-pipeline is one that is called from within another pipeline, like a subroutine. A sub-pipeline usually has a single start node and single end node. Any changes the sub-pipeline makes to the pipeline dictionary affects the dictionary of the calling pipeline directly.
- Pipelines can have more than one start node.
 - If they do, each start node within a pipeline must have a unique name.
 - The start node name is used to identify the wanted pipeline.
- For example, `AddToBasket-Start1` calls the start node `Start1` of the pipeline `AddToBasket`.
- If parameters are passed to a pipeline via an HTTP request, these parameters, i.e., key names, must be defined in the start node properties.
- If a pipeline is called as a sub-pipeline (using a call node), then all dictionary values of the calling pipeline which are accessed by the sub-pipeline should be declared as start parameters.

Start node properties are displayed and edited in the Properties View. To view all properties:

1. Navigate to toolbar of Properties tab.

2. Select Filters icon

Selection Needed dialog starts.

3. Deselect all filters.

4. Confirm dialog with OK.

Properties view displays all properties unfiltered.

Start nodes have the following properties:

■ **Description (optional)**

A descriptive text that you can maintain for your own documentation purposes.

■ **Name (required)**

An external identifier for the start name; must be unique within the pipeline.

■ Session Mode (required; value = persistent | transient | volatile)

Defines a session type for the pipeline.

Persistent sessions write session information to the database, and therefore have session fail over. If the server should stop during a session, the information is still available. Persistent sessions are used in pipeline activity occurring on the application server. User login, ordering, payment, etc., require a persistent session with information written to the database.

CAUTION: If you set a pipeline to persistent session, do not use interaction continue nodes in the pipeline.

Transient sessions hold data in memory, but do not write it to the database. This means that pipelines in the transient mode do not have session fail over. A transient session would be used, for example, when an unrecognized user browses a catalog. Transient sessions can be upgraded to persistent sessions; for example, once the user stops browsing anonymously and logs in.

Volatile sessions are equivalent to not using a session at all; they start and stop the session within the course of the pipeline.

■ Call Mode (required; value = public | private | include)

Controls the way a pipeline may be accessed via this start node.

Public mode permits access from an external source using (for example) HTTP requests. Public start nodes are marked with a little globe (🌐).

Private mode permits access from an internal source such as another pipeline, using call or jump nodes.

Include mode allows access via remote include calls (Web Adapter includes, "<WAINCLUDE>").

■ Strict

Strict pipelines will not pass the entire pipeline dictionary. Strict start nodes define their required input parameters. In this way, quasi an interface is created. Thus the reuse is improved.

■ Visibility

The visibility setting offers a possibility to restrict availability of the pipeline node. This setting is not supported yet.

Table 59. Visibility Settings

Property	Description
Cartridge	The node is accessible only from the current cartridge.
Private	The node is accessible only from the current pipeline.
Protected	The node is accessible only from the current pipeline and overwriting pipelines, whose overwrite mode is set to inherit.
Public	There are no restrictions on access to the pipeline.

■ Parameters

Parameters tab allows to manage node's parameters. Via parameters tab, parameters can be created, added, removed and sorted.

■ Permissions

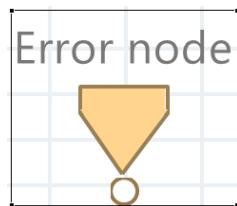
This security setting determines what permissions a user must have to call this pipeline.

Note that, the Permissions passed unlinked. A user needs all specified permissions. If the setting is *not* set, the pipeline *cannot* be called. Setting the context element to NONE means that no permission necessary.

Error Node

Error nodes are special start nodes with name "Error". Error nodes are used to define pipeline-specific error pipelines.

Figure 129. End Node



Error node properties are displayed and edited in the Properties View. To view all properties:

1. Navigate to toolbar of Properties tab.

2. Select Filters icon

Selection Needed dialog starts.

3. Deselect all filters.

4. Confirm dialog with OK.

Properties view displays all properties unfiltered.

Error nodes have the following properties:

■ Description (optional)

A descriptive text that you can maintain for your own documentation purposes.

■ Name (required)

Required: Name has to be "Error".

■ Session Mode (required; value = persistent | transient | volatile)

Defines a session type for the pipeline.

Persistent sessions write session information to the database, and therefore have session fail over. If the server should stop during a session, the information is still available. Persistent sessions are used in pipeline activity occurring on the application server. User login, ordering, payment, etc., require a persistent session with information written to the database.

CAUTION: If you set a pipeline to persistent session, do not use interaction continue nodes in the pipeline.

Transient sessions hold data in memory, but do not write it to the database. This means that pipelines in the transient mode do not have session fail over. A transient session would be used, for example, when an unrecognized user

browses a catalog. Transient sessions can be upgraded to persistent sessions; for example, once the user stops browsing anonymously and logs in.

Volatile sessions are equivalent to not using a session at all; they start and stop the session within the course of the pipeline.

■ Call Mode

Error nodes should be private to disallow external access.

■ Strict

Strict pipelines will not pass the entire pipeline dictionary. Strict error nodes define their required input parameters. In this way, quasi an interface is created. Thus the reuse is improved.

■ Visibility

The visibility setting offers a possibility to restrict availability of the pipeline node. Still not fully supported.

Table 60. Visibility Settings

Property	Description
Cartridge	The node is accessible only from the current cartridge.
Private	The node is accessible only from the current pipeline.
Protected	The node is accessible only from the current pipeline and overwriting pipelines, whose overwrite mode is set to inherit.
Public	There are no restrictions on access to the pipeline.

■ Parameters

Parameters tab allows to manage node's parameters. Via parameters tab, parameters can be created, added, removed and sorted.

■ Permissions

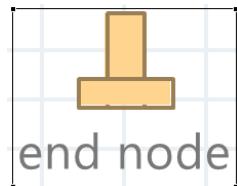
This security setting determines what permissions a user must have to call this pipeline.

Note that, the Permissions passed unlinked. A user needs all specified permissions. If the setting is *not* set, the pipeline *cannot* be called. Setting the context element to NONE means that no permission necessary.

End Node

End nodes are used to end pipelines and sub-pipelines without invoking a template.

Figure 130. End Node



End nodes do not clear the pipeline dictionary, and are thus often used to end sub-pipelines. Sub-pipelines usually end with end nodes; storefront pipelines always end with an interaction node or a jump node.

The end node name property gives each end node a unique identifier. It is used in the case of a sub-pipeline with branches that result in multiple end nodes (see *Call Node*).

The end node name property is also used in conjunction with debugging pipelines.

End nodes have the following properties:

■ **Description (optional)**

A descriptive text that you can maintain for your own documentation purposes.

■ **Name**

The name is an identifier that can be used multiple times within a pipeline.

■ **Strict**

Strict pipelines will not pass the entire pipeline dictionary. Strict end nodes define the output parameters, which they pass on. In this way, quasi an interface is created. Thus the reuse is improved.

■ **Return Value**

For strict end nodes: defines which return values are returned in the pipeline dictionary.

Stop Node

A stop node ends the execution of a pipeline.

Figure 131. Stop Node



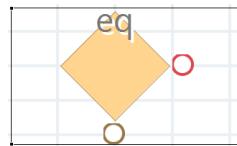
It is used to handle error conditions in a sub-pipeline that require all calling pipelines to also be aborted. Transactions are rolled back when a stop node is encountered.

A stop node functions as an "emergency break" since it ends all pipelines. Conversely, end nodes only stop the execution of the sub-pipeline, and processing continues one level higher, in the calling pipeline. Without using a stop node, the same result can only be achieved by throwing an exception from within a pipelet.

Decision Node

A decision node compares two values in the pipeline dictionary or checks for a value in the pipeline dictionary. The result directs the flow of the pipeline into a "yes" branch or a "no" branch.

Figure 132. Decision Node



Decision nodes work with key-value pairs in the pipeline dictionary to direct the pipeline flow in two branches, based on whether the result is true ("yes" branch) or false ("no" branch).

For example, a decision node might check whether the payment status of an order is authorized. If the status is 1, authorized, the "yes" branch is used and the pipeline continues, e.g. the shipping status might be checked next. If the status is 0, not authorized, the "no" branch is used (for example, a message may be called to alert the user that payment has not been authorized).

Decision nodes can check for the existence of any values, whether simple strings or complex objects. They can also compare values that are stored as simple strings or Java number objects.

A decision node is labeled automatically by choosing its operator.

Decision node properties are displayed and edited in the Properties View. To view all properties:

- 1. Navigate to toolbar of Properties tab.**

- 2. Select Filters icon .**

Selection Needed dialog starts.

- 3. Deselect all filters.**

- 4. Confirm dialog with OK.**

Properties view displays all properties unfiltered.

NOTE: Depending on the comparison operator, not all properties listed below may be visible.

Decision nodes have the following properties:

- Description (optional)**

A descriptive text to be used for your own documentation purposes.

- Condition Key (required)**

The dictionary key to compare.

- Condition Item (required)**

Enter the name of a dictionary key in this field to compare its value with the condition key value in the pipeline dictionary. If the key does not exist in the pipeline dictionary, a `PipelineExecutionException` is thrown.

- Condition Value**

Enter a value in this field to be compared with the value associated with the key specified in the condition key field.

- Operator**

Use one of the following operators:

Table 61. Decision Node Comparison Operators

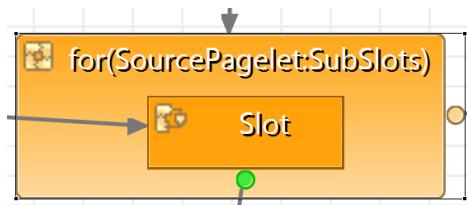
Operator	Description
(String) Equals	For string comparisons.
(String) Not Equals	For string comparisons.
(String) Empty	For string comparisons.
Equals	For numeric comparisons.
Not Equals	For numeric comparisons.
Lower Than or Equals <=	For numeric comparisons.

Operator	Description
Lower Than <	For numeric comparisons.
Greater Than or Equals >=	For numeric comparisons.
Greater Than >	For numeric comparisons.
Defined	Checks existence of an item.
Undefined	Checks existence of an item.
Has Elements (loop)	Checks whether an iterable object has elements.

Loop Node

Loop nodes are used to perform logic on all objects in an iterator. The iterator has already been stored in the pipeline dictionary.

Figure 133. Loop Node



■ Entry Point

The entry point of the node, marked by the arrow pointing on the outer element takes the first element from the iterator and stores it in the Element dictionary key. The DO transition is then followed.

■ DO (green circle)

The transition from the DO connector (green circle) performs the desired logic on the element and then leads back to the loop node through the LOOP connector.

■ LOOP

LOOP connector, represented by the arrow pointing on the inner element, checks the iterator to see if any elements remain to be iterated through. If so, the next element from the iterator is stored in the Element dictionary key and the DO transition is followed. If not, the NEXT transition is followed.

■ NEXT

This transition is followed in order to end the loop once all the elements in the iterator have been run through, at which point the iterator is reset to the first element. The NEXT connector is represented by the yellow circle.

Loop node properties are displayed and edited in the Properties View. To view all properties:

1. **Navigate to toolbar of Properties tab.**

2. **Select Filters icon**

Selection Needed dialog starts.

- 3. Deselect all filters.**
- 4. Confirm dialog with OK.**

Properties view displays all properties unfiltered.

Table 62. Loop Node Properties

Property	Description
Description	Optional: Enter a description for your own documentation purposes.
General Loop	The name of the iterator that is stored in the pipeline dictionary.
Loop Entry	The current element is written to this key during each iteration of the loop.

Interaction End Node (Interaction Node)

The interaction nodes (continue and end) are used to call a storefront template at a particular point in a pipeline. We recommend using interaction end nodes exclusively, as interaction continue nodes cannot be made session fail over-safe.

Figure 134. Interaction End Node



Interaction end nodes have the following qualities and behaviors:

- Interaction end nodes call a storefront template to be displayed.
- Interaction end nodes can display templates with known names or the template may be dynamic, i.e., showing different templates depending on the circumstance.
- The pipeline always ends and the pipeline dictionary is cleared after an interaction end node.

Compare this with the behavior of interaction continue nodes (see *Interaction Continue Node*).

Interaction end node properties are displayed and edited in the Properties View. To view all properties:

- 1. Navigate to toolbar of Properties tab.**

- 2. Select Filters icon .**

Selection Needed dialog starts.

- 3. Deselect all filters.**

- 4. Confirm dialog with OK.**

Properties view displays all properties unfiltered.

Interaction end nodes have the following properties:

■ **Description (optional)**

A descriptive text you can use for your own documentation purposes.

■ Template

Referenced Name: required for non-dynamic template. To use this option, the template must have a constant name that is always used.

Object Path: required for dynamic templates. Use this option if the template to be used is based on variable data determined by the selected pipeline.

■ Interaction Processor

Choose from `HTMLInteractionProcessor` or `XMLInteractionProcessor`. By default, `HTML` processing is selected.

■ Dynamic (value = true | false)

Choose `false` if the template to be called is known; that is, the same template is always used and does not depend on variable information determined by the pipeline (see *Template | Referenced Name* above).

Choose `true` if the template to be used is based on variable information determined somewhere else within the pipeline dictionary. For example, if a template is selected by the method `SetDynamicTemplate`.

■ Buffered Template (value = true | false)

Defines the way the Web adapter delivers templates, either in *buffered* or *streamed* mode.

The default is buffered mode (`true`).

In order to save system memory, enable streaming mode by setting this property to `false`.

Also, for templates that rely on the output from XML pipelines for catalog data, or for templates with a lot of content, the streamed response delivered instantaneously improves storefront response time compared with the buffered default option.

■ Transaction Required (value = false | true)

"Transaction" in this context refers to a specified database transaction. Generally, transactions are specified either within pipelets or placed on connectors; they are very rarely used directly within templates.

Choose `false` for most situations. This is also the default value.

Choose `true` if a new database transaction is opened within the template itself. An example for this kind of behavior would be a template containing *JavaScript* code that requires writing data to the database. However, adding this kind of logic to templates is not encouraged in general.

Interaction Continue Node

Interaction continue nodes invoke templates just as interaction end nodes do. However, note the following important caveat:

CAUTION: Interaction continue nodes cannot be made session fail over safe. Therefore, it is recommended that you always use interaction end nodes instead!

Figure 135. Interaction Continue Node

Interaction continue nodes do not clear the pipeline dictionary, but can be used as start nodes for the next pipeline.

Interaction continue node properties are displayed and edited in the Properties View. To view all properties:

1. Navigate to toolbar of Properties tab.

2. Select Filters icon .

Selection Needed dialog starts.

3. Deselect all filters.

4. Confirm dialog with OK.

Properties view displays all properties unfiltered.

Interaction continue nodes have the following properties:

■ **Description (Optional)**

Enter a description for your own documentation purposes.

■ **Template**

Referenced Name: required for non-dynamic template. To use this option, the template must have a constant name that is always used.

Object Path: required for dynamic templates. Use this option if the template to be used is based on variable data determined by the selected pipeline.

■ **Interaction Processor**

Choose from `HTMLInteractionProcessor` or `XMLInteractionProcessor`. By default, `HTML` processing is selected.

■ **Dynamic (value = true | false)**

Choose `false` if the template to be called is known; that is, the same template is always used and does not depend on variable information determined by the pipeline (see *Template | Referenced Name* above).

Choose `true` if the template to be used is based on variable information determined somewhere else within the pipeline dictionary. For example, if a template is selected by the method `SetDynamicTemplate`.

■ **Buffered Template (value = true | false)**

Defines the way the Web adapter delivers templates, either in *buffered* or *streamed* mode.

The default is buffered mode (`true`).

In order to save system memory, enable streaming mode by setting this property to `false`.

Also, for templates that rely on the output from XML pipelines for catalog data, or for templates with a lot of content, the streamed response delivered instantaneously improves storefront response time compared with the buffered default option.

■ **Transaction Required (value = false | true)**

"Transaction" in this context refers to a specified database transaction. Generally, transactions are specified either within pipelets or placed on connectors; they are very rarely used directly within templates.

Choose `false` for most situations. This is also the default value.

Choose `true` if a new database transaction is opened within the template itself. An example for this kind of behavior would be a template containing *JavaScript* code that requires writing data to the database. However, adding this kind of logic to templates is not encouraged in general.

■ **External Name (Required)**

Assign a unique name to the start portion of the interaction continue node. On the workspace, this name is joined with the name of the template. When template designers program a URL call to this node, they use the pipeline name with the start name.

■ **Session Mode (required; value = persistent | transient | volatile)**

Defines a session type for the pipeline.

Persistent sessions write session information to the database, and therefore have session fail over. If the server should stop during a session, the information is still available. Persistent sessions are used in pipeline activity occurring on the application server. User login, ordering, payment, etc., require a persistent session with information written to the database.

CAUTION: If you set a pipeline to persistent session, do not use interaction continue nodes in the pipeline.

Transient sessions hold data in memory, but do not write it to the database. This means that pipelines in the transient mode do not have session fail over. A transient session would be used, for example, when an unrecognized user browses a catalog. Transient sessions can be upgraded to persistent sessions; for example, once the user stops browsing anonymously and logs in.

Volatile sessions are equivalent to not using a session at all; they start and stop the session within the course of the pipeline.

Call Node

Call nodes allow a pipeline to use other pipelines and maintain a single pipeline dictionary. When a call node is used, the called pipeline is executed and completed, then processing returns to the call node in the original pipeline and continues.

Figure 136. Call Node

Call nodes have the following qualities and behaviors:

- **A call node invoke another pipeline from within a pipeline.**

When this relationship between the two pipelines is established, the called pipeline is referred to as a sub-pipeline. Double-click a defined call node to open the target pipeline in a new editor window.

- **Multiple call nodes and calls in sub-pipelines can be used.**

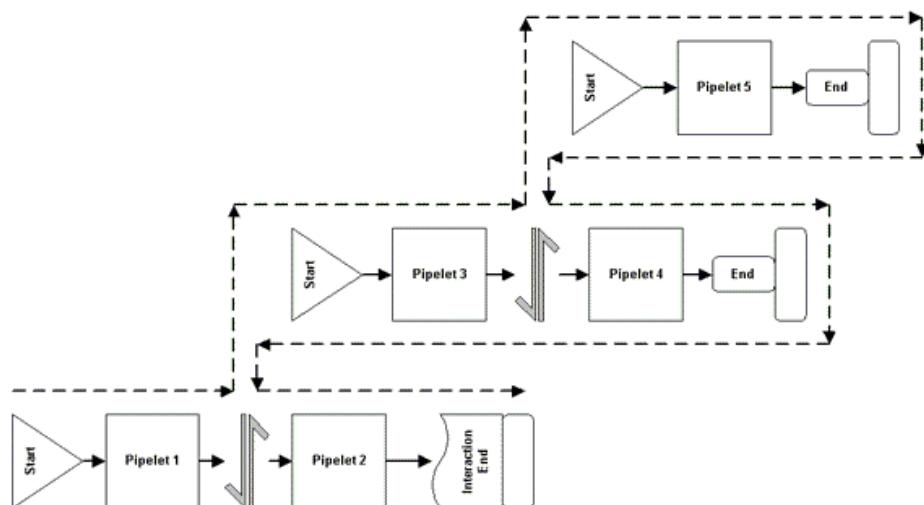
The execution flow always runs through the sub-pipeline, then returns to the call node and continues through the calling pipeline.

- **Call nodes can have multiple exit points.**

A sub-pipeline that branches can end its execution at different end nodes which each have a unique name (see *End Node*). A call node can have multiple exit points which represent these different end nodes.

- **Call nodes must have the pipeline and start node name of the called pipeline defined in the node's Properties View.**

- **For security reasons, by default, dynamic call nodes are only allowed to call *public* start nodes. If you want to call a *private* start node, you need to explicitly grant this right to the call node.**

Figure 137. Data Flow Through Called Pipelines

Call node properties are displayed and edited in the Properties View. To view all properties:

1. **Navigate to toolbar of Properties tab.**

2. **Select Filters icon .**

Selection Needed dialog starts.

3. Deselect all filters.

4. Confirm dialog with OK.

Properties view displays all properties unfiltered.

Call nodes have the following properties:

■ **Description (Optional)**

Enter a description for your own documentation purposes.

■ **Start Node**

Referenced Name: required for non-dynamic start node. To use this option, the start node must have a constant name that is always used.

Object Path: required for dynamic start node. Use this option if the start node to be used is selected based on variable data determined by the selected pipeline.

■ **Parameter Bindings**

The values assigned to parameters.

■ **Return Value Bindings**

The values assigned to return values.

Jump Node

Jump nodes are used to jump the business flow from one pipeline to another, without losing the pipeline dictionary. The difference between call and jump nodes is that jump nodes do not return the business flow to the original pipeline.

Figure 138. Jump Node

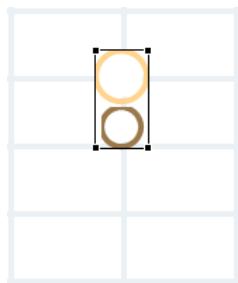


Jump node properties are defined very similar to call node properties. Jump nodes have the following qualities and behaviors:

- **Jump nodes transfer the business flow from one pipeline to another while maintaining the active pipeline dictionary.**
 - **Like call nodes, jump nodes must have the pipeline and start node name of the called pipeline defined in the node's Properties View.**
- See *Call Node*.
- **For security reasons, by default, dynamic jump nodes are only allowed to call *public* start nodes. If you want to call a *private* start node, you need to explicitly grant this right to the jump node.**
 - **Double-click a defined jump node to open the target pipeline in a new editor window.**

Join Node

Join nodes bring together two or three flows of a pipeline.

Figure 139. Join Node

For example, an order pipeline may use a decision node in confirming the user registration. If the user is registered, the pipeline continues; if not, the pipeline can register the user and then continue placing the order. A join node can bring the "no" branch back to the order creation part of the pipeline.

Transitions

Transitions are the arrow symbols connecting nodes and pipelets together in the workspace. Transitions can be used to set a database transaction explicitly at a particular point in a pipeline.

Transition properties are displayed and edited in the Properties View. To view all properties:

1. Navigate to toolbar of Properties tab.

2. Select Filters icon .

Selection Needed dialog starts.

3. Deselect all filters.

4. Confirm dialog with OK.

Properties view displays all properties unfiltered.

Transitions have the following properties:

■ **Description (Optional)**

Enter a description for your own documentation purposes.

■ **Transaction automatically get a qualified name composed by the pipeline name and the name of the preceding node or its connector.**

■ **Transaction Handling (Optional)**

Used to set explicit database transactions. The possible settings are summarized in the following table.

Table 63. Transaction Property Values

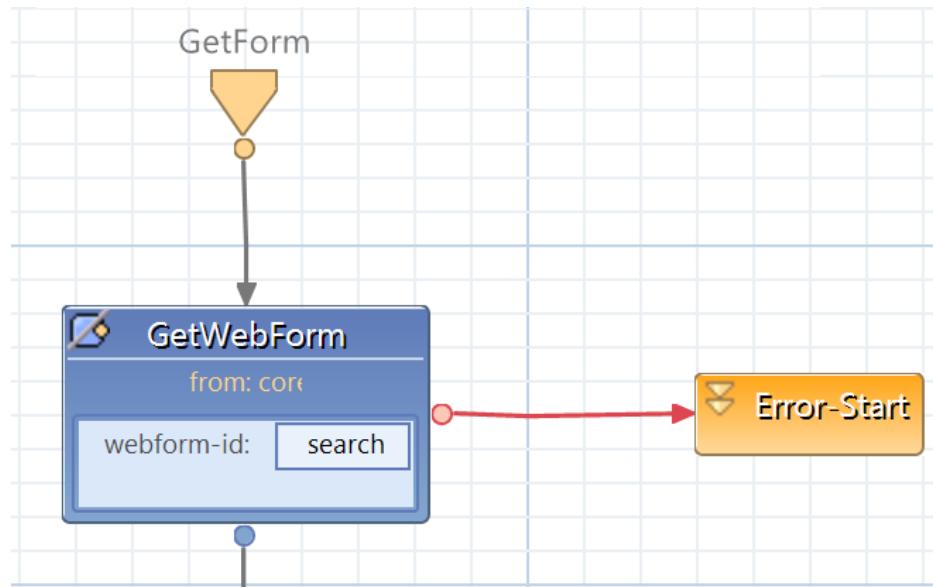
Value	Description
None	Default setting. If you do not need an explicit transaction, this value is set automatically and does not need to be changed.
Begin 	Ensures that a transaction is active. If a transaction is already active, the nested transaction counter is incremented.

Value	Description
Commit 	Decrements the nested transaction counter and commits the transaction to the database when the outermost nested transaction is reached. If no transaction is active, nothing happens.
Flush Transaction (Store + Cache Clear) 	Stores the current transaction to the database without committing it. Additionally the transactional ORM cache is cleared.
Rollback Transaction 	Used to undo any change done on processed data to the beginning of the outermost nested transaction.
Store Transaction 	Stores the current transaction to the database without committing it.
Transaction Savepoint (Commit + Begin) 	Used to write any data that has been changed within the current transaction. Any following rollback will only be effective to this point.

Connectors

Connectors are the points where transitions enter or exit a node or pipelet, and are symbolized by small circles or the node itself.

Figure 140. Connectors



Every node except the start node has at least one entry point, represented by the node/pipelet itself; every node except the end node, jump node, and interaction end node has at least one exit point (next).

Disconnected connectors are shown as empty circles. Connected connectors are shown as a filled circle. Error connectors and "no"-next-successors of decision-nodes shown in red. Loop entry's next connectors are shown in green. Pipelet default connectors are shown in blue.

ISML Template Development

Templates and ISML

What Are Templates?

Templates form the heart of the Intershop 7 presentation layer which is responsible for converting the results of a business process into a response that can be sent back to the client. Typically, this results in an HTML page being displayed through the Web-front.

Each such response is based on a template. Templates define the skeleton of a Web page: the page structure, the style used on the page, and static content elements. In addition, templates often contain variable elements to accommodate dynamic content, i.e. content specific to a particular request, user, and/or session. Dynamic content is based on information stored in the pipeline dictionary and added to the page at the time the response is actually generated.

Templates and ISML

Templates make extensive use of a proprietary language extension referred to as Intershop Markup Language (ISML):

- ISML is an extended markup language based on JSP and is primarily used to add dynamic content to a page.
- ISML provides means to retrieve data from the pipeline dictionary, to trigger the execution of pipelines, and to embed templates inside other templates.
- Accounting for the variable parts of a template, ISML makes it possible to derive an infinite number of pages from a single template, it also provides an essential prerequisite for features such as personalization.

For a detailed introduction to ISML, see *Intershop Markup Language (ISML)*.

Template Types

Templates generate different types of responses to a request. The following template types can be distinguished:

■ Web Front Templates

Web front pages are standard HTML documents sent by a server to a user (usually through a browser). Most templates for an Intershop 7 Web front are used to control the page display. This type of template generates "text/html" documents. The content type, `HTML` or `plain`, is set automatically by an interaction-end or interaction-continue node in the pipeline. A Web front page that is generated from an HTML template is sent to the customer's browser using HTTP. The content type of that document (`Web front page`) is specified in the HTTP header and is set to `text/html` by default.

■ Templates with Static Content

These templates are standard HTML documents containing information that is not likely to change, such as a General Terms and Conditions section. Instead of using templates with static content, you should rather store static files as such, namely in the `static` folder of a site or a cartridge. This increases the performance of your system and saves resources.

■ Templates with Binary Content

A special ISML tag `<ISBINARY>` is available which makes it possible for pipelines to output binary content, e.g. a graphics file or a PDF document. The appropriate content type (e.g. `application/pdf`) has to be set by the template programmer in the template defining or referencing the binary content. See `</ISBINARY>` for details.

■ E-Mail Templates

E-mails are plain-text messages sent from a server to an e-mail client. The browser or e-mail client needs to know what sort of document it is about to receive in order to interpret and display the data correctly. E-mail documents send order confirmations or invoices to your customers or suppliers. The content type, `e-mail`, is set automatically by the `SendMail` pipelet. E-mails generated by this pipelet are sent to another server using the SMTP protocol. The type of e-mail templates is set to `text/plain` by default. Even though many popular e-mail clients are able to display HTML code, plain text is advised for building mail templates, ensuring that any recipient can display the message correctly. If you want to send e-mail containing HTML code, set the content type explicitly to `text/html` in the Intershop Studio, otherwise an e-mail client displays the HTML source code in the message. See `</ISCONTENT>` for details on setting the type explicitly.

Template Deployment

The storage location of templates and static content (such as images) depends on whether they are intended for system-wide or site-specific deployment.

■ System-Wide Template Deployment

Templates designed for system-wide deployment are stored in the folder `<IS_HOME>\share\system\cartridges\<cartridge_name>\release\templates`, with an additional subfolder depending on the locale the templates are designed for (e.g., `\templates\en_US`).

■ Site-Specific Template Deployment

In order to enable site-specific customization, templates can be bound to a particular site which means that they are available to that particular site only. Templates designed for site-specific deployment are stored in the folder <IS_HOME>\share\sites\<site_name>\<version_id>\templates\en_US, with an additional subfolder depending on the locale the templates are designed for (e.g., \templates\en_US).

NOTE: Site-specific resources always take precedence over resources deployed to a cartridge. Hence, when processing a request, the system first searches for site-specific versions of a template before attempting to locate a template in the cartridge directory area.

Templates and Localization

Locales

An Intershop 7 Web front application (storefront or back office) can exist in different variations for different locales. Each locale of a Web front has its own set of templates stored in uniquely named folders below the \templates folder, such as \templates\en_US or \templates\de_DE. Static files stored in the sub-folder \static are arranged according to the same logic.

By default, Intershop 7 supports the locales "English (United States)" (en_US) and "German (Germany)" (de_DE). The en_US locale serves as the system lead locale.

If you want to support other locales, change locales, or share templates between two locales, you have to create the new locale and create the template folder to store the templates in.

For information on how to create a new locale, see the \prod Administration and Configuration Guide.

Directory names for locale-specific templates consist of a language and a country code as specified in the standard ISO 639 (language) and ISO 3166 (country). The table below shows a number of sample folder names:

Table 64. Sample locales and their folder names

Locale	Folder Name
English/United Kingdom	en_GB
English/Ireland	en_IE
German/Austria	de_AT
Spanish/Argentina	es_AR

The Default Directory

A special directory exists for templates: the default / directory. Templates stored in this folder serve as a fallback. They are called in case a locale-specific template cannot be found. In an Intershop 7 standard installation, the template set for the lead locale en_US is stored in the default folder, and not in the folder.

Fallbacks for the Template Lookup Process

When searching for a template or static content resource, the following fallback rules apply:

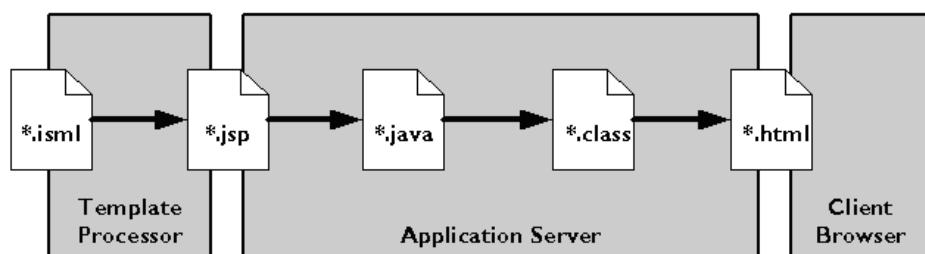
- 1. Site-specific resources take precedence over resources stored in a cartridge.**
- 2. Within a site or cartridge, the system first searches for the resource in a folder with matching language and country code (e.g. en_US or de_DE)**
- 3. If not found, the system searches for the resources in a folder with matching language code (e.g. en or de). Hence, resources which should be used for different locales sharing the same language code (e.g. de_DE, de_AT) can be deployed in a folder using the language code only (e.g. de).**
- 4. If not found, the system searches for the resources in a folder with language and country code matching the system's lead locale. For example, if the current locale is de_DE and the lead locale is en_US, the system searches the folder en_US for resources it cannot find in the folders de_DE or de.**
- 5. If not found, the system searches for the resources in a folder with the language code matching the language code of the lead locale (e.g. en or de).**
- 6. If not found, the system searches for the resources in the default folder.**

The Template Conversion Process

When a Web front page is requested by a browser, the page must be compiled, generating an HTML file from an .isml file, using Java Server Pages (JSP) and servlet creation as intermediary events. This process is transparent to both the template designer and the user.

Intershop 7 performs the initial stages of the conversion process (conversion if ISML to Java class file) the first time a Web front page is requested. Alternatively, templates can be all precompiled, e.g. before server startup. Conversion of class files to HTML is performed at runtime.

Figure 141. Template Conversion Process



Converting ISML Templates to Java Classes

The initial stages are as follows:

■ Converting ISML Templates to JSP Files

The template processor's ISML-to-JSP compiler reads the ISML templates and translates them to JSP files. During the conversion process, ISML is translated into Java code.

■ Compiling JSP Files to Java Source Files

The page compiler of the application server merges inline Java code with HTML, producing a Java source file (*.java). The resulting file only contains Java source code, with HTML embedded into Java print-like statements.

■ Compiling Java Source Files to Java Class Files

The Java source files are finally compiled into Java class files that can be executed by the server.

Converting Java Classes to HTML

If a page requested by the client cannot be served from the page cache, the Intershop 7 servlet engine executes the Java class files which have been compiled from the ISML templates that make up the requested page. The Java class files produce the content that is returned to the client.

Static and Dynamic Content Conversion

Because the template files are built from a combination of HTML and ISML tags and expressions, the final page presented in the Web front contains code processed and converted at runtime to pure HTML. Other elements of the page, such as the HTML headings and text, images called with HTML tags, or any other part of the page that is not affected by the conversion process, is considered static content.

In contrast, when ISML tags call variables to generate a specific list at runtime, e.g., a list of products, they look up object properties in the pipeline dictionary to present product details or other session-dependent data "on the fly". URLs are another type of information that the system parses at runtime to compose content that changes from session to session. These are examples of dynamic content.

Character Encoding in the Conversion Process

This section describes the character encoding rules which apply when an ISML template is compiled. The section also discusses character encoding rules used to interpret content (such as form data or post data) sent from a client to an Intershop 7 system.

Character Encoding for Reading ISML Templates

Before converting ISML templates to JSP files, the ISML-to-JSP compiler first reads the ISML templates which are going to be processed. When reading the ISML

templates, the ISML-to-JSP compiler uses the following fallbacks to determine the ISML template file encoding:

1. Read the first two bytes of the ISML file.

If the file starts with the byte sequence "0xFF 0xFE", the file was written in the Unicode little-endian format => UTF-16LE.

If the file starts with the byte sequence "0xFE 0xFF", the file was written in the Unicode big-endian format => UTF-16BE.

2. Read the first three bytes of the ISML file.

If the file starts with the byte sequence "0xEF 0xBB 0xBF", the file was written in the Unicode format => UTF8.

3. Read the first 1024 bytes of the file.

If the file prefix contains an <ISCONTENT> tag, check if a "charset" attribute is set with this tag. Take the value from the "charset" attribute and check if it describes a valid HTML encoding. If it does, convert the HTML encoding into a Java encoding and use this encoding.

If no <ISCONTENT> "charset" attribute is found, use the default encoding of the underlying operating system as determined by the Java function call System.getProperty("file.encoding").

Character Encoding in ISML-to-JSP Conversion

After an ISML template has been read from the file, it is converted to a JSP file which is stored in the pagecompile directory of the cartridge to which the template belongs. With respect to character encoding, it is important that the Java file encoding (which is used to write the JSP file) and the JSP page header encoding value (e.g. <@page ... contentType="...;charset=(encoding)">=) refer to the same character set.

Note that the JSP page header encoding is an HTML encoding name, while the Java file encoding is a Java encoding name. For instance, "windows-1256" is an HTML encoding and "Cp1256" the corresponding Java encoding. Both refer to the same character set.

Determining the MIME Type of the ISML File

The JSP page encoding is dependent on the MIME type of the ISML template. Therefore, the MIME type needs to be determined first, using the following fallback rules:

1. **Read the first 1024 bytes of the file. If the file prefix contains an <ISCONTENT> tag, check if a "type" attribute is set at this tag. Take the value from the "type" attribute.**
2. **If no <ISCONTENT> "type" attribute is found, check the header of the ISML file. If there is an XML header like (<?xml version="1.0" ...?>) in the first 1024 Bytes of the ISML file, the MIME type is "text/xml".**
3. **If no XML header was found, the default MIME type "text/html" is assumed.**

Determining the JSP Page Encoding

Dependent on the MIME type, the JSP page encoding can now be determined, using the following fallback rules:

- **The MIME type is "text/html"**
 1. **Use the HTML character encoding defined in the appserver.properties file by the key intershop.template.DefaultContentEncoding.**
 2. **If the property intershop.template.DefaultContentEncoding does not describe a valid HTML character encoding, use UTF-8 as default value.**
- **The MIME type is "text/xml"**
 1. **Check whether the XML header includes an encoding attribute describing a valid HTML character encoding. If it does, use this encoding.**
 2. **Use the HTML character encoding defined in the appserver.properties file by the key intershop.template.DefaultContentEncoding.**
 3. **If the property intershop.template.DefaultContentEncoding does not describe a valid HTML character encoding, use UTF-8 as default value.**
- **The MIME type is not one of the above and starts with "text/"**
 1. **If the file prefix contains an <ISCONTENT> tag, check if a "charset" attribute is set at this tag. Take the value from the "charset" attribute**

and check if it describes a valid HTML encoding. If it does, use this character encoding to write the JSP file.

2. **Use the HTML character encoding defined in the appserver.properties file by the key intershop.template.DefaultContentEncoding.**
 3. **If the property intershop.template.DefaultContentEncoding does not describe a valid HTML character encoding, use UTF-8 as default value.**
- **All other MIME types**
1. **Use the HTML character encoding defined in the appserver.properties file by the key intershop.template.DefaultContentEncoding.**
 2. **If the property intershop.template.DefaultContentEncoding does not describe a valid HTML character encoding, use UTF-8 as default value.**

Character Encoding in Java Source and Class File Conversion

The JSP file is handed over to the servlet engine of the application server to compile Java source files from JSP files, and Java class files from Java source files.

According to the JSP 2.0 specification, the servlet engine must read the JSP file using the provided JSP page encoding (<@page ... contentType="...;charset=(determined encoding)">). When writing the Java source files, the default behavior of the JSP compiler is to use UTF-8 encoding. Note that it is possible to configure the encoding to use for Java source file creation, via the init parameter "javaEncoding" in the servlet engine's configuration file (<IS_HOME>/share/system/config/servletEngine/conf/web.xml). However, in the context of Intershop 7, the default value should not be changed.

The servlet engine's JSP servlet also creates the resulting class files out of the Java files. It should be noted that the Javac compiler (which compiles the Java class file) must have set the compiler switch "-encoding" to the same encoding which is used to write the Java source file (typically UTF-8). This is also important if the pre-compilation of ISML templates is performed with ANT tasks when building the cartridge.

Decoding Post Data

As a result of a request from a client browser, the class file compiled from an ISML template is executed at the server side, generating the content which is delivered back to the client browser. To indicate the character encoding of the transmitted content, the HTTP response includes an HTTP header "Content-Type".

The value of "Content-Type" corresponds to the JSP page encoding

```
<@page ... contentType="...;charset=(encoding)">
```

of the corresponding JSP file, e.g.

```
text/html; charset=utf-8
```

The browser uses this information to display the received content.

However, the character encoding for content sent in the opposite direction (from a client browser to the Intershop 7 Servlet and JSP container) is not defined as strictly, due to the HTTP specification. Hence, browsers from different vendors behave differently when sending form input data back to the Web server. This makes it

necessary to manipulate the received request data, especially if the data is post data submitted by a form. The decoding of post data is done for all HTTP request parameter names and values. The data is transformed assuming an HTML character encoding which is determined in the following way:

1. Use the HTML character encoding defined in the `appserver.properties` file by the key `intershop.template.DefaultContentEncoding`.
2. If the property `intershop.template.DefaultContentEncoding` does not describe a valid HTML character encoding, use UTF-8 as default value.

Page Caching

Many Web front pages do not change very often and do not depend on who is viewing the page. After compilation, the Web adapter can cache these pages to minimize response time and enhance the performance of your system. If the page is cached, the Web server can present it immediately, without running the pipeline that generates the page.

Basic Principles

The Web adapter page cache is divided in two parts:

■ Pipeline Cache

The pipeline cache stores the content generated by pipeline requests. For content to be cached, the respective templates have use the `<ISCACHE>` tag (see *Page Cache Administration* and `</ISCACHE>`). The `<ISCACHE>` also determines the expiration date for the cached page. Hence, the expiration date can be set individually per template. With each application server response, the expiration date for the returned content is transmitted to the Web adapter.

■ Static Cache

The static cache is used to store static content resources. The expiration date is configured centrally for each channel. If set to 0, static content will not be cached.

Figure 142. Setting the maximum age for static content for a channel

Page Cache Settings	
Maximum age of static content defines the time span (in seconds), static content is cached by the web adapter until it is requested again from the application server. Set it to 0, to disable the caching of static files (graphics etc.)	
Enabled page caching allows the web adapter to cache certain pre-compiled storefront pages. This option improves storefront performance.	
Enable page cache keyword processing to mark certain storefront pages as cacheable by the web-adaptor with different keywords. These pages can later selectively removed from the cache by entering the keywords and clicking 'Invalidate Page Cache'.	
Maximum age of static content:	<input type="text" value="86,400"/>
Page caching:	<input checked="" type="checkbox"/>
Page cache keyword processing:	<input type="checkbox"/>

NOTE: Using the page cache efficiently (i.e., wherever possible) is key to increasing the overall throughput rate, hence key to good performance.

Page Cache Administration

Page cache administration involves the following activities.

■ Enabling or Disabling Page Caching for a Channel

The channel administrator enables or disables the page cache for each channel individually. This is done in the channel back office. See the respective channel user guide for details. When you have enabled page caching, you can use this feature in the templates.

NOTE: During development of templates, page caching should be disabled to see changes immediately. By default page caching is enabled.

■ Enabling Page Caching for a Template

For a Web front page to be cached, you must add the ISML tag `<ISCACHE>` to the appropriate template. The tag can be located anywhere in the template. If multiple `<ISCACHE>` tags of the type `daily` or `relative` exist, the one with the shortest caching period takes precedence. In these cases, debug logging messages are produced. For more information on the `<ISCACHE>` tag, see [<ISCACHE>](#).

NOTE: If the page caching preference is not enabled in the Intershop 7 back office, the `<ISCACHE>` tag is ignored when it is encountered in a template.

■ Invalidate Page Cache

The Intershop 7 back office is also used to invalidate the page cache. Invalidating the page cache means that cached pages have to be generated by the application server the next time they are called. You have the option of invalidating the page cache entirely, or selectively based on pre-defined keywords. See [Selective Page Cache Deletion](#), for details.

Page Caching Restrictions

In general, any Web front page that shows buyer or session-related information should not be cached.

NOTE: Because the system cannot decide whether a Web front page contains data that can be cached or not, it is the responsibility of the template designer to use the `<ISCACHE>` tag properly.

The list below shows two examples of when the template designer should not use the `<ISCACHE>` tag:

■ Web fronts in which the entry page uses a frame set

■ Web front pages using USC2 encoding

You can direct the system to encode the output of a template using a specific character encoding (see `<ISINCLUDE>` for more information). The Web adapter does not support caching of Web front pages encoded with USC2 (a 2 byte, 16-bit, encoding form defined by ISO/IEC 10646-1:1993, constituting the canonical encoding form for Unicode character data).

■ Web front pages displayed by interaction continue nodes

Web front pages displayed by interaction continue nodes of a pipeline are never be cached. If the respective templates have the `<ISCACHE>` tag set, the application server ignores this information.

By default, Intershop 7 enables the page caching feature. This also applies to requests using the HTTP POST method to transmit user input. These requests are assumed to require a dynamically-generated response. The `<eserver>\share\system\config\cluster\webadapter.properties` file contains a parameter defining the maximum content length of POST requests that can be cached by the Web adapter.

Personalized Page Caching

In Intershop 7, personalized pages can be distinguished from non-personalized ones. With personalized pages, you can have many different instances of the same template (depending on the number of user groups). The content differs between the pages, e.g., showing different product prices to different groups of users. Non-personalized pages contain the same content for all users. There is only one instance of the underlying template.

To personalize templates, the `personalized` attribute must be appended to the `<ISCONTENT>` tag. This attribute indicates whether a template is personalized. See `</ISCONTENT>` for more information on the `personalized` attribute.

If a template is personalized, the Web adapter caches the page depending on the personalization group to which the user belongs. Each group has a unique personalization group ID (PGID), the value of which depends on the provider implementation. The PGID is stored in a cookie or the URL, in parallel with the SID. You can set a personalization group ID using the `DeterminePersonalizationGroup` pipelet in the `Login` pipeline.

Personalized page caching is particularly important for Web front pages with session-related information. Since the PGID (in contrast to the session ID) is used to generate the page cache access key, using personalized page caching helps to make sure that session-specific content from the page cache is displayed only to members of the respective personalization group.

Selective Page Cache Deletion

In Intershop 7, you have the following page cache invalidation options:

- **Invalidate the page cache entirely**
Invalidates the page cache for all pages on your system.
- **Invalidate the page cache selectively based on keywords**
Invalidates only pages that have been marked with certain keywords.
- **Invalidate the page cache selectively using a full-text index**

Intershop 7 provides the possibility to have a full-text index generated over the content of the page cache. If such an index has been generated, you can provide keywords in order to selectively delete those pages containing the keyword from the page cache. Selective page cache deletion based on a full-text index is less precise compared to the keyword-based deletion mechanism. However, it does not require you to explicitly mark templates with keywords.

NOTE: Intershop recommends to use the selective page cache deletion feature for smaller updates only (e.g., changing the product price). Do not use the option for mass data handling, such as to clear the page cache after importing large amounts of data. Also, do not use the option for invalidating a large number of pages when you change one of the ISML templates. If you plan to

make regular changes to your templates, use the <ISCACHE> tag to restrict the life-time of the pages so that they automatically become invalid.

Two ways are available to mark templates with keywords to be used for selective page cache deletion: by assigning keywords using a pipelet, or by including the tag <ISCACHEKEY> in templates.

NOTE: Make sure to use unique keywords to prevent pages from being deleted undesirably.

■ Assign Keywords With Pipelets

Using the AddPagecacheKeyword pipelet, you can automatically assign keywords to a page generated by a pipeline. Add the pipelet to the respective pipeline and enter keywords as pipelet properties. You can enter static and dynamic keywords. Static keywords are provided via the pipelet configuration. Dynamic keywords are provided as pipelet parameter. If static keywords are provided, dynamic keywords are ignored.

■ Using the <ISCACHEKEY> tag

The <ISCACHEKEY> tag is used to assign keywords to specific templates (see <ISCACHEKEY>).

Intershop Markup Language (ISML)

What is ISML?

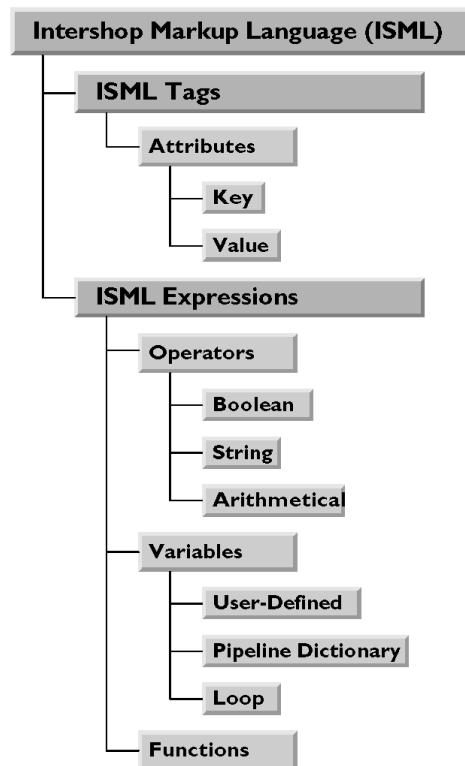
Templates combine HTML with a proprietary language extension referred to as Intershop Markup Language (ISML):

- ISML is an extended markup language based on JavaServer Pages technology and is primarily used to add dynamic content to a page.
- ISML provides means to retrieve data from the pipeline dictionary, to trigger the execution of pipelines, and to embed templates inside other templates.
- Accounting for the variable parts of a template, ISML makes it possible to derive an infinite number of pages from a single template, it also provides an essential prerequisite for features such as personalization.

NOTE: Being based on JSP technology, templates can directly embed JSP code (<% Your JSP code %>). See here for details.

ISML Language Elements

ISML includes two basic language elements, ISML tags and ISML expressions.

Figure 143. The components of ISML

■ ISML Tags

ISML tags define commands that describe how Intershop 7 should embed dynamic data into a page, and how to format this data together with regular HTML code.

An ISML tag has a specific name that always starts with the prefix "IS," such as `<isprint>` or `<isset>`. A tag can have several attributes to control its behavior. ISML tags and their attribute names are case insensitive. However, the values of attributes are often case sensitive. See *ISML Tags* for a complete reference of standard ISML tags.

■ ISML Expressions

ISML expressions encode arithmetical, Boolean, and string operations, as well as special function calls. They are used to perform calculations, to compare values, or to produce new values in combination with functions. ISML expressions are evaluated at runtime every time output is generated from a specific template.

Expressions can be composed of:

- **Template Variables**
See *Template Variables* for details.
- **Constants**
See *Constants* for details.
- **Operators**
See *Operators* for details.
- **ISML functions**

See *ISML Functions* for details.

Expression Example

A sample ISML expression looks as follows:

```
#len(sub:DisplayName)>15#
```

It consists of a function (`len()`), a template variable (`sub:DisplayName`), and an operator (`>`).

An expression always evaluates to a value of a specific data type as shown in the table below:

Table 65. Data Types

Data Type	Description	Scope
<code>double</code>	Numeric data type with a size of 64 bits.	minimum value: ±4.94065645841246544E-224, maximum value: ±1.79769313486231570E308
<code>boolean</code>	Logical data type.	True or false.
<code>string</code>	Alphanumeric data type.	Maximum size is 32K.

Expression Conventions

Please note the following regarding ISML expressions:

- ISML expressions must be surrounded with pound signs (#).

Example:

```
=#len(sub:DisplayName) > 15#
```

- Do not use expressions outside of tags, in order to be sure that the expressions are properly parsed by HTML editors during the design phase.
- This is the universal syntactic description of an ISML expression:
`ISML expression := # <operation> #`
- You can use different elements to build ISML expressions, see *Table 66, “Elements used to build ISML expressions”*.

NOTE: Currently, no type check of Intershop 7 template expressions is provided during compile. If types do not match, a runtime exception will occur.

Expression Syntax

The following table lists the different elements that can be used to build ISML expressions.

Table 66. Elements used to build ISML expressions

Element	Syntactic Description
Operation	<code><arithmetic compare> <string compare> <arithmetic expression> <string expression> <Boolean expression></code>
Arithmetic Compare	<code><simple expression> <ac operator> <simple expression></code>

Element	Syntactic Description
String Compare	<simple expression> <sc operator> <simple expression>
Arithmetic Expression	<simple expression> (<ae operator> <simple expression>)*
String Expression	<simple expression> (<se operator> <simple expression>)*
Boolean Expression	[NOT] <simple expression> (<be operator> [NOT] <simple expression>)*
Simple Expression	((<function name> ([<operation>] (, [<operation>])*)) <Template Variable> <UserVariable> <constant> (<operation>))
ac Operator	<= >= != < > ==
sc Operator	EQ NE
ae Operator	+ - * /
se Operator	. (period)
be Operator	AND OR
Constant	(" <a String> ") (' <a String> ') (<a Float>)
Function Name	hasElements isSSEnabled getValue url urlex action parameter WebRoot isDefined getHeader getCookie hasNext pad trim lcase ucase len val stringToHtml split replace
Simple Name	'a-zA-Z' ('a-zA-Z0-9_')*
ISML Template Identifier	(<subdirectory name> /)* <filename without extension> [.isml]

Formatting Expression Results

If the result of an expression is printed out to an HTML page, it is often useful if you can specify how this should be done. For example, a date can be formatted in different ways depending on the store's locale:

- 12 March 1999
- 12.03.1999
- 03/12/99

For formatting values of the data types Money, Number, Quantity, Date, and Message, Intershop 7 uses specific standard formatter classes. *Table 67, "Existing standard formatter classes"* lists the data types and their appropriate formatter class.

Table 67. Existing standard formatter classes

Type of Expression Result	Standard Formatter Class
Money	StaticMoneyFormat
Number	DecimalFormat
Quantity	StaticQuantityFormat
Date	SimpleDateFormat
Message	MessageFormat

General

Please mind the following general notes regarding formatters:

- All formatter are additionally customized by regional settings.
- Objects of classes other than Money, Date, Message, Quantity and Number will not be formatted and the result of formatting these objects is not standardized.
- The EURO-styles are a temporary work-around for the direct support of multi-currency pages.

A standard formatter class needs a special formatter string. There are three ways to provide such a string:

- Specifying one of the formatter strings pre-defined by a formatter class (they are called styles).
- Using the default style of a formatter class.
- Providing a user-defined formatter string.

Using Styles

Styles are pre-defined formatter strings, which are defined in standard formatter classes. You can specify a style via the `style` attribute of `<ISPRINT>`, or in combination with `getValue()`.

StaticMoneyFormat

This class is used to format expression results of type Money. See *Table 68, "Valid style identifiers for StaticMoneyFormat"* for valid styles. As shown in the table, the style identifier is capable of converting between the Euro currency and a predecessor currency, the Deutsche Mark, at the fixed rate of 1 Euro to 1.95583 Deutsche Mark.

Table 68. Valid style identifiers for StaticMoneyFormat

Valid Style Identifier	Description	Example
MONEY_SHORT	Two digits following decimal separator. No currency symbol.	Input: 3333, Formatted string: "3,333.00"
MONEY_LONG	Two digits following decimal separator. Leading currency symbol.	Input: 3333, Formatted string: "\$3,333.00"
EURO_SHORT	Currency converted into Euro. Two digits following decimal separator, included in brackets. No currency symbol.	Input: 3DM, Formatted string: "(1.53)"
EURO_LONG	Currency converted into Euro. Two digits following decimal separator, included in brackets. Leading currency symbol.	Input: 3DM, Formatted string: "(€ 1.53)"
EURO_COMBINED	Concatenation of MONEY_LONG and EURO_LONG.	Input: 3DM, Formatted string: "DM 3.00 (€ 1.53)"

DecimalFormat

This class is used to format expression results of type Number. See *Table 69, "Valid style identifiers for DecimalFormat"* for valid styles.

Table 69. Valid style identifiers for DecimalFormat

Valid Style Identifier	Description	Example
INTEGER	No digits after decimal separator. No decimal separator.	Input: 2200.1234, Formatted string: "2,200"
DECIMAL	Exactly two digits after decimal separator.	Input: 2200.1234, Formatted string: "2,200.12"

StaticQuantityFormat

This class is used to format expression results of type Quantity. See *Table 70, "Valid style identifiers for StaticQuantityFormat"* for valid styles.

Table 70. Valid style identifiers for StaticQuantityFormat

Valid Style Identifier	Description	Example
QUANTITY_SHORT	0 to 3 digits after decimal separator. No quantity symbol.	Input: 3333 kg, Formatted string: "3,333" Input: 3333.1 kg, Formatted string: "3,333.1" Input: 3333.1234 kg, Formatted string: "3,333.123"
QUANTITY_LONG	0 to 3 digits after decimal separator. Following quantity symbol.	Input: 3333 kg, Formatted string: "3,333 kg" Input: 3333.1 kg, Formatted string: "3,333.1 kg" Input: 3333.1234 kg, Formatted string: "3,333.123 kg"

SimpleDateFormat

This class is used to format expression results of type Date. See *Table 71, "Valid style identifiers for SimpleDateFormat"* for valid styles.

Table 71. Valid style identifiers for SimpleDateFormat

Valid Style Identifier	Description	Example
DATE_SHORT	Date without clock time in short format.	Formatted string: "09/25/99"
DATE_LONG	Date without clock time in long format.	Formatted string: "SEP 25, 1999"
DATE_TIME	Clock time.	Formatted string: "7:55:55 PM"

For date/time input fields, the styles listed in *Table 72, "Date/time input style identifiers"* are available.

Table 72. Date/time input style identifiers

Style Identifier	Description	Example
DATE_INPUT	Date without clock time.	Formatted string: "09/25/99"
TIME_INPUT	Clock time.	Formatted string: "7:55:55 PM"

Style Identifier	Description	Example
DATE_TIME_INPUT	Date and clock time.	Formatted string: "9/25/99 7:55:55 PM"

MessageFormat

This class is used to format expression results of type Message. The formatter of this class cannot be customized. It has a standard behavior. For more information, see the Java documentation in <IS.INSTANCE.SHARE>/docs.

Using the Default Style of a Formatter Class

The default style of a formatter class is used if nothing else has been specified by the user, neither a style, nor a user-defined formatter style. See *Table 73, "Default styles"* for a list of default styles.

Table 73. Default styles

Standard Formatter Class	Default Style
StaticMoneyFormat	MONEY_LONG
DecimalFormat	DECIMAL
StaticQuantityFormat	QUANTITY_SHORT
SimpleDateFormat	DATE_SHORT
MessageFormat	This class has a standard behavior. For more information, see the Java documentation in <IS.INSTANCE.SHARE>/docs.

Specifying User-defined Formatter Strings

A formatter string is explicitly given by the `formatter` attribute of <ISPRINT> or in combination with `getValue()`. For more information about valid placeholders, see the Java documentation in <IS.INSTANCE.SHARE>/docs. See *Table 74, "User-defined formatter strings"* for sample user-defined formatter strings.

Table 74. User-defined formatter strings

Standard Formatter Class	Example for User-Defined Formatter String
StaticMoneyFormat	Scheme: "* #,##0.0#" Input: 3, Formatted string: "\$ 03.00" Input: 3333.123, Formatted string: "\$ 3,333.12"
DecimalFormat	Scheme: "#,#00.0#;(-#,#00.0#)" Input: 3, Formatted string: "03.0" Input: -3333.333, Formatted string: "-3,333.33"
StaticQuantityFormat	Scheme: "#,#00.0# *;(-#,#00.0# *)" Input: 3, Formatted string: "03.0 kg" Input: 3333.333, Formatted string: "3,333.33 kg"
SimpleDateFormat	Scheme: "EEE, MMM d, 'yy", Formatted string: "Wed, Jul 10, '96" Scheme: "h:mm a", Formatted string: "12:08 PM" Scheme: "K:mm a, z", Formatted string: "0:00 PM, PST" Scheme: "yyyy.MMMMMdd.GGG hh:mm aaa", Formatted string: "1996.July.10 AD 12:08 PM"

Standard Formatter Class	Example for User-Defined Formatter String
MessageFormat	Cannot be customized.

Encoding Expression Results

It is possible to use national characters, such as Greek or Cyrillic letters, as well as mathematical symbols and any non-ASCII characters in HTML documents. To ensure that a Web browser can interpret and display these characters correctly, they need to be encoded using HTML representations. In order to do this, you need to replace characters, such as "<," ">," or "&" with named character entities as defined in the HTML 4.01 standard. Because the result of an expression can be a string that might contain a special character expression, results are automatically encoded if you use `<ISPRINT>` to output them. See the example below, which shows the exact string returned by `<ISPRINT>` for a string containing a quote sign. The example assumes that the template variable `product:name` stores the value `Nokia 447X Pro 17"` Monitor.

The following ISML code in a template:

```
<ISPRINT value="#Product:Name#">
```

generates this HTML code:

```
"Nokia 447X Pro 17&quot; Monitor"
```

which is displayed by the browser as:

Nokia 447X Pro 17" Monitor

Expressions outside `<ISPRINT>` are never automatically encoded. In this case, if you need to encode characters, you have to use the `stringToHtml()` function.

NOTE: If you do not encode expressions outside of `<ISPRINT>`, then you might be vulnerable to Cross Site Scripting.

Template Variables

Template variables are placeholders for dynamic content. They are used to refer to data stored in the pipeline dictionary and to be displayed on the page generated from the template. The pipeline dictionary stores these data objects as key-value pairs. Via the key, template variables identify a data object in the pipeline dictionary.

If the data object is simple, e.g., a string, it can be identified directly through the key, e.g., `#ProductID#`. If the data object is complex, the key must be further specified by a colon-separated list of attributes. A simple example (`#Product:Name#`) has already been presented in the preceding section; a slightly more complex one is shown below.

```
<isprint value="#Profile:AddressBook:DefaultAddress:City#">
```

More complex still is the possibility to pass arguments to a data object:

```
<isset name="AddressName" value="Address1">
<isprint value="#Profile:AddressBook:AddressByName(AddressName):City#">
```

A special kind of template variables are so-called loop variables. Loop variables refer to an iterator object in the pipeline dictionary or the current session. Loop variables can only be used within a loop defined via the `<isloop>` tag, as shown below.

```
<isloop iterator="Hotdeals" alias="hds" preview="1">
  <isprint value="#hds:name#"><br>
</isloop>
```

Template variables use object path expressions. See *Object Path Expressions* for a detailed description.

Constants

Constants are used within expressions. There are two different types of constants:

- Numerical Constants
- String Constants

ISML expressions can use both numerical and string constants.

Numerical Constants

Numerical constants are simply numbers, such as 12 or 3.6. In the example `#len(sub:DisplayName) > 15#`, the number 15 is a numerical constant.

String Constants

Encoding of Special Characters in String Constants

A string constant must be enclosed by either single quotes ('') or double quotes (""). We recommend to use single quotes, because double quotes cause problems with some HTML editors.

If a string constant includes single or double quotes as valid characters, these need to be encoded by doubling them, as demonstrated in the examples below:

1. SQL-like encoding (recommended)

To use a single quote inside a string that is bordered by single quotes, use two single quotes:

```
'this is a string '' with a single quote in the middle'
```

To use a double quote inside a double-quoted string, use two double quotes.

```
"this is a string "" with a double quote in the middle"
```

2. Java-like encoding

To use a single quote inside a string that is bordered by single quotes, put a backslash before the single quote character.

```
'Single Quote: /\'
```

To use a double quote inside a string that is bordered by double quotes, put a backslash before the double quote character.

```
"Double Quote: /\"
```

3. Special Intershop 7 encoding

This encoding may only be used in cases where both Java-like and SQL-like encoding schemes are causing problems with a template editor. The encoding is similar to standard Java encoding (with backslash) and uses the character

's' for single quotes and 'd' for double quotes. The standard Java codes /b /t /n /f /r // and /u are supported too. See your Java documentation for more information.

```
'Single Quote: /s'  
"Double Quote: /d"
```

Operators

An operator is a symbol that represents a specific action. For example, a plus sign (+) is an operator that represents addition. Operators are used within expressions and allow you to manipulate numerical and string values stored as variables or constants. In case one or more operators are not of the required types as described below, a runtime exception will occur.

Arithmetic Operators

The standard arithmetic operators are addition, subtraction, multiplication and division. Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. Arithmetic operators work the same in ISML expressions as they do in other common programming languages. *Table 75, "Arithmetic operators"* shows a list of arithmetic operators.

NOTE: Division by zero is not defined and will result in a runtime exception.

Operands for arithmetic operators must be classes of type `number` and their subclasses, e.g., `money`. The result is of type `Double`.

Table 75. Arithmetic operators

Operator	Description	Examples Returning Value 5
+	Addition	<code>var + 3</code>
*	Multiplication	<code>2.5 * var</code>
/	Division	<code>10 / var</code>

The examples in the above table assume that `var` has been assigned the value 2.

Arithmetic Comparison Operators

These operators compare numerical operands and return a Boolean value based on whether or not the comparison is true. *Table 76, "Arithmetic comparison operators"* shows valid operators.

Operands for arithmetic comparison operators must be classes of type `number` and their subclasses, e.g., `money`. The result is of type `Boolean`.

Table 76. Arithmetic comparison operators

Operator	Description	Examples Returning True
<code>==</code>	Equal: Returns true if the operands are equal.	<code>var == 2</code>
<code>!=</code>	Note Equal: Returns true if the operands are not equal.	<code>var != 0</code>
<code>>=</code>	Greater Than or Equal To: Returns true if the left operand	<code>3 >= var; 2 >= var</code>

Operator	Description	Examples Returning True
	is greater than or equal to the right operand.	
<=	Less Than or Equal To: Returns true if the left operand is less than or equal to the right operand.	var <= 5; var <= 2
>	Greater Than: Returns true if the left operand is greater than the right operand.	12 > var
<	Less Than: Returns true if the left operand is less than the right operand.	1 < var

The examples in the above table assume that variable var has been assigned the value 2.

Logical Operators

Logical operators are used to build complex expressions that evaluate to true or false.

Type of operands: Boolean; result type: Boolean.

See *Table 77, "Logical operators"* for valid operators.

Table 77. Logical operators

Operator	Description	Examples Returning True
AND	Returns TRUE if both operators are true.	var_true AND var_true
OR	Returns TRUE if at least one of the two operators is true.	var_true OR var_false var_false OR var_true
NOT	Returns TRUE if its single operator is false, otherwise it returns FALSE	NOT var_false

The examples assume that variable var_true has been assigned the value of true and var_false the value of false.

String Concatenation Operator

Use the Concatenation Operator to concatenate two strings and return another string that is their union. If an operand is not a string, it is automatically converted into a string.

Type of operands: String, result type: String.

See *Table 78, "Concatenation operator for strings"* for valid operators.

Table 78. Concatenation operator for strings

Operator	Description	Example
.	Period: Returns the concatenation of two string operands.	"Inter". "shop" returns "Intershop"

String Comparison Operators

These operators compare string operands and return a logical value based on the comparison. The value is equal only if both combined strings contain exactly the same characters. In the case that an operand is not of type string, it is automatically converted into a string.

Type of operands: String, result type: Boolean.

See *Table 79, "Comparison operators for strings"* for valid operators.

Table 79. Comparison operators for strings

Operator	Description	Examples Returning True
EQ	Returns TRUE if the two string operands are equal, otherwise FALSE.	"string" EQ "string"
NE	Returns TRUE if the two string operands are not equal, otherwise FALSE.	"Inter" NE "shop"

ISML Functions

Functions perform a specific task and return a specific value when the template is processed to generate its output. ISML functions are, for example, used to dynamically generate URLs for links to other storefront pages (`url()`, `urlex()`), or to operate on strings. ISML functions can be categorized into the following groups:

■ Functions used to build URLs

`url()`, `urlex()`, `action()`, `parameter()`, `webRoot()`, `webRootEx()` `contentURL()`, `contentURLEX()`, `sessionlessurl()`, `sessionlessurlex()`

■ Functions used for formatting values of expressions

`stringToHtml()`, `stringToXml()`, `stringToWml()`, `getValue()`

■ Functions for checking the existence of variables, templates, and store properties

`hasElements()`, `hasLoopElements()` `isSSSEnabled()`, `isDefined()`, `hasNext()`, `existsTemplate()`

■ Functions used to manipulate strings

`pad()`, `trim()`, `lcase()`, `ucase()`, `len()`, `val()`, `split()`, `replace()`, `stringToHtml()`, `stringToWml()`, `stringToXml()`, `val()`

■ Functions used to retrieve data

`getHeader()`, `getCookie()`, `getText()`, `getTextEx()`, `paramMap()`, `paramEntry()`

See *ISML Functions* for a complete reference of all available ISML functions, their syntax and usage.

Creating Templates

Templates are designed in the Template Editor. When creating a new template, a simple wizard helps to collect required information such as the template name and the cartridge to which the template belongs.

Start the Template Wizard

To start the template wizard:

1. In the Cartridge Explorer, right-click the cartridge project for the new template to open the context menu.
2. From the context menu, select New | ISML Template.

The template wizard is displayed. If you have not selected a cartridge before starting the template wizard, the template wizard first prompts you to select the cartridge for the template.

Set General Template Properties

The template wizard allows you to set general template properties as described in the table below:

Figure 144. General Template Properties

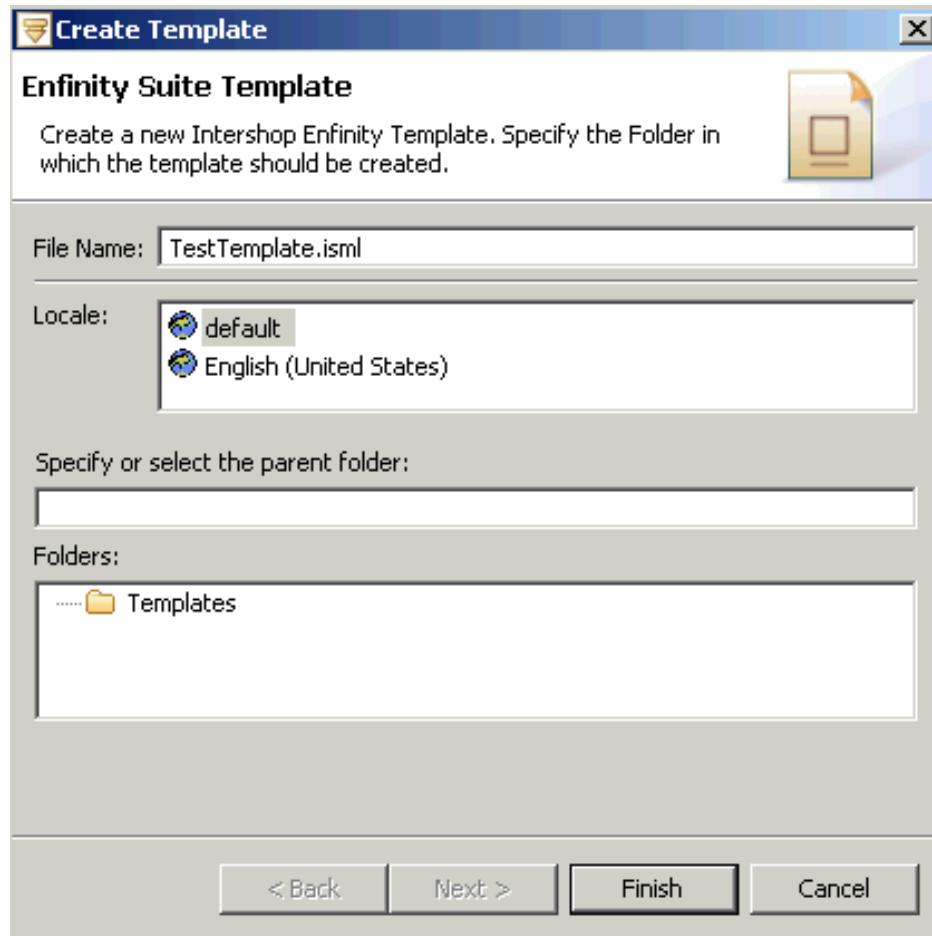


Table 80. General Template Properties

Property	Description
File Name	The file name must be unique within the locale for which the template is intended. A valid file name must have the extension <code>.isml</code> .

Property	Description
Locale	This setting determines in which folder below <IS_SOURCE>/<cartridge_name>/staticfiles/cartridge/templates the template is generated. For example, when selecting "default", the template will be generated in <IS_SOURCE>/<cartridge_name>/staticfiles/cartridge/templates/default. The setting also determines the display name of the template in the Cartridge Explorer. For example, the template "TestTemplate" for locale "de_DE" will have the display name "TestTemplate de_DE" in the Cartridge Explorer.
Parent Folder	Defines the parent folder for the new template. If no parent folder is provided, the template is generated directly below <IS_SOURCE>/<cartridge_name>/staticfiles/cartridge/templates/<locale_id>.

NOTE: Intershop Studio offers all locale IDs that have been configured for Intershop 7 on the Locales preference page. For details on configuring locales, see here. Note that Intershop Studio does not affect locale settings on the development or target system. For information on how to configure locales in Intershop 7, see the Intershop 7 Administration and Configuration Guide.

Once all properties are defined, click Finish. This opens a new Template Editor window for the new template. Moreover, the new template is now accessible both in the Cartridge Explorer and the Template View.

Working with Templates

Creating Template Folders

When working with larger template sets, templates are typically organized in folders. To create a new folder for templates:

- 1. In the Cartridge Explorer, right-click the Templates folder inside the cartridge project to open the context menu.**
- 2. From the context menu, select New | Template Folder.**

Provide folder name. Select locale and parent folder.

- 3. Click Finish to create the new folder.**

The folder can now be used as parent folder for templates.

Using the Content Assist

You can use content assist at various places when writing template code or comments. To activate the content assist, press Ctrl + Space. The content assist displays a floating window with a list of possible completions for the code at the current cursor position. Select a suggested element to insert in the file.

See *Content Assist* for a description of the different contexts in which content assist is available.

Using the Quick Fix Function

Use the Quick Fix function to automatically solve frequently occurring problems. Available quick fixes can be checked and selected directly from the Template Editor, or by selecting the respective problem in the Problems view. To apply quick fixes to task list entries:

- 1. Right-click on a warning or error in the Problems view.**
- 2. From the context menu, select Quick Fix ...**

The Quick Fix dialog opens.

NOTE: If the command in the context menu is grayed out, no quick fix is available.

- 3. Choose one of the options to fix the problem and click OK.**

The fix is carried out and the task removed from the list.

The quick fix function can also be invoked directly from the Template Editor. Place the cursor inside the expression that contains the error and press Ctrl + 1 to open the list of available quick fixes, including a preview of the proposed solution.

NOTE: If the automatic build is disabled, Intershop Studio does not automatically update warning and error markers in the code (including proposed quick fixes). To update these markers after modifying your code, build your project manually as described in here. To update these markers for a single resource only, right-click the element in the Cartridge Explorer to open the context menu and select Check Element.

Open Referenced Elements

When working with a template, you can directly open templates, pipelines or static files that are referenced in the template. To open a referenced resource:

- 1. Place the cursor inside the reference (for example, the pipeline name).**
- 2. Press the F3 key.**

The referenced resource opens in a new editor.

Explore References and Dependencies

Intershop Studio offers the possibility to explore dependencies between the current template and other resources. For example, you can identify templates which are overridden by the current templates, or pipelines which call the current template.

To explore references and dependencies, open the Intershop 7 Dependency Hierarchy View and select one of the options described in *Intershop 7 Dependency Hierarchy View*.

Using Pre-Defined Code

To facilitate template development, you can create pre-defined text or code blocks that are made available when using the content assist or quick fix feature. To create a pre-defined block of code:

- 1. From the menu bar, select Window | Preferences to open the Workbench Preferences dialog.**
- 2. In the left panel, select Intershop Studio | ISML Source | Templates.**
- A list of all available templates is displayed.
- 3. Click New to define a new template.**
- 4. Enter name and description for the new template and select the context in which the template should be made available.**

The context determines where the cursor has to be placed for the content of the template to be available (e.g. inside a comments section or inside an ISML tag). Moreover, depending on the selected context, the template belongs to the content assist or the quick fix function.

- 5. Enter the content of the new template, then click OK.**

You can insert variable elements such as the name of the current user or the current date and time.

Templates containing pre-defined code blocks can be exported and imported to share them across team members. To export a template, select the templates and click Export. To import templates, locate the templates in the file system and click Import.

Editing Templates on Remote Servers

Intershop Studio provides the possibility to directly edit templates on remote servers. This features (also referred to as "hot edit") enables you, for example, to modify templates, but also pipelines or pipelets or static resources, on a non-development system (e.g. test or production system).

NOTE: Remote editing should be used to diagnose problems and fix smaller issues. When dealing with larger issues, consider using the remote development feature instead (for details, see *Remote Server Configuration*).

For remote editing, Intershop Studio provides a specialized perspective, the Intershop 7 Remote Editing perspective. With respect to the explorers, views and editors, the Intershop 7 Remote Editing perspective is similar to the Cartridge Development perspective.

To edit a template on a remote system

- 1. Configure a remote server connection**

See *Remote Server Preferences* for details.

- 2. In the workbench, click Window | Open Perspective | Other. Select the Intershop 7 Remote Server Perspective from the list.**

Having established the connection to the remote server, the cartridges are displayed in the Cartridge Explorer.

Styleguide

Source Code Formatting Conventions

The following conventions are intended to increase the readability of the source code of ISML templates. Note that the list is not exhaustive. For more discussion of these and related issues, see *Intershop Markup Language (ISML)*.

ISML

- Use standard ISML tags with lower case.
- Use mixed-case tag names for self defined ISML tags.
- Avoid using JSP. Business functionality should be implemented in pipelines, not in templates.
- Always declare the scope for <ISSET>, prefer usage of scope="request".
- Use code compacting

```
<iscontent compact="true">
```

- Always declare parameters for <ISMODULE>, use lower case parameter names, prefer the usage of strict modules.
- Each template must begin with:

```
<iscontent type="text/html" charset="UTF-8">
```

- for pages that depend on the personalization group id, add the personalized="true" attribute to the <ISCONTENT> tag
- Even with <ISCONTENT> defining UTF-8 as encoding, use only ASCII characters in ISML files. Always mask special characters, e.g. use 'ä' instead of 'ä'.
- Enable caching for static templates using:

```
<iscache type="relative" hour="24">
```

- Always use an ISPRINT> tag to format and encode output.

To prevent cross-side scripting attacks, expressions within "#" without an enclosing <ISPRINT> are generally not allowed.

Correct:

```
User Comment: <isprint value="#GuestbookEntry:Comment#" />
```

Wrong:

```
User Comment: #GuestbookEntry:Comment#
```

- Avoid subtags in attribute values as this hampers proper template localization.

Correct:

```
<ISMessageBox message="You have not selected any auctions. ...  
  \u003cbr/\u003eUse the checkboxes to select the auctions, ...  
  then click "Cancel" again.">
```

Wrong:

```
<ISMessageBox message="You have not selected any auctions.<br/> ...  
  Use the checkboxes to select the auctions, then click ...
```

```
&quot;Cancel&quot; again.">
```

- Write ISML templates in an XML conform way, e.g. treat ISML as a transformation, not a grammar. Do not mix ISML tags and HTML output tags, instead try to keep them separate.

Correct:

```
<isif condition="#ConfigurationParameterDefinition:Multiple eq 'true'#">
    <select class="select" name="ConfigurationParameter_...
        #ConfigurationParameterDefinition:Name#" multiple="multiple"> ...
<iselse>
    <select class="select" name="ConfigurationParameter_...
        #ConfigurationParameterDefinition:Name#"> ...
</isif>
```

Wrong:

```
<select class="select"
    name="<isprint value="#'ConfigurationParameter_...
        ConfigurationParameterDefinition:Name#">" ...
<isif condition="#ConfigurationParameterDefinition:Multiple ...
    eq 'true'#"multiple="multiple"</isif> ...
```

Formatting Rules

- Use tabs instead of spaces and do not convert tabs to spaces.
This is because spaces consume much more disk space and increase the number of transferred bytes between server and client dramatically.
- Remove spaces and/or tabs at the end of lines.
- To improve readability, try to visualize the page structure by adding markers and comments to the source code.
- Always correctly indent HTML and ISML tags, except for setting form values.

Correct:

```
<table>
    <tr>
        <td>
            <isif ...>
                <input ... value="foo"/>
            </isif>
        </td>
    </tr>
</table>
<table>
    <tr>
        <td>
            <input ...value="<isif ...>foo</isif>" />
        </td>
    </tr>
</table>
```

Wrong:

```
<table>
    <tr>
        <td>
            <isif ...><input ... value="foo"/></isif>
        </td>
    </tr>
</table>
```

```
<table>
  <tr>
    <td>
      <input ... value="
        <isif ...
          foo
        </isif>" />
    </td>
  </tr>
</table>
```

- include HTML comments for compiled html pages
- include ISML comments for documenting the source for further development
- For forms, always specify a name and use the method "post".
- Use template includes wherever possible. This way, complex pages are broken down into smaller components that are easier to re-use.
- Don't surround button texts with spaces.

Correct:

```
<input type="submit" name="apply" value="Apply" />
```

Wrong:

```
<input type="submit" name="apply" value=" Apply" />
```

XHTML Standard

The following guidelines are intended to make ISML templates compliant with the XHTML standard.

- Include an appropriate <!DOCTYPE> definition:

```
=<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

or

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ...
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

- Use the meta tag to declare the document encoding:

```
=<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
```

- Use lower case names for tags and attributes.
- All attributes must have a value, including those attributes that typically don't take values. For example, nowrap has to be replaced by nowrap="nowrap".= The following attributes have to be checked: compact, checked, declare, readonly, disabled, selected, defer, ismap, nohref, noshade, nowrap, multiple, noresize.
- Attribute values have to be included into double quotes.
- Enclose all url() and urlex() method calls in an <ISPRINT> tag to ensure that all "&" parameter separators are encoded to the XHTML conform "&".

```
<a href=<ISPRINT value="#URL(Action('Pipeline-StartNode'), ...
  Parameter('Param1', Value1))#">" class="...">Link</a> ...
```

- Close all open tags according to the XML standard.
- Use nesting according to the XML standard.

- Reserved characters must be replaced by their HTML codes.
- Include a css definition.
- Define color-, font-, width- and other attributes that relate to the visual appearance of a page in the css file only.
- Avoid using JavaScript. Do not use JavaScript to implement business logic.
- Input-tags such as buttons and hidden-input-fields should be defined within a form and a surrounding <td>-tag.

Template Naming

This section provides recommendations regarding template naming. All templates should start with the name of the persistent object that has its data edited or displayed. In the file system, templates should also be grouped into sub-directories according to these persistent objects, for example:

```
/en  
/user  
/catalog  
/category  
/product  
/awf  
/departments
```

Intershop recommends the following naming scheme for templates:

■ **<ENFINITY:BusinessObjectName>List**

This template is used whenever a list of business objects is to be displayed.

Example:

```
AWFList, DepartmentList, BuyingOrgList
```

■ **New<ENFINITY:BusinessObjectName>**

This template is used whenever a new business object is to be created.

Example:

```
NewAWF, NewDepartment, NewBuyingOrg
```

■ **<ENFINITY:BusinessObjectName>**

This template is used whenever the details of an existing business object are to be shown. It should be returned by the PrepareUpdate start node.

Example:

```
AWF, Department, BuyingOrg
```

■ **<ENFINITY:BusinessObjectName><ENFINITY:AssociatedBusinessObjectName>List**

This template is used whenever a list of associated business objects is to be displayed. This template corresponds to a tab in the detail view of a business object.

Example:

```
AWFSequenceList, DepartmentUserList, CostCenterAddressList, ProductBundledProductList
```

■ **New<ENFINITY:BusinessObjectName><ENFINITY:AssociatedBusinessObjectName>**

This template is used whenever details of an associated business object are to be displayed. Append the prefix 'New' in case you are creating a new associated business object.

Example:

NewAWFSequence, NewDepartmentUser, NewCostCenterAddress, NewProductBundledProduct

■ **<ENFINITY:BusinessObjectName>Select<ENFINITY:AssociatedBusinessObjectName>**

Use this template name in case of multi-page selection wizards.

<ENFINITY:BusinessObjectName> represents the business object for which associated businesses object or object relation is to be created.

<ENFINITY:AssociatedBusinessObjectName> represents the associated business object to be selected.

Example:

DepartmentSelectUser, DepartmentSelectRole, ProductSelectCatalog, ProductSelectCategory

■ **Browse<ENFINITY:BusinessObjectName>**

Use this template to browse the content of a business object. The content can be divided into multiple lists (i.e. products and sub-categories of a catalog category).

Example:

BrowseCatalogCategory, BrowseDepartment

Web Forms

Each screen that creates or updates an entity should use a web form to collect submitted form parameters.

If the user enters a wrong input value or does not provide a mandatory input value, an according error message must be displayed together with the the previously entered data.

To display the previously entered data, use only the webform. Do not use individual form parameters.

Example:

Correct:

```
<tr>
  <td class="fielditem2">Description:&nbsp;</td>
  <td class="table_detail">
    <textarea name="RoutingRule_ShortDescription"
              rows="5" cols="70" class="inputfield_en">
      <isif condition="#(RoutingRuleForm:ShortDescription:Value NE '')#>
        <isprint value="# RoutingRuleForm:ShortDescription:Value #">
      <iselse>
        <isprint value="#RRule:ShortDescription#">
      </isif>
    </textarea>
  </td>
</tr>
```

Wrong:

```
<tr>
```

```

<td class="fielditem2">Description:&nbsp;</td>
<td class="table_detail">
    <textarea name="RoutingRule_ShortDescription"
              rows="5" cols="70" class="inputfield_en">
        <isif condition="#isDefined(RoutingRule_ShortDescription)#">
            <isprint value="#RoutingRule_ShortDescription#">
        <iselse>
            <isprint value="#RRule:ShortDescription#">
        </isif>
    </textarea>
</td>
</tr>

```

Template Editor

User Interface

The Template Editor is a powerful tool for developing and modifying ISML templates. The Template Editor provides a range of features to assist with template development tasks, including:

■ Syntax highlighting

Syntax highlighting is used, for example, to separate ISML tags and ISML expressions, and to mark attribute values.

■ Code completion and content assist

At various places, code completion and content assist are available, for example, to insert images, pipelines, or object path expressions.

■ Template preview

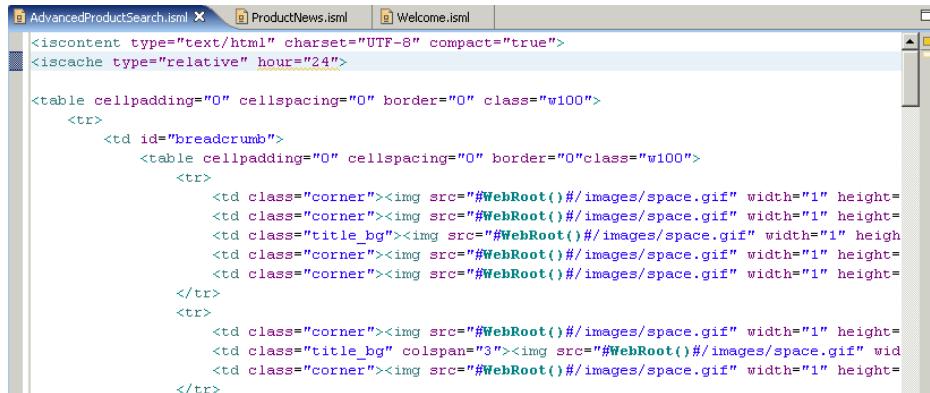
A preview function is available to control general aspects of template design.

■ Instant access to documentation.

When hovering over an ISML tag or expression, over an object path expression or a referenced pipeline, short descriptions for the respective element are made available, drawn from JavaDocs or the Intershop 7 documentation.

General properties of the Template Editor can be customized via the *ISML Template Preferences*.

Figure 145. Template Editor



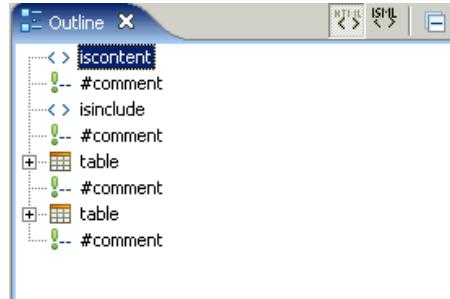
Outline View for Template Editor

In the context of the Template Editor, the Outline View provides a structural view on the template that is currently active in the editor, including the possibility to expand and collapse parts as needed. You can select two different structure trees:

■ HTML-Based

This view computes a tree based on the hierarchical organization of HTML tags. The view is used to check the overall structure of the template.

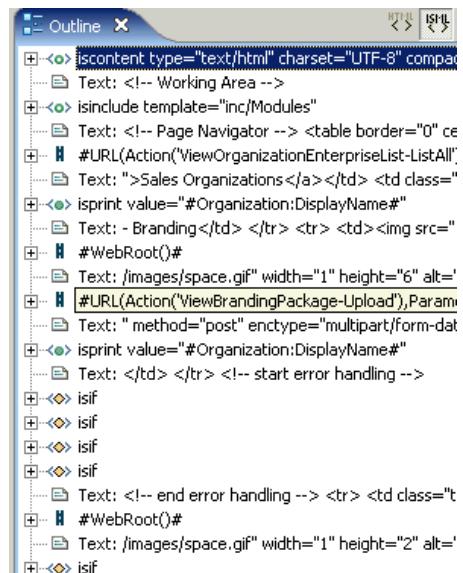
Figure 146. HTML-Based Outline View for Template Editor



■ ISML-Based

This view computes a tree which displays ISML-tags as top-level elements. This view is used to directly access and check ISML tags.

Figure 147. ISML-Based Outline View for Template Editor



Toolbar options available in the context of the Template Editor include:

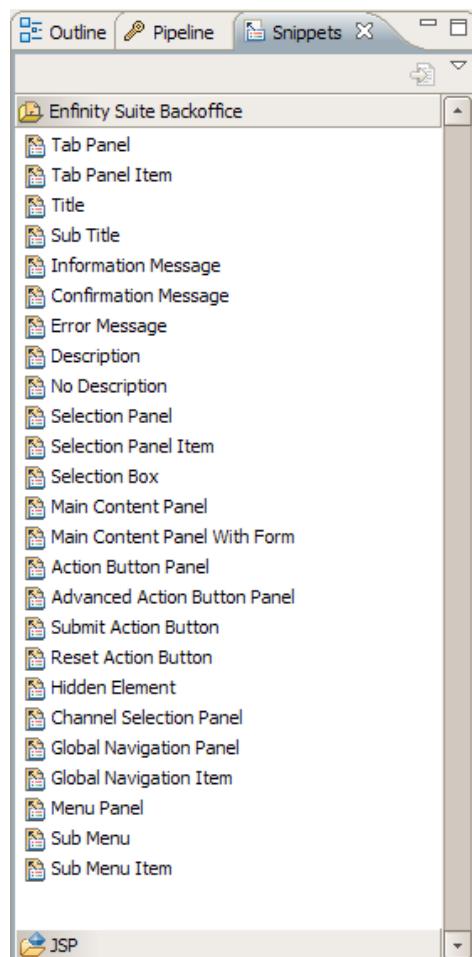
Table 81. The Outline View Toolbar Icons

Icon	Description
	Switches to the HTML-based structural outline.
	Switches to the ISML-based structural outline.
	Collapses all elements in HTML-based structural outline.

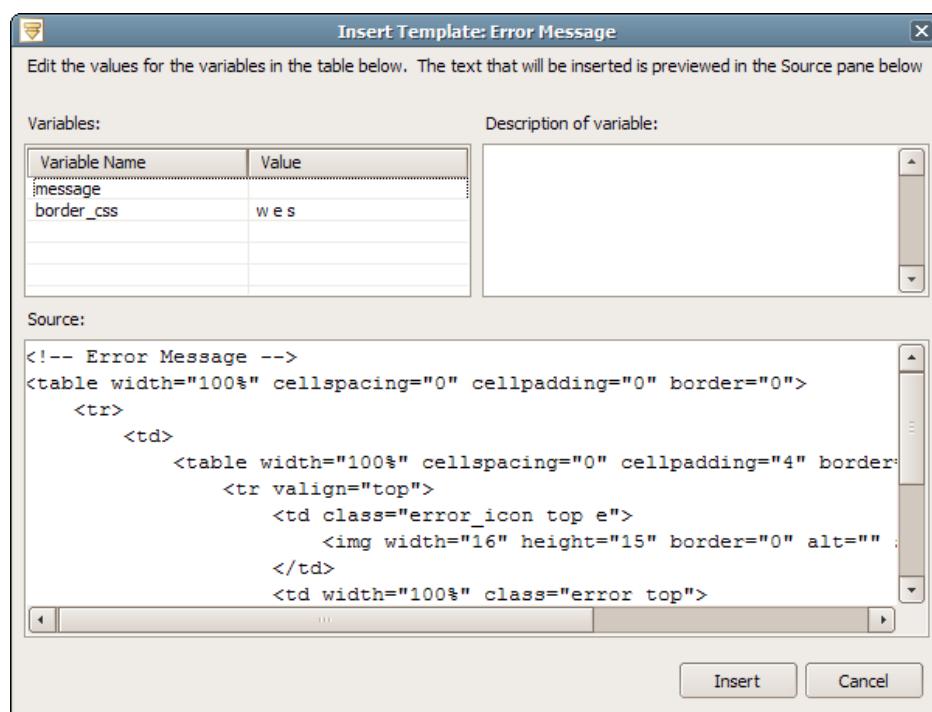
Snippet View for Template Editor

To ease the development of back-office templates, the Template Editor provides the snippet view. The snippet view offers easy access to complex yet frequently used HTML/ISML code fractions, which can be included in templates via simple drag-and-drop actions.

Figure 148. List of ISML snippets for back-office templates



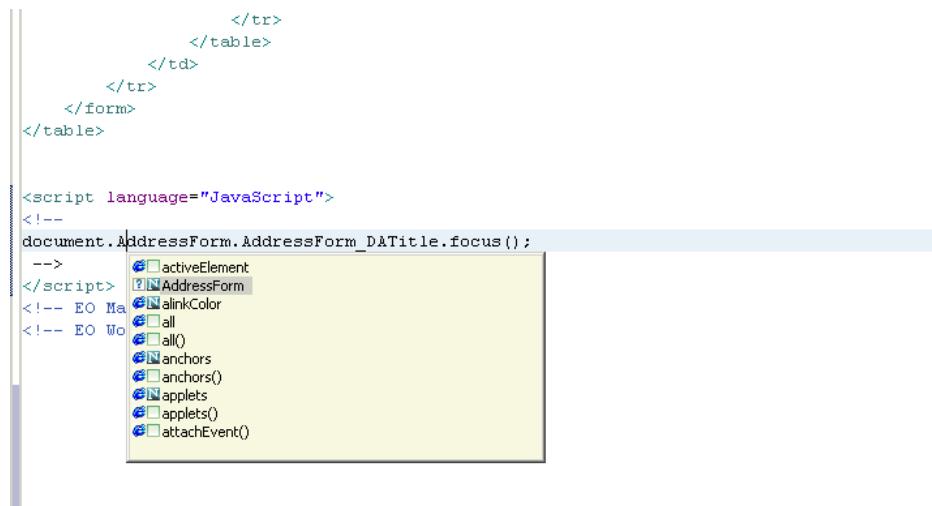
Some of the ISML snippets contain variables. In this case, a dialog opens that allows for entering the corresponding values.

Figure 149. Editing ISML snippet variable values

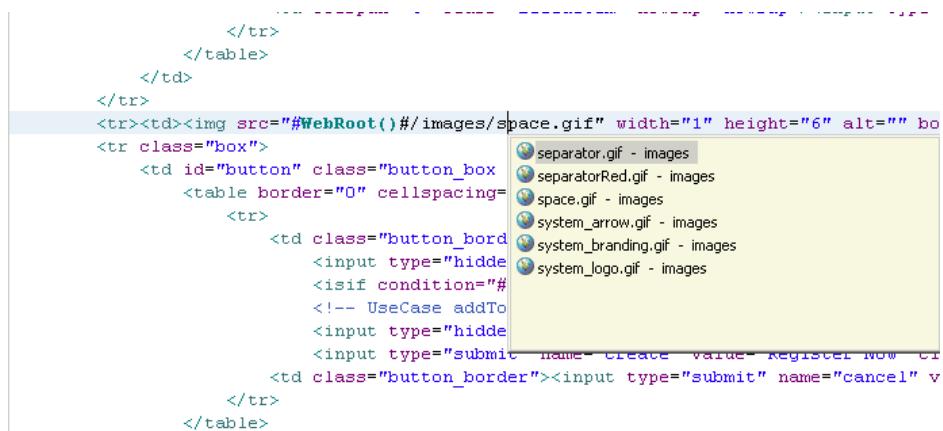
Content Assist

You can use content assist at various places when writing template code or comments. To activate the content assist, press **Ctrl + Space**. The content assist displays a floating window with a list of possible completions for the code at the current cursor position. Select a suggested element to insert in the file.

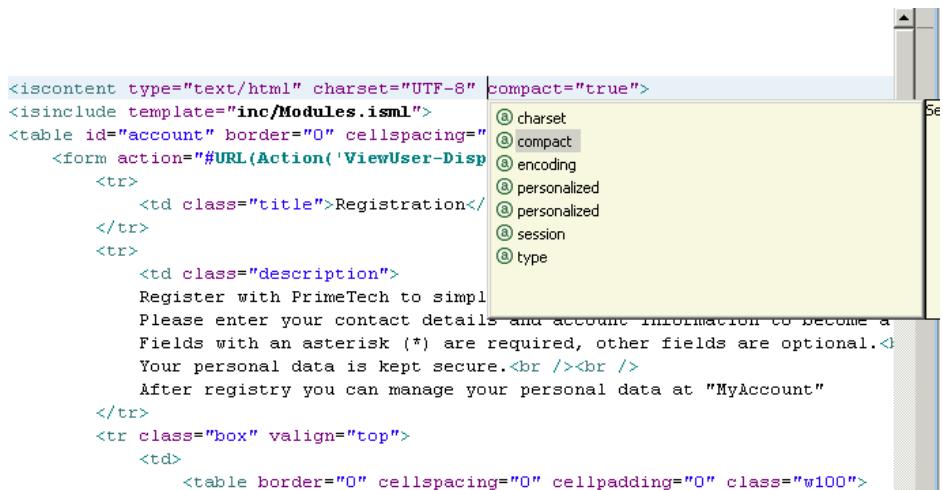
Content assist is available inside JavaScript code blocks, as well as in Java code inside templates.

Figure 150. JavaScript content assist

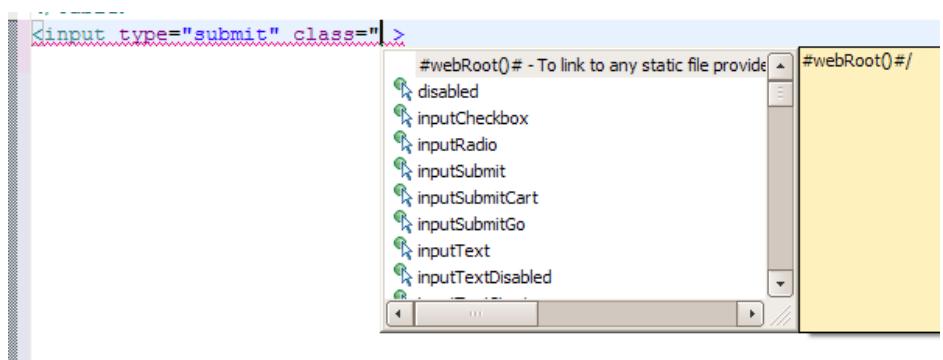
Content assist is available to locate images accessible in the current web root directory.

Figure 151. Content assist to locate images

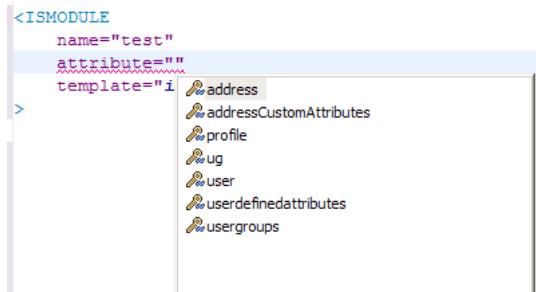
Content assist is available to access available attributes for the current ISML tag.

Figure 152. Content assist to access attributes for ISML tags

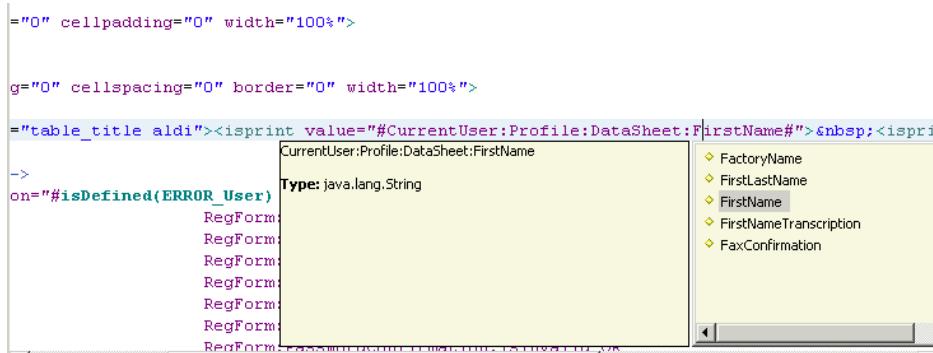
Content assist is available to access values for the tag attributes "class" and "id", which are delivered by CSS stylesheets.

Figure 153. Content assist to access values for class attributes

Content assist is available to access available attributes for the current `<!ISMODULE>` tag.

Figure 154. Content assist to access ISMODULE attributes

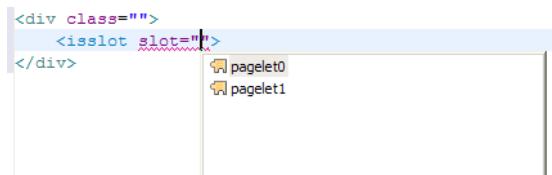
Content assist is available to access the internal structure of object path expressions.

Figure 155. Content assist to access object path expressions

Content assist is available to access available pipelines and templates.

Figure 156. Content assist to access pipelines and templates

Content assist is available to retrieve the names of available slots when editing the slot attribute in <ISSLOT> tags.

Figure 157. Content assist to access available slots

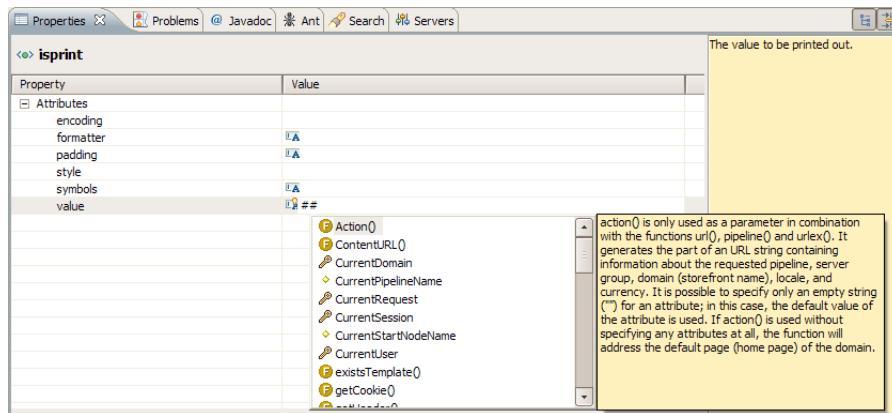
Properties View Cell Editors

ISML or HTML tag attributes can be edited in the properties view. The following attributes now provide powerful cell editors that feature content assist or browsing capabilities:

■ Attributes that can contain ISML expressions

Provide content assist to specify ISML functions or to access the pipeline dicitonary.

Figure 158. Editing an isprint expression



■ href

For images, allows for browsing for a static file and provides field content assist. For anchors, allows for browsing for a static file and provides field content assist as well as offers a dialog for accessing a pipeline dictionary value.

■ img

Allows for browsing for a static file and provides field content assist.

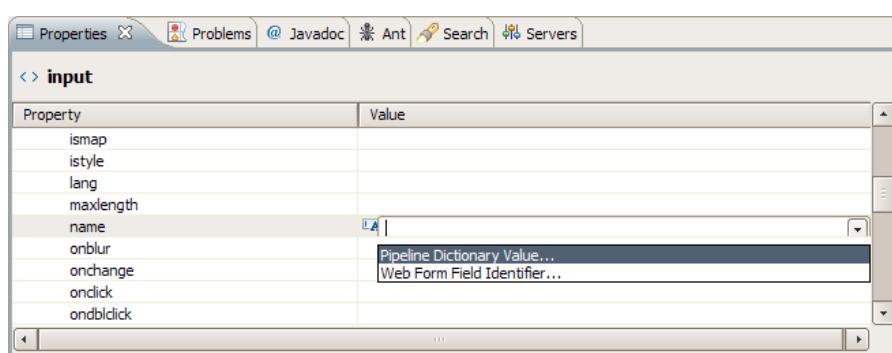
■ action

In forms, offers a dialog for specifying a pipeline URL and provides field content assist.

■ name

Offers a dialog for accessing a pipeline dictionary value as well as allows for browsing for a Web form field ID.

Figure 159. Editing a name ISML attribute



- **value**

In input fields, offers a dialog for accessing a pipeline dictionary value as well as allows for browsing for a Web form field ID.

ISML Reference

ISML Tags

<ISBINARY>

The <ISBINARY> tag allows you to define templates that generate binary pipeline output. This makes it possible, for example, to return PDF documents or image files as pipeline result.

In a template, the <ISBINARY> tag can only be combined with tags that affect the HTTP header, such as <ISCONTENT> and <ISCACHE>. All text content is ignored.

NOTE: The ISML designer is responsible, to set the correct <ISCONTENT> MIME type in a template using <ISBINARY>.

Syntax

```
<isbinary
(
    file      = "( {String} | {ISML expression} )" |
    stream    = "{ISML expression}" |
    resource  = "( {String} | {ISML expression} )" |
    byte      = "{ISML expression}"
)
[ downloadname = "( {String} | {ISML expression} )" ]
```

Examples

The following sample reads the content from a pdf file and sends it as response:

```
<ISBINARY file="d:/foobar.pdf"/>
```

The next sample reads the content from an InputStream and sends it as response:

```
<ISBINARY stream="#aStreamObject#/"/>
```

The next sample reads the content from a resource found in the classpath and sends it as response:

```
<ISBINARY resource="com/intershop/beehive/core/dbinit/internal/icon.gif"/>
```

Finally, the sample below sends a byte array as response, e.g. can be read from a persistent object (blob):

```
<ISBINARY bytes="#MyObject:bytes#/"/>
```

Next, a complete ISML template example is shown. The template returns a PDF documents which is read from the file foobar.pdf. Note, that the content is made cacheable.

```
<ISCONTENT type="application/pdf">
<ISCACHE type="relative" hour="24">
<ISBINARY file="D:/foobar.pdf"/>
```

If the optional "downloadname" attribute is set, a header like

```
response.setHeader("Content-Disposition", "attachment; filename=\"downloadname\"");
```

is generated into the JSP code. The header is used to force the browser to display a file download dialog with the given download file name.

NOTE: Make sure that the content type is set to something different than "text/html" or "image/gif" (e.g. <iscontent type="application/octet-stream">) because some browsers first evaluate the content type and display the content instead of offering a download dialog as desired.

Typically, templates generating binary pipeline output are included in other templates. For example, assume the snippet above defines a template referenced by the interaction end node of the pipeline `BinaryOutput-PDF`. In a different template, you can then create a link to the document `foobar.pdf` by calling the pipeline as shown below:

```
<iscontent type = "text/html">
This is text
<a href="#URL(Action('BinaryOutput-PDF'))#">View the foobar.pdf file</a>
This is text
```

Required Attributes

One of the following attributes is required:

- **file**

```
file = filename | ISML expression
```

Specifies a filename referencing a binary file, e.g. an image, to be read and sent as response.

- **stream**

```
stream = ISML expression
```

Specifies a `java.io.InputStream` object to be read and sent as response.

- **resource**

```
resource= string | ISML expression
```

Specifies a binary resource found in the classpath, e.g., `com/intershop/beehive/core/dbinit/internal/icon.gif`.

- **byte**

```
byte= ISML expression
```

Specifies a byte array (`byte`) object containing binary content to be read and sent as response.

<ISBREAK>

This tag is only allowed as child element of `<ISLOOP>`. For details, see `<ISLOOP>`, `<ISNEXT>`, `<ISBREAK>`.

<ISCACHE>

To improve the performance of the online storefront, the template designer can direct the system to hold specific storefront pages in a cache.

The requested storefront page is retrieved from the cache without running the pipeline that invokes the template and generates the desired page. For more information about page caching and page caching restrictions, see *Page Caching*.

If you want a storefront page to be cached after the Java class file is generated in response to the first request for the template, you must invoke the `<ISCACHE>` tag in the template. The tag allows you to specify when the cached page should expire from the cache; after a fixed period of time or daily at a particular time.

The tag can be located anywhere in the template. See *Page Caching* for a discussion on the interaction between page caching and ISML template inclusion.

Syntax

```
<iscache
    type      = "( relative | daily | forbidden )"
    [ hour    = "( {Integer} | {ISML expression} )" ]
    [ minute = "( {Integer} | {ISML expression} )" ]
```

NOTE: When specifying time, the 24 hour clock is used. This means that 0 is equal to midnight, 1 is equal to 1 am, 2 is equal to 2 am, etc., up to 2359, which is equal to 11:59 pm.

Examples

A page based on a template that contains the following line of code would expire from the cache after 90 minutes:

```
<ISCACHE type = "relative" hour = "1" minute = "30">
```

A page based on a template that contains the following line of code would expire from the cache every day at 0630 hours (6:30 am):

```
<ISCACHE type = "daily" hour = "6" minute = "30">
```

A page based on a template that contains the following line of code would expire from the cache every day at 2330 hours (11:30 pm):

```
<ISCACHE type = "daily" hour = "23" minute = "30">
```

Required Attributes

The `type` attribute is required:

```
type = relative | daily | forbidden
```

`relative` allows you to specify a certain period of time, in minutes and hours, after which the page will be deleted from the cache.

`daily` allows you to specify a specific time when the page will be deleted from the cache.

`forbidden` allows you to explicitly exclude the template from being cached. This helps to make sure that no none-cachable content is stored in the page cache even in complex ISML include structures.

NOTE: If multiple `<ISCACHE>` tags of the type `daily` or `relative` exist, the one with the shortest caching period takes precedence.

Optional Attributes

The following attributes are optional:

■ **hour**

```
hour = integer
```

If the type attribute is set to `daily`, the `hour` value must be an integer ranging from 0 to 23. If type is set to `relative`, all integer values greater than or equal to 0 are valid (the default value is 0, meaning that either the page is never cached or only the `minute` attribute is relevant).

■ **minute**

```
minute = integer
```

If the type attribute is set to `daily`, the `minute` value must be an integer ranging from 0 to 59. If type is set to `relative`, all integer values greater than or equal to 0 are valid (the default value is 0, meaning that either the page is never cached or only the `hour` attribute is relevant).

<ISCACHEKEY>

The `<ISCACHEKEY>` tag allows you to assign a keyword to a specific storefront template. You can use this keyword to invalidate those pages that are based on the template to which the keyword has been assigned. Use either a pipelet or the `<ISCACHEKEY>` tag for invalidating page caching selectively. See *Selective Page Cache Deletion*, for more information on how to implement a `AddPagecacheKeyword` pipelet.

CAUTION: Do not hold too many pages in cache (e.g., one million) and use too many keywords per page (e.g., 10 keywords). Otherwise, the performance of your system will decrease significantly.

Syntax

```
<iscachekey
(
    keyword = "( {String} | {ISML expression} )"
    object  = "( {String} | {ISML expression} )"
)
>
```

Examples

The first example shows how to bind the `PrimeTechSpecials_hotdeals` keyword to the current page (= the URL that requested the page):

```
<ISCACHEKEY keyword="PrimeTechSpecials_hotdeals">
```

For every incoming request, the Web adapter generates a URL hash and sends it to the application server. The application server then uses this hash to assign the keyword to the request URL. The keyword is simply added to the Intershop 7 URL.

If an Intershop 7 pipeline recognizes a keyword, all pages that are generated by calling the pipeline are removed from the cache.

When you invalidate the keyword, the page is removed from the cache (see *Selective Page Cache Deletion*, for more information).

The second example shows how to assign a number of product-related keywords to the `product_item_1` template:

```
<!-- TemplateName : product_item_1.isml -->
<iscache type="relative" hour="48">
<iscachekey keyword="product_item_1">
<iscachekey keyword="#Product:SKU#">
<iscachekey keyword="#Product:Name#">
```

In the example, the template contains a static keyword (`product_item_1`) and two dynamic keywords (`Product:SKU`, `Product:Name`). If you change the specified product details (e.g., SKU, name), all pages that are based on the `product_item_1` template are removed from the cache, and hence can be updated regularly.

NOTE: Keywords of static content are automatically assigned to images via the path name.

Required Attributes

The following attributes are required:

- **keyword**

```
keyword = "( {String} | {ISML expression} )"
```

String or ISML expression for defining a keyword. A keyword uses the Unicode standard. Its length must not exceed 2000 byte.

- **object**

```
object = "( {String} | {ISML expression} )"
```

<ISCOMMENT>

Use `<ISCOMMENT>` to document your templates while you are writing them, to include reminders or instructions for yourself and others who work with the system. HTML comments are created by surrounding the text with the character strings `<!--` and `-->`. This works well for each template user, but from the programmer's perspective, that commenting method provides no confidentiality. Anyone can use a browser's `View | Source` menu to see the HTML code, including the comments. If you don't want to broadcast your comments to the outside world, use the `<ISCOMMENT>` tag for your comments. Anything enclosed in an `<ISCOMMENT> ... </ISCOMMENT>` structure is not parsed by the template processor and does not appear in the generated storefront page.

Syntax

```
<ISCOMMENT> your comments </ISCOMMENT>
```

Example

```
<ISCOMMENT> this comment will not be sent to the browser </ISCOMMENT>
```

<ISCONTENT>

`<ISCONTENT>` modifies the HTTP header of the generated output stream sent to the browser or e-mail client. The HTTP header is identified by its MIME type. In most

cases, you don't need to set the type because this is done automatically by the node or pipelet that calls the template. These nodes set the MIME header as follows:

■ **interaction continue node, interaction end node**

Set the MIME type to `text/html`.

■ **SendMail pipelet**

Sets the MIME type to `text/plain`.

If you want to overwrite these defaults, or if you want your template to generate a different type of output (e.g. when using the `<ISBINARY>` tag), you can explicitly specify another MIME type in the `type` attribute.

Additionally, the `<ISCONTENT>` tag:

- sets the character encoding used by the ISML-to-JSP compiler when reading the template (see *Character Encoding in the Conversion Process* for more information on character encoding),
- allows HTML, XML or WML encoding for the whole template,
- removes unnecessary whitespace by means of the `compact` attribute

You can also dynamically select `<ISCONTENT>` settings.

NOTE: Due to a JSP limitation, the `<ISCONTENT>` tag must be placed before any resulting content in the ISML page (including whitespace characters).

Syntax

```
<iscontent
  [ compact      = "( true      | false        )" ]
  [ templatemarker = "( true      | false        )" ]
  [ type         = "( {String} | {ISML expression} )" ]
  [ charset       = "{String}" ]
  [ httpstatus    = "( {Integer} | {ISML expression} )" ]
  [ session       = "( true      | false        )" ]
  [ personalized   = "( true      | false        )" ]
  [ encode         = "( on |off | html | xml | wml    )" ]
>
```

Example

This example sets the content type and the character set to be used by the ISML-to-JSP compiler:

```
<ISCONTENT type = "plain/text" charset = "iso-8859-1">
<ISCONTENT compact = "true"
```

Optional Attributes

`<ISCONTENT>` has the following optional attributes:

■ **compact**

`compact = true | false`

This attribute ensures that all extra white spaces are removed during the template conversion process. See *The Template Conversion Process* for details and examples.

■ **templatemarker**

```
templatemarker = true | false
```

If set to **true** (default), the resulting markup is surrounded by the following two comment lines:

```
<!-- BEGIN TEMPLATE $templateName$ -->
<!-- END TEMPLATE $templateName$ -->
```

If set to **false**, the above comments are omitted from the output.

NOTE: The default value for this attribute can be set using the intershop.template.PrintTemplateMarker property in IS_SHARE/system/config/cluster/appserver.properties.

■ **type**

```
type = String | ISML Expression
```

Specifies the MIME type of the generated output stream. The default value depends on the template type, generally storefront pages are set to plain/HTML and e-mail output is set to plain/text.

■ **charset**

```
charset = String
```

Defines the character set assumed by the ISML-to-JSP compiler when reading the template. See *Character Encoding in the Conversion Process*.

■ **httpstatus**

```
httpstatus = Integer | ISML expression
```

■ **session**

```
session = true | false
```

Default value is **true**. The session attribute results in a `<@page session="true"...>` entry in the JSP file. The servlet engine is able to provide session context information to Java Beans. Such data would be meaningful when the session is originated by an HTTP request. Templates that are not associated with an HTTP request will provide no meaningful context data. Therefore, the session flag should be set to **false** for all pipelines executed by means other than an HTTP request.

■ **personalized**

```
personalized = true | false
```

Templates can be marked as *personalized* or *non-personalized*. See *Personalized Page Caching* for more information on the caching behavior of personalization pages. To activate the personalization feature, the **personalized** attribute must be appended to the `<ISCONTENT>` tag. This attribute indicates whether a template contains personalized content:

```
<iscontent personalized="true">
```

This marks the page for a specific group of users, where the number of cached pages coincides with the number of user groups in your system. For example, if you have ten different user groups, the system caches the page showing

product prices ten times according to the specific requirements of the user groups. You cannot select or deselect one or more user groups for page caching.

```
<iscontent personalized="false">
```

This page is identical for all users. Regardless of how many user groups exist, a specific page is cached only once.

NOTE: By default, the value for the personalized attribute is set to false.

■ encode

```
encode = on | off | html | xml | wml
```

Default value is **on** for the selected content type. Naming a content type, for example **html** has the same affect as using the **on** flag. All data processed after the first occurrence of **<ISCONTENT>** are affected by this setting. **encode** converts characters that could cause a syntax conflict. For example, when set to **html** the ampersand symbol "&" is converted to &.

NOTE: It is possible, though rarely helpful, to generate one type of content through the **type** command and then **encode** the output with a different content type. Generally, the values **on** and **off** will suffice.

<ISCOOKIE>

<ISCOOKIE> is used to set a cookie in your system. A cookie is a message given to the Web browser by the Intershop 7 system. A cookie allows you to store user-related information on the buyer's system, such as storefront preferences for a single buyer. Cookies can also be used to simplify the login procedure, so buyers don't have to type in their names and passwords each time they access the storefront. To accomplish this, the cookie would store a unique user ID on the buyer's system. After the cookie is stored, the browser returns it every time the buyer requests the URL of the issuing server.

Syntax

```
<ISCOOKIE
    name      = "( {String} | {ISML expression} )"
    value     = "( {String} | {ISML expression} )"
    [ comment = "( {String} | {ISML expression} )" ]
    [ domain  = "( {String} | {ISML expression} )" ]
    [ path    = "( {String} | {ISML expression} )" ]
    [ maxAge  = "( {String} | {ISML expression} )" ]
    [ version = "( {String} | {ISML expression} )" ]
    [ secure   = "( on        | off           )" ]>
```

You can use the following functions to retrieve additional data:

- Use the **getCookie()** function to read the cookie sent back by the client (see **getCookie()**).
- Use the **getHeader()** function to return the value of a request header field of the current request (see **getHeader()**).

Examples

The following example shows how to set a cookie that can be used later to identify a customer:

```
<ISCCOOKIE
    name = "UserID"
    value = "#Buyer:UUID#"
    comment = "your international customer ID"
    domain = ".foo.com"
    path = "/acme"
    maxAge = "10000"
    version = "0"
    secure = "on"
>
```

The following example shows how to use a cookie that has been sent with the current request.

```
<ISIF condition = "#isdefined(getCookie("UserID"))#">
    <ISREDIRECT location="#URL(Action('LoginPanel'))#">
</ISIF>
```

Required Attributes

The following attributes are required:

- **name**

`name = string | ISML expression`

Specifies a name for the cookie. Names starting with "\$" are not allowed.

- **value**

`value = string | ISML expression`

Specifies the value that should be stored by the cookie, e.g., a user ID.

Optional Attributes

The following attributes are optional:

- **comment**

`comment = string | ISML Expression`

Allows you to document the intended use of the cookie. For example, if a cookie contains private information about a user, use the `comment` attribute to mark the cookie accordingly. The user can then inspect the information to decide whether to accept the cookie or not.

- **domain**

`domain = String | ISML Expression`

Specifies the domain for which the cookie is valid. An explicitly specified domain must always start with a dot. If not specified, the browser will set the attribute `domain` to the domain that issues the cookie.

- **path**

`path = String | ISML expression`

Specifies the subset of URLs to which this cookie applies. Pages outside of that path cannot use the cookie.

- **maxAge**

`maxAge = String | ISML expression`

Defines the lifetime of the cookie in seconds. After the specified time elapses, the buyer's browser can discard the cookie. A value of zero means the cookie can be discarded immediately.

■ **version**

```
version = String | ISML expression
```

Identifies the version of the specification which the cookie conforms to. The default value is 0.

■ **secure**

```
secure = on | off
```

The secure attribute is a flag indicating that a cookie should only be used with a secure server, such as SSL. Default value is off.

<ISIF>, <ISELSEIF>, <ISELSE>

The <ISIF> tag group allows you to create conditional ISML code. <ISIF>, and its supporting tags, <ISELSEIF> and <ISELSE>, will likely be the tags you use most in your templates. These are also called business flow functions because they control the business logic's flow.

Every <ISIF> tag must have a matching </ISIF> tag. <ISIF> uses operators to compare values. For more information about operators, see *Operators*. To perform more complex comparisons, you can combine conditions within parentheses using logical operators. You can compare any values accessible through variables, as well as numerical and string constants.

The <ISELSE> and <ISELSEIF> tags are optional. You may use as many <ISELSEIF> tags as needed in an <ISIF> statement, but you can only use one <ISELSE>, which must always be the last comparison performed.

Syntax

```
<isif condition = "{ISML expression}">
  ... some HTML and ISML code ...
[<elseif condition = "{ISML expression}">
  ... some HTML and ISML code ...
]*[<else>
  ... some HTML and ISML code ...
]</isif>
```

Examples

The example shows some possible conditional entries in a template (with and without using operators):

```
<isif condition="#hasLoopElements(Basket:ProductLineItems)#">
<isif condition="#isDefined(Error_PasswordConfirmation)#">
<isif condition="#Error_PasswordConfirmation EQ 'MandatoryValueNotAvailable'#">
<isif condition="#(isdefined(ERROR_FormValues)) OR (isdefined(ERROR_User))#">
<isif condition="#(isdefined(ERROR_FormValues) AND isdefined(ConfirmMail)) OR...
  (not(isdefined(ERROR_FormValues))))#">
```

This example shows how to alternate the colors of the rows in a table. The current color is stored in the user-defined variable, `color`.

```
<ISSET name="color" value="#00FFFF#">
<table>
  <ISLOOP iterator="basket:product">
    <tr>
      <td bgcolor="#color#">
        <ISPRINT value = "#product:name#">
      </td>
    </tr>
    <ISIF condition="#color EQ '#00FFFF'">
      <ISSET name="color" value="#00CCFF#">
    <ISELSEIF condition="#color EQ '#00CCFF'">
      <ISSET name="color" value="#0099FF#">
    <ISELSE>
      <ISSET name="color" value="#00FFFF#">
    </ISIF>
  </ISLOOP>
</table>
```

The result of a conditional expression must be a Boolean value:

```
<ISIF condition ="#Products:Price==0#"> Special free gift.
<ISELSEIF condition ="#Products:Price<100#">
  Special deal.
<ISELSE>
  Today's low price.
</ISIF>
```

Required Attributes

The `condition` attribute is required:

`condition` = Boolean ISML expression

For `<ISIF>` and `<ISELSEIF>`, specify an expression that must evaluate to a Boolean value. If the expression evaluates to `true`, the enclosed code is executed, otherwise it is not. `<ISELSE>` has no attributes.

<ISFILE>

Only allowed as a child element of `<ISFILEBUNDLE>`. See `<ISFILEBUNDLE>`, `<ISFILE>`, `<ISRENDER>`.

<ISFILEBUNDLE>, <ISFILE>, <ISRENDER>

The `<ISFILEBUNDLE>` tag allows you to bundle multiple smaller source files into one big file and to configure processors for the bundled files. `<ISFILEBUNDLE>` has the two supporting tags `<ISFILE>` and `<ISRENDER>`.

Each file bundle must have one `<ISRENDER>` tag. `<ISRENDER>` provides a render method for all resulting files of this file bundle. The variable `File:Name` (see *Example*) is the name of the bundle or the name from one of the configured files.

A file bundle must have one or more `<ISFILE>` tags. `<ISFILE>` is used to declare single files to be processed via its `name` attribute.

Syntax

```
<isfilebundle
```

```

        name      = "{relative target path}"
        processors = "{comma separated default processor list}"
    >
    <isrender>
        ... some HTML and ISML code ...
    </isrender>
    [ <isfile
        name = "{relative source path}"
        [ processors = "{comma separated processor list}" ] />
    ]*
</isfilebundle>
```

Example

```

<isfilebundle name="/bundles/css/CSS_Bundle.css"
    processors="CSSCompressor">
    <isrender>
        <link type="text/css"
            rel="stylesheet"
            href="#webRoot()/.File:Name#"
            media="all">
    </isrender>
    <isfile name="/css/CSS_File3.source.css" />
    <isfile name="/css/CSS_File4.source.css"/>
    <isfile name="/css/CSS_File5.source.css"/>
    <isfile name="/css/CSS_File6.source.css"/>
</isfilebundle>

<isfilebundle name="/bundles/js/JS_Bundle.js"
    processors="JSCompressor, SemicolonAppender">
    <isrender>
        <script src="#webRoot()/.File:Name#"></script>
    </isrender>
    <isfile name="/js/JS_File3.source.js" processors="none"/>
    <isfile name="/js/JS_File4.source.js"/>
    <isfile name="/js/JS_File5.source.js"/>
    <isfile name="/js/JS_File6.source.js"/>
</isfilebundle>
```

<ISFILEBUNDLE> Attributes

<ISFILEBUNDLE> has the following attributes:

■ **name (required)**

name = "{relative target path}"

Specifies the relative path to the bundled file.

■ **processors (required)**

processors = "{comma separated default processor list}"

Specifies the components that should process the declared files. The value must be a comma separated list of valid processor names. The processors are executed in reverse order (from last to first).

The following three processors are available:

- CSSCompressor

Minimizes a CSS file.

- JSCompressor

Minimizes a JavaScript file.

- SemicolonAppender

Appends a semicolon (';') to the end of each file. Normally there is no need to do this. But for bundling of some JavaScript files it is necessary to add a ";" at the end of the file. It is recommended to fix this issue in the files directly instead of using this SemicolonAppender.

<ISFILE> Attributes

<ISFILE> has the following attributes:

■ **name (required)**

`name = "{relative source path}"`

Specifies the relative path to the source file.

■ **processors (optional)**

`[processors = "{comma separated processor list}"]`

Overrides the `processors` attribute specified with the <ISFILEBUNDLE> tag.
Specify this attribute to define a file specific processor list.

For available list of processors, see <ISFILEBUNDLE> Attributes .

You can also set `processors` to `None` to skip processing this file.

<ISINCLUDE>

The <ISINCLUDE> tag directs Intershop 7 to include the contents of one template inside another. The template being included can be as complex as an entire template, or as simple as a single line of HTML code.

NOTE: The include concept enables Web designers to reuse code and avoid unnecessary redundancy. Moreover, template inclusion allows Web designers to maximize page cache usage by splitting up complex pages into parts that can be cached and parts which cannot be cached.

There are two ways of referencing templates for inclusion:

1. local include

To reference local templates for inclusion, use either the `template` or the `templatesDirectory` attribute.

`template` can be used to specify the path and the file name of a single template to be included. The specified path has to be relative to your template's root directory.

Example:

```
<ISINCLUDE template="default/Template2.isml">
```

`templatesDirectory` can be used to specify a path to a directory with templates to be included. All templates residing in the given directory will be included sorted by name.

Example:

```
<isinclude templatesDirectory="product/editor/generalTab"/>
```

Local includes are resolved by the application server. The application server assembles the response out of the including and the included templates

and returns the result to the Intershop 7 Web Adapter. Note that this also implies that the entire servlet response is cached (if possible), consisting of the including and the included templates.

2. remote include

Content can also be included via a HTTP or HTTPS connection, this is called a remote ISML include. Use the URL attribute to specify a pipeline or a valid URL.

The processing of remote ISML includes depends on the details of the pipeline or resource call. In the most common case, the remote ISML include triggers a pipeline request addressed to the same Intershop 7 cluster (using the same protocol and port) which has processed the original request for the template that contains the <ISINCLUDE> element. In the example below, the remote include triggers the execution of the MyPipeline-Start pipeline.

```
<ISINCLUDE url="#URL(Action('MyPipeline-Start'))#">
```

Initial processing of a template containing such an <ISINCLUDE> proceeds as follows:

- a. **The application server converts the <ISINCLUDE> element into a corresponding <WAINCLUDE> element.**

This is why remote ISML includes like this are also referred to as remote Web Adapter includes, or WA-includes, for short. The syntax of a <WAINCLUDE> call reads as follows:

```
<wainclude url = "..." [username = "..." password = "..."]>
```

- b. **The application server returns the template content including the <WAINCLUDE> elements to the Web Adapter.**
- c. **The Web Adapter parses the returned content for <WAINCLUDE> elements.**
- d. **If the content to be produced by the called pipeline is already stored in the cache, the requested page is retrieved from the page cache. If not, the Web Adapter invokes the application server to execute the requested pipeline.**
- e. **The Web Adapter assembles the complete page and sends it to the client.**

NOTE: With Web Adapter includes as sketched above, it is the Web Adapter which assembles the servlet response. This is in contrast to local includes (using the template attribute), in which the application server assembles the servlet response. This has important implications for the caching behavior of pages, as discussed in more detail in here.

Remote ISML includes cannot be resolved using Web Adapter includes if one of the following conditions applies:

- The remote ISML include targets a resource not located on the current Intershop 7 cluster.
- The remote ISML include has to be resolved using a protocol other than the protocol of original request (e.g. HTTPS instead of HTTP).

In contexts like this, the application server opens a HTTP or HTTPS connection in order to resolve the <ISINCLUDE> directly, a process also referred to as remote server include. Hence, similar to local includes, the servlet response is assembled by the application server, and not by the Web Adapter. By default, Intershop 7

automatically decides which method to use in order to resolve a remote ISML request: a Web Adapter include or a remote server include.

Note that in special contexts, remote ISML includes always have to be resolved using a remote server include, since the servlet response is not delivered via the Web Adapter. This is the case e.g. with Mail templates. In contexts like this, you have to force Intershop 7 to resolve remote ISML includes via a remote server include. Otherwise, the servlet response contains <WAINCLUDE> elements which the client cannot resolve. Use the <ISINCLUDE> attribute `mode` with value `server` to force a remote server include.

CAUTION: Do not force Intershop 7 to use remote server includes in contexts which would allow for Web adapter includes as well. Wherever possible, leave it up to Intershop 7 to decide which include mode to use.

Syntax

```
<isinclude
(
(
    template = "( {ISML template identifier} | {ISML expression} )"
)
|
(
    templatesDirectory = "(relative directory path)"
)
|
(
    url = "( {URL} | {ISML expression} )"
[
    username = "( {string constant} | {ISML expression} )"
    password = "( {string constant} | {ISML expression} )"
]
[ mode = "( server | automatic )" ]
)
[ dictionary = "( {string constant} | {ISML expression} )" ]
)
>
```

Example

The following example shows a standard local include of a template:

```
<ISINCLUDE template="inc/blueBar">
```

A remote include triggering another pipeline to show hot deals on a storefront page might look like this:

```
<ISINCLUDE url="#URL(
    action('BrowseCatalog-Hotdeals'),
    Parameter('catalogCategoryID','Storefront')
)#">
```

Dynamic template selection. The name of the referenced template is built during runtime.

```
<ISINCLUDE template = "#'ProductTemplates/Template'.(Product:TemplateNumber)#">
```

Required Attributes

<ISINCLUDE> has the following required attributes:

- `template` | `templatesDirectory`

```
template = "( {ISML template identifier} | {ISML expression} )"
templatesDirectory = "(relative directory path)"
```

Use `template` to specify the name and location of a template to be included. You can either hard code its name or use an ISML variable identifier.

Use `templatesDirectory` to specify a path to a directory containing multiple templates you want to include. The order in which templates are included depends on the name of the templates.

- `url`

```
url = string | ISML expression
```

Specifies a URL for including a remote resource (e.g. a pipeline) via a hard-coded string or an ISML variable identifier.

Optional Attributes

The following attributes are optional:

- `username` and `password`

```
username = string constant | ISML expression
password = string constant | ISML expression
```

Use `username` and `password` in combination to provide HTTP authentication.

Both attributes are only relevant when the required attribute is a URL and the Web server uses basic HTTP authentication.

The pipelet `DetermineBasicAuthentication` stores the HTTP `username` and `password` for the current pipeline call in the pipeline dictionary. When `DetermineBasicAuthentication` is used in a pipeline, remote templates that also reference the target template can refer to the pipeline variables `HTTPUsername` and `HTTPPassword`.

- `dictionary`

```
dictionary = string constant | ISML Expression
```

Specifies the pipeline dictionary to be used.

```
mode = server | automatic
```

Setting the mode to `automatic` (default) leaves it up to Intershop 7 to decide which method to use in order to resolve a remote ISML include: Intershop 7 will use Web Adapter includes whenever possible, and remote server includes otherwise only, e.g. because the remote ISML include targets a resource or pipeline on a remote host. Setting the mode to `server` forces Intershop 7 to use a remote server include. Note that this is only reasonable in very specific contexts, as discussed above.

<ISLOOP>, <ISNEXT>, <ISBREAK>

Loops are blocks of code that are executed repeatedly until a specific condition is met. Use `<ISLOOP>` to loop through the elements of a specified iterator. For example, using `<ISLOOP>` you can list iterable data like categories, products, shipping, or payment methods on a Web page.

Multiple `<ISLOOP>` statements can be nested to construct more complex loop structures.

`<ISLOOP>` has the two supporting tags `<ISBREAK>` and `<ISNEXT>`. Both tags can only be used inside the `<ISLOOP>` tag.

Use `<ISBREAK>` within an `<ISLOOP>` construct to unconditionally terminate the loop. If `<ISBREAK>` is used in a nested loop, it will only terminate the inner loop.

Usually, the `<ISBREAK>` tag is used within `<ISIF>` tags to terminate a loop when a certain condition is met.

Use `<ISNEXT>` to jump forward to the next element in an iterator. In nested loops, this tag affects only the iterator of the inner loop. In case an iterator has already reached its last element, or an iterator is empty when an `<ISNEXT>` is processed, the loop is terminated instantly.

Syntax

```
<isloop
    iterator = "{ISML variable identifier}"
    [ alias = "{simple name}" ]
    [ counter = "{counter name}" ]
>
    ... some HTML and ISML code ...
    [<isnext>]*
    [<isbreak>]
</isloop>
```

Examples

`<ISLOOP>`

Use `<ISLOOP>`'s `alias` attribute to avoid naming conflicts:

```
<ISLOOP iterator = "basket:Offering" alias = "Off">
    <ISPRINT value = "#Off:Name#"> <BR>
</ISLOOP>
```

Use `<ISLOOP>`'s `counter` attribute to define a variable that gets incremented by one on each loop iteration:

```
<isloop iterator="iterator1" alias="element1" counter="c1">
<h3><isprint value="#element1#"></h3>
<ul>
    <isloop iterator="iterator2" alias="element2" counter="c2">
        <li>
            <isprint value="#c1#">.
            <isprint value="#c2#" />.
            <isprint value="#element1#" />
            <isif condition="#c2 == 1#">
                (This is the first element of the inner list)
            </isif>
        </li>
    </isloop>
</ul>
</isloop>
```

In conditional templates, `<ISLOOP>` is often used in combination with the `hasElements()` and `hasNext()` functions.

`hasElements()` checks if an iterator stored in the pipeline dictionary has any elements.

`hasNext()` checks whether the actual iterator instance in the current loop has any more elements. This is useful in cases where the last loop element needs special treatment. See `hasElements()`, `hasLoopElements()` and `hasNext()` for details.

```
<ISLOOP iterator="foo">
    ...
    <ISIF condition="#hasNext(foo)">
        ... // do something except for last line
    </ISIF>

    ...
</ISLOOP>
```

<ISBREAK>

In this example, `<ISBREAK>` terminates the loop after displaying three product names, even if the products iterator contains more than three products:

```
<ISSET name="counter" value="#0#">
<ISLOOP iterator="basket:product">
    <ISPRINT value="#product:name#"> <BR>
    <ISSET name="counter" value="#counter + 1#">
    <ISIF condition="#counter >= 3#">
        <ISBREAK>
    </ISIF>
</ISLOOP>
```

<ISNEXT>

Here `<ISNEXT>` is used to output the names of two products one after another:

```
<ISLOOP iterator= "Products">
    <ISPRINT value="#Products:Name#">
    <ISNEXT>
    <ISPRINT value="#Products:Name#"> <BR>
</ISLOOP>
```

`<ISNEXT>` can also be used to organize products in a two-column table.

After data from one loop variable is printed out in the left table column, `<ISNEXT>` jumps forward to the next element in the list, and prints into the right table column. After that, the loop returns to its beginning to generate the next row of the table.

```
<isloop iterator="products" alias="hds">
    <tr>
        <!-- first column of hotdeals -->
        <td valign="top" width="50%">
            <ISPRINT value="#hds:Name#"><br>
            <ISPRINT value="#hds:ShortDescription#">
        </td>
        <isnext>
        <td>
            ...
        </td>
        <td valign="top" width="50%">
            <!-- second column of hot deals -->
            <ISPRINT value="#hds:Name#"><br>
            <ISPRINT value="#hds:ShortDescription#">
        </td>
    </tr>
</isloop>
```

Required Attributes

```
iterator = ISML variable identifier
```

Use this attribute to specify an iterator. Attributes of the iterator can be used in the loop within expressions. The iterator can be one of the following classes:

- **com.intershop.beehive.foundation.util.Iterable**
- **java.util Enumeration**
- **java.util Iterator**
- **java.util Collection**

Optional Attributes

The following attributes are optional:

- alias

```
alias = simple name
```

Allows you to define an alias name that can be referenced instead of the often unwieldy variable identifier.

- counter

```
counter = counter name
```

Allows you to define a variable that contains the number of the current element in the iterator. The variable starts with 1.

<ISMODULE>

Use **<ISMODULE>** to declare custom tags in your ISML templates. The declaration can be located anywhere in the template, as long as it appears before the first usage of the declared tag. Multiple declarations of the same tag do not interrupt template processing, the last one is used. You can also define a custom tag in an included template and use it afterwards in the including template.

Custom tags can be used like any other tags. In fact, a custom tag is an included template that can have user-defined attributes. Therefore, every custom tag definition is stored in a separate file. This method is similar to an include template.

Custom tag attributes are slightly restricted and generalized in their behavior and syntax, as follows:

- They may be strings or ISML expressions that are converted to strings during runtime.
- They are always optional. During runtime, the template processor does not check if attributes are missing. If such a check is needed, you have to code it explicitly.
- They can be accessed in the definition template simply by name. This method is similar to how user-defined variables are accessed.

Syntax

```
<ismodule
```

```

template      = "( {ISML template identifier} | {ISML expression} )"
name         = "{simple name}"
[ attribute = "{simple name}" ]*
(
    [ strict  = "false" ]
)
|
(
    strict      = "true"
    [ returnattribute = "{simple name}" ]*
)
>

```

Required Attributes

The following attributes are required:

■ **template**

```
template = ISML template identifier | ISML expression
```

Defines a path and a name for the ISML file implementing the tag. Relative paths are expanded from the server's template root directory. Physical paths are not allowed.

NOTE: Template file names and folder names cannot contain spaces.

■ **name**

```
name = string
```

Name of the custom tag. Custom tags are always declared without the **IS** prefix, e.g., **MYTAG**. However, when using the custom tag in a template, you must first write the prefix like this: <**ISMYTAG**>. Custom tags can use either case.

■ **strict**

```
strict = true | false
```

This parameter can be used to control parameter handling.

If set to **false** (default), the behavior does not change compared to earlier releases:

- The whole pipeline dictionary is passed to the module.
- If a parameter is passed to a module, an already existing variable with the same name as the attribute will be overwritten in the dictionary.
- After the execution of the module, the variable is completely removed from the dictionary, even if the variable existed before (e.g. an output parameter of a pipelet).

If set to **true**, strict has the following effect:

- A new empty dictionary is passed to the module only initialized with the specified attributes.
- The original pipeline dictionary will be restored after the execution of the module.

NOTE: If no value is specified for the strict attribute, the value **false** is assumed by default. This ensures backwards compatibility. However, it is strongly recommended to use value **true** when defining new modules, as this guarantees transparent processing of the pipeline dictionary.

Optional Attributes

The following attributes are optional:

■ **attribute**

```
attribute = simple name
```

(Unlimited) Specifies attributes you want your custom tag to have.

NOTE: Because all attribute names stored in the pipeline dictionary are lowercase, also use lowercase names in the definitions.

returnattribute

```
returnattribute = <simple name>
```

With *strict=true*, **<ISMODULE>** tags can return values. Using **returnattribute**, the names of dictionary keys to be returned can be specified. Specified return attributes can be mapped into the pipeline dictionary of the caller.

Examples

This first example demonstrates the declaration, the implementation, and the application of a custom tag **bgcolor**, which is used for inserting specifically colored backgrounds.

Before you can use a custom tag in a template, it needs to be declared in the same template (*sample1.isml*). The tag declaration must include a reference to a separate template file (*background.isml*) containing the actual code of the tag. In this case, the *background.isml* file has been stored in a separate folder named *TagExtension*.

Here is the contents of the *sample1.isml* file:

```
<---! tag declaration --->
<ISMODULE template = "TagExtension/background.isml"
    name = "bgcolor"
    attribute = "color">
    strict = "true"

<---! tag usage --->
<ISBGCOLOR color = "green">
```

And here, the contents of the *background.isml* file:

```
<---! tag implementation --->
<ISIF condition="#isDefined(color)#">
    
<ISELSE>
    
</ISIF>
```

This next example shows **<ISMODULE>** combined with server-side scripting:

The custom tag can be used to store key-value pairs in the pipeline dictionary of the current pipeline. Declaration and usage of the custom tag takes place in the *sample2.isml* template, where the actual code of the custom tag has been stored in a separate template file, *put.isml*.

Following is the code in the *sample2.isml* file:

```
<!--- tag declaration --->
<ISMODULE template = "put.isml"
    name = "put"
    strict = "true"
```

```

        attribute = "keyname"
        attribute = "keyvalue">

<!-- tag use --->
<ISPUT keyname = "foo" keyvalue = "foobar">

<!-- tag test --->
<ISPRINT value= "#foo#">
```

And finally, the code in the *put.isml* file:

```

<%
    if (getObject("keyname")==null)
        throw new ServletException("missing key name");
    if (getObject("keyvalue")==null)
        throw new ServletException("missing key value");
    getPipelineDictionary().put((String)getObject("keyname"),
        getObject("keyvalue"));
%>
```

<ISPIPELINE>

<ISPIPELINE> can be used to call a pipeline in an ISML template. The resulting pipeline dictionary of the executed pipeline is stored in the current pipeline dictionary, using a key provided by the attribute alias.

Syntax

```

<ispipeline
    pipeline = "( {String} | {ISML expression} )"
    [ params = "{ISML expression}" ]
    alias     = "( {String} | {ISML expression} )"
>
```

Example

In this example, <ISPIPELINE> executes the start node `LoginCheck` of pipeline `MyTestPipeline`. Two parameters (`param1`, `param2`) are submitted to the pipeline via a `Map` object. The pipeline dictionary that results from executing `MyTestPipeline-LoginCheck` is stored in the current pipeline dictionary under the `resultDict` key.

```

<ISCreateMap mapname="ParamMap"
    key0="param1" value0="foo"
    key1="param2" value1="bar">

<ispipeline
    pipeline="MyTestPipeline-LoginCheck"
    params="#ParamMap#"
    alias="resultDict" />
```

Required Attributes

The following attributes are required:

- `pipeline`

`pipeline = String | ISML expression`

This attribute specifies the name of the pipeline to be called. Allowed are text values and ISML expressions. If no start node is specified, the default start node name "Start" is taken.

- alias

```
alias = String | ISML expression
```

This attribute specifies the key under which the pipeline dictionary resulting from the execution of the called pipeline is stored in the current pipeline dictionary.

Optional Attributes

```
params = ISML Expression
```

The attribute specifies the pipeline parameters. Allowed are ISML expressions only. The parameters can either be a `java.util.Map` object implementation which is stored in the pipeline dictionary and contains all the parameter key value pairs, or a parameter set which is defined manually using the `paramMap()`, `paramEntry()` and `parameter()` functions.

<ISPLACEHOLDER>

`<ISPLACEHOLDER>` is used to define spots in an ISML template where the Web adapter post-processing will fill in content that was defined through matching `<ISPLACEMENT>` tags within the current request's HTML response. The connection between an `<ISPLACEHOLDER>` and the matching `<ISPLACEMENT>` entries is done through the `<ISPLACEHOLDER>` ID.

Syntax

```
<ISPLACEHOLDER
  id          = "( {String} | {ISML expression} )"
  [ prepend    = "( {String} | {ISML expression} )" ]
  [ separator  = "( {String} | {ISML expression} )" ]
  [ append     = "( {String} | {ISML expression} )" ]
  [ preserveorder = "( true | false )" ]
  [ removeduplicates = "( true | false )" ]
>
```

Examples

Defining Meta Tags

This example aggregates all values that belong to `<ISPLACEHOLDER>` with `id="keywords"` while removing duplicates. This way several components on a page can influence the content of the meta tag for keywords.

The respective `<ISPLACEMENT>` values are separated by commas. By specifying the `prepend` and `append` attributes this tag will only be generated if the according `<ISPLACEMENT>` tags are present in the HTML. The order of the `<ISPLACEMENT>` values is given by the order of the `<ISPLACEMENT>` tags within the HTML since no additional ordering algorithm is applied (`preserveorder="false"`). This can be done since there are no ordering dependencies between keywords.

```
<ISPLACEHOLDER id="keywords" prepend="#" <meta name="keywords" ...
      content="#" separator=", " append="#" /> "#" preserveorder="false"/>
<ISPLACEMENT placeholderid="keywords">foo</ISPLACEMENT>
```

```
<ISPLACEMENT placeholderid="keywords">bar</ISPLACEMENT>
```

Using the code above, the web adapter post-processing results in the following HTML:

```
<meta name="keywords" content="foo, bar"/>
```

NOTE: To use double quotes ("") within the attribute values of prepend, separator, or append some form of escaping is needed. The two options that work with ISML are as follows:

```
<ISPLACEHOLDER id="keywords" prepend="#<meta name="keywords" ...  
content="#" separator=", " append="#"/>'#" preserveorder="false"/>  
<ISPLACEHOLDER id="keywords" prepend="#<meta name=\\"keywords\\\" ...  
content=""#" separator=", " append="#\"/>'#" preserveorder="false"/>
```

In contrast to this, the following example will *not* work:

```
<ISPLACEHOLDER id="keywords" prepend="<meta name=\\"keywords\\\" ...  
content="" separator=", " append="\\"/>'" preserveorder="false"/>
```

Defining JavaScript References

The following example illustrates the usage of `<ISPLACEHOLDER>` to define a spot within the HTML result where the aggregated JavaScript references (content of the `<ISPLACEMENT>` tags with `placeholderid="JS"`) will be inserted.

When inserting the values, duplicates will be removed and the order of the JavaScript references will be preserved. By specifying the `prepend`, `separator` and `append` attributes the required HTML syntax for the JavaScript references is generated for each `<ISPLACEMENT>` value while the `<ISPLACEMENT>` tag only provides the path to the JavaScript reference:

```
<ISPLACEHOLDER id="JS" prepend="#<script type="text/javascript" ...  
src="#" separator="#"></script><script type="text/javascript" ...  
src="#" append="#"></script># />  
  
<ISPLACEMENT placeholderid="JS">1.js</ISPLACEMENT>  
<ISPLACEMENT placeholderid="JS">2.js</ISPLACEMENT>
```

Alternatively, you could reach the same result without using the `prepend`, `separator`, and `append` attributes if the `<ISPLACEMENT>` values already include the complete markup for a JavaScript reference:

```
<ISPLACEHOLDER id="JS">  
  
<ISPLACEMENT placeholderid="JS"><script type="text/javascript" ...  
src="1.js"></script></ISPLACEMENT>  
<ISPLACEMENT placeholderid="JS"><script type="text/javascript" ...  
src="2.js"></script></ISPLACEMENT>
```

The result after the post-processing is:

```
<script type="text/javascript" src="1.js"></script>  
<script type="text/javascript" src="2.js"></script>
```

Required Attributes

`id` = String | ISML expression

This attribute specifies the ID of the placeholder. Here the Web adapter's post-processing will insert only the content of `<ISPLACEMENT>` tags that refer to this ID with their respective `placeholderid`.

Optional Attributes

The following attributes are optional:

■ **prepend**

```
prepend = string | ISML expression
```

If defined, the prepend value will be inserted in the HTML markup where the <ISPLACEHOLDER> is defined. After this, the values of the respective <ISPLACEMENT> tags are inserted.

■ **separator**

```
separator = string | ISML expression
```

If defined, the separator value will be used to separate the values of the respective <ISPLACEMENT> tags.

■ **append**

```
append = string | ISML expression
```

If defined, the append value will be inserted in the HTML markup after the values of the respective <ISPLACEMENT> tags are inserted.

■ **preserveorder**

```
preserveorder = true | false
```

The default value is true. This attribute determines if a special ordering algorithm should be applied to the list of aggregated <ISPLACEMENT> values.

This is needed amongst others to keep the right order for loading dependencies of JavaScript or CSS files. In addition, this algorithm removes duplicates too.

For the ordering algorithm to work it is required that the <ISPLACEMENT> tags are grouped by their page/component/template instance. If the application of the ordering algorithm is not needed (e.g for meta tag values) it can be disabled by specifying preserveorder="false".

■ **removeduplicates**

```
removeduplicates = true | false
```

The default value is true. With this attribute the aggregation of the <ISPLACEMENT> values can be influenced in such a way that duplications will be omitted. removeduplicates is not needed in combination with preserveorder="true" since the ordering algorithm already removes duplicates. This attribute is useful if the ordering is not needed but duplicates should be removed e.g. for keywords.

Note, for the removal of duplicates to work the contents of the individual <ISPLACEMENT> tags have to be exactly the same because the Web adapter performs only a simple text comparison.

The following example shows two JavaScript references that are semantically duplicates but would not be handled as such since the plain text comparison yields unequal values.

```
<script type="text/javascript" src="shopfunctions.js"></script>
<script src="shopfunctions.js" type="text/javascript"></script>
```

To avoid this kind of problems, you may prefer the first of the JavaScript reference examples (see *Defining JavaScript References*) where only the real references are defined by the <ISPLACEMENT> tags and the rest is handled by the prepend, append, and separator values.

<ISPLACEMENT>

<ISPLACEMENT> is used to mark content within an ISML template that will be aggregated and moved to the according <ISPLACEHOLDER> spots by the Web Adapter post-processing.

Syntax

```
<isplacement
    placeholderid = "( {String} | {ISML expression} )"
>
    ... some HTML and ISML code ...
</isplacement>
```

Required Attributes

placeholderid = string | ISML expression

Specifies the ID of the placeholder this entry belongs to. The Web adapters post-processing will insert the aggregated content of all <ISPLACEMENT> tags with that placeholderid at the matching <ISPLACEHOLDER> spot.

Examples

Use the <ISPLACEMENT> tag to define a CSS reference that will be aggregated at the matching <ISPLACEHOLDER> tag.

```
<ISPLACEMENT placeholderid="CSS">
    <link rel="stylesheet" type="text/css" href="/css/shop.css" />
</ISPLACEMENT>
<ISPLACEMENT placeholderid="CSS">
    <link rel="stylesheet" type="text/css" href="/css/slider.css" />
</ISPLACEMENT>
```

Use the <ISPLACEMENT> tag to define content for the meta tag for keywords that will be aggregated at the matching <ISPLACEHOLDER> tag.

```
<ISPLACEMENT name="keywords">Shirts</ISPLACEMENT>
<ISPLACEMENT name="keywords">
    <isprint value="#Product:Name#" />
</ISPLACEMENT>
```

<ISPRINT>

<ISPRINT> outputs the results of ISML expressions and template variables. Even though it is possible to output expression results without <ISPRINT>, you should always use it, because it contributes to optimizing your template code.

The <ISPRINT> tag does the following:

- **automatically formats the output string**

When printing expression results using <ISPRINT>, the output is formatted according to its type by an appropriate formatter class. For more information

about formatter classes, see *Formatting Expression Results*. You can use three formatter strings to control the output format:

- The default formatter string of the formatter class.
- One of the pre-defined formatter strings (styles) of the formatter class.
- A formatter string that you have defined.

■ automatically encodes the output string

<ISPRINT> automatically encodes the result of expressions and the contents of ISML variables, except for user-defined variables and string constants. During the encoding any special characters, such as ;, <, >, &, =, including any named characters of HTML 4.01 (Unicode 160-255) are converted into their HTML counterparts. For example, the character > would be converted into the string >.

For more information, see *Encoding Expression Results*.

NOTE: The encoding is always done after the formatting.

The following example shows a string containing quotation marks, and the corresponding string that is returned by the <ISPRINT> tag (it is assumed that the template variable product:name has the value 'Nokia 447X Pro 17" Monitor'):

The ISML code in the template is:

```
<ISPRINT value="product:name">
```

The HTML code generated is:

```
"Nokia 447X Pro 17&quot; Monitor"
```

And the browser will render this text as:

```
Nokia 447X Pro 17" Monitor
```

Expressions outside <ISPRINT> tags are not automatically encoded. In this case, special characters must be converted with the `stringToHtml()` function (see `stringToHtml()`).

Syntax

```
<isprint
  [ encoding = "( on | off )" ]
  value = "{ISML expression}"
  [
    (
      style      = "{Style Identifier}"
    )
    |
    (
      formatter = "{Format String}"
    )
  ]
  [ symbols = "({Symbol Declaration} | {ISML expression})" ]
  [ padding = "({Padding Constant} | {ISML expression})" ]
>
```

Examples

The first example demonstrates how to use <ISPRINT> to output an expression that returns a string. The returned string is formatted as defined by either the default

formatter (line one), or by using a style (line two), or by a user-defined formatter string (line three).

```
<ISPRINT value = "#Basket:Offering:Price#">
<ISPRINT value = "#Basket:Offering:Price#" style = "EURO_LONG">
<ISPRINT value = "#Basket:Offering:Price#" formatter = "* #00.0#">
```

The usage of the symbols attribute:

```
<isprint value="#aNumber#" formatter="#.#" 
         symbols="GroupingSeparator=';DecimalSeparator=,">
<isprint value="#aDate#" formatter="h:mm a"
         symbols="AmPmStrings={ante meridiem, post meridiem}">
```

The usage of the padding attribute:

```
<ISPRINT value = "#Product:Price#" padding = "-10"> //string output: " 5.00 EUR"
<ISPRINT value = "#Product:Price#" padding = "+10"> //string output: "5.00 EUR"
<ISPRINT value = "#'foolish'" padding = "+3">      //string output: "foo"
```

The usage of the encoding attribute:

```
<ISPRINT value = "#'>new<#" encoding = "on">      //string output: ">new<"
<ISPRINT value = "#'>new<#" encoding = "off">     //string output: ">new<"
```

Required Attributes

value = ISML Expression

Specifies the expression you want to output with <ISPRINT>.

Optional Attributes

The following attributes are optional:

■ **encoding**

encoding = on | off

Default value is on. With this attribute, you can explicitly switch automatic encoding on and off. Intershop 7 supports encoding in HTML, XML and WML. Even if encoding is turned off, the functions `stringToHtml()`, `stringToXml()` and `stringToWml()` can be used to encode individual strings.

■ **style**

style = style identifier

Use this attribute to specify a style identifier. See *Formatting Expression Results*, for a list of valid style identifiers. Instead of using the `style` attribute, you can alternatively define a formatter string using the `formatter` attribute.

■ **formatter**

formatter = string

Use this attribute to define a formatter string to control how <ISPRINT> outputs your expression's result. For information on building your own formatter string, refer to *Formatting Expression Results*. If `formatter` is used, `style` must be omitted.

■ **symbols**

symbols = symbol declaration

Use this attribute to modify the format used for printing. The value of the `symbols` attribute can be a symbol declaration or an ISML expression that results in a symbol declaration. This attribute supports all properties of the types `char`, `String` and `String` of `java.text.DecimalFormatSymbols` and `java.text.SimpleDateFormatSymbols`.

For symbol declarations, the following syntax is allowed:

```
<SymbolsDecl> ::= <SymbolDecl> ( ';' <SymbolDecl> )*
<SymbolDecl> ::= <SymbolName> '=' <ValueDecl>
<ValueDecl> ::= <Value> | <ValueList>
<ValueList> ::= '{' <Value> ( ',' <Value> )* '}'
```

NOTE: The values can contain escaped characters. Use '\ for escaping.

■ padding

`padding = padding constant`

This is used only with mail templates, namely templates using `plain` rather than `html` type, to define field width and other spacing issues. For example, when printing a list of product names using a loop statement, you can define a constant field width for every element of the list. The value for padding can be any positive or negative number. The absolute value of `PaddingConstant` defines the field width. A positive value produces left-aligned output, a negative value produces right-aligned output. If the output string is greater than the field size, the output string will be truncated at its right end.

<ISREDIRECT>

Use `<ISREDIRECT>` to redirect the browser to a specified URL.

Syntax

```
<isredirect
    location      = "( {URL String} | {ISML expression} )"
    [ httpstatus = "( {Integer}      | {ISML expression} )" ]
>
```

Example

The following example checks whether the buyer has enabled cookies in his browser or not. Depending on the result, the browser is redirected:

```
<ISIF condition = "#ISDEFINED(getCookie('UserID'))#"
       <ISREDIRECT location="#URL(Action('LoginPanel'))#">
<ISELSE>
    <!-- cookies are not enabled --->
</ISIF>
```

Required Attributes

`location = URL String | ISML expression`

Use the `location` attribute to specify a target URL used by the browser to send a new request.

NOTE: Some browsers do not support redirecting from HTTP to HTTPS. To circumvent this problem, use the HTML meta tag refresh instead:

```
<html>
</head>
<meta http-equiv="refresh" content="0; URL='#URLEX(...)#'">.
...
```

Optional Attributes

`httpstatus = Integer | ISML expression`

Specifies the status code of the HTTP response.

<ISRENDER>

This tag is only allowed as child element of `<ISFILEBUNDLE>`. See `<ISFILEBUNDLE>`, `<ISFILE>`, `<ISRENDER>` for details.

<ISSELECT>

`<ISSELECT>` can be used for easily implementing drop-down list boxes in an HTML form. `<ISSELECT>` replaces the HTML tag `<SELECT>` and its supporting tags.

With the `iterator` attribute you can specify an iterator, whose elements appear in the list box. One element in the list box can be preselected by specifying a condition with the `condition` attribute. Make sure that the specified condition evaluates to `true` for only one element of the list, otherwise the browser may not display the list properly.

Syntax

```
<isselect
  name      = "( {a name} | {ISML expression} )"
  iterator   = "{ISML variable identifier}"
  [ condition = "{ISML expression}" ]
  value     = "{ISML expression}"
  description = "{ISML expression}"
  [ encoding  = "( on | off )"      ]
  [ disabled   = "( true | false )" ]
```

Example

In this example, `<ISSELECT>` adds all elements of the iterator to an HTML list box that is generated.

```
<form action = ... >
  <ISSELECT name = "PaymentMethodUserSelection"
            iterator = "PaymentMethodChoices"
            condition = "#PaymentMethod == PaymentMethodChoices#"
            value = "#PaymentMethodChoices:UUID#"
            description = "#PaymentMethodChoices:Description#">
  ...
</form>
```

Required Attributes

The following attributes are required:

- **name**

`name = simple name`

Specify a name for your list box. Note that the name you specify here is used when the browser sends the user selection as a key value pair to the server. Therefore, the specified `name` must be equal to the input parameter required by the pipeline that will process the form values.

■ **iterator**

```
iterator = loop variable
```

Specifies a loop variable. All elements of the loop variable are added to the list box.

■ **value**

```
value = string
```

Text sent back by the browser to the server as the value of a key-value pair. This can be used, for example, to return an internal product ID instead of the product name displayed to the buyer. The string specified by `value` is stored in the pipeline dictionary. The appropriate key is determined by the `<ISSELECT>` attribute name.

■ **description**

```
description = string
```

Text that is displayed to the buyer in the drop-down list box.

Optional Attributes

The following attributes are optional:

■ **condition**

```
condition = Boolean ISML Expression
```

If `condition` is true, the appropriate list element is pre-selected. This has its counterpart in the `selected` attribute of the `<option>` HTML tag. Make sure that only one list element matches the condition to avoid unknown results in different browsers.

■ **encoding**

```
encoding = on | off
```

Default value is on. With this attribute you can explicitly switch automatic HTML encoding of the output on and off. Encoding means converting any special characters, such as ; <, >, &, =", as well as any named characters of HTML 4.01 (Unicode 160-255) into their HTML counterparts. For example, `<ISPRINT>` converts the character > to the string >.

■ **disabled**

```
disabled = true | false
```

If set to true, this attribute will disable the select box. The default value is false.

<ISSET>

`<ISSET>` can be used to define and set a user-defined variable.

The first occurrence of <ISSET> in a template declares and sets the variable, there is no other special tag needed for declaration. In case the variable already exists, <ISSET> resets it to the specified value.

The special attribute `scope` is available to determine the scope of the user-defined variable. `scope` is optional and can be either `request` or `session`.

■ **request scope (== default scope)**

Scope `request` means that the variable is accessible within the current request only. The value of the variable is stored in the pipeline dictionary. Using `request` scope is useful in case you want to work with a temporary variable which does not need to be persisted to the database. This prevents the session object from being polluted with temporary data and renders additional session synchronization with the database unnecessary.

■ **session scope**

Scope `session` means that the variable is accessible to all templates during a particular storefront session. The lifetime of a session variable ends with a session. A session variable can be used in expressions and is identified by its simple name. Values can now be of any type, not just Double and String. But note that non-serializable objects stored at session scope are not written to the database when the session is made persistent.

Syntax

```
<isset
  value  = "( {String} | {ISML expression} )"
  name   = "{simple name}"
  [ scope = "( request | session )" ]
>
```

Examples

The following example shows how <ISSET> is used to define and initialize a variable named `color`. The value of `color` is the string `#A0CC99`. As no value for attribute `scope` is defined, `request` scope is assumed by default.

```
<isset name="color" value="#'#A0CC99'">
```

In the next example, if a variable was already defined by a former <ISSET> tag (first line), a second <ISSET> resets the value of the variable (second line):

```
<isset name="counter" value="#0#" >
<isset name="color" value="#counter + 1#">
```

Finally, the examples below illustrate the usage of the `scope` attribute:

```
<!-- the value goes to the session -->
<isset scope="session" name="foo" value="bar">
<isset name="foo" value="bar">

<!-- the value goes to the pipeline dictionary -->
<isset scope="request" name="foo" value="bar">
```

Required Attributes

The following attributes are required:

■ **name**

```
name = simple name
```

Specifies the name of a user-defined variable. Mind the following rules for naming user-defined variables:

- Identifiers must start with a letter (a-z, A-Z).
- Any following character can be a letter, number, or underscore.
- User-defined variables are case-sensitive.

■ **value**

```
value = ISML Expression
```

Specifies a value to be stored in the variable. Value can be of any type. Note, however, that non-serializable objects stored at session scope will not be written to the database when the session is made persistent.

Optional Attributes

```
scope = session | request
```

Default value is session. Specifies the scope of a user-defined variable. In case of session scope, the value is stored to the session. In case of request scope, the value is stored to the pipeline dictionary.

<ISNEXT>

This tag is only allowed as child element of <ISLOOP>. See <ISLOOP>, <ISNEXT>, <ISBREAK> for details.

<ISTEXT>

<ISTEXT> is used as a placeholder for localized text strings that can be provided by sites and cartridges.

<ISTEXT> defines a key and a locale that maps onto a localization resource. These resources are defined as key value pairs that are stored in the format of Java property resource bundles (*.properties files).

Cartridge-specific localization resource bundles are stored in <IS_SOURCE>/<cartridge>/staticfiles/cartridge/localizations/ from where they are read when the cartridge is loaded.

For example, a localization resource may look like this:

```
// <IS_SOURCE>/<cartridge>/staticfiles/cartridge/localizations/ ...
button_en.properties

buttons.apply=Apply
product.retail_set.part.text=The product {0} is currently part of ...
another retail set {1}.
test.choice=There {0,choice,0#are no files|1#is one file|1<are ...
{0,number,integer} files}.
```

The numbers enclosed within curly braces designate placeholders for dynamic content to be inserted during template processing, see the parameter attribute below.

Additional description files can include descriptions of messages or contexts that can help translation agencies to identify the context of the key. These descriptions will have the same key as the translated text:

```
// IS_SOURCE/cartridge/staticfiles/cartridge/localizations/
    button_descriptions.properties ...
```

buttons.apply=A button label to apply changes in a form

During development, you may enable the continuous reloading of localization resources in your appserver.properties file:

```
// IS_SHARE/system/config/appserver.properties

# reloading interval 5 seconds
intershop.localization.CheckContent=5000
```

Syntax

```
<istext
  key      = "( {String} | {ISML expression} )"
  [ encode   = "( on | off | encoding handler )" ]
  [ locale   = "( {String} | {ISML expression} )" ]
  [ parameter0 = "( {String} | {ISML expression} )" ]
  [ parameter1 = "( {String} | {ISML expression} )" ]
  [ parameter2 = "( {String} | {ISML expression} )" ]
  [ parameter3 = "( {String} | {ISML expression} )" ]
  [ parameter4 = "( {String} | {ISML expression} )" ]
  [ parameter5 = "( {String} | {ISML expression} )" ]
  [ parameter6 = "( {String} | {ISML expression} )" ]
  [ parameter7 = "( {String} | {ISML expression} )" ]
  [ parameter8 = "( {String} | {ISML expression} )" ]
  [ parameter9 = "( {String} | {ISML expression} )" ]>
```

Examples

The following example illustrates the usage of the <ISTEXT> tag in an ISML template:

```
<istext key="product.retail_set.part.text" parameter0="#Product:SKU#" ...
        parameter1="#RetailSet:SKU#" />
<istext key="test.choice" locale="#Locale#" parameter0="#CountOfFile#" />
```

Required Attributes

key = String | ISML expression

Unique identifier of a localization resource as defined in a localization resource bundle.

Optional Attributes

The following attributes are optional:

- parameter0 .. parameter9

```
parameter<position> = String | ISML expression
```

Specifies a parameter to be injected into the returned string at the given position.

- **encoding**

```
encode = on | off | encoding handler
```

Specifies the encoding scheme to be used. The following encoding handlers are supported:

- html (default)
- js
- none
- locale

```
locale = String | ISML expression
```

Defines the locale of the resource to be looked up. If not specified, the current template processing locale will be used.

<IS{Module Name}>

Use <IS{Module Name}> tags to insert custom-modules defined using <ISMODULE>. See </ISMODULE> for details.

Syntax:

```
<is{module name}
  [ {attribute name} = "{String} | {ISML expression}" ]*
>
```

ISML Functions

action()

There are two ways to generate dynamic content:

■ Using custom servlets

This is done through the ISML function `servlet()`. See the section `servlet()`, for more information.

■ Using pipelines

This is done through the ISML function `action()`. See the remainder of this section.

`action()` is only used as a parameter in combination with the functions `url()` and `urlex()`. It generates the part of an URL string containing information about the requested pipeline, server group, domain (storefront name), locale, and currency. The attributes you can specify are listed below. It is possible to specify only an empty string ("") for an attribute; in this case, the default value of the attribute is used. If `=action()` is used without specifying any attributes at all, the function will address the default page (home page) of the domain.

NOTE: All attribute values you can specify for `action()` are case sensitive.

Syntax

```
action()
action(<action>)
action(<action>,<servergroup>)
action(<action>,<servergroup>,<domain>)
action(<action>,<servergroup>,<domain>,<locale>)
action(<action>,<servergroup>,<domain>,<locale>,<currency>)
```

Examples

The following example shows a simple pipeline call:

```
<a href="#URL(action('CheckBasketReadyToOrder-Start'))#">
... link text to be displayed ...
</a>
```

The next example shows a pipeline call with a change of locale and currency:

```
<a href="#URL(action('CheckBasketReadyToOrder-Start','','','en_US','USD'))#">
```

This example shows a pipeline call with a change of domain (storefront name):

```
<a href="#URL(action('CheckBasketReadyToOrder','server group',
... 'TransactionStore'))#">
```

Optional Attributes

`action = string`

The string defines the name of the target pipeline and an appropriate start node. The default value is the default pipeline (and its start node) of the current storefront (the string "Default-Start" is returned).

NOTE: The name and start node of the default pipeline is hardcoded in the RequestHandlerServlet. To modify the default pipeline/start node for your site, you can use, for example, the URL rewriting or cartridge assignment mechanisms.

`server group`

The string defines the type of the target server group, e.g., Web Front or Back Office Servers. The default value for server group is the current server group of the session.

`domain = string`

The string defines the name of the target domain (storefront name). The default value is the current domain of the session.

`locale = string`

String that defines the locale of a store. The default value is the current locale of the request.

`currency = string`

The string defines the currency of shown prices. The default value is the current currency of the session.

ContentURL()

The ContentURL function is used to reference static content in the static content directory of the current unit, e.g., share/sites/<site_name>/<version-id>/static/units/<unit_name>/static.

Syntax

```
ContentURL(<contentRef>)
ContentURL(<contentRef>, <locale>)
ContentURL(<contentRef>, <locale>, <servergroup>)
```

Required Parameters

content reference

References static content (e.g., /images/calc.gif).

Optional Parameters

locale

This parameter can be used to override the default locale. If left unspecified, the following fallback is used:

- 1. the request's current locale**
- 2. the lead locale**
- 3. the locale will be designated as a missing parameter ("")**

servergroup

This parameter can be used to override the default server group. If left unspecified, the first value from the application server property `intershop.server.assignedToServerGroup` is taken.

Examples

The following example shows how to reference an image of an offer within a unit:

```

```

When the template is processed, the content reference replaces the ISML expression `Product:OfferedProduct:Image`. You can reference a product image with or without mentioning the unit it belongs to (e.g., `Officeland:/images/calc.gif` or `/images/calc.gif`). This is shown below, along with the possibility to provide a locale ID:

```

```

In the example, the ISML function `ContentURL` parses the string parameter given in brackets to extract unit and path names, and produces the following URL:

```
/is-bin/intershop.static/<group>/<current_site>/-/<locale>/images/calc.gif  
/is-bin/intershop.static/<group>/<current_site>/-/<locale>/images/calc.gif
```

The abstract result of the `ContentURL()` function parsing the string parameter is the following:

```
<WebRoot_url>/<group>/<site>/<unit>/<locale>/<path>
```

Unit and path names are separated by a colon, as shown below:

```
<unit_name>:<path>
```

For example, the following reference

```
<a href="#ContentURL('PrimeTech:zips/p4366a.zip', 'de_DE')#"#>
```

is resolved to

```
http://<>/is-bin/intershop.static/WFS/PrimeTech-PrimeTechSpecials-Site/...
PrimeTech/de_DE/zips/p4366a.zip
```

The elements of this URL are explained in *Table 82, "URL Elements"*:

Table 82. URL Elements

URL Element	Description
<WebRoot_url>	Specified by the property intershop.template.(Https)WebRootURL, the default value of which is /is-bin/intershop.static (see the appserver.properties file).
<group>	Server group
<site>	Current site
<unit>	Unit specified by the ContentURL parameter
<locale>	Locale specified by the ContentURL parameter. If no locale is provided, the fallback described above is used.
<path>	Resource path; Intershop 7 business objects (e.g., Products) hold relative paths to the static content that is assigned to them. The syntax of such a path name is <unit_name>:<path>.

If there is no unit element in the ContentURL parameter, the unit name is replaced by a dash ('-').

When static content is requested, the static content URL, e.g., /is-bin/intershop.static/<group>/<current_site>/<specified_unit>/<locale>/images/calc.gif is parsed. The site, unit, locale, and path elements are extracted.

Static content is searched for in the following order:

- Unit,
- Superordinate site,
- All cartridges assigned to this site.

ContentURLEx()

The ContentURLEx() function complements the ContentURL() function. It is also used to reference static content. However, in contrast to ContentURL(), the ContentURLEx() function always returns an absolute URL, hence is used in templates in which an absolute URL is required, such as in mail templates.

Syntax

```
ContentURLEx(<protocol>,<host>,<port>,<server_group>,<locale>,<contentRef>)
```

Required Parameters

protocol

This parameter can be used to override the default protocol ("HTTP"). The value is ignored (even if set) if *intershop.template.WebRootURL* or *intershop.template.HTTPSWebRootURL* (in *config/cluster/appserver.properties*) already specifies an absolute URL.

host, port

If these parameters are left unspecified, the values from the application server properties `intershop.WebServerURL` (in case of "HTTP") or `intershop.WebServerSecureURL` (in case of "HTTPS") are taken. The values for these parameters are ignored (even if set) if `intershop.template.WebRootURL` or `intershop.template.HTTPSWebRootURL` (in `config/cluster/appserver.properties`) already specifies an absolute URL.

server group

This parameter can be used to override the default server group. If left unspecified, the first value from the application server property `intershop.server.assignedToServerGroup` is taken.

locale

This parameter can be used to override the default locale. If left unspecified, the following fallback is used:

- 1. Get the locale from the current request.**
- 2. Get the lead locale.**
- 3. Set the locale as missing parameter ("").**

NOTE: If <protocol>, <host>, <port>, <server group> or <locale> are left unspecified or passed as empty strings, the according values are taken from the current request, or, if this fails, from the `appserver.properties` file.

content reference

References static content (e.g., `/images/calc.gif`).

Examples

The following ISML structure

```

```

produces the following result:

```

```

existsTemplate()

This function can be used to check whether a template exists in the context of the current site or not. The function returns `true` if the template is accessible in the template lookup path of the current site. Otherwise, the function returns `false`. Use the expression inside `<isif>` structures.

Syntax

```
existsTemplate(String)
```

Example

```
<isif condition="#existsTemplate('inc/DBbreadcrumbTrail')#">
```

The template name can also be build dynamically using ISML string concatenation:

```
<isif condition="#existsTemplate('product/ProducTabs_'.CurrentChannel:TypeCode)#">
```

Use the function to implement a fallback from a specialized template to generic template. For example, consider the product detail page of the Consumer Channel which has tabs that do not appear in other channels. Using the `existsTemplate()` function, a generic product detail page can be constructed, including a fallback like:

```
<isif condition="#existsTemplate('product/ProducTabs_'.CurrentChannel:TypeCode)#">
  <isinclude template="#'product/ProducTabs_'.CurrentChannel:TypeCode#">
<iselse>
  <isinclude template="product/ProducTabs">
</isif>
```

getCookie()

Every time the client sends a request to the server and a cookie is defined for the requested path, the cookie is also sent to the server with the current request.

The function `getCookie()` returns the value of such a cookie. You need to specify the name of the desired cookie as a function attribute. In case a cookie with the specified name has not been received, the function returns an empty string.

See `</ISCOOKIE>` about information on how to create a cookie on the client side.

Syntax

```
getCookie( <Key String - operation> )
```

Example

The example below first checks if a cookie with the `MyCookie` name exists, then it prints out the value of the specified cookie.

```
<ISIF condition = "#isDefined(getCookie('MyCookie'))#">
  <ISPRINT value = "#getCookie('MyCookie')#">
</ISIF>
```

getHeader()

Every time the client sends a request to the server, the HTTP protocol is used. The client can pass additional information about the request, and about the client itself, in request header fields. The function `getHeader()` returns the value of a request header field of the current request. You have to specify the name of the desired field as a function attribute. In case the specified header field does not exist, the function returns an empty string. For additional information, consult the HTTP 1.0 specification, available at the World Wide Web Consortium Web site, <http://www.w3.org>.

Syntax

```
getHeader( <Key String - operation> )
```

Example

In this example, `getHeader()` returns the value assigned to the header field `User-Agent`.

```
<ISPRINT value = "#getHeader('User-Agent')#">
```

getText(), getTextEx()

Are used to localize an attribute value of a (possibly custom) ISML tag. For that purpose `getText()` returns the plain text behind a given language neutral key. The function accepts the following arguments:

Table 83. `getText()` Parameters

Parameter	Description
key	a unique key for a translation
locale	an optional parameter to get the translation in a specific locale
site	an optional parameter indicating for which site the translated text should be presented

The following variations of the `getText()` function are possible:

```
getText(key : String) : String
getText(key : String, locale : String) : String
getText(key : String, locale : String, site : Domain) : String
```

`getTextEx()` supports another parameter named `params` that can be assigned an array of Objects to be injected into the translation.

The following variations of the `getTextEx()` function are possible:

```
getTextEx(key : String, ParameterList(params ... : Object)) : String
getTextEx(key : String, locale : String,
          ParameterList(params ... : Object)) : String ...
getTextEx(key : String, locale : String, site : Domain,
          ParameterList(params ... : Object)) : String ...
```

getValue()

This function can be used to format the result of an ISML expression. The value of an ISML expression is returned by `getValue()` as a formatted string. The standard formatter classes control how the string is formatted.

There are three ways to control the formatter classes:

- **use pre-defined formatter string**

The standard formatter classes come with pre-defined formatter strings. These are called styles.

- **use custom formatter string**

You can define and use your own formatter string with the `formatter` class.

- **use default formatter string**

Each formatter class has one default style. This style will be used if you do not specify anything else (neither a style nor a custom formatter string).

See *Formatting Expression Results* for more information.

Syntax

```
getValue( <value> )
```

```
getValue( <value>, <style> )
getValue( <value>, <formatter> )
```

Required Attributes

`value = ISML Expression`

Specify the ISML expression to be formatted. Usually this will be just a template variable.

Optional Attributes

The following attributes are optional:

■ **style**

`style = style identifier`

Specify a valid style to be used for formatting the expression result. See *Formatting Expression Results* for a list of valid styles.

■ **formatter**

`formatter = formatter string`

Specify a formatter string to control the formatting of the expression result. See the Java documentation in `<IS.INSTANCE.DIR>\docs` for more information about creating formatter strings.

Examples

The first example shows one of the styles in the formatter class:

```
<ISPRINT value="#getValue(Basket:TotalPrice,MONEY_LONG)#">
```

The next example shows how to specify a formatter string:

```
<ISPRINT value="#getValue(Offering:Weight,'#,##0.00')#">
```

Because neither a style nor a formatter string has been specified, the default style of the formatter class is used for formatting in the following example:

```
<ISPRINT value="#getValue(Purchaser:Address:Zip)#">
```

hasElements()*, *hasLoopElements()

The ISML function `hasElements()` allows you to check whether an iterator as stored in the pipeline dictionary contains any elements. This is particularly useful before looping through an iterator using a `<ISLOOP>` statement. If your iterator contains a number of elements different from 0, the function returns `true`, otherwise `false`. Note that `hasElements()` and `hasLoopElements()` are synonyms.

Syntax

```
hasElements( <ISML variable identifier> )
```

Example

```
<ISIF condition = "#hasElements(Basket:Offerings)#">Your basket contains:
```

```
<ISLOOP iterator = "Basket:Offerings">
    <ISPRINT value = "#Offerings:Name#"><BR>
</ISLOOP>
<ISELSE>
    <!-- Your basket contains nothing --!>
</ISIF>
```

hasNext()

The ISML function `hasNext()` allows you to check whether the iterator instance currently used in a loop contains any elements. This is useful, for example, in case the last loop element requires special treatment.

Syntax

```
hasNext( <loop iterator identifier> )
```

Example

```
<ISLOOP iterator="foo">
    ...
    <ISIF condition="#hasNext(foo)#">
        ... (do something except for last line)
    </ISIF>
    ...
</ISLOOP>
```

NOTE: If an alias name has been defined for the iterator, this alias has to be used for `hasNext()` as well.

isDefined()

This function checks if an ISML expression results in a valid existing value. Usually, this function checks a variable for its existence (either server or user defined). It can also be used with HTTP header functions to check for the existence of special keys.

Syntax

```
isDefined( <ISML expression> )
```

Example

This example shows how `isDefined()` can check whether a buyer has a basket or not.

```
<ISIF condition="#isDefined(Basket)">
    <ISCOMMENT> View Basket Content </ISCOMMENT>
<ISELSE>
    <!-- Basket does not exist --!>
</ISIF>
```

isSSEnabled()

With `isSSEnabled()`, you can check whether or not server-side scripting is enabled for the current store. If server-side scripting is enabled, the function returns `true`, otherwise `false`.

Syntax

```
isSSEnabled()
```

Example

```
<ISIF condition = "#isSSEnabled()#>
<%
// Java code
%>
<ISELSE>
<!-- Server-Side Scripting not enabled --!>
</ISIF>
```

lcase()

Returns a string that is converted to lowercase.

Syntax

```
lcase( <source string> )
```

Example

The following example returns the string "hello!"

```
<ISPRINT value = "#lcase('Hello!')#">
```

len()

Returns the number of characters in a string. If the source string is empty, `len()` returns 0.

Syntax

```
len( <source string> )
```

Example

The following example returns the value 9.

```
<ISPRINT value = "#len(' Hello! ') + 1#">
```

pad()

Justifies text to the left or right side of a field. For example, if you need to print out prices in a list, you probably want them exactly aligned at one side.

`pad()` has two attributes. The first specifies the source string to be processed. The second defines the field width as a positive or negative number of characters. A positive number produces left-aligned text, a negative number right-aligned text. If the source string is longer than the padding value, the output string is truncated on the right.

Syntax

```
pad( <source string>, <padding value> )
```

Example

```
<ISPRINT value = "#pad(Product:Price,-10)">
// returned string: " 5.00 EUR"
```

Required Attributes

The following attributes are optional:

- **source string**

```
source string = String
```

The string to be processed.

- **padding value**

```
padding value = Integer
```

Either a negative or positive integer value.

parameter()

Use `parameter()` in combination with the `url()` and `urlex()` functions. It returns an encoded URL key-value pair that is used to transmit data to the requested pipeline.

Syntax

```
parameter( <key>, <value> )
```

Example

This example shows a pipeline call with parameter passing.

```
<a href="#URL(action('PLProcessBasketActions'),
    parameter('action','remove'),
    parameter('quantity','1'),
    parameter('offeringID',Offering:UUID))#">
... link text to be displayed ...
</a>
```

Required Attributes

```
key = string
```

String for defining the name of the parameter.

```
value = string
```

String for defining the value of the parameter.

paramMap(), paramEntry()

Use `paramMap()` in combination with the `paramEntry()` functions to provide parameter sets for the `<ISPIPELINE>` tag. Note that this function cannot be used in other element contexts.

Syntax

```
ParamMap(ParamEntry('foo','bar'), ParamEntry('bar','foo'))
```

Example

In the example below, a pipeline to be executed in a template is defined, with the parameters passed to the pipeline using the `paramMap()` function.

```
<ispipeline
    pipeline="MyTestPipeline-LoginCheck"
    params="#ParamMap(ParamEntry('foo','bar'), ParamEntry('bar','foo'))#"
    alias="resultDict" />
```

Required Attributes

`key = string`

String for defining the name of the parameter.

`value = string`

String for defining the value of the parameter.

replace()

`replace()` can be used to reformat string values of attributes which consist of multiple elements, such as "red,green,blue,white". The function identifies a separator (e.g., a comma) and replaces it with a different separator (e.g., a white space). Going by example, the string value "red,green,blue,white" would be turned into "red green blue white", with the comma replaced by a white space.

Syntax

```
replace(<string variable>, <pattern>, <replacement>)
```

Note that `<pattern>` may be a regular expression to find multiple separators (e.g., comma and semicolon) and complex separators (e.g., ---) at once.

Example

The following example replaces a comma by a white space when printing the value of the string variable `foo`:

```
<ISPRINT value="#replace(foo, ',', ' ')#">
```

servlet()

It is sometimes desirable to generate dynamic content via servlets instead of pipelines. To do this, the ISML function `servlet()` is used.

It generates the part of an URL string that addresses Java application components that are downloaded, on demand, to the part of system that needs them.

As with `action()`, the function `servlet()` is only used within `url()` and `urlex()`.

Syntax

```
Servlet(String servletName)
Servlet(String servletName, String serverGroupName)
```

Example

The following example shows how to call the LineChart servlet, which generates a new URL with the requested servlet and the server group encoded.

```

```

The `servlet()` function within the `url()` function returns the following URL:

```
http://<webserver-host>:<port>/intershop.servlet/WFS/LineChart?title=Memory& ...
```

This URL contains the name of the servlet (`LineChart`) and the server group, e.g., `WFS`, which is also the default server group.

With the `intershop.servlet` mapping to `LineChart`, the Web adapter can address the `LineChart` servlet in an application server that is part of the `WFS` server group. Therefore, the Web adapter uses this URL:

```
http://<appserver-host>:<port>/servlet/LineChart?title=Memory& ...
```

sessionlessurl(), sessionlessurlex()

`sessionlessurl()` and `sessionlessurlex()` are used in templates to dynamically generate URLs as part of a link. As opposed to `url()` and `urlex()`, these functions generate URLs without session IDs.

For details about the function syntax, attributes, etc., refer to `url()` and `urlex()`.

split()

`split()` is used to break down the value of a complex string variable into individual elements which can then be treated individually. This allows you to iterate through complex string values, e.g., in a loop. Elements are broken down according to a specified pattern which serves as separator.

Syntax

```
split(<string variable>, <pattern>)
```

Note that `<pattern>` may be a regular expression to find multiple separators (e.g., comma and semicolon) and complex separators (e.g., ---) at once.

Example

In the following example, the string value of variable `foo` is broken down into individual elements. Elements are separated by a comma. Each element (except the last) is then printed with a following `
` tag:

```
<isset scope="request" name="lines" value="#split(foo, ',')#">
<isloop iterator="lines" alias="line">
  <isprint value="#line#">
  <isif condition="#hasNext(line)#"><br></isif>
</isloop>
```

stringToHtml()

The ISML function `stringToHtml()` returns the result of a string expression after converting any special characters, such as "<", ">", and "&" into their appropriate HTML encoding sequences. For details see *Encoding Expression Results*.

Syntax

```
stringToHtml(<operation>);
```

Example

This example shows that although the automatic encoding of <ISPRINT> has been disabled by setting encoding to off, you can encode the result of an expression by explicit use of the function stringToHtml().

```
<ISPRINT value = "#stringToHtml('<<<')#" encoding = "off">
// returned string: "<<<"
```

stringToWml()

stringToWml() returns the result of a string expression after converting any special characters, such as "<", ">", "&", and "\$" into their appropriate WML encoding sequences. For more information see *Encoding Expression Results*.

Syntax

```
stringToWml(<operation>);
```

Example

This example shows that although the automatic encoding of <ISPRINT> has been disabled by setting encoding to off, you can encode the result of an expression by explicit use of the function stringToWml().

```
<ISPRINT value = "#stringToWml('<<$')#" encoding = "off">
// returned string: "<<$$"
```

stringToXml()

stringToXml() returns the result of a string expression after converting any special characters, such as "<", ">", and "&" into their appropriate XML encoding sequences. For more information see *Encoding Expression Results*.

Syntax

```
stringToXml(<operation>);
```

Example

This example shows that although the automatic encoding of <ISPRINT> has been disabled by setting encoding to off, you can encode the result of an expression by explicit use of the function stringToXml().

```
<ISPRINT value = "#stringToXml('<<<')#" encoding = "off">
// returned string: "<<<"
```

trim()

Strips white space from the beginning and end of a string.

Syntax

```
trim( <source string> )
```

Example

The following example returns the string "Hello!":

```
<ISPRINT value = "#trim(' Hello !')#">
```

ucase()

Returns a string that is converted to uppercase.

Syntax

```
ucase( <source string> )
```

Example

The following example returns the string "HELLO!"

```
<ISPRINT value = "#ucase('Hello!')#">
```

url()

Web pages usually have links allowing a user to navigate from one page to another.

A link in a "standard" Web page includes a URL pointing to another Web page. If the user clicks on a link, the browser sends a request containing the URL to a server. In an Intershop 7 site, however, links do not reference other pages. Instead, Intershop 7 links trigger pipelines which again can generate storefront pages as a response.

The functions `url()` and `urlex()` are used in a template to dynamically generate a URL that is part of a link. Any reference in a storefront page needs to be generated by one of these function calls. Use the attribute `action` for generating that part of a URL that contains information about the storefront name, language/locale, currency, etc. The parameter attribute can be used to extend the URL with additional key-value pairs which can forward additional data to the called pipeline.

The `url()` function uses the protocol that triggered the pipeline. Therefore, if the pipeline was triggered via an HTTP request, the URL will use the HTTP protocol. If the pipeline was triggered via HTTPS, then that protocol will be used. By contrast, the `urlex()` function allows you to arbitrarily specify a protocol, independently of the one used in the current pipeline request.

Syntax

```
url( <action - operation> )
url( <action - operation>,(<parameter - operation>) )
```

Example

This example shows how to build a link in a storefront page to call a pipeline.

```
<a href =
  "#url(action('ViewBasketHistory-Start'))#"
  ... link text to be displayed ...
</a>
```

Required Attributes

action - operation

Result of an `action()` function call. For more information, see `action()`.

Optional Attributes

parameter - operation

Result of a `parameter()` function call. The `parameter()` attribute may be included as often as needed in this function call. For more information, see `parameter()`.

urlex()

`urlex()` extends the capabilities of `url()` by allowing you to specify a particular protocol, host, and port for the URL to be generated.

For instance, if you want to make sure that sensitive data like credit card numbers are transmitted safely, you might use the secure protocol HTTPS (rather than HTTP). In this case, you use the `urlex()` function to specify that a protocol other than HTTP should be used.

Syntax

```
urlex( <protocol>, <portNumber>, <action - operation> )
urlex( <protocol>, <portNumber>, <action - operation>,
       (<parameter - operation>) )
urlex( <protocol>, <host>, <portNumber>, <action - operation> )
urlex( <protocol>, <host>, <portNumber>, <action - operation>,
       (<parameter - operation>) ) ...
```

Example

```
a href =
  "#urlex('https','443',action('ViewBasketHistory-Start'))#"
  ... link text to be displayed ...
</a>
```

Required Attributes

The following attributes are required:

■ protocol

`protocol = string`

String that defines the part of a URL that contains information about the protocol. For example, you can set `protocol` to HTTPS or FTP.

■ host

`host = string`

String that defines the part of a URL that contains information about the host (for example, www.intershop.com).

■ **portNumber**

`portNumber = string`

String that defines the TCP/IP port number of the target pipeline.

NOTE: If <protocol>, <host> or <portNumber> are left unspecified or passed as empty strings, the according values are taken from the current request, or, if this fails, from the appserver.properties file.

■ **action**

`action - operation`

Result of an `action()` function call. For more information, see `action()`.

Optional Attributes

`parameter - operation`

Result of a `parameter()` function call. The `parameter()` attribute may be included as often as needed in this function call. For more information, see `parameter()`.

val()

`val()` returns the numbers contained in a string in floating point representation. If the string contains non-numeric characters, a zero is returned.

Syntax

```
val( <Source String Operation> )
```

Example

The following example returns the value -2.20:

```
<ISPRINT value = "#val('-3.2') + 1#">
```

WebRoot()

This function is used to reference static content, e.g., images or other external files, within ISML templates.

`WebRoot()` points to the static content directory of the current site (e.g. `share/sites/site_name/version-id/static`) or the cartridge-based content directories, where reference material is stored.

`WebRoot()` returns the `intershop.template.WebRootURL` property of the storefront and the current locale as a string. You can configure what `WebRoot()` returns globally, in the `config/cluster/appserver.properties` file.

By default, `WebRoot()` returns `/is-bin/intershop.static`.

For example, the following ISML code:

```

```

Results in the following HTML code:

```

```

Syntax

```
WebRoot()  
WebRoot(<servergroup>)  
WebRoot(<servergroup>, <locale>)
```

Optional Parameters

The following attributes are optional:

■ **servergroup**

`servergroup`

This parameter can be used to override the default server group. If left unspecified, the first value from the application server property `intershop.server.assignedToServerGroup` is used.

■ **locale**

`locale`

This parameter can be used to override the default locale. If left unspecified, the following fallback values will be used instead:

1. **the request's current locale**
2. **the lead locale**
3. **the locale will be designated as a missing parameter ("")**

Examples

To reference an external text file, e.g. a CSS definitions file:

```
<link rel="stylesheet" href="#WebRoot()#/general.css" type="text/css">
```

To reference an image of an offer within a site:

```

```

To reference the `george.gif` image in the `MyStorefront.isml` template. Locate the file `george.gif` in your file system, then add the following code to your template:

```

```

Save the changes and test the result in the storefront.

WebRootEx()

The `WebRootEx()` function complements the `WebRoot()` function. It is also used to reference static content. However, in contrast to `WebRoot()`, `WebRootEx()` always returns an absolute URL, hence is used in templates in which an absolute URL is required, such as in mail templates.

Syntax

```
WebRootEx(<protocol>,<host>,<port>,<servergroup>,<locale>)
```

Required Parameters

The following parameters are required:

■ **protocol**

This parameter can be used to override the default protocol ("HTTP").

The value of `protocol` is ignored (even if set) if `intershop.template.WebRootURL` or `intershop.template.HTTPSWebRootURL` (set in `config/cluster/appserver.properties`) already specify an absolute URL.

■ **host, port**

If these parameters are left unspecified, the values for `intershop.WebServerURL` (in case of "HTTP") or `intershop.WebServerSecureURL` (in case of "HTTPS") as specified in the `appserver.properties` are used.

The values for these parameters are ignored (even if set) if `intershop.template.WebRootURL` or `intershop.template.HTTPSWebRootURL` (set in `config/cluster/appserver.properties`) already specify an absolute URL.

■ **servergroup**

This parameter can be used to override the default server group. If not specified, the first value from the application server property `intershop.server.assignedToServerGroup` will be used.

■ **locale**

This parameter can be used to override the default locale. If left unspecified, the following fallback values will be used:

- 1. the request's current locale**
- 2. the lead locale**
- 3. the locale will be designated as a missing parameter ("")**

Examples

The following ISML code:

```

```

Pagelet Development

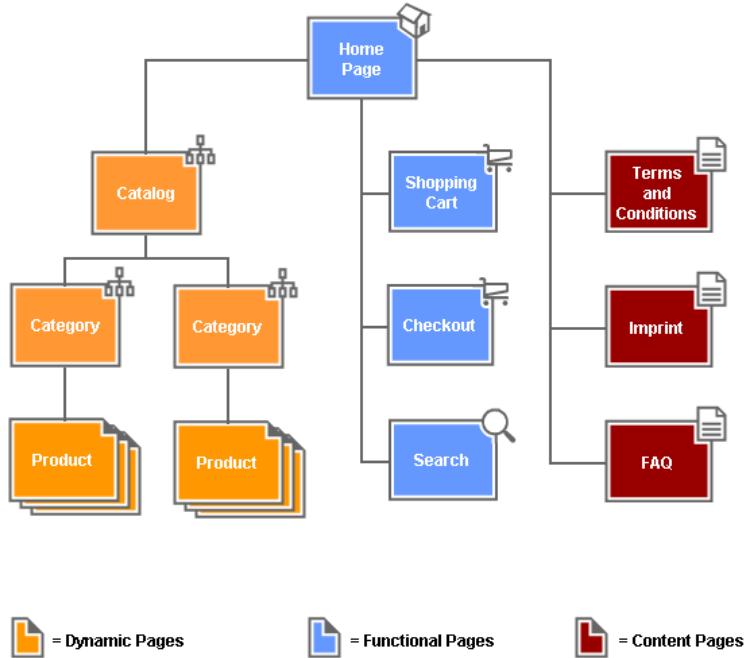
Content Management System (CMS) Development with Pagelets

Intershop 7 provides a content management system, which is based on a set of models and their implementations. Generally, they enable developers to model "pagelets", i.e., the page and component building blocks, for a storefront application, and users to create and edit the corresponding page and component instances in the Intershop 7 back office.

This section is addressed to Intershop Studio developers. It is intended to promote the understanding of the basic concepts of CMS development with Intershop Studio and their effects in the Intershop 7.

Introduction

Figure 160. Simple storefront structure



The diagram above shows a simple shop storefront structure with its elements partitioned in the way they are managed by CMS. The structure itself, i.e. the relations between the nodes or pages is not part of CMS, but is rather implicitly encoded in the viewing pipelines and ISML templates that come with the storefront application. The CMS is relevant when you want to arrange storefront content so that it is manageable and flexible (dependent on criteria such as time, user, promotions, etc.).

Even this reduced view of a storefront illustrates that many different versions and CMS modules are needed.

With a sufficient understanding of the concepts, as well as the generation and configuration of CMS blocks in Intershop Studio and the Intershop 7 back office it is possible to estimate the development needs of your storefront.

As already indicated, the different steps of CMS development take place at different times in different system layers.

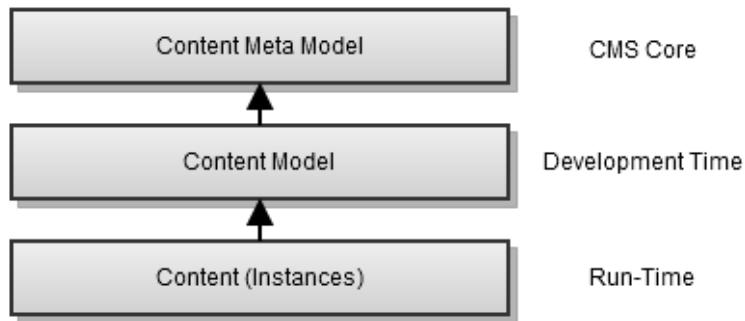
CMS Layer Model

Content development takes place in a model-driven manner, i.e. content instances are created and linked with each other as specified by a Content Model (or definition, from this perspective). The content model in turn represents the structure of elements and their relations according to certain rules and type restrictions. These types and rules are defined in a further model (class) that serves as the model for the content model (object) and is thus called Content Meta Model.

NOTE: Remember that a class is the blueprint from which individual instances are created. Such an instance is also referred to as an object.

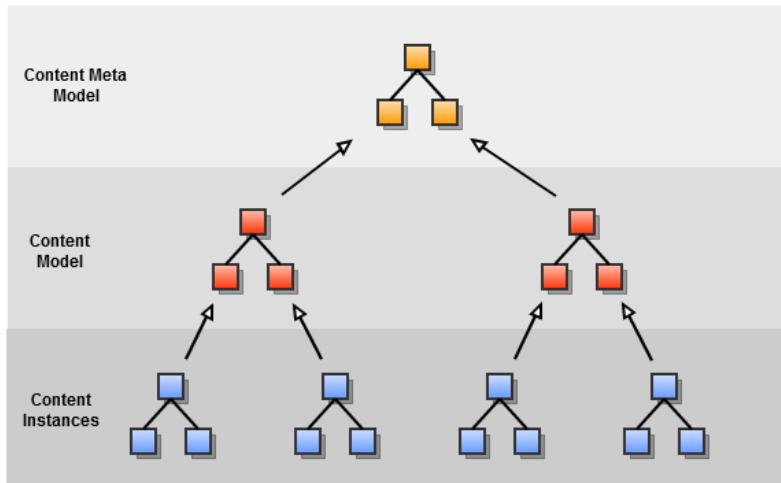
As is shown in the picture below, the Content Meta Model acts as a class for the object Content Model. The Content Model acts as a class for the object Content Instance.

Figure 161. Content Layer and Period



The content meta model embodies the core of the entire CMS. All content management and development tools as well as the entire content rendering in the storefront are either directly or indirectly based-on, or derived from it. It is part of the Intershop 7 platform and as such is fixed and cannot be changed by content developers or editors.

Figure 162. Layer's artifacts and their dependence



All items at the content instance layer and content model layer have a counterpart at the next higher layer. These inter-layer relations may be regarded as an 1:n relation with the single part of the relation being the model of the multiple part, i.e. one content meta model element has several counterparts at the content model layer, and each content model element has several counterparts at the content instance layer. Similar to classes and their objects in an object-oriented

programming language, the inter-layer relation in CMS may be also considered as a kind of "instance-of" relation.

Content Model Development

Intershop 7 provides a new content model that offers enormous flexibility for CMS developing. This content model is based on minor constituent parts, called content model elements. These content model elements are the component building blocks (called artifacts) of the Intershop 7 content management system. These content model elements enable you to determine Intershop 7 storefront content, its structure and configuration in Intershop 7 back office.

Basically, in Intershop Studio content model are created from the content meta model, edited and stored as *.pagelet2 files in cartridges. These content model are models for content instances derived in Intershop 7 back office. In other words, Intershop Studio creates the blueprints and only their derivations are used in Intershop 7 back office.

An content instance used in Intershop 7 is always derived from a content model (definition). Of course the instance inherits all properties of its blueprint. The instances are then stored in the database.

In simple terms Intershop Studio defines the options to compose the constituent parts of the CMS in Intershop 7.

NOTE: Many problems occur because people are talking about the same thing with different terms (albeit from different perspectives). Please ensure that you have a firm grasp of the terms used in this section and you have a good understanding of the offered features.

Content Model Elements

Pagelet Definitions

A pagelet can be considered as "container" which embody "real" content that is rendered in the storefront.

Page Variant Definition

A Page Variant Definition is a blueprint for a page variant used in Intershop 7. It can reference slot definitions. So the content instance (page variant) contains slots. Page Variant definition can reference a page entry point definition. So the content instance (page variant) can be assigned to pages (the instances of page entry point definitions or the instance of a system page entry point definition) only. Vice versa, a page variant cannot be added to an include nor to a slot. A page variant constitutes the "container" object that holds the actual content and must be assigned to a page.

Component Definition

A Component Definition is a blueprint for a component used in Intershop 7. A component constitutes a snippet of content. Component definition can reference slot definitions. So the content instance (components) contains slots. A component definition can have "component entry point extensions". These CEP extensions

pointing on component entry point definitions (include definitions) and slot definitions. So the content instance (component) fits into these includes or slots.

Content Entry Point Definition

Content Entry Points can be considered of as a spot where pagelets can be added.

Page Entry Point Definition

A Page Entry Point Definition is the blueprint for a page used in Intershop 7. Pages are accessible by a unique ID directly from a pipeline. They function as an element onto which (multiple) page variants can be attached. In contrast to includes or slots, pages cannot be part of a pagelet. That is, pages are the "root" element of a web page.

Component Entry Point Definition (Include Definition)

A Component Entry Point Definition is the blueprint for an include used in Intershop 7. Includes are accessible by a unique ID directly via an ISML tag. They function as an element onto which (multiple) components can be attached.

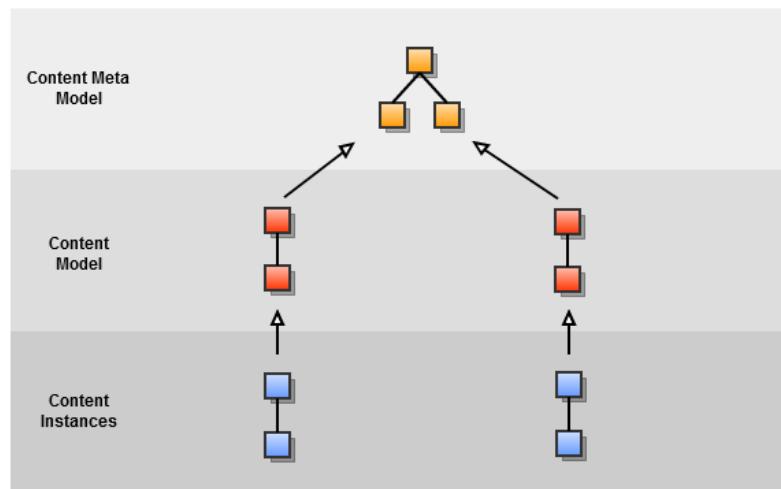
System Page Entry Point Definition (System PEP Definition) and System Component Entry Point Definition (System CEP Definition)

A System PEP Definition is a blueprint for a system page.

Typical examples of system pages are checkout pages, my account pages or even the home page of an online shop.

A System CEP Definition is a blueprint for a system include.

Figure 163. Instantiation of system managed page definition and include definition



In contrast to the standard content elements, the system managed elements do not have multiple instances but rather only have a single instance (compare with *Figure 162, "Layer's artifacts and their dependence"*).

The system itself guarantees the existence of a single instance over the entire life cycle of system managed elements. This content instance has the same unique

ID like its content model. By this unique ID the instance can be referenced during development time in Intershop Studio. That is, a system page or a system include cannot be deleted or created in Intershop 7 back office. In this sense, system managed content entry points constitute development time artifacts.

Slot Definition

A Slot Definition is a blueprint for a slot used in Intershop 7. A slot is used inside of a pagelet (page variant or component) to add other components.

A slot definition is referenced in the page variant definition or component definition. So the slot will be part of the content instance. In Intershop 7 back office other components can be assigned to the slot.

NOTE: A slot is always part of a pagelet, i.e. they cannot exist on their own in Intershop 7 back office.

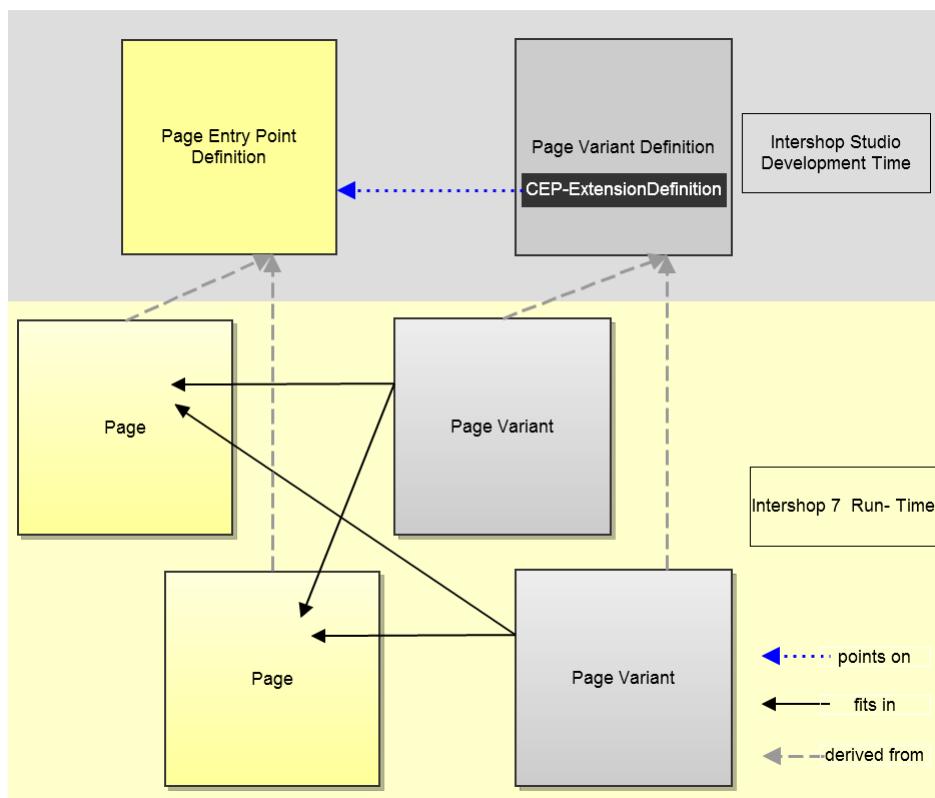
Relationship of Content Instances

This section shows how the different artifacts interact and outlines the effects of their interaction.

Page and Page Variant

As long as a page variant definition references a page entry point definition, their content instances (page variant and page) will fit together. The page represents a identified point in the storefront. So one task of a page is to link a page variant directly into the storefront structure at defined nodes. In other terms, the page variant requires a page, because it can be assigned to pages only.

An other possibility to ensure compatibility between page and page variant is the use of call parameter interfaces. If the content models (page entry point definition and page variant definition) reference the same call parameter interface definition, their content instances (page and page variant) support this call parameter interface. Thus, the two content instances are compatible.

Figure 164. Page and Page Variant

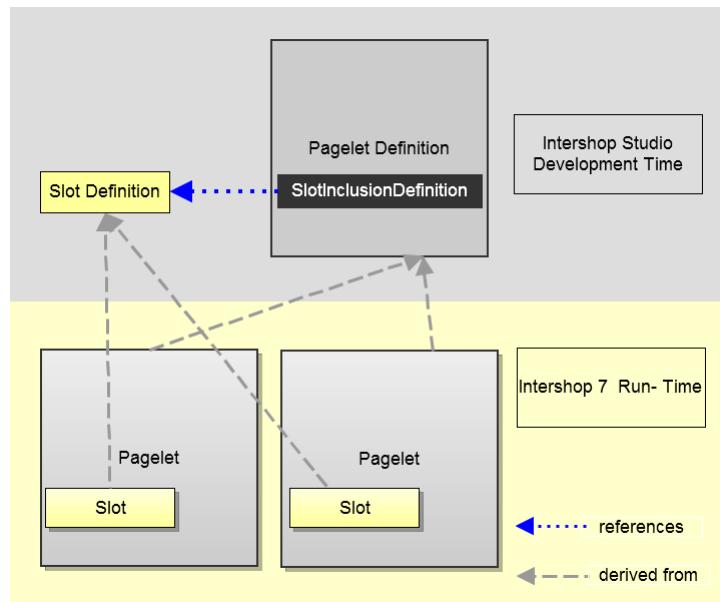
Different page variants can be dynamically assigned to the same page. Controlled by business logic, different content can be shown in the storefront with the same Uniform Resource Locator (URL). As a consequence, this offers the possibility to show different content on the same web page, for example, depending on time or user group.

System Page

As described in the section *System Page Entry Point Definition (System PEP Definition)* and *System Component Entry Point Definition (System CEP Definition)*, a system page is the only instance of a system page entry point definition, and the system itself guarantees the existence of this single instance. In this way it is ensured that these entry points cannot be deleted and that cannot exist more than one instance in Intershop 7, which is intended for certain web pages. Make sure, that you still have to add page variants to those system pages. They initially exist but they are empty.

Pagelet (Page Variant/Component) and Slot

Figure 165. Pagelet and Slot



If a page variant definition or a component definition references a slot definition, its content instance (page variant or component) contains a slot. Such a reference pointing a slot definition from a pagelet definition is called slot inclusion definition. Of course, depending on their definition, a page variant or a component can have multiple slots. This will require the number of slot inclusions within the pagelet definition.

Slot and Component

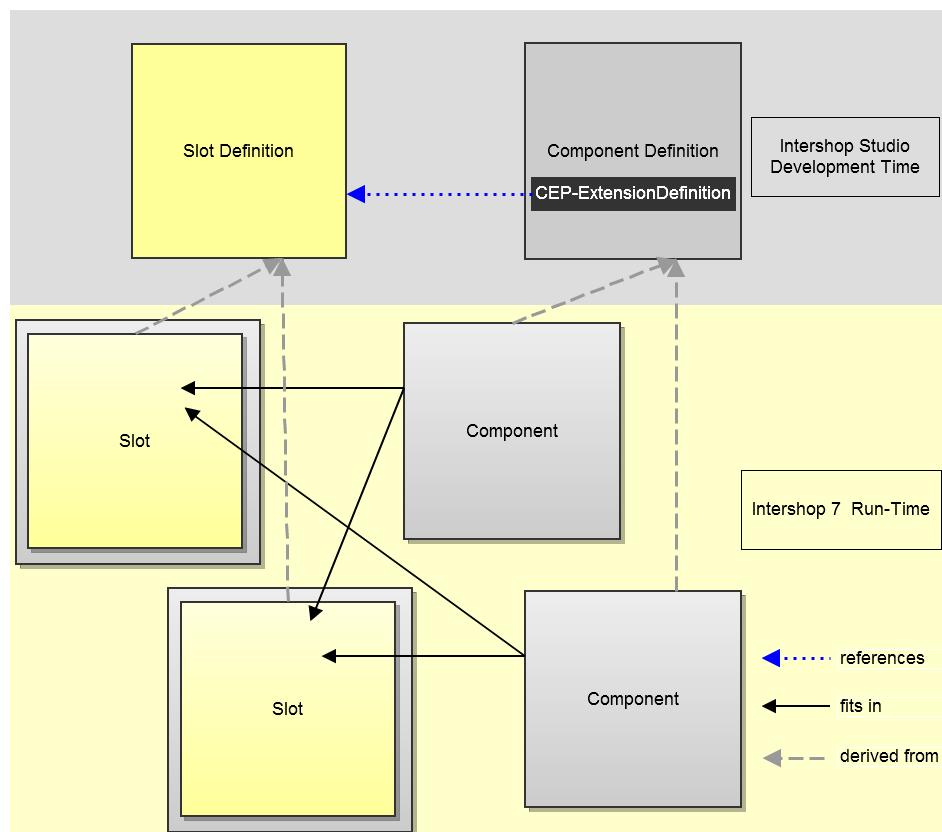
Slots are designed to be filled with components. The components provide the actual content displayed on the final page.

In Intershop 7 back office any component which fits into a slot can be assigned to this slot. The number of components displayed in the slot depends on the configuration of the slot.

There are two different concepts to make a component fit into a slot.

Content Entry Point Extension Definition

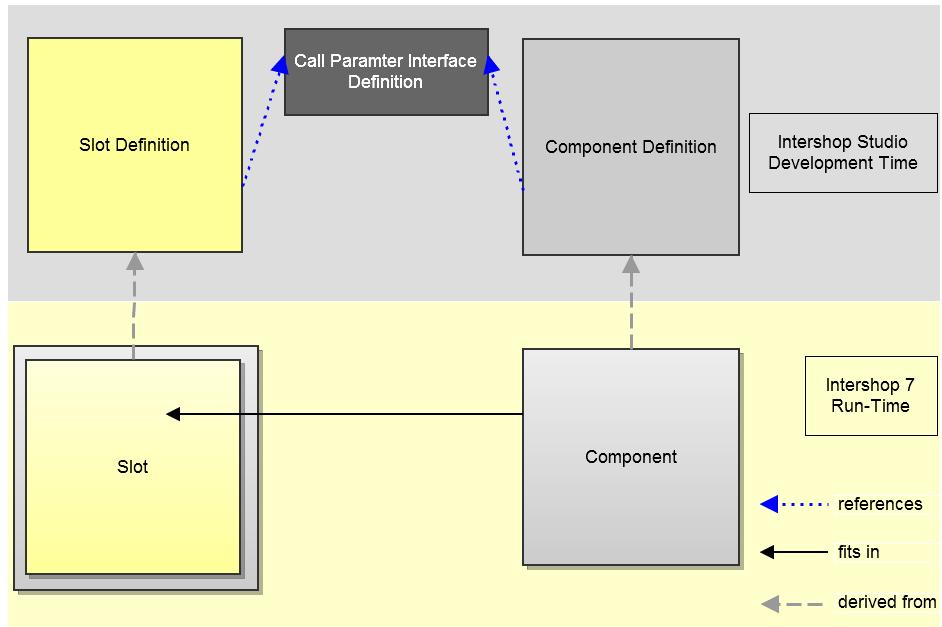
Figure 166. CEP Extension



One way to make a component fit into a slot is to target the slot definition explicitly from the component definition. If a component definition contains a slot extension definition that is targeting the slot explicitly, the content instance (component) can be assigned to the slot in the Intershop 7 back office.

Call Parameter Interfaces

Figure 167. Call Parameter Interfaces



Another way to specify a component to fit into a slot is the use of call parameter interfaces.

In Intershop Studio you can reference call parameter interface definitions within slot definitions and component definitions. So the content instance (component) supports the call parameter interface. Any content instance (component) supporting the same call parameter interface as a slot, fits into the slot and can be assigned in Intershop 7 back office.

NOTE: In Intershop 7 back office a slot can not exist on its own. A slot is always part of a pagelet (page variant or component). To assign a component to a slot, the slot must be ingredient of a "larger" pagelet (variant page or component). See *Pagelet (Page Variant/Component)* and *Slot*.

If a component definition is targeting a slot definition explicitly via slot extension definition, the content instance (component) can only be assigned to the specified slot. If a existing component should be used in other slot, the content model (component definition) has to be adjusted. The component definition must have a further slot extension definition pointing on the other slot definition. This may be inflexible and cause maintainance problems.

Existing component definitions do not have to be adjusted to a new slot definition, as the new slot definition can reference the call parameter interface which is also referenced by the component definition.

Note, that using call parameter interfaces to make components fit into slots may be more difficult in Intershop Studio, because you cannot see directly into which slots a component fits. You have to list all slots which support the same call parameter interface.

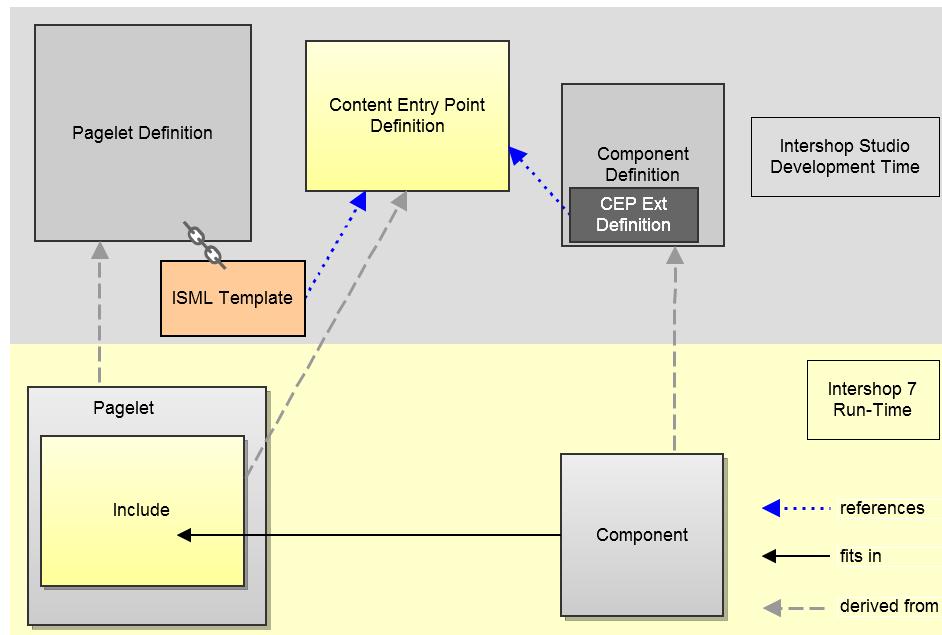
Composition

As described above, a pagelet (page variant or component) can contain slots. In these slots other components can be filled, i.e. these components can contain more

slots and so on. As long as the right pagelet definitions are defined in Intershop Studio, in Intershop 7 a content developer can compose these pagelets to every intended web structure.

Component and Component Entry Point (Include)

Figure 168. Component and Component Entry Point (Include)



In Intershop Studio, a component definition can reference a component entry point definition (include definition). This effects, that the content instances fit together. Another pagelet definition's ISML template can reference this component entry point definition via ISML-tag `<iscontent include...>`. The pagelet definition and the include definition are firmly connected in Intershop Studio. This connection is not visible within the pagelet definition. The connection is only visible in the ISML template.

By assigning the content instance of the component definition (component) to the content instance of the component entry point definition (include), the component is bound to the larger pagelet via the include.

Another possibility to ensure compatibility between component entry point (include) and component is the use of call parameter interfaces. If the content models (component entry point definition and component definition) reference a call parameter interface definition, their content instances (component entry point (include) and component) support this call parameter interface. Thus, the two content instances are compatible.

Advanced Concepts

Context Object Relation (COR)

A context object relations allow to define different "views" in various contexts for business objects.

To define a new "view", COR itself has to be defined in Intershop Studio as a content model. A content entry point definition can be extended to reference this COR.

Content editors in Intershop 7 back office cannot see a context object relation. They only see includes and pages and can assign a pagelet to such an content entry point, implicitly making use of CORs. The COR finds the appropriate content entry point and thereby the desired pagelet, in which structure the content will be rendered.

A pagelet definition may extend several CORs. This leads to more available "views" in Intershop 7.

Other CMS related Resources

Pipelines

Several pipelines are used in context of content element definitions.

Render Pipelines

Render Pipelines are assigned by page definitions and are therefore content-specific.

Pagelet Entry Point Pipeline

These pipelines define the lookup strategy for context object relations for special pieces of content like a product summary view in a product list. "Context object relation", another concept provided by the CMS framework (see *Context Object Relation (COR)*). The implementation of the lookup strategies is specific to the storefront.

During run-time the desired content entry point for a pagelet can be sought by the pipeline.

ISML Templates

Similar to pipelines, ISML templates are referenced by pagelet definitions (component definitions or page variant definitions) as render template which will produce the HTML output of the page variants or components.

Content Templates

Templating is a concept that offers the ability to compose complex structures from CMS elements.

In addition, content templates are a way to reduce the enormous complexity of the structural composition of pagelet nestings.

Templating is a task for content developers in Intershop 7 back office.

Essential to this templating in Intershop 7 back office are three concepts:

■ Page Template

Page templates constitute page variant "models". A page template is based on a page variant definition, created in Intershop Studio, thus defining the structure

and rendering parameters for the page variants derived from the template. Page templates can be "prefilled" with components (which cannot be modified in derived page variants) and can hold placeholders.

■ Component Template

Component templates constitute component "models". A component template is based on a component definition, created in Intershop Studio, thus defining the structure and rendering parameters for the components derived from the template. Component templates can be "prefilled" with other components (which cannot be modified in derived components) and can, if allowed by the component definition, hold placeholders.

The use of component templates allows for reuse. You do not need to create each page from top to bottom as a standalone template, but can reuse component templates.

■ Placeholder

Placeholders define editable areas in page templates, component templates, page variants and components. If provided by their corresponding definitions, page or component templates can "propagate" their slots to their derived page variants or components via placeholders.

Content templates are based on pagelet definitions with all its dependencies, configurations, etc., especially the slots referenced in them (see *Relationship of Content Instances*). Depending on the needs of the storefront all the necessary content model have to be created and configured in Intershop Studio.

NOTE: In Intershop Studio a content model that will be instantiated first as a content template is not different to a content model, which is immediately instantiated as a page-variant or be component in Intershop 7 back office.

Pagelet Model Editor

The Pagelet Model Editor offers the possibility to edit all pagelet2-based constituent parts of CMS.

Pagelet2 models are generated, edited and managed as separate files in the Intershop Studio. This makes it possible to reuse the models in Intershop Studio and reuse their instances in Intershop 7 back office and to compose complex nestings of their instances in Intershop 7 back office.

User Interface

Pagelet Development Perspective

Intershop Studio supports various perspectives for different tasks. Perspectives are arrangements of menus and toolbars, views and editors. They are highly configurable and custom configurations can be saved and loaded. The perspective from which the following descriptions derive is the "Pagelet Development"

perspective. To clearly understand the arguments, it makes sense to apply this perspective.

- 1. Click Open Perspective icon .**

- 2. Select Other.**

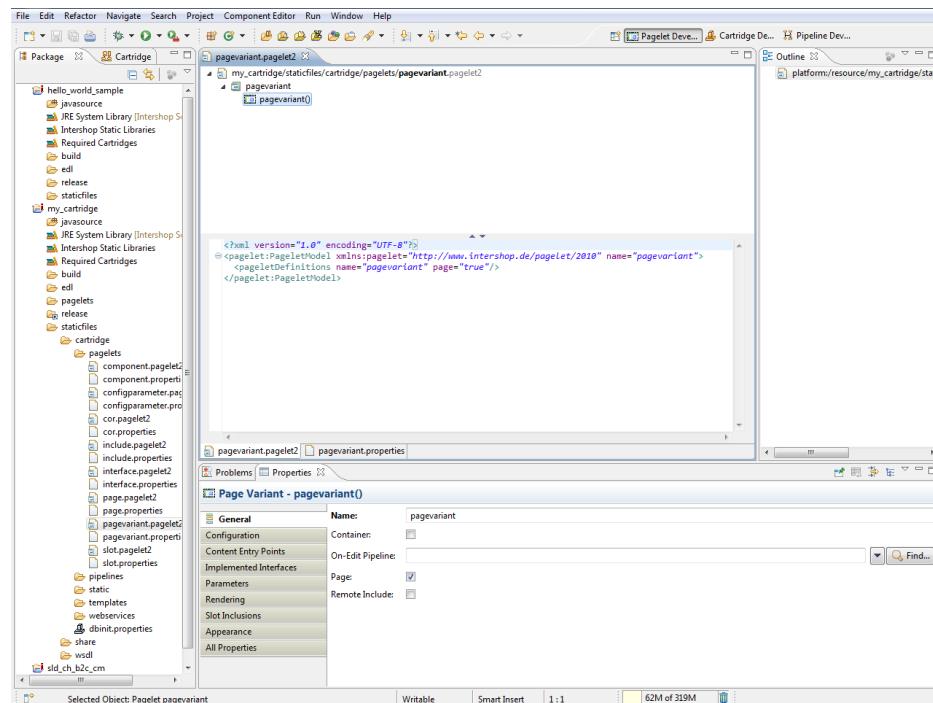
Open Perspective dialog starts.

- 3. Select Pagelet Development.**

- 4. Confirm dialog with OK.**

Pagelet Development perspective is applied.

Figure 169. GUI with Pagelet Development Perspective

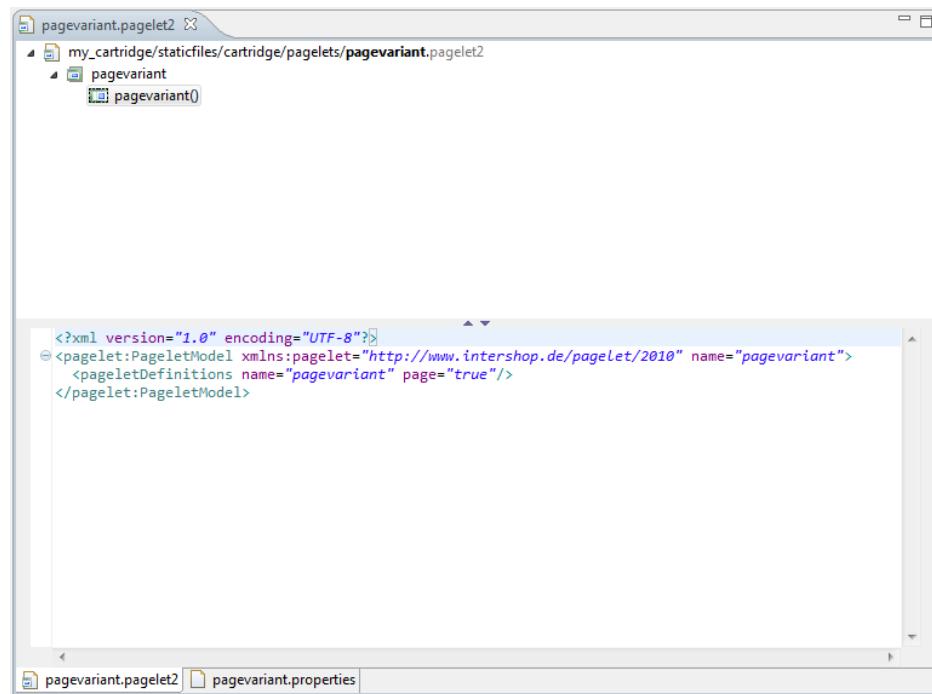


Central Pagelet Model View

This view contains the main parts. The upper section displays a graphical representation of the pagelet within an expandable data tree structure. The pagelet elements are represented here by specific icons and designations. This view can be used for the navigation between the elements of a pagelet. Via context menu (right click a pagelet's element) the elements can be edited, e.g. by adding new child elements.

The lower section offers an XML plain text editor with syntax highlighting. A pagelet can be edited by writing the necessary elements, attributes, etc. with a plain text editor.

There buttons under the editor enables views to switch between a pagelet2 file and its properties file. A pagelet's properties file is created automatically when a pagelet is created. It is an appendage that is important for the localization.

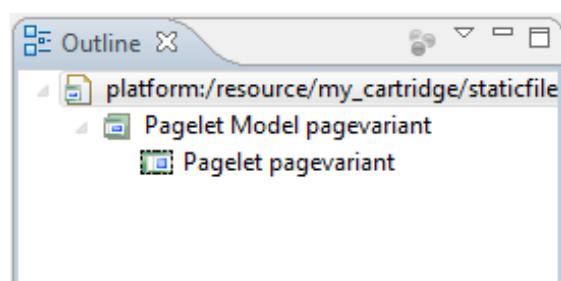
Figure 170. Central Pagelet Model View

Properties View

Depending on the chosen element of the pagelet model, the Properties View shows various tabs. These tabs contain different properties that can be directly adjusted via buttons, dialogs or text boxes.

Outline View

The Outline View of the Pagelet Model Editor offers a data tree structure of the current pagelet. These data tree structures can also be used for navigation. In contrast to Central Pagelet View, the element's terms are written out. This may be a simpler means in some cases to orient yourself in the data tree structure.

Figure 171. Outline View

Open a Pagelet2 Model

Double click a pagelet2 model in Package Explorer. Intershop Studio opens the model with the editor which was used to edit the model in the previous session.

This should be the Pagelet Model Editor. If, for any reason, it is not the Pagelet Model Editor:

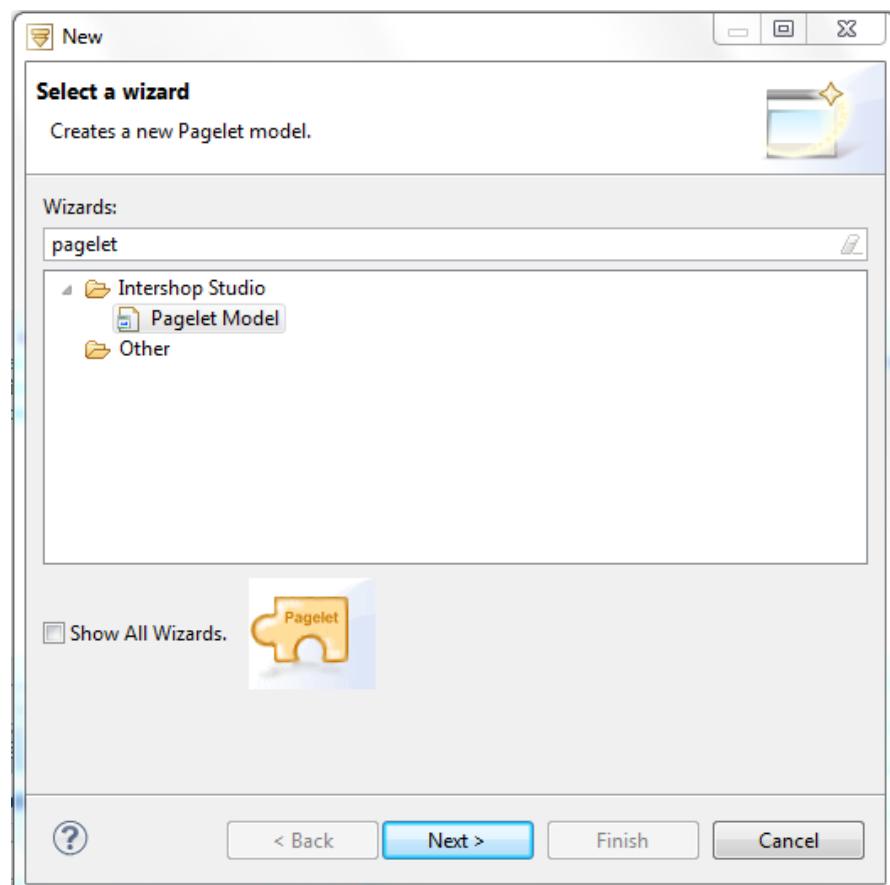
1. **Right click the pagelet2 model.**
2. **Click Open with.**
3. **Click Other.**
4. **Select Pagelet Model Editor from Editor Selection dialog.**
5. **Confirm dialog with OK.**

The pagelet2 model is opened in Pagelet Model Editor.

Create a new Pagelet2 Model

1. **Select the desired cartridge in the Cartridge Explorer.**
 2. **Click File.**
 3. **Click New.**
 4. **Click Other.**
- The New dialog opens.
5. **Type "Pagelet Model" in the text box.**

Figure 172. Pagelet Model Wizard



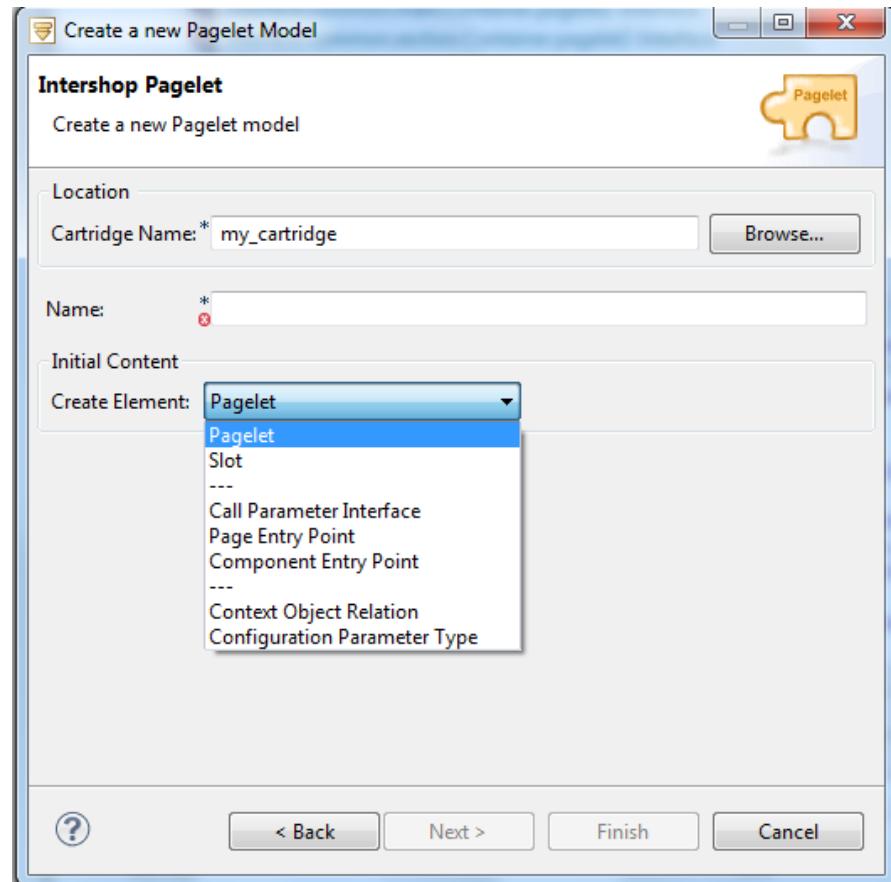
6. Select Pagelet Model.**7. Confirm dialog with Next.**

The Create a new Pagelet Model dialog opens.

8. Select the correct cartridge.

If the cartridge is selected in first step, it is preselected in Create a new Pagelet Model dialog. Otherwise the cartridge has to be chosen from Select Cartridge dialog.

Figure 173. Create a new Pagelet model dialog

**9. Name the pagelet model.****10. Select the desired pagelet model type from the Create Element drop-down-list.**

Available are: Pagelet, Slot, Call Parameter Interface, Page Entry Point, Component Entry Point, Context Object Relation and Configuration Parameter Type.

11. Confirm dialog with Finish.

The new pagelet model is opened in Pagelet Model Editor.

Edit a Pagelet2 Model

The configuration possibilities of a pagelet model depend on the type of the pagelet model. As described above the pagelets can be configured in the central pagelet model view (within the data tree via context menu or in the plain text editor via typing xml code) or in the properties view.

At the beginning of a model has three levels in the data tree structure. The highest level is the file level. The file level shows the file path. The middle level is the Pagelet Model level. The lower level is the definition or element level.

Figure 174. Data Tree Structure of Pagelet Model



The following descriptions refer to the configurations at the third level, the element level. Open or create a pagelet model. Expand the data tree. Select the element level. Select Properties tab to display the Properties view.

Pagelet Definition (Content Unit)

Page variant definition and component definition are made from the same meta model.

General Tab

Figure 175. General tab of page variant definition/component definition

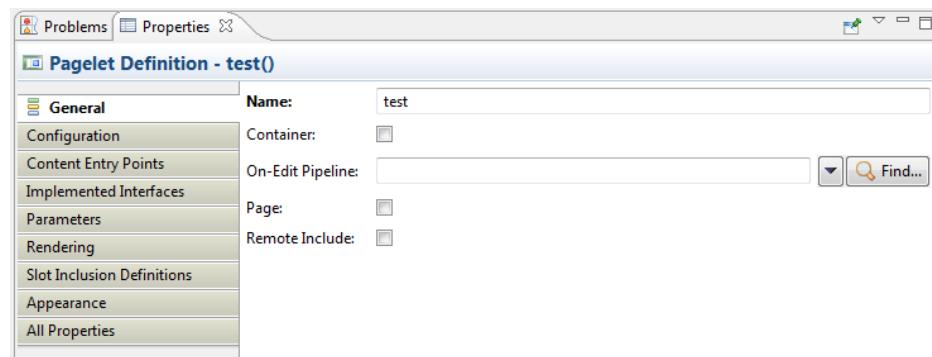


Table 84. Pagelet's General Tab

Property	Description
Name	Edit Name to change pagelet model's name attribute. Default value is the chosen file name.
Container	Set Container flag to <code>true</code> effects that instances of this pagelet definition do not have an explicit parameter mapping. This setting is useful for structuring elements, which will pass the content of the pipeline dictionary unfiltered. The default value is <code>false</code> .
On-Edit Pipeline	Values that are defined in Intershop 7 back office can be validated via On-Edit Pipeline.
Page	Set page flag to <code>true</code> will specify the pagelet definition as a page variant definition. The default value is <code>false</code> and specifies the pagelet definition as a component definition.

Property	Description
Remote Include	Set Remote Include flag to <code>true</code> makes it possible that the instances of this pagelet model are rendered separately. This allows, e.g., to render a parent component only once a day again, but the instance of this pagelet model every hour. The default value is <code>false</code> .

Configuration Tab

Configuration tab allows to manage element's configuration parameters. Via Configuration tab, configuration parameters can be created, added, removed and sorted. Configuration Parameters are the actual information-bearing elements of the CMS. These define the configuration options of the pagelet's instance in Intershop 7 back office.

1. Click New.

Select Element dialog starts.

2. Select a matching item.

Available are New Configuration Parameter and New List.

3. Select New Configuration Parameter.

4. Confirm dialog with OK.

New Configuration Parameter dialog starts.

1. Insert a name.

2. Select the required java type.

Available java types are: String, Float, Character, Boolean, Double, Integer, Long, Short, Date and Regular Expression Pattern. The default value is String.

3. Reference the type of this configuration parameter.

In the supplied cartridges are configuration parameter types included. Of course, you can create new or edit existing configuration parameter types (see *Configuration Parameter Type*).

4. Set a default value.

5. Type in a short description.

6. Type in the display name.

Table 85. Properties of New Configuration Parameter dialog

Property	Description
Optional	Set flag to <code>true</code> makes it an optional configuration parameter. That is, for the content manager in Intershop 7 back office it is not necessary to enter this value. The default value is <code>false</code> , which makes it a required parameter value. The content manager in Intershop 7 back office must enter this value.
Localizable	With this setting it is defined, whether the values of the parameter are localizable or not. That is, whether the parameter's value, which can be entered in the Intershop 7 back office, can be translated.
Visible	Set this flag to <code>false</code> effect, that the parameter is invisible in Intershop 7 back office. So the content manager cannot enter its value. This setting can be helpful to implement a configuration

Property	Description
	option that is utilized only in the future by the flag set to <code>true</code> . The default value is <code>true</code> . Note: If a parameter is not visible, you should make sure that it is optional. Required parameters, which cannot be entered in Intershop 7 back office, because they are invisible, cause errors.
Multiple	This setting determines whether the content manager in Intershop 7 back office can specify a single value or multiple values for this parameter. The default value is <code>false</code> .

In addition to simple parameters and parameter lists are possible. Parameter lists allow for multiple selection in Intershop 7 back office, eg a drop-down list or multiple check boxes.

Content Entry Points Tab

This tab allows to manage definition's content entry points. You can reference content entry point definition's, like page definitions, include definitions or slot definitions, as a content entry point extension to a content unit (page variant definition or component definition). To do so ensures that the instance of the content unit (page variant or component) in Intershop 7 back office can be assigned to the instance, the page, slot or the include.

NOTE: This does *not* mean, that the instance of the content unit definition contains a slot, page, or include. It means that the instance fits into a slot, include or page. Content entry point extensions, like slot extension or include, should only be referenced within a component definition. A page variant represents an HTML page. Therefore page variants should be assigned to pages only. It would be pointless to enable a page variant definition to produce instances that fit into slots or includes.

To get further information how to create a content unit definition, whose instance, contains slots, see section *Slot Inclusions Tab*.

Parameter Mapping:

The component definition references via its content entry point extension a content entry point (slot definition or an include definition). So the instance, the component fits into this slot or include, derived from the definition. Due to different naming of specific parameters, it may be necessary to reclassify parameters. This is done by parameter mapping. A parameter mapping links an appropriate pipeline dictionary object (source) to the call parameter (target).

Page variant definition and page definition behave as the same as component definition and include definition.

1. Right click content entry point extension in the data tree structure.

Context menu opens.

2. Select Parameter Mappings | Parameter Mapping.

A new parameter mapping is added to the content entry point extension. Parameter mapping's general tab is open.

3. Type in Source Parameter's name.

The source parameter is listed outside the content unit definition (page variant definition or component definition). This parameter is part of the pipeline dictionary. The parameter can be noted within the content entry point definition (page definition, include definition or slot definition).

4. Type in Target Parameter's name.

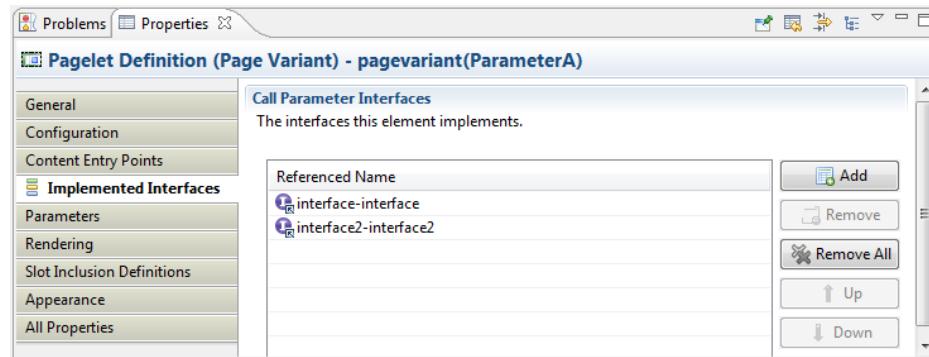
The target parameter is listed within the content unit definition. A content assist, listing all defined call parameters of the pagelet, is available.

Configuration parameter can be mapped in the same way, except that in second step's selection is Configuration Mappings / Parameter Mapping.

Implemented Interfaces Tab

This tab allows to add, remove and sort preconfigured interfaces.

Figure 176. Implemented Interfaces tab



Parameters

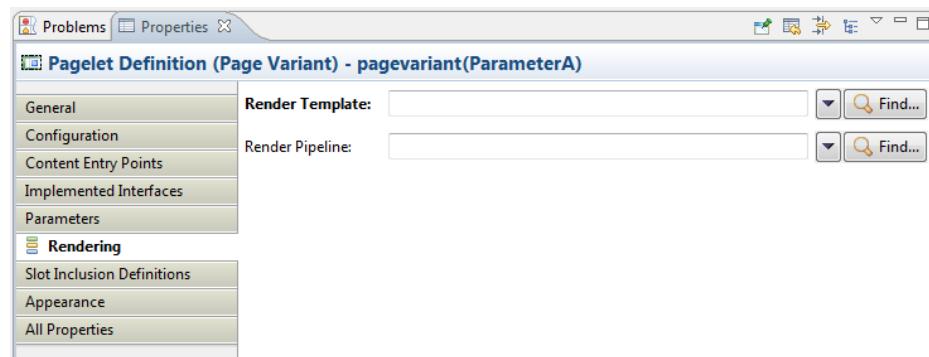
This tab allows to create, add, remove and sort call parameters.

NOTE: The assigning of call parameters is simplified by the use of interfaces considerably, see *Call Parameter Interfaces*.

Rendering

This tab allows to add pagelet's render template and render pipeline.

Figure 177. Rendering tab



Slot Inclusions Tab

This tab allows for managing slot inclusions. Reference a slot definition within a content unit definition effects, that the instance (component or page variant) contains a slot.

NOTE: The referenced slot must be of course available. This means that a corresponding slot definition must be created. To get further information how to configure a slot definition, see *Slot Definition*.

Parameter Mapping:

The content unit definition (page variant definition or component definition) references via its slot inclusion a slot definition. So the instance, the page variant or the component contains a slot, derived from the slot definition. Due to different naming of specific parameters, it may be necessary to reclassify parameters. This is done by parameter mapping. A parameter mapping links an appropriate pipeline dictionary object (source) to the call parameter (target).

- 1. Right click slot inclusion in the data tree structure.**

Context menu opens.

- 2. Select Parameter Mappings | Parameter Mapping.**

A new parameter mapping is added to the slot inclusion. Parameter mapping's general tab is opened.

- 3. Type in Source Parameter's name.**

A content assist, listing all defined call parameters of the content unit definition (page variant definition or component definition), is available.

- 4. Type in Target Parameter's name.**

The target parameter is listed outside the content unit definition, but within the referenced slot definition.

Configuration parameter can be mapped in the same way, except that in second step's selection is Configuration Mappings / Parameter Mapping.

Appearance Tab

This tab allows to edit the Display Name and the Group and to write down a short Description to the pagelet. These information are written to the pagelet's properties file and important for localization.

All Properties Tab

This tab lists all properties have been set in the other tabs. By double clicking the value of a property you can set or edit all properties within the All Properties tab.

Slot Definition

General Tab

Figure 178. Slot's General Tab

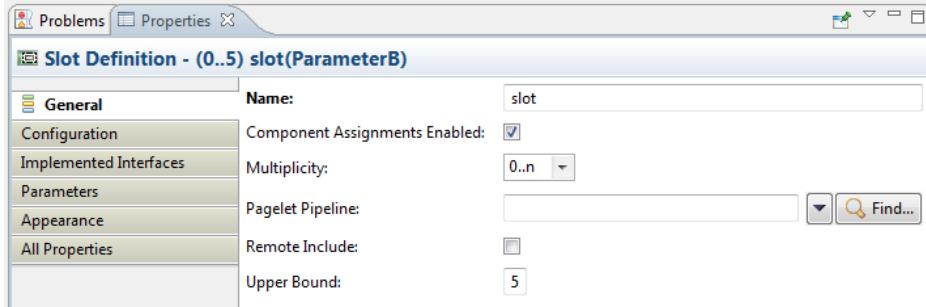


Table 86. Slot's General Tab

Property	Descriptions
Name	Edit Name change slot definition's name attribute. Default value is the chosen file name
Components Assignments Enabled	If a pagelet pipeline is used that would prevent the assignment of components to the instance of this slot definition in Intershop 7 back office, this flag must be set to <code>true</code> to permit the explicit assignment by the content manager Intershop 7 back office.
Pagelet Pipeline	A pagelet pipeline allows the fetching of objects according to the business logic from the database. This setting allows the reference to a pipeline. Via a pipeline it is possible to determine components of this slot.
Multiplicity	Editing the Multiplicity allows the configuration of the number of components that are rendered in this slot in the storefront at the same time. A multiplicity from 0 to 1 allows the rendering of a single component. That means, the content manager in Intershop 7 back office can assign a lot of components, but only one component can be rendered at the same time. A multiplicity from 0 to n allows the rendering multiple components at the same time.
Upper Bound	Upper Bound defines a maximum value for the variable n. For example: set Multiplicity to 0 to n and set upper bound for n to 5 will have the affect, that, if assigned in Intershop 7 back office, 5 components can be rendered in the slot in the storefront at once.
Remote Include	Setting the Remote Include flag to <code>true</code> makes it possible that all components which are assigned to the instance of this slot definition, the slot, are rendered separately. The default value is <code>false</code> .

All other tabs provide the same functionality as demonstrated in the section *Pagelet Definition (Content Unit)*.

Call Parameter Interfaces

An interface is a set of parameters. These parameters make it possible to define compatibility between individual pagelet2 models. The parameters may be defined in the pagelet2 models themselves. But in this case a parameter must be

defined in each parent element and child element. The use of interfaces allows the definition of these parameters only once and just references the interface in the pagelet2 models.

General Tab

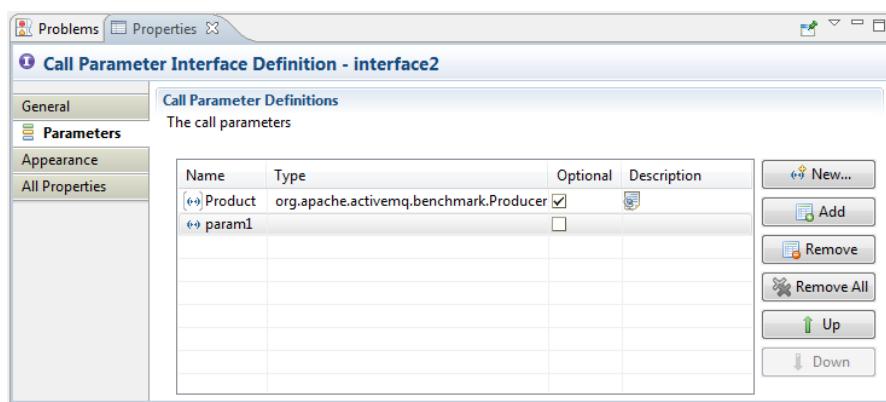
The interface General tab only offers the editing of the interface name. As described, the intent of interfaces is the allocation of call parameters. For this purpose, the parameters must be assigned to the interface and configured.

Interface's Parameter Tab

This tab allows for the management of an interface's call parameters.

1. Select Parameters tab.

Figure 179. Manage interface's parameters



2. Click New.

Select Element dialog starts.

NOTE: A ready-configured parameter can be added, if it is already in use anywhere in the cartridge.

3. Select New Parameter.

4. Confirm dialog with OK.

New Parameter dialog starts.

5. Insert parameter's name.

6. Select parameter's java type.

7. Type in a short description.

8. Specify whether the parameter is optional or required.

9. Confirm dialog with OK.

The parameter is added to the interface.

Of course, an interface can contain several parameters.

Parameters can be removed from the interface.

- 1. Select a parameter.**

- 2. Click Remove.**

The parameter is removed from the interface.

If an interface contains several parameters, parameters can be sorted.

- 1. Select a parameter.**

- 2. Click Up.**

The parameter slips into the list one position up.

- 3. Click Down.**

The parameter slips into the list one position down.

All other tabs provide the same functionality as known from the section *Pagelet Definition (Content Unit)*.

Page Entry Point Definition (Page Definition)

General Tab

Figure 180. Page Entry Point's general tab

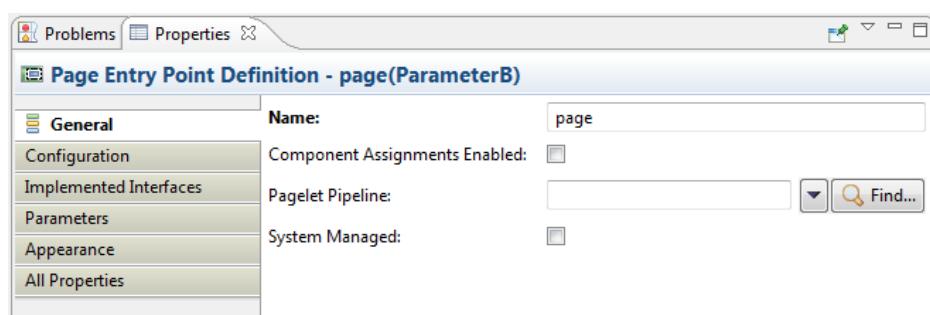


Table 87. Page Entry Point's General Tab

Property	Description
Name	Edit Name change page entry point definition's name attribute. Default value is the chosen file name.
Components Assignments Enabled	If a pagelet pipeline is used, which would prevent the assignment of page variant to the instance of this page definition in Intershop 7 back office this flag set to true, thereby permitting the explicit assignment by the content manager Intershop 7 back office.
Pagelet Pipeline	A pagelet pipeline allows the fetching of objects according to the business logic from the database. This setting allows to reference a pipeline. Via a pipeline it is possible to determine page variants of this page entry point definition.
System Managed	This flag when set to <code>true</code> , turns the page entry point definition into a system-page entry point definition. A system managed page entry point is a development time artifact with an unique ID. That is, unlike all non-system managed pagelet2 models, the instance of system managed page entry point is not created by a content

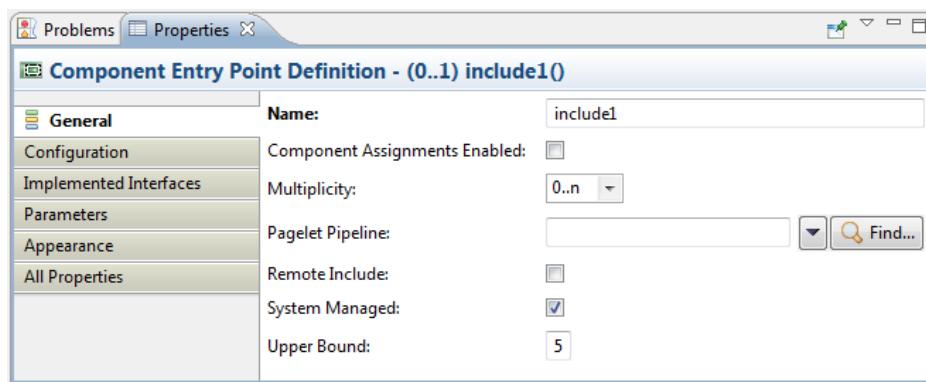
Property	Description
	manager in Intershop 7 back office, but are already present when the server starts. The system page entry point definition has only one instance, system page, in Intershop 7 back office. The existence of this system page, is managed by the system itself. That is, a system page cannot be deleted or created in Intershop 7 back office. The default value of the System Managed setting is false .

All other tabs provide the same functionality as known from the *Pagelet Definition (Content Unit)*.

Component Entry Point Definition (Include Definition)

General Tab

Figure 181. Component Entry Point's General Tab



The General Tab of content entry point definition offers largely the same options as the *General Tab* of slot definition. Furthermore, a content entry point can be system managed like a page entry point (see section *General Tab*). The functions of these settings are the same.

All other tabs provide the same functionality as known from the section *Pagelet Definition (Content Unit)*.

Context Object Relation Definition

General Tab

Figure 182. Context Object Relation's general tab

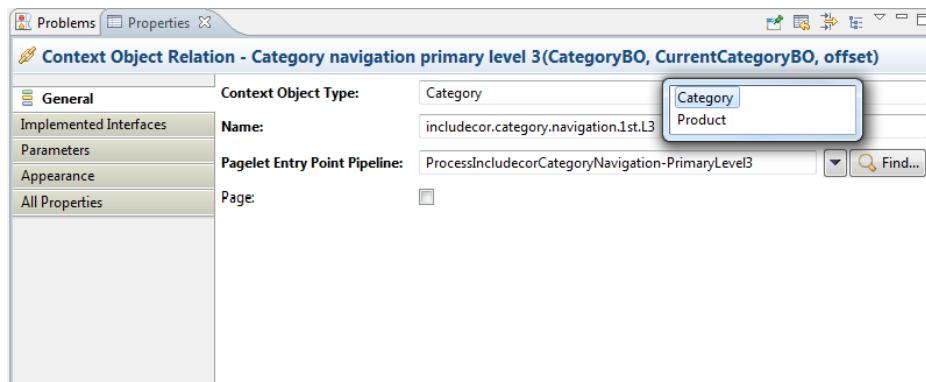


Table 88. Context Object Relation's General Tab

Property	Description
Context Object Type	References the type of the element. There are two types available: Product or Category.
Name	Edit Name change context object relation definition's name attribute. Default value is the chosen file name.
Pagelet Entry Point Pipeline	Here, the pipeline is referenced to find the desired content entry point.
Page	This setting determines whether this is a context object relation for a page entry point or component entry point. The default value is <code>false</code> which effect a context object relation for a component entry point (include). In Intershop 7 back office the content manager does not see any context object relation. The content manager assigns include or a page to a content unit (page variant or component). This flag determines in which "list" (page or include) the context object relation is to be found. Note: Set this flag depending on whether it is a page entry point or component entry point pointed by the referenced pipeline.

All other tabs provide the same functionality as known from the section *Pagelet Definition (Content Unit)*.

Configuration Parameter Type

As described above, are already various configuration parameters types included in the supplied cartridges. Of course, new configuration parameters types can be defined. To configure a new configuration parameter type: create an appropriate pagelet2 model.

General Tab

Figure 183. Configuration Parameter Type's General Tab

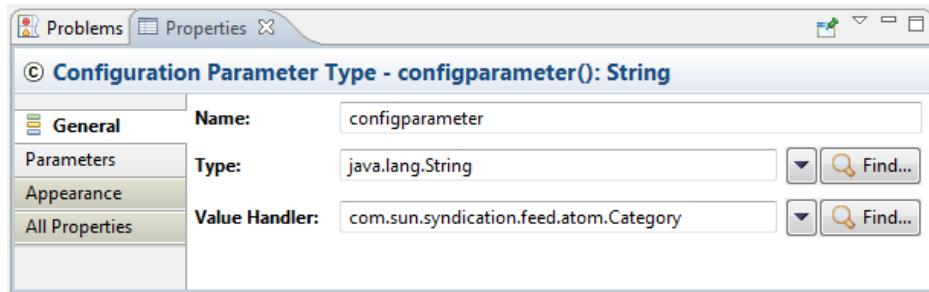


Table 89. Properties of Configuration Parameter Type's General Tab

Property	Description
Name	Edit name to change configuration parameter type's name attribute. Default value is the chosen file name.
Type	Reference the java type of the configuration parameter type.
Value Handler	Reference the value handler of the configuration parameter type

All other tabs provide the same functionality as known from the section *Pagelet Definition (Content Unit)*.

Delete a Pagelet2 Model

1. Right click the pagelet2 file in Cartridge Explorer.
2. Select Delete from context menu.
3. Confirm the dialog with OK.

The pagelet file is deleted.

NOTE: You have to delete pagelet's properties file too.

Copy a Pagelet2 Model

1. Right click the pagelet2 file in Cartridge Explorer.
2. Select Copy from context menu.
3. Navigate to an other folder.
4. Right click and Paste.

The pagelet file is copied.

NOTE: You have to copy pagelet's properties file too.

Move a Pagelet2 Model

1. Right click the pagelet2 file in Cartridge Explorer.
2. Click Refactor.
3. Click Move.

Move dialog opens.

- 4. Navigate in Move dialog box to an other folder.**
- 5. Confirm dialog with OK.**

The pagelet file is moved.

NOTE: You have to move pagelet's properties file too.

Rename a Pagelet2 Model

- 1. Right click the pagelet2 file in Cartridge Explorer.**
 - 2. Click Refactor.**
 - 3. Click Rename.**
- Rename Resource dialog opens.
- 4. Type in a new name.**
 - 5. Confirm dialog with OK.**

The pagelet file is renamed.

NOTE: The file extension must be .pagelet2.

NOTE: You have to rename pagelet's properties file too.

Creating Extensions

Extensions

What are Extensions?

Intershop 7 provides a mechanism to easily extend the functionality of pipelines, templates, queries, and other code artifacts by defining extension points for plugging in foreign code.

An extension point is a point in the code, which marks the place where an extension can plug in. Extension points exist for several code artifacts, like pipelines, templates, queries and others.

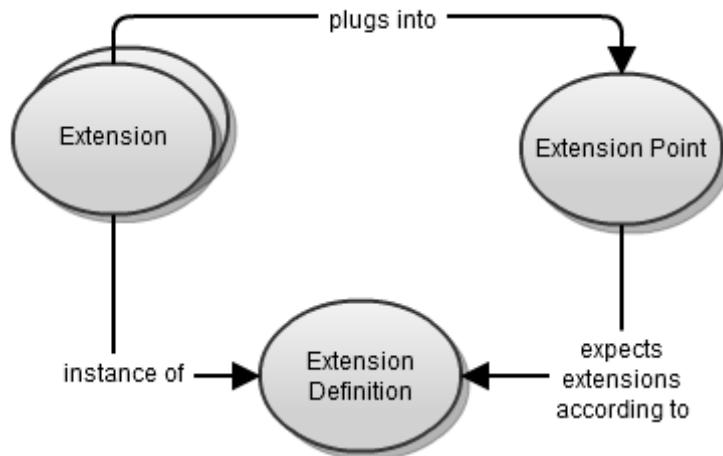
An extension is the code fragment that plugs into an extension point. An extension always corresponds to the type of the extension point, e.g. an extension pipeline plugs into a pipeline extension point.

Extensions provide the following advantages over previously available extension mechanisms:

- you can extend existing functionality coded in pipelines and templates belonging to different independent cartridges
Because multiple cartridges are involved, the alternative approach of overwriting pipelines and templates is not applicable in this case.
- Extension points are a unified extension mechanism that works for different types of code artifacts. Currently extensions of the following artifact types exist: pipeline, ISML template and Java extensions.

How the Extension Mechanism Works

The following diagram shows the general functionality of the extension mechanism:

Figure 184. Extensions and Extension Points

An extension point expects extensions according to an extension definition. This definition basically consists of a type name, which corresponds to the type of the artifact to be extended, and an identifier.

Extensions, which plug into the extension point, are valid instances of this extension definition, that is, they are of the given type and refer to the given identifier. For Example, a pipeline "Foo" defines an extension point node with the short identifier "Bar". The extension definition has the type "pipeline" and the full-qualified identifier "Foo-Bar". Into this extension point all extensions will plug into, which are extension pipelines and refer to "Foo-Bar".

The creation of an extension point is specific to the type of artifact the extension point is created for. For example, a pipeline extension point is a pipeline specific feature while the ISML template extension point is a specific feature of the ISML template engine. However, the way how extensions are bound to these extension points follows a common approach. The bindings must be defined in Intershop Studio and the resulting extension definition file is an XML file with a common format for all types of extension points.

Each extension definition has a priority. This is an integer value which is used to prioritize the extensions and thus to influence the order of processing. An extension with a higher priority is processed before one with a lower priority. If multiple extensions have the same priority then the order of processing is undefined. This feature is especially useful for template extensions, e.g. when tabs or menu items need to be sorted. It is recommended to use a range from 1 to 100, but this is no hard coded limit. The default value is 50.

The following sections explain the existing extension types in more detail and describe how additional extension types can be implemented.

Pipeline Extensions

Extension pipelines plug into pipeline extension points, which are pipeline nodes similar to call nodes.

The process of creating pipeline extension points and extension pipelines is explained in *Extending Pipelines*.

ISML Template Extensions

Template extension points are defined using the `extensionpoint` attribute of the `ISINCLUDE` tag.

The process of creating ISML template extension points and appropriate template extensions is explained in *Extending ISML Templates*.

Java Extensions

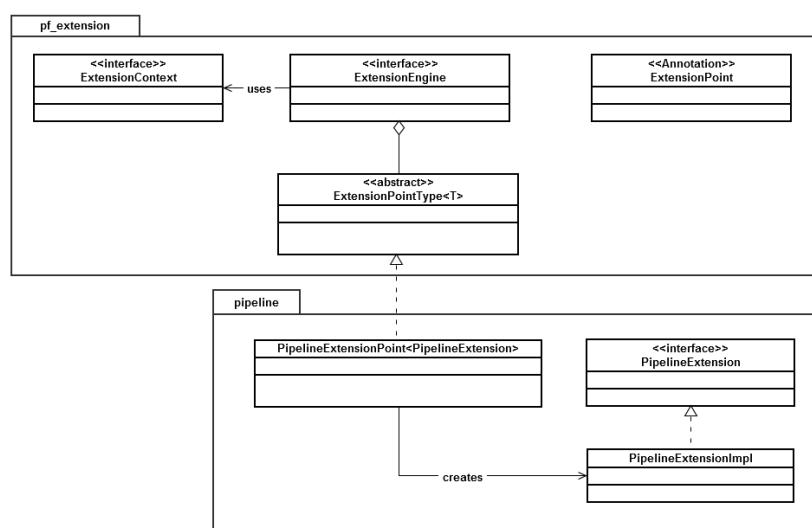
Java class extension points provide the ability, to plug custom code snippets into existing functionality written in Java. To define an extension point in Java the method which processes the extensions must be annotated appropriately.

The process of creating Java extension points and extensions which plug into this extension point is explained in *Extending Java Code*.

Defining New Extension Types

It is also possible to create new extension point types for other artifacts existing in Intershop 7 or developed later. To understand how a new extension point type is added some knowledge about the extension API (as show in the diagram below) is required.

Figure 185. Extension API



The diagram shows two packages: the cartridge `pf_extension` and the *cartridge pipeline*. The cartridge pipeline is only part of the diagram to show how the integration of a "real" extension works.

The `pf_extension` cartridge basically defines the infrastructure to work with extensions: an engine, a context, and an extension point type.

The *extension engine* is responsible for registering extension point types and lookup extensions. The extension engine uses the capabilities of the EMF engine to perform the registration and lookup of extensions. Extension point types must be registered explicitly at the engine.

The *context* must be provided to lookup extensions. The context provides a list of extension set IDs, which are basically cartridge names, sorted in the lookup order.

Extension point types are declared using the `ExtensionPointType` class and are registered at the extension engine. One extension point type instance corresponds to the type attribute used in extension point bindings created in Intershop Studio.

A new extension point type can be introduced by implementing the according extension point type class and registering the new type at the extension engine. The extension engine can lookup extensions of registered extension point types only. The extension point type is also responsible for creating an extension instance of a given class.

When the extension engine determines all extensions of a given extension point, then it invokes the according method to create the extension instances. For that purpose the implementation of an extension point type instance is parametrized with the extension interface class.

To illustrate an implementation case, the class diagram above also contains the classes of the pipeline extension. The `PipelineExtension` interface defines methods to get a pipeline name and a start node name. The `PipelineExtensionPoint` creates an instance of `PipelineExtension`, that's why this class is parametrized accordingly. When requested, An instance implementing the interface `PipelineExtension` is created by the pipeline extension point type.

Extending Pipelines

You can extend a pipeline with extension pipelines from other cartridges. This task comprises the following steps:

- 1. Add a pipeline extension point node to the pipeline you wish to extend.**
- 2. Create and implement an extension pipeline.**
- 3. Bind the extension pipeline to the pipeline extension point.**

These steps are explained in more detail in the following sections.

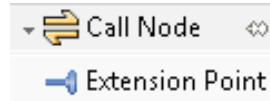
Create a Pipeline Extension Point

You need to add a pipeline extension point node to the pipeline you want to extend. Make sure, that the extension point ID of the extension point node is unique within the pipeline.

Do the following:

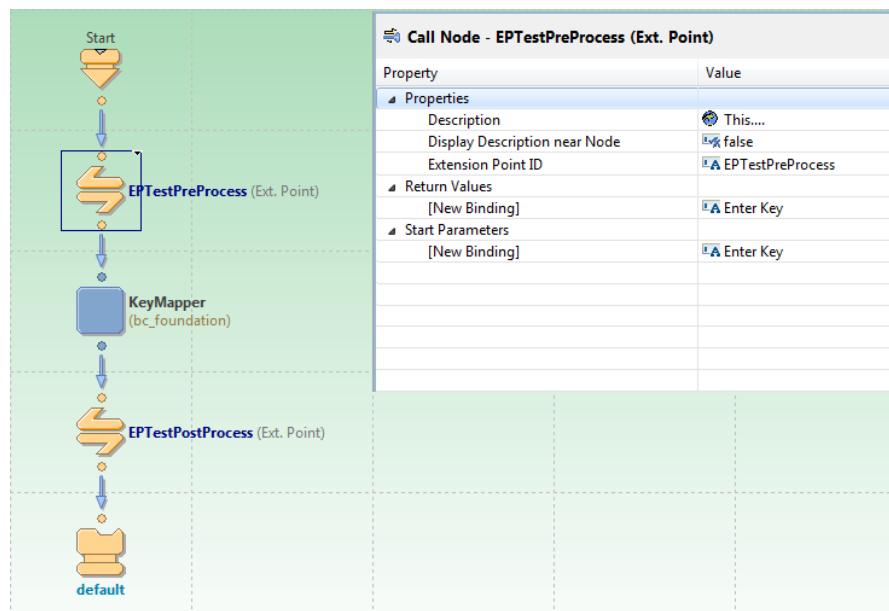
- 1. Open the pipeline in the Intershop Studio pipeline editor.**
- 2. In the Pipeline Components palette, expand the "Call Node" item, so the "Extension Point" node becomes visible.**

Figure 186. Extension Point Node in Pipeline Components Palette



- 3. Add the extension point node at the desired place in the pipeline and wire the transitions.**
- 4. Set a unique Extension Point ID and save the pipeline.**

Figure 187. Adding an Extension Point Node to a Pipeline



Create and Implement an Extension Pipeline

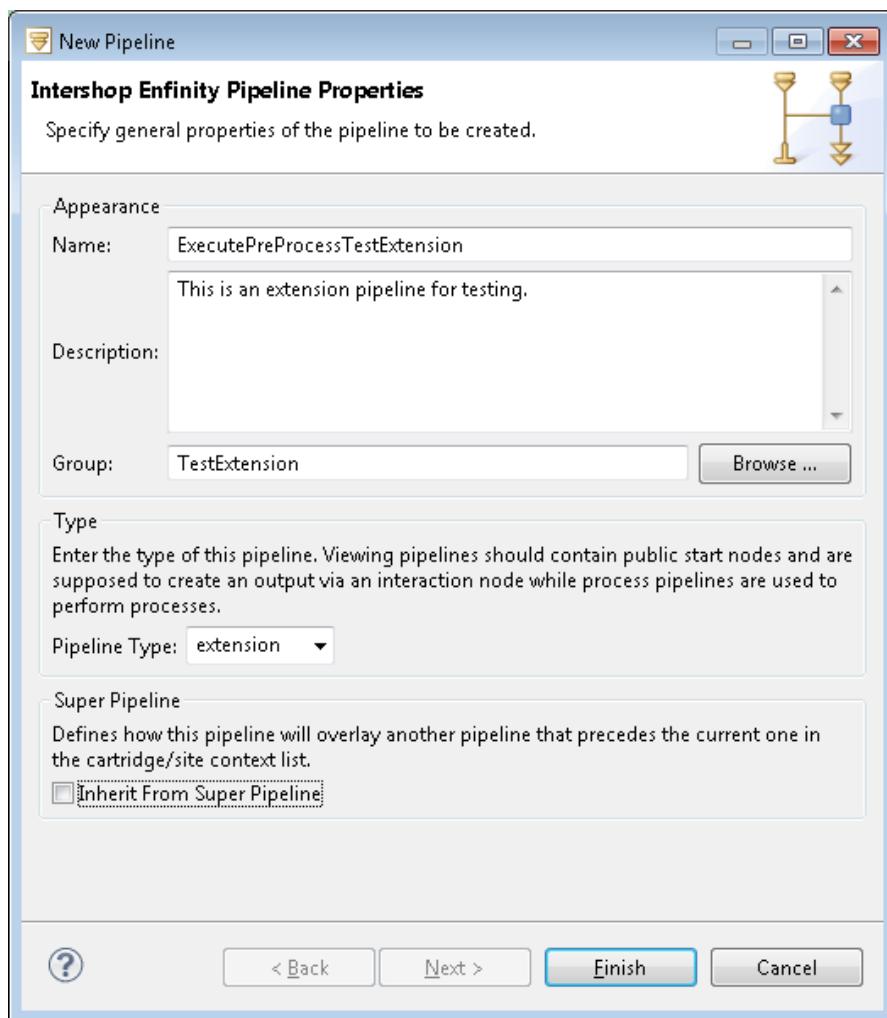
To create an extension pipeline:

- 1. Start the pipeline wizard.**

See *Start the Pipeline Wizard* [book_all.org] for details.

- 2. Specify general pipeline properties, such as pipeline name and description.**

Choose a pipeline name that describes what the pipeline does. Don't choose a generic name or a name that corresponds to the extension point ID as this can result in accidentally overwriting pipelines at runtime. For example, a good name would be `InitializePaypalOnChannelCreation` instead of `OnChannelCreationHook`.

Figure 188. Creating an Extension Pipeline

3. Set the pipeline type to "extension".

4. Click Finish to create the pipeline.

NOTE: In order to make the extension pipeline available to the extended pipeline, make sure that the cartridge containing the extension pipeline is part of the lookup path of the according application type.

Once the pipeline has been created, you can implement it. When doing so, keep the following things in mind:

- An extension pipeline must have only private start nodes.
- An extension pipeline must have only strict start and end nodes.
- An extension pipeline must not contain interaction nodes or interaction continue nodes.
- An extension pipeline must have an end node named "next".
- An extension pipeline may have an end node named "error".

Bind the Extension Pipeline to the Extension Point

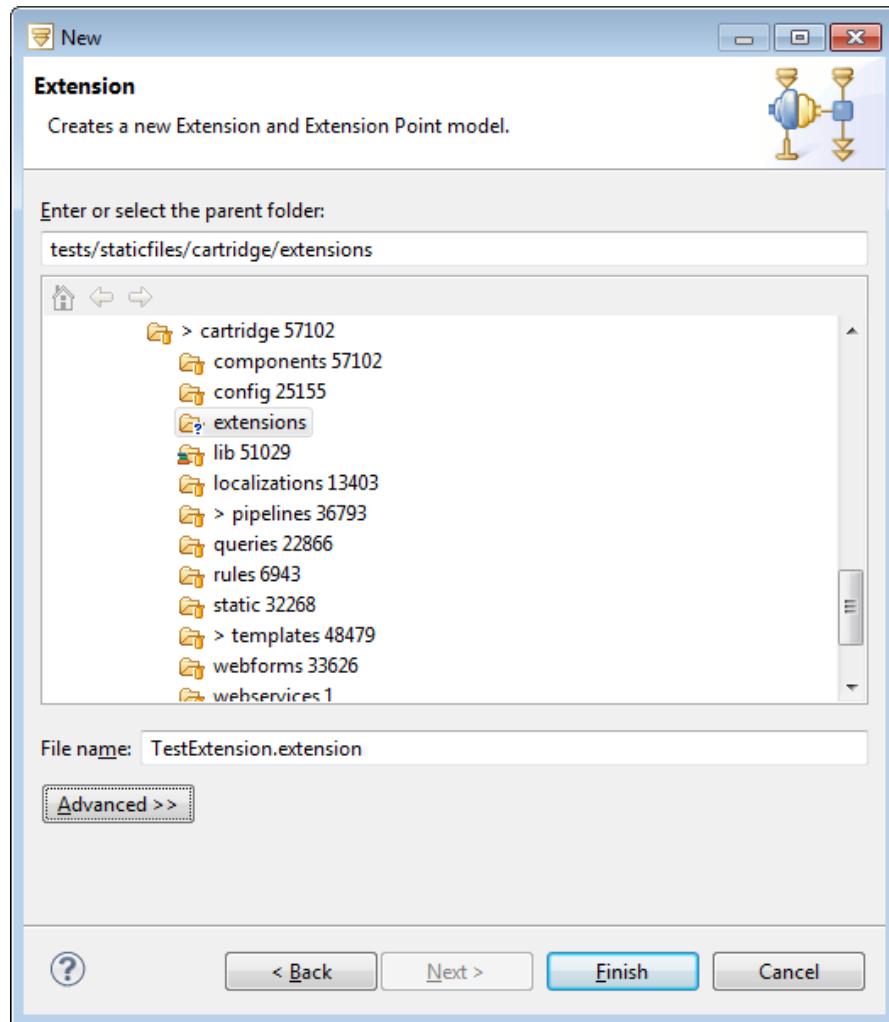
Bind the extension pipeline to the previously created extension point by doing the following:

1. Create a new extension definition file.

The file must be created in the *staticfiles/cartridge/extensions* directory of the target cartridge.

From the Intershop Studio menu bar, choose New | Other | Extension, select the correct directory, and set the correct file name. Click Finish.

Figure 189. Creating a Extension Definition File



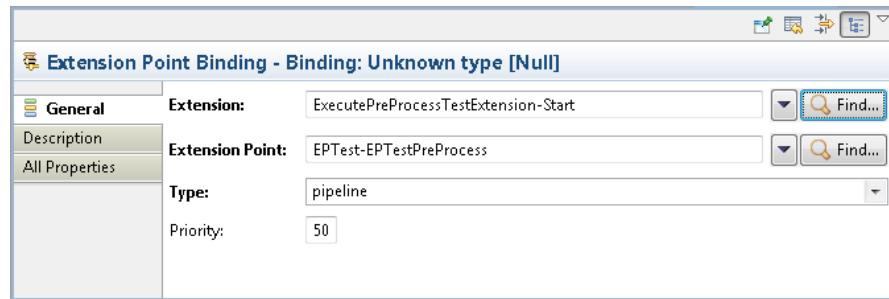
2. Create an extension point binding.

In the cartridge explorer, navigate to the *staticfiles/cartridge/extensions* directory and right-click the new extension definition file. From the context menu, select New Child | Extension Point Binding.

Figure 190. Creating an Extension Point Binding

3. Set the extension point binding's properties.

In the cartridge explorer, highlight the extension point binding, then set its general properties in the properties view.

Figure 191. Extension Point Binding Properties

In the properties view:

- Set the type to "pipeline".
- Set the priority to a value between 1 and 100.

The priority allows you to influence the order in which extensions are processed. Extensions with a higher priority are processed first.

- Select the extension point where the extension pipeline plugs into.

Use the Find button to select a pipeline extension point. In the browse dialog, you can use a wildcard pattern ("**") to display all possible extension point nodes that are available.

- Select the extension pipeline similar to the step above.

After performing the described steps, the source of the extension definition file (TestExtension.extension) contains something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<extensionpoint:ExtensionPointModel
    xmlns:extensionpoint="http://www.intershop.de/extensionpoint/2011"
    name="TestExtension">
<extensionBindings
    type="pipeline"
    extensionPoint="EPTest-EPTestPreProcess"
    extension="ExecutePreProcessTestExtension-Start"/>
</extensionpoint:ExtensionPointModel>
```

Extending ISML Templates

You can extend an ISML template with extension templates from other cartridges. This task comprises the following steps:

- 1. Define an ISML extension point in the template you wish to extend.**
- 2. Create and implement an extension template.**
- 3. Bind the template to the template extension point.**

Define an ISML Extension Point

An ISML template extension point is an <ISINCLUDE> tag with an extension point ID. Add the <ISINCLUDE> tag to the template. Make sure that the extension point ID is unique within the template.

Example:

```
...<isinclude extensionpoint="ProductTabsExtensions"/>...
```

Create and Implement an Extension Template

First create the extension template. The ISML template has no specifics, it is similar to any other local include template.

When implementing the template, you should also inspect the code of the template that declares the extension point to make sure that the extension template fits into the specific HTML structure of the extended template.

Choose a template name that describes what the template does. Don't choose a generic name or a name that corresponds to the extension point ID as this can result in accidentally overwriting templates at runtime. For example, a good name would be `ProductTabRevenue.isml` instead of `ProductTabExt.isml`.

NOTE: In order to make the extension template available to the extended template, make sure that the cartridge containing the extension template is part of the lookup path of the according application type.

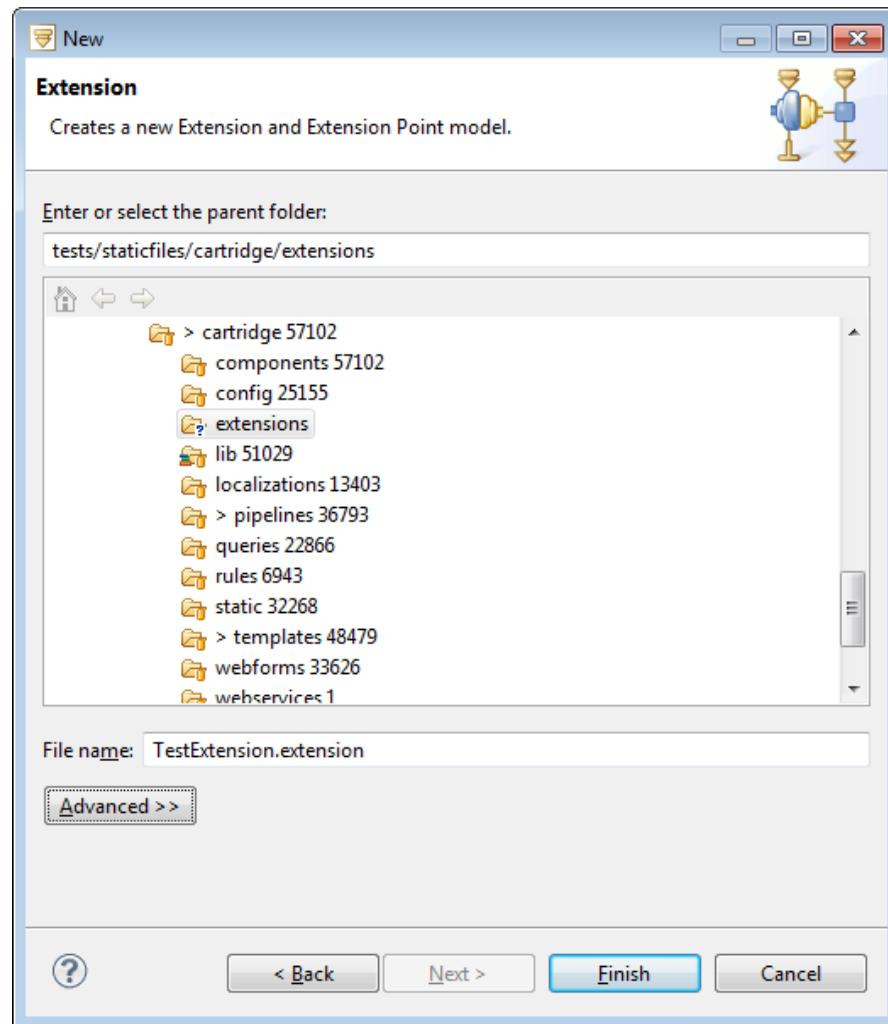
Bind the Extension Template to the Extension Point

Bind the extension template to the previously created extension point by doing the following:

- 1. Create a new extension definition file.**

The file must be created in the `staticfiles/cartridge/extensions` directory of the target cartridge.

From the Intershop Studio menu bar, choose **New | Other | Extension**, select the correct directory, and set the correct file name. Click **Finish**.

Figure 192. Creating a Extension Definition File

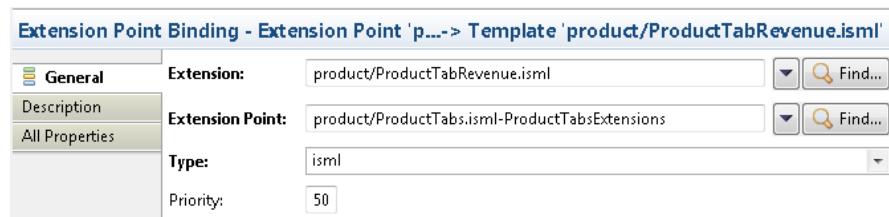
2. Create an extension point binding.

In the cartridge explorer, navigate to the *staticfiles/cartridge/extensions* directory and right-click the new extension definition file. From the context menu, select New Child | Extension Point Binding.

Figure 193. Creating an Extension Point Binding

3. Set the extension point binding's properties.

In the cartridge explorer, highlight the extension point binding, then set its general properties in the properties view.

Figure 194. Extension Point Binding Properties

In the properties view:

- Set the type to "isml".
- Set the priority to a value between 1 and 100.

The priority allows you to influence the order in which extensions are processed. Extensions with a higher priority are processed first.

- Select the extension point where the extension pipeline plugs into. Use the Find button to select a template extension point. In the browse dialog, you can use a wildcard pattern ("**") to display all possible extension point nodes that are available.
- Select the extension template similar to the step above.

After performing the described steps, the source of the extension definition file (TestExtension.extension) should contain something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<extensionpoint:ExtensionPointModel
    xmlns:extensionpoint="http://www.intershop.de/extensionpoint/2011"
    name="TestExtension">
    <extensionBindings
        type="isml"
        extensionPoint="product/ProductTabs-ProductTabsExtensions"
        extension="product/ProductTabRevenue.isml"/>
</extensionpoint:ExtensionPointModel>
```

Extending Java Code

You can extend Java Code with an extension class from another cartridge. This task comprises the following steps:

- 1. Define an Java extension point in the Java code you wish to extend.**
- 2. Create and implement an extension class.**
- 3. Bind the extension class to the Java extension point.**

These steps are detailed in the following sections based on the following example: The product manager implementation defines a Java extension point for a product deletion handler. Custom code can plug into this extension point and execute custom code while the product is deleted, e.g. deleting product related instances of custom classes.

Define a Java Extension Point

You declare and implement a Java extension point by doing the following:

1. Create an extension interface.

The extension interface has two functions: It defines the API for Java extension which will plug into the extension point and it is used together with the extension point ID to lookup the extensions.

The extension interface must be part of the public API of the cartridge which declares the extension points. Otherwise extension implementations would have to refer to an internal class, which is forbidden.

The extension interface usually defines methods for each task an extension may perform. For the example introduced above, a interface definition might look like this:

```
// ProductDeletionHandler.java
public interface ProductDeletionHandler extends Extension
{
    public void deleteProduct(Product p);
}
```

2. Implement the method that processes the extensions and annotate the method as an extension point.

Example:

```
@ExtensionPoint(type=ProductDeletionHandler.class,
               id="ProductMgr.removeProduct")
private void callProductDeletionHandlers(Product p)
{
    ComponentMgr cMgr = NamingMgr.getManager(ComponentMgr.class);
    ExtensionEngine extEngine = (ExtensionEngine)cMgr. ...
        getGlobalComponentInstance("ExtensionEngine");
    ApplicationTypeImpl app = (ApplicationTypeImpl)(AppContextUtil...
        getCurrentAppContext().getApp());
    ExtensionContext ctx = new ExtensionContext(app. ...
        getResourceSetIdentifiers(), app.getID());

    List<ProductDeletionHandler> eps = extEngine. ...
        getJavaExtensions(ProductDeletionHandler.class,
        "ProductMgr.removeProduct", ctx);
    for (ProductDeletionHandler pdh: eps)
    {
        pdh.deleteProduct(p);
    }
}
```

NOTE: The Java annotation marks the extension point in the source code. The extension point ID must be unique. Intershop Studio uses the extension point declaration to build a qualified name from the interface name and the extension point ID, which must be a unique string.

Create and Implement an Extension Class

You need to create an extension class that implements the extension interface. The class must have a public default constructor. Based on our example, an according implementation could look like this:

```
// ProductDeletionHandlerImpl.java
public class ProductDeletionHandlerImpl implements ProductDeletionHandler
{
    public ProductDeletionHandlerImpl() {}
```

```
@Override  
public void deleteProduct(Product p)  
{  
    // execute custom code  
}  
}
```

NOTE: You have to make sure that the code does not interfere with other existing extensions or the caller's code.

Bind the Extension Class to the Extension Point

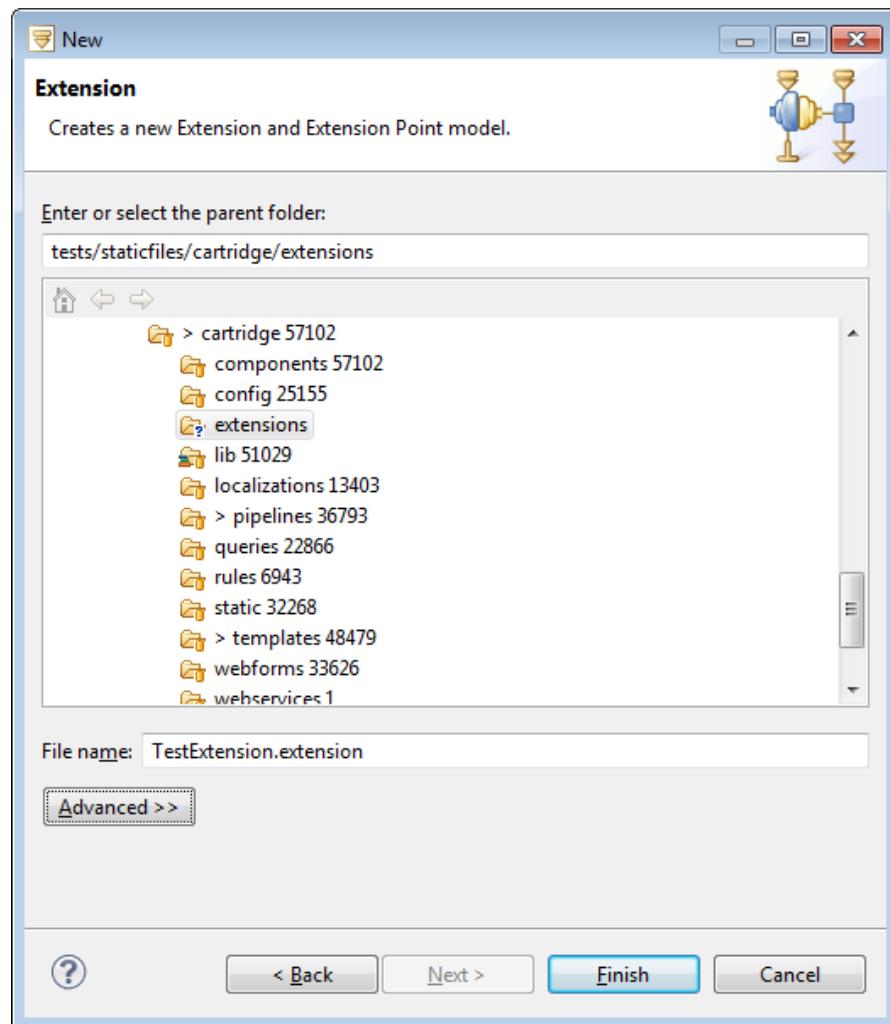
Bind the Java extension code to the previously created extension point by doing the following:

1. Create a new extension definition file.

The file must be created in the *staticfiles/cartridge/extensions* directory of the target cartridge.

From the Intershop Studio menu bar, choose New | Other | Extension, select the correct directory, and set the correct file name. Click Finish.

Figure 195. Creating a Extension Definition File



2. Create an extension point binding.

In the cartridge explorer, navigate to the `staticfiles/cartridge/extensions` directory and right-click the new extension definition file. From the context menu, select New Child | Extension Point Binding.

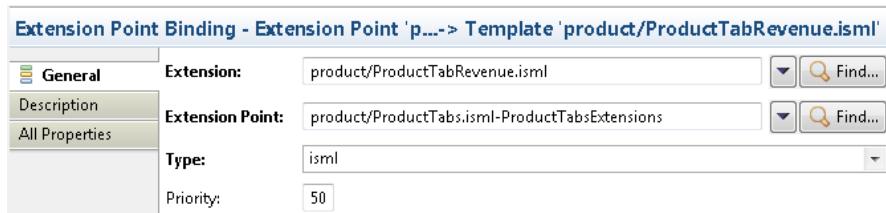
Figure 196. Creating an Extension Point Binding



3. Set the extension point binding's properties.

In the cartridge explorer, highlight the extension point binding, then set its general properties in the properties view.

Figure 197. Extension Point Binding Properties



In the properties view:

- Set the type to "java".
- Set the priority to a value between 1 and 100.

The priority allows you to influence the order in which extensions are processed. Extensions with a higher priority are processed first.

- Select the extension point where the extension pipeline plugs into.

Use the Find button to select a template extension point. In the browse dialog, you can use a wildcard pattern ("**") to display all possible extension point nodes that are available.

- Select the extension template similar to the step above.

After performing the described steps, the source of the extension definition file (`TestExtension.extension`) should contain something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<extensionpoint:ExtensionPointModel
    xmlns:extensionpoint="http://www.intershop.de/extensionpoint/2011"
    name="TestExtension">
    <extensionBindings
        type="java"
        extensionPoint="com.intershop.beehive.xcs.capi.product. ....
            ProductDeletionHandler-ProductMgr.removeProduct"
        extension="tests.units.basic.com.intershop.beehive.xcs.internal....
            product.ProductDeletionHandlerImpl"/>
</extensionpoint:ExtensionPointModel>
```

Query Development

Queries

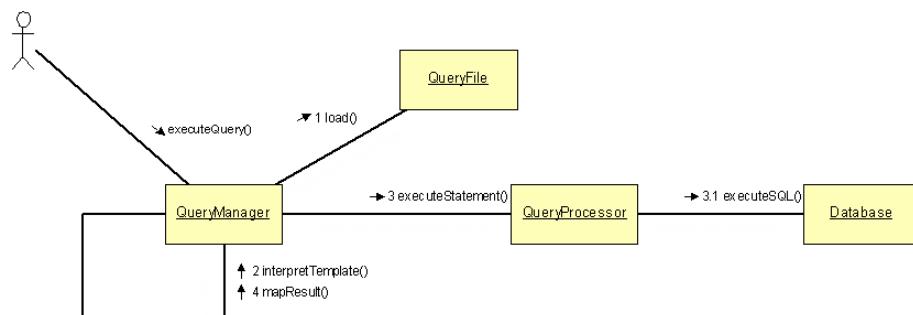
Intershop 7 provides a file-based query framework. Its main features include, among others:

- Support for several search engines
- Support for multiple data sources
- Support for customizable SQL-based features as well as for `select` and `update` queries
- Convenient maintainability, customizability and extendibility
- File system-based query definition

In the new query framework, the number of pipelines and the number of pipelets within the pipelines is reduced to a minimum. The actual queries are defined using XML-based query files. The query files contain query templates in the native language of the associated search/query engine. To ease customization, query files can be overridden using the same site-specific lookup mechanism as used for pipeline and template lookup.

The *QueryMgr* is responsible for loading and executing the query templates and the result mapping; the *QueryProcessor* performs the actual query.

Figure 198. Query framework overview



For the query framework to work, two new application server properties are introduced:

```
intershop.queries.CheckSource=true
intershop.queries.Preload=true
```

Usage Patterns

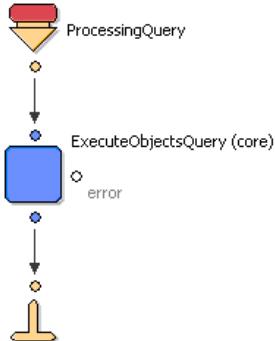
Generally, the framework provides the following pipelets for query execution: *ExecuteCountQuery*, *ExecuteObjectsQuery*, *ExecutePageableQuery* and *ExecuteUpdateQuery*. Additional ("helper") pipelets are *LoadQuery*, *VerifySearchTerm* and *UpdateDictionary*.

Upon editing query pipelines, consider the following:

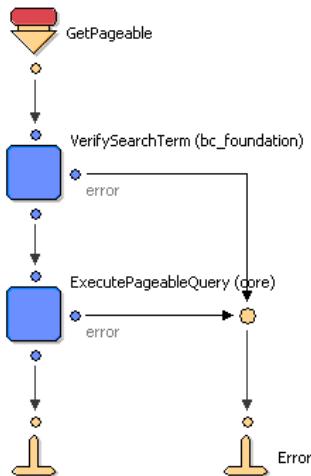
- You should not declare a data source for query execution pipelets, except the current ORM connection; the default data source is used automatically. Declaring the data source at the pipelet would overwrite the declaration defined by the query file.
- Avoid, as far as possible, text processing in order to prevent SQL injection security leaks.
- Usually, there is no need to change a pipeline when customizing a query. Allow this only if the new query needs some *QueryProcessor*-dependent context, like a different JDBC connection, for example.

The following graphic shows a pipeline performing a simple query that is used, for example, during a processing pipeline or for a single page search result. In this case, a single execution pipelet covers all requirements. Usually, a simple query uses the current ORM connection.

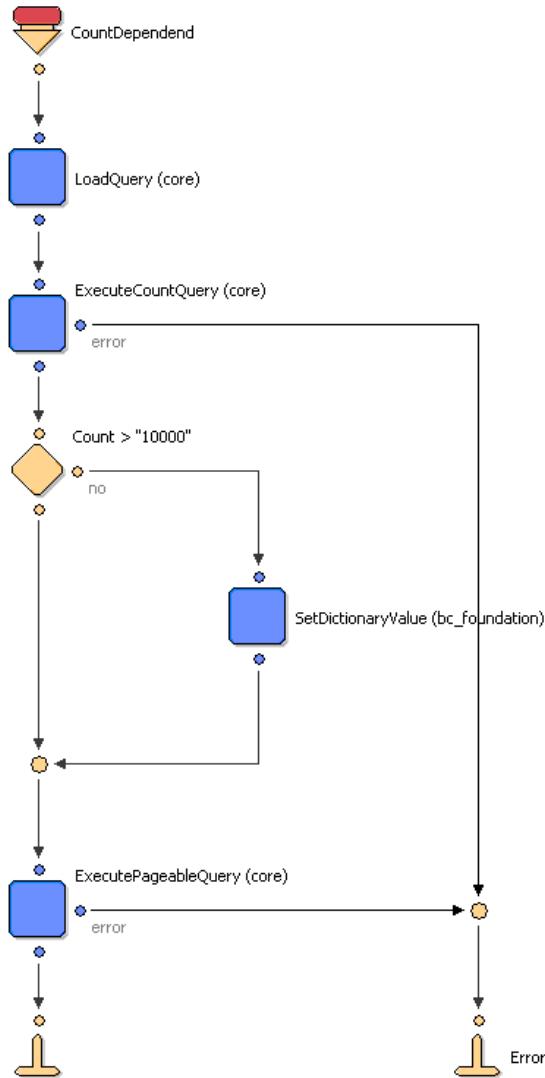
Figure 199. Simple query



The next graphic illustrates a simple pageable query. This type of queries can be used, for example, for the most of Intershop 7's list views with a moderate amount of data. The *VerifySearchTerm* pipelet parses the user input, *ExecutePageable* performs the search. In this case, there is usually no need to declare a data source.

Figure 200. Simple pageable query

The following graphic illustrates a query with count dependency that is used when the query for sorted objects becomes inperformant on a large result set, e.g., products. In this example, the LoadQuery pipelet is used to allow for reusing the actual result of the count query for the pageable.

Figure 201. Query with count dependency

Query Files

Query files are XML files suffixed with `.query`. As they normally should not be edited directly, Intershop Studio provides a dedicated query file editor (see *Creating and Editing Query Files*). Query files are stored in the site and cartridge structure in the subdirectory `queries`, in parallel to the `pipelines` and `templates` directories. Subdirectories are supported.

Query files are designed to hold only one query in terms of business. This means, a file can contain more than one search/query engine statement template. Depending on the execution context, the `QueryManager` implicitly selects the actually required template.

A query file consists of:

- an input parameter declaration,
- a return mapping,

- a processor section, and
- one or more query templates.

For more details about query files, refer to *Query Files*.

User Search Expressions

The process to build the query actually executed by the *QueryProcessor* comprises two steps: First, a search term entered by a user is translated into a *SearchExpression*. Then, this *SearchExpression* is passed to the query, where the query template or a parameter handler translates it into the actual query to be executed.

The *VerifySerachTerm* pipelet transforms the search term into the *SearchExpression*. The following syntaxes are supported:

- Fuzzy: Word
 - Wildcard: Wo?d*
 - Exact: +Word or "Word A"
 - Exclusion: -Word or -"Word A"
 - Logical AND: LCD Monitor
 - Logical OR: LCD | CRT Monitor
- Searches for "Monitor AND (LCD OR CRT)"

Features and symbols can be configured in the pipelet properties. The system-wide defaults are defined in *bc_foundation.properties*.

Public APIs and Usage

This section explains the public APIs of the query framework and describes how to use them.

Pipelets

VerifySearchTerm

This pipelet converts a search string entered by a user into an expression that can be used in query templates. This allows for defining queries without regard to the actual search string provided by a user.

Table 90. VerifySearchTerm

	Name	Optional	Type	Description
Configuration	AllowFuzzySearch	Yes	Boolean	Default: Yes
	AllowWildcards	Yes	(No, Multi, Single, Multi & Single)	Default: Multi & Single
	RequiredCharacters BeforeWildcard	Yes	Integer	Default: 0
	AllowExclusions	Yes	Boolean	Default: Yes
	DefaultOperator	Yes	(AND, OR)	Default: AND

	Name	Optional	Type	Description
	AllowSubExpressions	Yes	Boolean	Default: Yes
Input	UserSearchTerm	No	String	The site context.
Output	SearchExpression	Yes	SearchExpression	The created search expression.
	ErrorCode	Yes	String	An identifier for the error in case the term could not be translated into an expression, e.g., "LeftWildCard Forbidden"
Errors	In case the term cannot be translated into an expression, the error connector is used.			

LoadQuery

This pipelet loads a query from a file, according to the site context.

Table 91. LoadQuery

	Name	Optional	Type	Description
Configuration	QueryName	No	String	The name of the query (file name without extension and query file directory root).
Input	CurrentDomain	No	Domain	The site context.
Output	Query	No	Query	The loaded query.
Errors	In case the query could not be found or the file is invalid, an exception is thrown.			

UpdateDictionary

This pipelet builds a map with string objects as key. It can be used to build larger sets of parameters for query execution.

Table 92. UpdateDictionary

	Name	Optional	Type	Description
Configuration	Name_xx	Yes	String	The name for value xx in the created map.
Input	Value_xx	Yes	Object	The value to be stored under Name_xx.

	Name	Optional	Type	Description
	Dictionary	Yes	Map	A possibly already existing dictionary to be changed.
Output	Dictionary	No	Map	The created or changed dictionary.
Errors				

ExecuteCountQuery

This pipelet executes count queries.

Table 93. ExecuteCountQuery

	Name	Optional	Type	Description
Configuration	ParameterName_xx	Yes	String	The name for value xx.
	QueryName	Yes	String	The name of the query to be executed.
	DatasourceName	Yes	String	The name of a data source to get a JDBC connection from. Allowed values are values to be used with <code>JDBCDataSourceMgr.getConnection()</code> . Additionally, there is a predefined value to use the current ORM connection.
Input	ParameterValue_xx	Yes	Object	The value for ParameterName_xx.
	Parameters	Yes	Map	A map containing query parameter values.
	Query	Yes	Query	The query to be executed.
	Connection	Yes	Connection	A JDBC connection that is used to perform the query.
	CurrentDomain	Yes	Domain	The site context.
Output	Count	Yes	Integer	The value in case the query was executed successfully.
	ErrorCode	Yes	String	An identifier for the error when returned with PIPELET_ERROR.
Errors	In case no query is given and no query could be loaded by the given name, an exception is thrown.			

	Name	Optional	Type	Description
	In case the underlying search engine is not able to answer a formally correct query, the pipelet error exit is used and the ErrorCode is set, e.g., <code>TooManyValues</code> for a DRG-51030 Oracle error.			

ExecuteObjectsQuery

This pipelet executes a query for objects. The result is intended to be used only in the following pipeline. In this case, the element count and a sorting are usually not required.

Table 94. ExecuteObjectsQuery

	Name	Optional	Type	Description
Configuration	ParameterName_xx	Yes	String	The name for value xx.
	QueryName	Yes	String	The name of the query to be executed.
	DatasourceName	Yes	String	The name of a data source to get a JDBC connection from. Allowed values are values to be used with <code>JDBCDataSourceMgr.getConnection()</code> . Additionally, there is a predefined value to use the current ORM connection.
Input	ParameterValue_xx	Yes	Object	The value for ParameterName_xx.
	Parameters	Yes	Map	A map containing query parameter values.
	Query	Yes	Query	The query to be executed.
	Connection	Yes	Connection	A JDBC connection that is used to perform the query.
	CurrentDomain	Yes	Domain	The site context.
Output	SearchResult	Yes	Iterator	The value in case the query was executed successfully.
	ErrorCode	Yes	String	An identifier for the error when returned with PIPELET_ERROR.
Errors	In case no query is given and no query could be			

	Name	Optional	Type	Description
	loaded by the given name, an exception is thrown.			
	In case the underlying search engine is not able to answer a formally correct query, the pipelet error exit is used and the ErrorCode is set, e.g., <code>TooManyValues</code> for a DRG-51030 Oracle error.			

ExecutePageableQuery

This pipelet executes a query for objects. As the result is intended to be displayed to users, it must support sorting and paging.

Table 95. ExecutePageableQuery

	Name	Optional	Type	Description
Configuration	ParameterName_xx	Yes	String	The name for value xx.
	QueryName	Yes	String	The name of the query to be executed.
	DefaultPageSize	Yes	Integer	The page size for the returned pageable. With a value less or equal 0, all elements are placed in one page.
	DatasourceName	Yes	String	The name of a data source to get a JDBC connection from. Allowed values are values to be used with <code>JDBCDataSourceMgr.getConnection()</code> . Additionally, there is a predefined value to use the current ORM connection.
Input	ParameterValue_xx	Yes	Object	The value for ParameterName_xx.
	Parameters	Yes	Map	A map containing query parameter values.
	Query	Yes	Query	The query to be executed.
	PageSize	Yes	Integer	The page size for the returned pageable.
	Connection	Yes	Connection	A JDBC connection that is used to perform the query.
	CurrentDomain	Yes	Domain	The site context.

	Name	Optional	Type	Description
Output	SearchResult	Yes	PageableIterator<Object>	The value in case the query was executed successfully.
	ErrorCode	Yes	String	An identifier for the error when returned with PIPELET_ERROR.
Errors	In case no query is given and no query could be loaded by the given name, an exception is thrown.			
	In case the underlying search engine is not able to answer a formally correct query, the pipelet error exit is used and the ErrorCode is set, e.g., <code>TooManyValues</code> for a DRG-51030 Oracle error.			

ExecuteUpdateQuery

This pipelet is intended to execute DML queries. This means, the query executes insert, update and delete statements.

Table 96. ExecuteObjectsQuery

	Name	Optional	Type	Description
Configuration	ParameterName_xx	Yes	String	The name for value xx.
	QueryName	Yes	String	The name of the query to be executed.
	DatasourceName	Yes	String	The name of a data source to get a JDBC connection from. Allowed values are values to be used with <code>JDBCDataSourceMgr.getConnection()</code> . Additionally, there is a predefined value to use the current ORM connection.
Input	ParameterValue_xx	Yes	Object	The value for ParameterName_xx.
	Parameters	Yes	Map	A map containing query parameter values.
	Query	Yes	Query	The query to be executed.
	Connection	Yes	Connection	A JDBC connection that is used to perform the query.
	CurrentDomain	Yes	Domain	The site context.

	Name	Optional	Type	Description
Errors	In case no query is given and no query could be loaded by the given name, an exception is thrown.			
	Errors during query execution are thrown as exception.			

For the pipelets *ExecuteCountQuery*, *ExecuteObjectsQuery*, *ExecutePageableQuery* and *ExecuteUpdateQuery*, the properties view allows for easily accessing the query parameters. Developers can directly edit the parameters passed to the query to be invoked or processed by the pipelet.

Query Files

Query files are XML files suffixed with .query. They are stored in the site and cartridge structure in the subdirectory *queries*, in parallel to the *pipelines* and *templates* directories.

As already outlined, a query file consists of an input parameter declaration, a return mapping, a processor section, and one or more query templates. The following example illustrates the query file contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<query>
<parameters>
    <parameter name="Domain" type="com.intershop.beehive.core.capi.domain.Domain"
        optional="false"/>
    <parameter name="SortLocale" type="com.intershop.beehive.core.capi.localization.LocaleInformation" optional="false"/>
</parameters>
<return-mappings>
    <return-mapping name="Rebate" type="orm" class="com.intershop.component.marketing.internal.rebate.PromotionPO">
        <return-attribute name="PromotionUUID"/>
    </return-mapping>
</return-mappings>
<processor name="OracleSQL">
</processor>
<template type="countedobjects">
    SELECT p.uuid as PromotionUUID, s.stringValue, COUNT(*) over() AS rowcount
    FROM promotion_av s, promotion p
    WHERE p.DomainID=<template-variable value="Domain:UUID"/>
        AND not exists (select * from abtestgroup ab where ab.promotionuuid=p.uuid)
        AND s.name (+)='displayName'
        AND s.localeid (+)=<template-variable value="SortLocale:LocaleID"/>
        AND p.uuid=s.ownerid (+)
    ORDER BY s.stringValue asc NULLS LAST
</template>
</query>
```

Parameter Declaration

The parameter section defines which values the query expects. It consists of a name, the expected type and a flag marking it as optional or required. Only values declared here are accessible in further query processing (parameter preprocessing by the *QueryProcessor* and template execution). The parameters are checked at runtime by the *QueryMgr*.

Example:

```
<parameters>
  <parameter name="TypeCode" type="java.lang.Integer" optional="false"/>
  <parameter name="ID" type="java.lang.String" optional="true"/>
  <parameter name="Domains" type="java.util.Iterator" optional="true"/>
  <parameter name="User" type="com.intershop.beehive.core.capi.user.User"
    optional="false"/>
</parameters>
```

Return Mapping

This section specifies which elements are returned in the resulting iterator of select queries. It defines not only the elements to be returned by the *QueryProcessor*, but also their transformation into higher level objects using the return mapping performed by the *QueryMgr*.

It is legal to return multiple values at once. In this case, row objects are returned that can easily be accessed via the get method or object path expressions. The column names are case-insensitive.

NOTE: The iterators returned by the *QueryMgr* will contain the plain objects instead of encapsulating them in rows when there is only one value per row.

Three mapping types are implemented:

- **rename**
Directly returns the value delivered by the *QueryProcessor*, optionally using a different name.
- **orm**
Builds persistent objects using the `getObjectByPrimaryKey` method of the according *ORM* factory.
- **constructor**
Calls the constructor for the given class applying the declared arguments.

The order of the attributes (the input parameters for the mappings) has to be the same as the primary key attributes for the *ORM* persistent object or the parameters of the constructor. And as there is no additional type mapping for these attributes, the *QueryProcessor* must deliver objects of the correct types.

If there is no return mapping defined, the *QueryMgr* will perform an identity mapping, which actually returns the original objects retrieved by the *QueryProcessor*. If only one value per row is returned, this value is automatically unwrapped.

Example:

```
<return-mappings>
  <return-mapping name="id" type="rename">
    <return-attribute name="sku"/>
  </return-mapping>
  <return-mapping name="product" type="orm" class="com.intershop.beehive.xcs.
    internal.product.ProductPO">
    <return-attribute name="uuid"/>
  </return-mapping>
  <return-mapping name="price" type="constructor" class="com.intershop.beehive.
    foundation.quantity.Money">
    <return-attribute name="currency"/>
    <return-attribute name="value"/>
```

```
</return-mapping>
</return-mappings>
```

In ISML templates, the returned objects can be accessed as follows:

```
<isloop iterator="SearchResultIterator" alias="Row">
  <isprint value="Row:ID">
  <isprint value="Row:Product:Name">
  <isprint value="Row:Price" style="MONEY_LONG">
</isloop>
```

Query Processor Declaration

This mandatory section defines which *QueryProcessor* performs the query. In addition, it allows for defining a configuration that is passed to the processor as a context. This configuration actually depends on the processor implementation and can be used, for example, for defining return types or the connection to the data source.

Furthermore, the query processor section provides for defining a query input parameter preprocessing. This can be used, for example, for translating a complex search expression into the native language of the underlying search engine.

Preprocessing the query input parameters prior to the template processing allows for evaluating the actual result in the template.

Example:

```
<processor name="OracleSQL">
  <processor-configuration name="readType.StartDate" value="Timestamp.GMT"/>
  <processor-preprocessing output="containsQuery" input="SearchExpression"
    processing="SearchExpression2Contains"/>
</processor>
```

Query Templates

Statements to be executed by the *QueryProcessor* are expressed with templates. For select queries, there are three types of statements:

- count

Intended to calculate integers as result.
- objects

Intended to return objects or manipulate data.
- countedobjects

Combined queries, intended to calculate a row count and return row(s) as result.

The *QueryMgr* decides which template is executed depending on the expected result and specified types. For update queries, all templates are executed in the order as they are defined in the file.

A query file can contain more than one query templates. In this case, however, they must refer to the same business task, e.g., searching for a certain object type or performing a specific task in the database. All templates then share the same parameter set, return mapping and processor configuration.

Templates are usually designed to produce a result that is ready to be used by the underlying search engine. This way, the *QueryProcessor* just has to pass the *QueryStatement* to the search engine.

A template constists of query text, a tag for accessing variables, flow control tags (`if`, `loop`, `call`) and a comment tag.

■ **variable Tag**

Passes dynamic values and predefined processing methods (`bind` and `text`) to the `QueryProcessor`. Custom `QueryProcessor`-specific methods are possible.

```
<template-variable value="text:xyz"/>
<template-variable value="text:xyz" processing="bind"/>
<template-variable value="ProductTable" processing="text"/>
<template-variable value="SearchExpression" processing="contains"/>
```

The value of required attribute `value` is an object path expression that is interpreted similarly as in ISML templates. This means that literals can be used. The attribute `processing` is optional. The processing method `bind`, which is the default, means that the `QueryProcessor` gets tokens of type `BindVariableToken`. Text tokens are produced for the processing method `text`. All other processing methods are passed to the `QueryProcessor` via the `CustomVariableToken`. This allows for implementing special handlings for values that cannot be handled meaningful in the query template. Custom `QueryProcessors` should also implement all preprocessing methods as custom processing methods.

■ **if Tag**

Used for conditional executions. In the future, logical expressions for object paths will be supported; currently only simple evaluations are possible.

```
<template-if condition="isDefined(SortByDisplayName)">
...
<if-elseif condition="isDefined(SortByUserName)" />
...
<if-else/>
...
</template-if>
```

The attribute `condition` is required, its is an object path expression. Branches are introduced using the `elseif` and `else` tags, which do not have any members.

■ **loop Tag**

Used for iterated executions.

```
<template-loop iterator="ProductDomains" alias="Domain">
  <loop-statement>...</loop-statement>
  <loop-separator>...</loop-separator>
</template-loop>
```

The two attributes `iterator` and `alias` are required. The value of `iterator` is an object path expression that returns an object to be iterated, like arrays, collections, enumerations or iterators. The `alias` defines the name for the current loop element during the loop execution. A possibly existing value with this name is hidden for the time of the loop execution, but is visible again after the execution.

■ **call Tag**

Allows for reusing common query parts.

```
<template-call name="inc/SearchExpression2Like">
  <call-parameter name="SearchExpression" value="SimpleSearchExpression"/>
  <call-parameter name="ColumnNames" value="'u.lastName u.firstName'"/>
  <call-parameter name="CaseInsensitive" value="true()"/>
```

```
<call-parameter name="ConcatenationOperator" value="'" AND "'"/>
</template-call>
```

The required attribute `name` contains the name of the query file without the `.query` postfix. Note that query files can be grouped in directories. For the query lookup, the same site context as for the current query is used. The `sub tags` parameter specify the values passed to the sub query for execution. The sub query can access only values defined here. The sub query can contain only one template, or else the interpreter aborts the execution.

■ comment Tag

Used to add comments to the query template that are not passed to the search engine.

```
<template>
  select uuid from product where
  domainid = 'xyz'
  <template-comment>and sku like 'abc%'<template-comment>
</template>
```

Although SQL statements have their own syntax for comments, the query framework cannot rely on a single syntax since it has to support any query language. Thus, the query framework uses its own comment tags in order to prevent comments from being submitted to the underlying search engine.

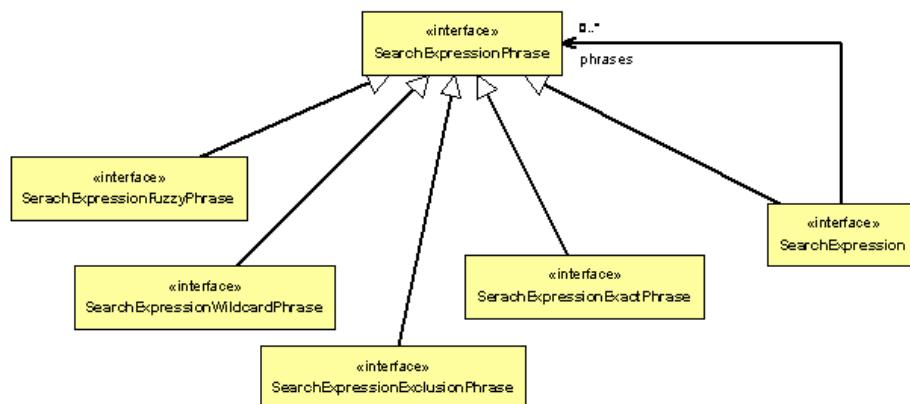
Public Java Interfaces

The public Java interfaces are located in `com.intershop.beehive.core.capi.query`. All functionality that is available with the pipelets is accessible at the Java API level.

SearchExpression

As user search expressions are not a core feature of the query framework, they are not located in the core cartridge but in `bc_foundation`, sub package `com.intershop.component.foundation.capi.search`. The provided classes are `SearchExpressionParser`, `SearchExpressionFactory` and `SearchExpressionParsingException`.

Figure 202. SearchExpression interface



QueryMgr

The *QueryMgr* is the central manager of the query framework. It provides the following methods: `loadQuery`, `executeCountQuery`, `executeObjectsQuery`, `executePageableQuery`, `executeUpdateQuery`, and `getQueryProcessor`.

■ ExecuteUpdateQuery

Triggers the `executeUpdateQuery` of the *QueryProcessor* for all templates in the given *Query* in the order as they are defined in the query file.

■ GetQueryProcessor

Returns the *QueryProcessor* instance registered under the given name. Each query file defines which processor can execute the templates.

Query

This interface represents a query. Generally, the *Query* holds two elements: the query definition (the representation of a query file) and the bound parameter values.

It also stores the query results. This allows, for example, for reusing the result of a count query for the object query in case it already contained the objects. Reusing the result is not possible, however, if the parameters have been changed.

QueryStatement

A *QueryStatement* is the result of the execution of a query template with the given parameter values, which is passed to the *QueryProcessor* for execution. A *QueryStatement* contains a list of tokens. Three types of tokens exist:

■ TextToken

A plain text fragment of the actual query for the underlying search engine.

■ BindVariableToken

A variable value for the actual query to be passed directly to the underlying search engine.

■ CustomVariableToken

A variable value with a processing method as specified in the query file to be used for special handling within the *QueryProcessor*.

QueryContext

The *QueryContext* contains additional information required by the *QueryProcessor* for the query execution. It can contain, for example, a JDBC connection and the processor configuration from the query file.

QueryParameterHandler

Before a query template is executed, the parameters can be preprocessed. A class responsible for preprocessing has to implement this interface.

QueryResult

A *QueryResult* is the result from a `select` query returned by the *QueryProcessor*. In addition to the rows, it also contains some meta information, e.g., the columns of the rows or the row count.

Row

A *QueryResult* has to return *Row* elements. According to the return declaration in the query file, the *QueryMgr* maps them to new *Rows* or to single elements. A *Row*, basically, constitutes a map that holds values for case insensitive names.

QueryProcessor

The *QueryProcessor* actually performs the queries. It is triggered by the *QueryMgr* from the `execute` methods. Upon customization, this interface can be implemented in order to link other search engines.

NOTE: Make sure that custom processors are subclasses of *AbstractQueryProcessor*.

Intershop 7 provides three *QueryProcessors*: the *ORMQueryProcessor* executes queries via Intershop 7's ORM layer, the *JDBCQueryProcessor* uses any JDBC connection in a generic way, and the *OracleSQLQueryProcessor* as a subclass provides particular support for Oracle databases, like custom processing for context index search.

■ ORMQueryProcessor

This query processor executes queries via the ORM layer of Intershop 7. It only supports `select` queries with the template types `objects` and `countedobjects`. The supported parameter handlers include `LastModifiedDate` and `SearchExpression2Contains`.

Queries are executed via the factory method `getObjectsBySQLJoin()`, which automatically assigns the alias `this` to the selection table that can be used within the `WHERE` condition. The processor has no configuration, but uses its own query syntax instead to extract the factory name and the parameters required by the `getObjectsBySQLJoin()` method. The processor returns the retrieved ORM objects encapsulated in rows under the key `ORMOBJECT`. This allows for defining additional return mappings.

Template syntax:

```
SELECT FROM <tablename | ORM factory name>
[JOIN <joins>]
[WHERE <conditions>]
```

■ JDBCQueryProcessor

The generic query processor can use any `JDBCConnection`. It has the following configuration properties:

- `dataSourceName` = the name of a datasource
- `readType.<column_name>` = [Array | AsciiStream | BigDecimal | BinaryStream | Blob | Boolean | Byte | Bytes | CharacterStream | Clob | Date | Date.GMT | Double | Float | Int | Long | Object | Ref | Short | String | Time | Time.GMT | Timestamp | Timestamp.GMT | URL]

Note also that:

- a returned column name `ROWCOUNT` is interpreted as the count of rows in the result set,
- if no connection or datasource name is given explicitly, the default data source is used,
- upon executing `select` statements, the shared read connection pool is used if there is no connection given explicitly.

■ OracleSQLQueryProcessor

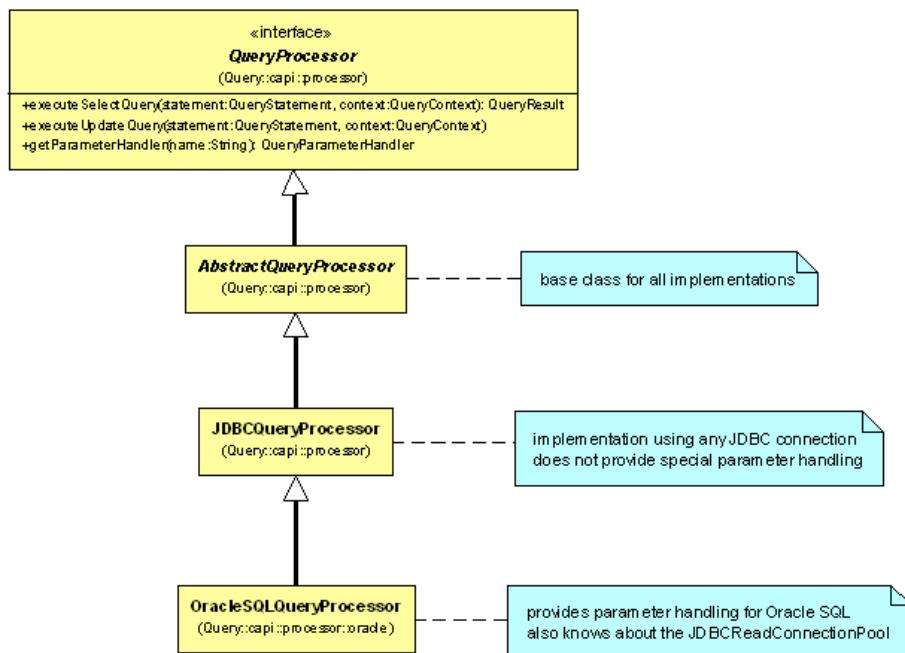
This subclass of `JDBCQueryProcessor` has the following extensions:

- handles the Oracle DRG-51030 error that can be thrown on `contains` queries,
- handles the GMT time conversion when `Date` objects are passed as bind parameters,
- supports the parameter handler `LastModifiedDate`, which makes an application server date comparable to the database `sysdate`,
- supports parameter handler `SearchExpression2Contains`, which builds a `contains` query from a `SearchExpression`.

Query processors and parameter handlers are registered in a file named `resources/<cartridge_name>/naming/queryprocessors.properties`, for example, specifying the following settings:

```
OracleSQL=com.intershop.beehive.core.capi.query.processor.oracle. \
    OracleSQLQueryProcessor
OracleSQL.contains=com.intershop.beehive.core.capi.query.processor.oracle. \
    ContainsHandler
```

Figure 203. QueryProcessor interface



The *QueryProcessors* provide the following methods: `executeSelectQuery`, `executeUpdateQuery` and `getParameterHandler`.

■ **executeSelectQuery**

Called by the *QueryMgr* when executing `count`, `objects` and `countedObjects` queries.

- Parameters: `QueryStatement` (the statement to be executed), `QueryContext` (context for the processor that holds the data given by the caller of the *QueryMgr* method and the processor configuration of the query file)
- Returns: `QueryResult` if the query was executed successfully
- Exceptions: `QueryRejectedException` (thrown in case the underlying search engine is not able to answer a formally correct query, e.g., the DRG-51030 Oracle error), or else any `RuntimeException`

■ **executeUpdateQuery**

Called by the *QueryMgr* when executing `update` queries.

- Parameters: `QueryStatements` (lists of statements to be executed), `QueryContext` (context for the processor that holds the data given by the caller of the *QueryMgr* method and the processor configuration of the query file)
- Exceptions: `RuntimeException` in case the query was not executed successfully

■ **getParameterHandler**

Returns an instance of `QueryParameterHandler` that is responsible for converting a query parameter value into a new value more suitable for the underlying search engine; and used by the *QueryMgr* to perform the preprocessing actions defined in the query file.

- Parameter: `processing` (the identifier of a processing method for the handler)
- Returns: the `QueryParameterHandler` or `null` if no processing with the specified name is defined

NOTE: The *AbstractQueryProcessor* already includes a working implementation of that method. Thus, there is usually no need for a custom implementation.

Creating and Editing Query Files

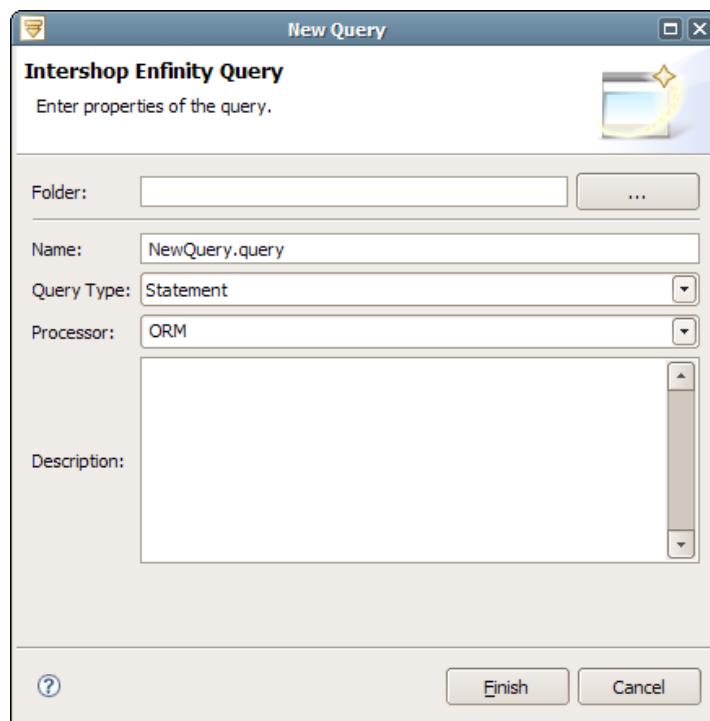
Intershop Studio provides the New Query wizard to create and the Intershop 7 Query Editor to edit query files.

Creating Query Files

For creating query files, the New Query wizard helps to collect the required basic information. To create a query file:

1. **In the Cartridge Explorer, right-click the cartridge project for the new query file to open the context menu.**
2. **From the context menu, select New | Query.**

The New Query wizard is displayed.

Figure 204. General query properties

3. Specify the general query properties.

The properties are described in the table below:

Table 97. General query properties

Property	Description
Folder	Defines the parent directory for the new query. If no parent directory is specified, the query file is created directly in <IS_SOURCE>/<cartridge_name>/staticfiles/cartridge/queries.
Name	The file name must be unique within the selected directory. A valid file name must include the extension .query.
Query Type	Specifies the query template type (none, objects, count, countedobjects). Depending on this type, an empty query template is created.
Processor	Specifies the query processor (ORM, OracleSQL or JDBC, see <i>QueryProcessor</i>).

4. Click Finish to save the query.

This opens the Intershop 7 Query Editor, displaying the details' overview of the new query. For information about editing the query details, refer to *Editing Query File Details*.

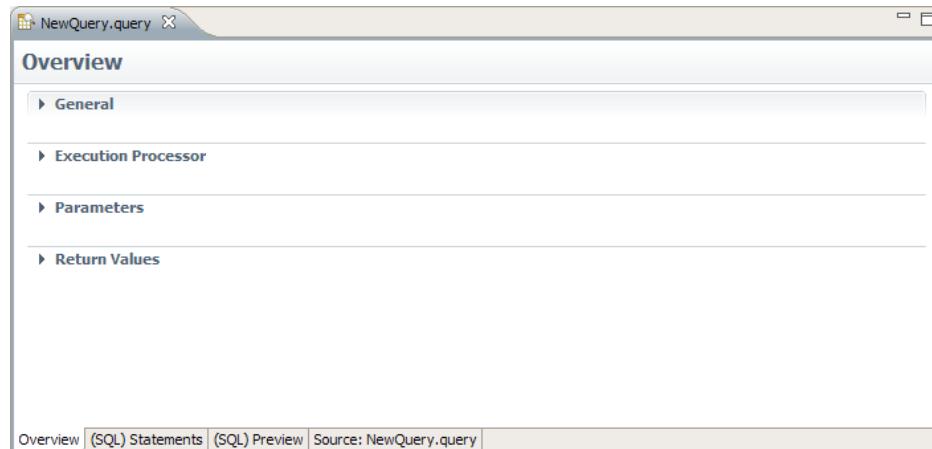
Editing Query File Details

Intershop recommends to edit query file details using the Intershop 7 Query Editor included with Intershop Studio. This editor integrates with the Eclipse Data Tools Platform for accessing data and provides the complete Eclipse-specific editing functionality like content assist, etc.

To open a query file for editing, you can either

- double-click the query file in the Cartridge Explorer, or
- right-click the query file in the Cartridge Explorer to open the context menu, and select Open With | Intershop 7 Query Editor.

Figure 205. Intershop 7 Query Editor Overview page



This opens the Intershop 7 Query Editor, displaying the details' overview of the selected query. To edit the details, you can either

- open the corresponding section from the overview in the editor, as described below, or
- select an element from the Workbench's Outline view and edit it in the Properties window.

General Query File Details

The General section provides a text field for editing the description for the query file. In the XML source, this description will be added as `description` attribute to the `<query>` element.

Execution Processor

This section provides a select box for defining which `QueryProcessor` performs the query. By default, you can choose between ORM, JDBC and OracleSQL.

To add a processor configuration or an input preprocessing instruction:

1. **In the Execution Processor section, click Execution Processor: Preprocessing or Execution Processor: Configuration as required.**

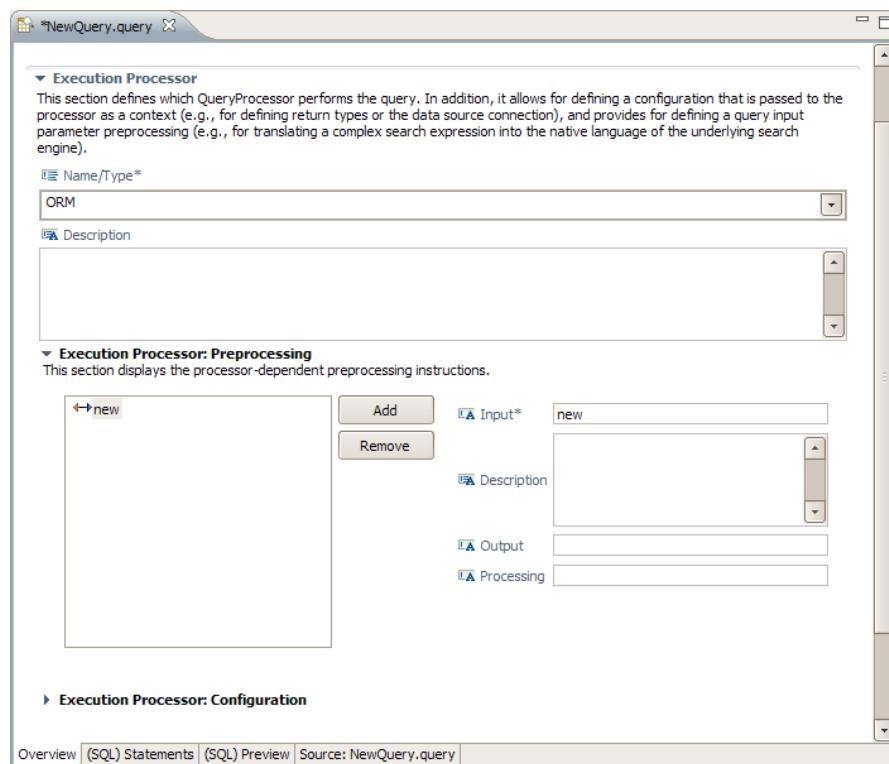
This opens a dialog to create and define preprocessing or configuration elements in the processor section of the query file.

2. **In the Preprocessing or Configuration section, click Add.**

This creates an empty preprocessing element, or accordingly a configuration element.

3. **Click the new configuration or preprocessing element in the list.**

This opens a dialog to specify the preprocessing or configuration attributes.

Figure 206. Defining execution processor preprocessings and configurations

The attributes are listed in the tables below:

Table 98. Query processor preprocessing

Attribute	Description
Input	The input value for the preprocessing.
Description	A description for the preprocessing (optional).
Output	The output value.
Processing	The processing type.

Table 99. Query processor configuration

Attribute	Description
Name	The configuration name.
Description	A description for the configuration (optional).
Value	The value that is passed to the processor as context.

4. Save the settings.

For more information about the query file processor section, refer to *Query Processor Declaration*. For detailed information about the processor API, refer to *QueryProcessor*.

Parameters

This section provides a dialog to create and define query parameters. Generally, parameters define which values the query expects.

To add a parameter:

- 1. In the Parameters section, click Add.**

This creates an empty parameter element.

- 2. Click the new parameter in the list.**

This opens a dialog to specify the required parameter attributes, as listed in the table below:

Table 100. Query parameter attributes

Attribute	Description
Name	The parameter name.
Description	A description for the parameter (optional).
Java Class	The expected object type, e.g., string or integer.
Optional	The flag defining the parameter as optional or required.

- 3. Save the settings.**

For more information about the query file parameter section, refer to *Parameter Declaration*.

Return Values

This section provides a dialog to create and define return mappings for the query. Generally, return mappings define which elements are returned in the resulting iterator of select queries and how they are transformed into higher-level objects.

To add a return mapping:

- 1. In the Return Values section, click Add.**

This creates an empty return mapping element.

- 2. Click the new mapping in the list.**

This opens a dialog to specify the attributes for the mapping, as listed in the table below:

Table 101. Query return mapping attributes

Attribute	Description
Name	The return mapping name.
Description	A description for the return mapping (optional).
Java Class	The expected object type, e.g., string or integer.
Mapping Type	The intended mapping type (rename, orm, constructor).

- 3. With the new mapping selected, click Add Parameter.**

This opens a dialog to specify the required return attributes, i.e., the input parameters for the mappings, like `uuid`, `sku`, etc.

- 4. Save the settings.**

For more information about the query file return mapping section, refer to *Return Mapping*.

SQL Statements

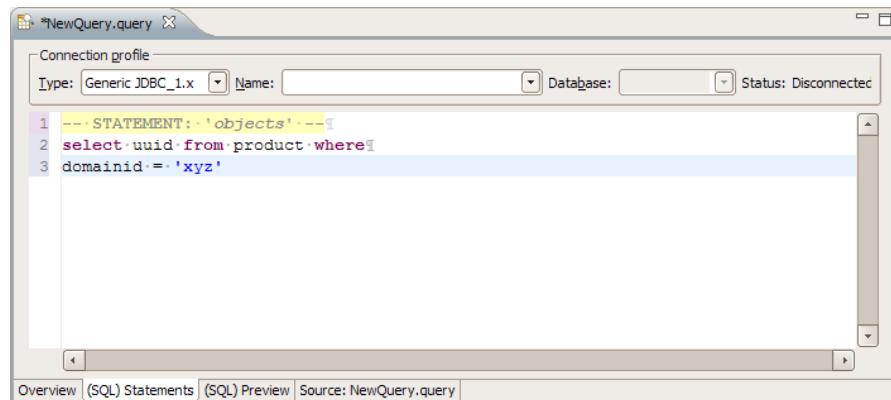
This tab provides an interface to create and edit the actual SQL statements to be passed to the search engine, as well as tags for accessing variables, flow control tags and comment tags. By default, three types of statements are supported: objects, count and countedobjects. The statement templates are defined upon creating the query using the wizard (see *Creating Query Files*).

To edit a query template:

- Open the SQL Statements tab of the Query Editor.**

This tab contains a connection profile dialog and an SQL editor window.

Figure 207. Editing SQL statements in the query editor



- Edit the connection profile as required.**

Specify a connection type, a connection name and the database to connect.

- Edit the SQL statement as required.**

For editing the SQL source code, you can use all Eclipse-specific features like content assist, data source integration, etc.

- If necessary, you can change the template type.**

Select the required type from the select box in the Type row of the Properties View.

- If necessary, enter a description for the new template element in the corresponding field in the Properties View.**

- Save the settings.**

To delete query template:

- In the query tree view in the Cartridge Explorer, navigate to the template to be deleted.**

- Right-click the template to be deleted.**

This opens the context menu.

- Select Delete from the context menu.**

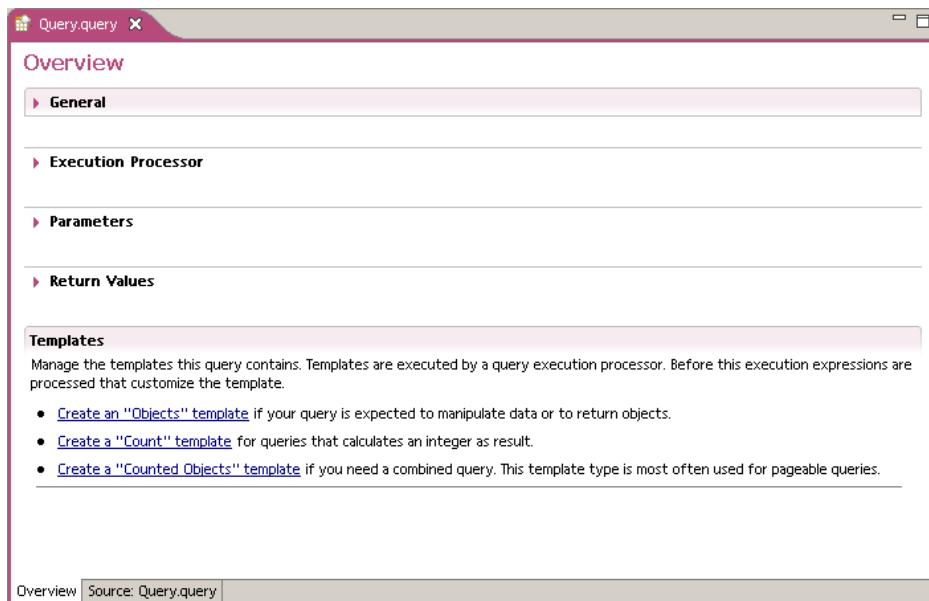
Click OK to confirm the deletion.

Query Editor

User Interface

The query editor allows you to create and manage queries defined for Intershop 7's new file-based query framework. This editor integrates with the Eclipse Data Tools Platform for accessing data and provides the complete Eclipse-specific editing functionality like content assist, etc.

Figure 208. Intershop 7 Query Editor Overview page



Sections exposed by the query editor are outlined below.

Table 102. Query Editor Sections

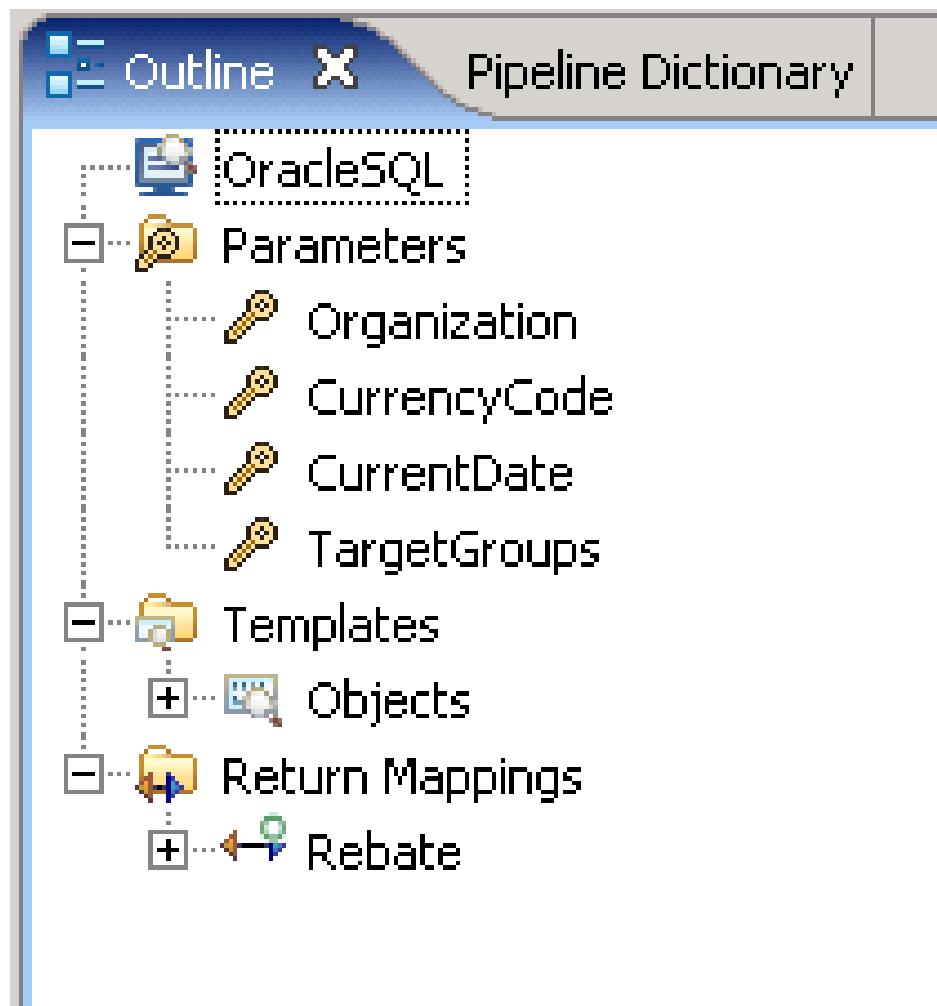
Section	Description
General	The General section provides a text field for editing the description for the query file. In the XML source, this description will be added as <code>description</code> attribute to the <code><query></code> element.
Execution Processor	This section provides a select box for defining which <code>QueryProcessor</code> performs the query. By default, you can choose between JDBC and OracleSQL.
Parameters	This section provides a dialog to create and define query parameters. Generally, parameters define which values the query expects.
Return Values	This section provides a dialog to create and define return mappings for the query. Generally, return mappings define which elements are returned in the resulting iterator of <code>select</code> queries and how they are transformed into higher-level objects.

Section	Description
Templates	This section provides an interface to create and edit the actual SQL statements to be passed to the search engine, as well as tags for accessing variables, flow control tags and comment tags.

Outline View for Query Editor

In the context of the Pagelet Editor, the Outline View provides a structural view on the query that is currently active in the editor, including the possibility to expand and collapse parts as needed.

Figure 209. Outline View for Pagelet Editor



Webform Development

Overview

Intershop 7 provides a new webforms framework to ease the creation, validation and processing of HTML webforms.

Additionally, the webform development process is aided by Intershop Studio in form of a webform creation wizard and a webform model editor.

HTML webforms resemble paper or database forms, because users fill out the forms using checkboxes, radio buttons, or text fields.

Webforms can be used as input templates when collecting data from users, e.g. the user enters address or credit card data, but also to query and display existing data, e.g. the user enters a search term and the query results (e.g. a record in a database) are displayed and layed out using a webform.

The new framework helps to significantly reduce the implementation effort in pipelines and templates needed to configure and create HTML webforms.

In comparison with the previous Intershop 7 webforms concept, the new framework has the following advantages:

- fosters the reuse of webform definitions within and across applications
- webforms are initialized within pipelines instead of templates
- reduces the number of pipelets needed to set up webforms resulting in less complex pipelines
- supports reusable and configurable validators to check if form data entered by the user is valid

Validators can be assigned to forms, form parameters, and form fields.

Validators may be used to perform the following types of checks:

- check if a form parameter or field exists
- check if a form parameter or field is within a specific range
- check if a compound form parameter can be built from its input fields
- perform complex cross checks across multiple form parameters and fields
- supports formatters

Formatters are used to transform a Java object (e.g. a Money object) into a string representation to be viewed by the HTML form.

- supports the creation of your own custom validators and formatters
- supports static and dynamic webforms

Static webforms always contain the same parameters that are defined at development time. Only the values for the parameters may change over the lifetime of the form.

Dynamic webforms do not have a fix set of parameters. Instead the parameters making up the form are determined dynamically at runtime.

- supports wizard-like multi-page webforms that preserve user entered data over multiple requests

- has built-in localization support

The validation of form fields can be based on configurable localizable rules.

- supports error handling

If the validation of the form fails, a localizable error message can be presented to the user.

The error message is defined as part of the webform definition, and not in the ISML template.

- supports binding data to webforms

Webforms can be mapped to business objects (extensible objects) allowing for easy pre-filling of forms.

- prevents cross-site request forgery attacks by protecting request parameters from user manipulations

General Process for Using Webform

The following steps outline the overall process for defining and using webforms. The example assumes the usage of two pipelines, one for creating and viewing the form in the first place, and a second one to validate and process the HTML form data returned by the client. Of course, other scenarios, where all functionality is implemented in a single pipeline are possible too.

The general process is as follows:

1. Create and configure the webform using Intershop Studio.

Intershop Studio features a wizard that helps you to create and setup webform models.

A webform model is a container that can hold the definitions of forms and their elements (form parameters and form fields). Additionally, webform models define validators and formatters to be bound to form elements.

Intershop Studio stores webform models in form of EMF based XML files that have the extension *.webform.

Intershop Studio contains also a dedicated webform editor for editing and configuring webform models.

2. If required, implement custom validators and formatters.

In most cases you will use the "primitive" validators and formatters that come with the platform. However, for special cases you can define your own custom validators and formatters.

3. Set up a pipeline for viewing the HTML form

The pipeline creates and pre-fills a form instance, then calls a ISML template to render the HTML form.

4. Set up a template for viewing the webform.

The display template contains the code for the HTML form. Via object path expressions of the form `Form:<ObjectName>:QualifiedName` parameters and fields of the form instance in the pipeline dictionary are mapped to request parameters of the HTML form.

The same template can also contain the code for displaying error messages to the user in case the webform data entered is invalid.

5. Set up a second pipeline to validate and process the HTML form values.

This pipeline validates and processes the HTML form values returned by the client.

The following sections explain each step outlined above in more detail.

Set Up a Webform Model

General

Webform models are containers that are used for defining and grouping related webform artifacts.

Those artifacts might be definitions for forms, validators, and formatters. For example you could have a webform model that defines different kinds of login forms, or you could group several validators into one webform model.

Webform models are created and edited using a dedicated Intershop Studio wizard (*Creating Webform Models*) and editor (*Editing Webform Models*).

Webforms are defined as EMF models in XML notation and stored in files with the extension `.webform`. In the directory structure of a cartridge, webform models reside in `release/webforms` next to the `pipelines` and `templates` directories. For example, the `sld_ch_b2c_app` cartridge has the following directory structure:

```
// some directories have been omitted

<cartridge_name>
  release/
    pipelines/
    templates/
    webforms/
      account/
      checkout/
      helpdesk/
      order/
      product/
      stores/
```

The following is an example of a webform model that defines a form with form parameters and assigns validators:

```
<?xml version="1.0" encoding="UTF-8"?>
<webform:WebForm
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xmlns:core="http://www.intershop.de/core/2010"
xmlns:webform="http://www.intershop.de/webform/2010"
name="LoginForm"

<forms name="LoginForm">
    <parameters xsi:type="webform:FormParameter" optional="false" ...>
        <parameter name="Login" type="java.lang.String">
            <validatorBindings validator="GlobalValidators-email"/>
        </parameters>
        <parameters xsi:type="webform:FormParameter" optional="false" ...>
            <parameter name="Password" type="java.lang.String">
                <validatorBindings validator="GlobalValidators-regexp">
                    <parameterBindings xsi:type="core:ParameterValueBinding" ...>
                        <parameter name="regExp" value=".{6,}"/>
                    </parameterBindings>
                </validatorBindings>
            </parameters>
        </forms>
    </webform:WebForm>

```

Note the following comments regarding the code sample above:

- The form model name is defined as `name` attribute of the root tag.
- the form model name is used to lookup a specific webform. It is recommended to set this name equally to the model's file name, in this example the file is saved as `LoginForm.webform`.
- Within one webform model multiple forms, validators, and formatters can be defined.

Creating Webform Models

The webform model wizard allows you to add a new webform model to a cartridge project. The wizard collects all required information and creates an according `*.webforms` file below the `/staticfiles/cartridge/webforms` directory.

To start the webform model wizard:

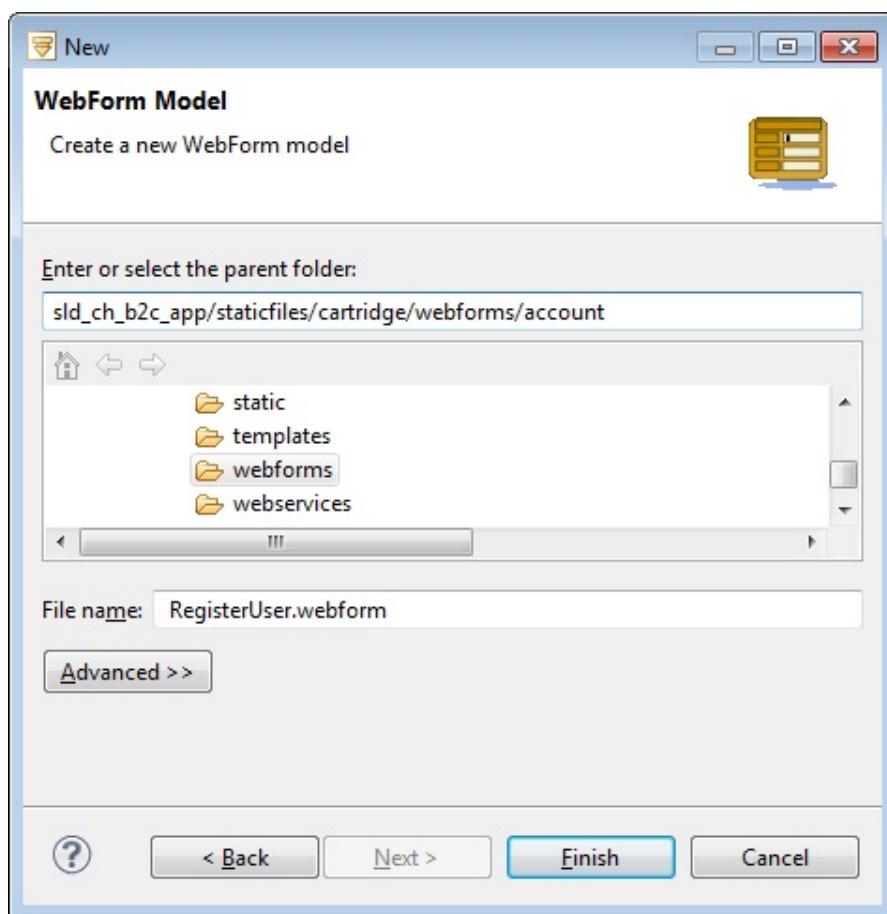
- 1. In the Cartridge Explorer, right-click the cartridge project for the new webform model to open the context menu.**
- 2. From the context menu, select New | Webform Model.**

The webform model wizard is displayed.

NOTE: If you have not selected a cartridge before starting the webform model wizard, the webform model wizard first prompts you to select the cartridge for the webform model.

- 3. Specify the name and the location for the new webform model.**

Give the webform model a unique and meaningful name. The name of the file will also be used as the webform model name.

Figure 210. Webform Wizard

Editing Webform Models

After creating the webform model, the webform model editor provides several ways to edit the model. The editor is divided into three areas:

- **a tree view**

Shows a structural view of the model, where you can add elements to the model by right-clicking an element and choosing element types from the context menu.

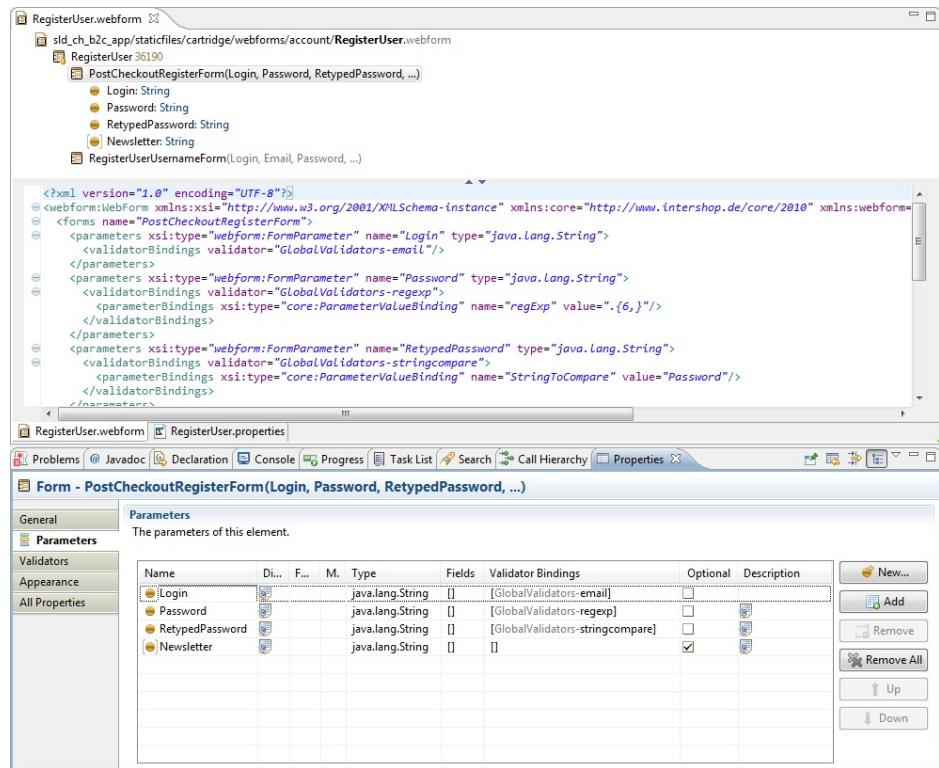
- **a source code editor**

Shows the source code of the webform model file allowing you to edit the XML directly.

- **a properties view**

Here you can edit the properties of the currently highlighted element.

The graphical editors allow you to select referenced resources and provide guidance when creating new elements.

Figure 211. Webform Model Editor

Optional: Implement Validators and Formatters

Intershop 7 comes with a number of pre-defined standard validators and formatters. They are defined in the two webform models `GlobalValidators.webform` and `GlobalFormatters.webform` respectively which are both part of the `core` cartridge, see the `core/staticfiles/cartridge/inc/` directory.

The following out of the box *validators* are available: `string`, `short`, `integer`, `long`, `double`, `date`, `percentage`, `money`, `quantity`, `regexp`, `email`, `url`, `required`, `equalsto`, `minlength`, `maxlength`, `datemin`, `datemax`, `integerrange`, `longrange`, `floatrange`, `doublerange`

The following out of the box *formatters* are available: `string`, `integer`, `long`, `float`, `bigdecimal`, `double`, `money`, `date`, `quantity`, `email`, `percentage`.

In some cases, you may want to create your own custom validators or formatters. This is a two step process:

1. **Define the validators and formatters in a webform model similiar to the standard ones.**
2. **Create according Java classes. These classes must either implement the Validator or the Formatter interface.**

Validators

A Validator tests the value of a given target object, usually a form parameter or a form field. The target object can also be an entire form in order to perform complex cross checks.

Validators implement the Interface Validator.

```
public interface Validator
{
    public enum ValidationResult {FAILED, SUCCESS, FAILED_CONTINUE, SUCCESS_STOP}
    public ValidationResult validate(ValidatorContext context);
}
```

The Validator interface defines the single method validate(). The method accepts a ValidatorContext as parameter and returns a ValidationResult.

Based on the returned validation result the validation process continues. ValidationResult can have one of the following values:

■ **SUCCESS**

The validation was successful, the validation process continues with the next validator.

■ **SUCCESS_STOP**

The validation was successful, the validation process stops. No further validators are called for the target object.

■ **FAILED**

The validation failed, the validation processing stops. No further validators are called for the target object.

■ **FAILED_CONTINUE**

The validation failed and the validation process continues with the next validator.

Based on the possible validation results, the following validation scenarios are possible:

- Stop the validation of the target object when the first validator fails.
- Stop the validation process without signalling a failure.

For example, you can check if a dependent parameter is set. If yes, then the current parameter must have a value (required). If not, then the current parameter must not be processed.

- Continue the validation after a failure.

This scenario is useful if you want to validate and possibly return error messages to the user for multiple input fields at once.

The Intershop 7 platform comes with a number of standard validators that are available through the resource `GlobalValidators` of class `com.intershop.beehive.core.capi.webform.ValidationMethods`.

The following is an example definition for a `DateTimeValidator` taken from a webform model:

```
...
<validators type="com.intershop.beehive.core.capi.webform.
    ValidationMethods$DateTimeValidator"
    name="datetime" message="error.datetime">
    <parameters name="day" type="java.lang.String"/>
    <parameters name="time" type="java.lang.String"/>
    <parameters optional="true" name="dateTimePattern"
        type="java.lang.String"/>
</validators>
...
```

Note the following comments regarding the code sample above:

- the `name` attribute of a validator definition is required.
It defines the identifying name of the validator and must adhere to the Java naming conventions (e.g. no white spaces).
It is recommended to use an all lowercase identifier.
- the required `type` attribute must point to a fully qualified Java class name of the validator.
- The optional `message` attribute specifies a validator specific error code that references an error message to be presented to the user in case the validation fails.
- The `parameters` definitions are used by the element checker to perform consistency checks of the webform definitions.
- Validators can be overwritten by validators in other cartridges. In this case the parameter list of the overwriting validator must contain the same parameters as the one overwritten. Additionally, the overwriting parameter list may define additional optional parameters.

Formatters

Formatters are used to transform the object value of a form parameter or a form field into a string representation. Formatters are usually triggered from within templates, like this: `TestForm:IntegerValue:FormattedValue`.

Formatters implement the Interface `Formatter`.

```
public interface Formatter
{
    public String format(FormatterContext context);
}
```

The `Formatter` interface defines the single method `format()`. The method accepts a `FormatterContext` as parameter and returns a `String` as result.

A number of pre-defined formatters can be found in the resource `GlobalFormatters`.

The following is a sample definition for a formatter from a webform model:

```
<formatters type="com.intershop.beehive.core.capi.webform.\
```

```
FormatterMethods$DateFormatter" name="date">
<parameters optional="true" name="style" type="java.lang.String"/>
<parameters optional="true" name="pattern" type="java.lang.String"/>
</formatters>
```

Create a Pipeline to Create a Form Instance

This pipeline instantiates a `Form` object based on a form definition from a webform model, then calls a template for displaying the resulting HTML webform to the user.

For the first task, the framework provides the pipelets `CreateForm` and `CreateListForm`. More information on the available pipelets can be found here [Webform Related Classes and Pipelets](#).

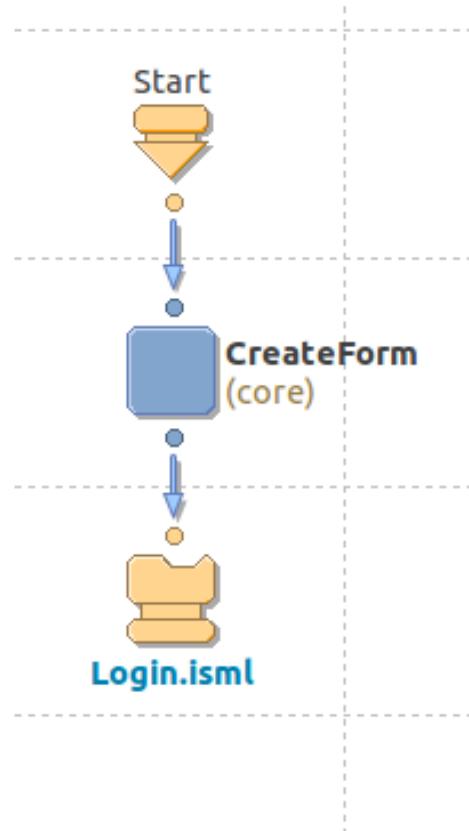
The form instance is stored in the pipeline dictionary.

In a next step, the form can be pre-filled with data. This can be done using pipelets such as `SetFormParameterValue`, `SetFormParameterValues`, or `MapObjectToForm`.

In a final step, the pipeline calls a template that shows the HTML webform to the user.

The following is an example pipeline that creates a `Form` instance from a webform definition:

Figure 212. Webform Creation Pipeline



This is the configuration of the `CreateForm` pipelet:

Figure 213. CreateForm Pipelet Configuration

Pipelet Node - CreateForm (core)	
Property	Value
Configuration	
FormName	LoginForm
FormResource	LoginForm
Dictionary Input Bindings	
Dictionary Output Bindings	
Form	LoginForm
Properties	

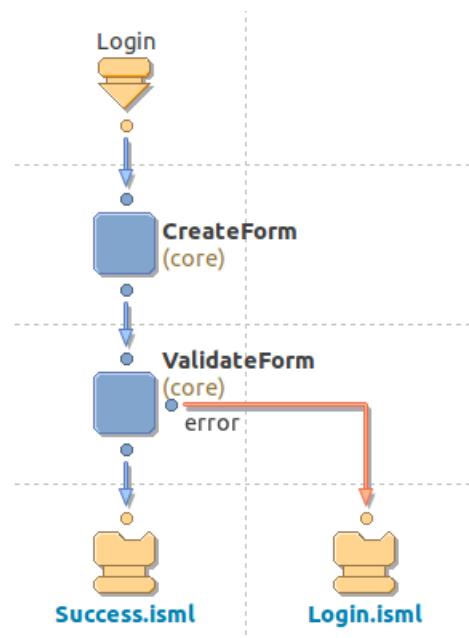
Create a Pipeline to Handle the Input

You need to set up a pipeline that validates and processes the HTML form data received with a client request.

When the client sends the HTML form data back to your application, it depends on the submission method (either GET or POST) how the data is transferred. Usually, you will use the default method GET which encodes the form data as URL request parameters.

The Intershop 7 Web adapter parses the contents of the HTML request and stores the result (including the HTTP request parameters from the HTML form) in the pipeline dictionary.

The following is an example pipeline that processes the form parameter values returned from the client as part of a HTTP request:

Figure 214. Webform Creation Pipeline

This is the configuration of the CreateForm pipelet:

Figure 215. CreateForm Pipelet Configuration

Pipelet Node - ValidateForm (core)	
Property	Value
► Configuration	
▼ Dictionary Input Bindings	
Form	dict(LoginForm)
► Properties	

The pipeline does the following:

Because form objects are stateless, forms are not stored in the session. Hence, the pipeline uses again a CreateForm pipelet to create a new Form instance similar to the one created in the first pipeline.

When the ValidateForm pipelet executes, it does the following:

- From the pipeline dictionary, it reads the submitted request parameter values and initializes the mapping form elements of the form object.
- It checks the form element values against their constraints
- It performs a type conversion, e.g. converts strings (initially all HTTP request parameter values are of type String) into the required object type of the form element, e.g. Integer or Money.

Depending on the result of the validation, the pipeline then branches to handle the different conditions.

If the validation succeeds, the Form instance in the dictionary is filled with the valid data from the HTML form. The pipeline could now continue to process the form data (e.g. persist the data in the database) and display a confirmation message to the user.

If the validation fails, you can use the form instance in the dictionary to re-display the HTML form to the user along with error messages. The user can then make corrections and resend the form to the application for another validation.

Set Up a Display Template

In the template, you define the HTML form and map the parameter and field values of the Form instance (stored in the pipeline dictionary) to the according request parameters of the HTML form.

The lookup of the input values from the pipeline dictionary is made by invoking object path expressions of the form Form:<ObjectName>:Qualifiedname (see *Object Path Expressions*).

The qualified name must be unique within one form, e.g.:

```
<Form-Name>[CONSBHK:Sub-Form-Name]<Parameter-Name>[CONSBHK:_Field-Name].
```

The same template can also contain the error handling (display of error messages on client side if validation fails) and can therefore be used by both pipelines.

The following is an example template that shows how to construct an HTML form and how to use object path expressions:

```
...
<!-- Error handling -->
<isif condition="#(LoginForm:isValid)">
    <!-- Usually error messages will be referenced by a key with
        <istext key=". /> to simplify localization -->
    <isif condition="#LoginForm:Password:isError("error.email")#"> ...
        Your login needs to be a valid email address.</isif>
    <isif condition="#LoginForm:Password:isError("error.regexp")#"> ...
        Your password is too short.</isif>
</isif>

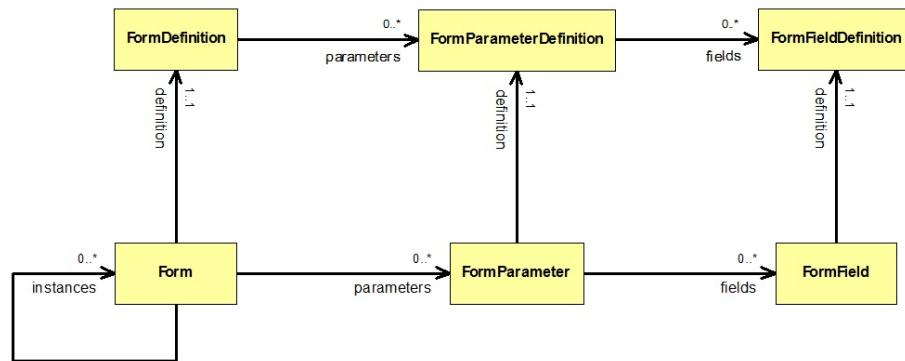
<!-- Start login form -->
<form action="#URLEX('https','','Action('ViewLogin-Login'))#" method="post" name=<isprint value="#LoginForm:Alias#">">
    <div>
        <label for=<isprint value="#LoginForm:Login:Qualified Name#">> ...
            Login:</label>
        <input type="text" id=<isprint value="#LoginForm:Login:Qualified Name#">
            name=<isprint value="#LoginForm:Login:Qualified Name#">
            value=<isprint value="#LoginForm:Login:Value#" /> ...
    </div>
    <div>
        <label for=<isprint value="#LoginForm:Password:Qualified Name#">> ...
            Login:</label>
        <input type="password" id=<isprint value="#LoginForm:Password:Qualified Name#">
            name=<isprint value="#LoginForm:Password:Qualified Name#">
            value=<isprint value="#LoginForm:Password:Value#" /> ...
    </div>
    <div>
        <button type="submit" value="Login" name="login">Send</button>
    </div>
</form>
...

```

Webforms Reference

Webform Related Classes and Pipelets

The webforms framework provides basic webform classes as depicted in the following diagram:

Figure 216. Basic Java classes provided by the webforms framework

The following pipelets are related to these classes:

■ **CreateForm:**

The configuration of the `CreateForm` pipelet defines a `FormDefinition` and returns a `Form` instance that holds this definition.

■ **CreateListForm:**

Creates a `Form` object based on the given webform model for one page of the given pageable of objects. Otherwise similar to `CreateForm`.

■ **ValidateForm:**

Fills form parameter a value of a given form object with submitted request parameter values, and converts them to the target objects and checks them against the constraints.

■ **FillForm:**

Fills the form parameters and fields with request form values (strings) without performing any validation steps or conversion operations.

■ **SetFormParameterValue:**

Set a parameter value of a given form object with a value determined within a pipeline.

■ **SetFormParameterValues:**

Similar to `SetFormParameterValue` but uses a map of parameter-value pairs.

■ **SetFormError:**

Sets an error code for the whole form or for a single field (form parameter).

■ **AddFormParameterDefinition:**

Adds an additional field to the webform.

■ **MapObjectToForm:**

Maps the values from an object to form parameter values.

■ **MapFormToObject:**

Maps form parameter values to an object.

■ **Validator (Interface)**

- defines a method for validating form data

- accepts a ValidatorContext object as input
- returns a ValidationResult as output

■ **Formatter (Interface)**

- defines a method for formatting form data
- accepts a FormatterContext object as input
- returns a String object as result

■ **class ValidationMethods**

Provides standard validators through its resource GlobalValidators.

Object Path Expressions

Table 103. Object path expressions supported by the Form class

Expression	Description
Form:<ID>, Form:get(<ID>)	In case a form holds form instances, the <ID> means the identifier for the form and a Form object is returned. In case there are no sub forms, the <ID> is the name of the form parameter and a FormParameter object is returned.
Form:Parameter(<ID>)	Returns the form parameter with given id.
Form:SubForm(<ID>)	Returns the sub form with given id.
Form:Elements	Returns an iterator of forms or parameters.
Form:ID	The name of the form (normally an empty string for the root form object).
Form:isSubmitted	Returns true after executing ValidateForm, else false.
Form:isValid	If validation failed or if any error code is explicitly set this will return true, else false. Note this aggregates all sub elements even forms or parameters.
Form:Errors(<ErrorID>)	Returns an iterator of direct sub elements providing the given ErrorID.
Form:FailedValidators	Returns a list of validators that caused a failed validation. This can be used to get additional information about the cause of the failed validation directly from the validator.
Form:InvalidForms	Returns a list of all invalid sub-forms.
Form:InvalidParameters	Returns a list of all invalid parameters.

Table 104. Object path expressions supported by the FormParameter class

Expression	Description
Parameter:<ID>, Parameter:get(<ID>)	Returns the FormField with the given <ID>. If no form fields are explicitly declared (since there is only one field for a parameter) a field named "Field" is automatically created.
Parameter:Fields	Returns an iterator of the fields.

Expression	Description
Parameter:ID	The name of the parameter.
Parameter:isValid	If the validation failed or an error code has explicitly set this will return true, else false.
Parameter:Errors(<ErrorID>)	Returns an iterator of fields providing this ErrorID.
Parameter:Errors	Returns an iterator of ErrorIDs holding this parameter.
Parameter:isError(<ErrorID>)	Returns true if the ErrorID is set for this parameter.
Parameter:QualifiedName	Returns the full qualified name containing form and parameter. If form has a parent also parent ID/ Alias will be part of the name.
Parameter:Value	The converted value.
Parameter:FailedValidator	Returns a list of validators that caused a failed validation. This can be used to get additional information about the cause of the failed validation directly from the validator.
Parameter:InvalidFields	Returns a list of all invalid fields.

Table 105. Object path expressions supported by the FormField class

Expression	Description
Field:ID	The name of the field.
Field:Value	The string representation to be used within ISML. Might be the converted parameter value or the submitted raw form value.
Field:isValid	If the validation failed or an error code has explicitly set this will return true, else false.
Field:Errors	Returns an iterator of ErrorIDs holding this field.
Field:isError(<ErrorID>)	Returns true if the ErrorID is set for this field.
Field:QualifiedName	Returns the full qualified name containing form, parameter and field. (e.g. OrderForm_Discount_Value for a form 'OrderForm' with a parameter of type money 'Discount' and a field 'Value'.
Field:FailedValidator	Returns a list of validators that caused a failed validation. This can be used to get additional information about the cause of the failed validation directly from the validator.

Web Service Development

Web Services

What Are Web Services?

Web services are software modules whose public interfaces and bindings are defined and described using XML. Web services can be discovered by other software systems using a Web service URL. These systems may then interact with the Web services in a manner prescribed by the Web service definition.

Web services enable you to automatically exchange data between two remote software components without any user interaction. Using Web services, developers can make well-defined business processes accessible to any internal or external Web service client. Organizations can communicate data without knowledge of each other's IT-systems behind the firewall. For example, one company can call another service to calculate the shipping costs for a package of a certain weight or size.

Web services are hosted on a server and offer the client a number of predefined functions. To use a Web service, the client must register with the server, call the requested function, and pass on the necessary parameters. After that, a Web service provides the client with the requested XML data. The client can integrate this data into his Web site as if the code came from his own server.

Web Service Technologies

The key concept of Web services is the Service-Oriented Architecture (SOA), which defines a model for accessing applications remotely through well-defined interfaces. Important standards on which Web services are based include

■ SOAP

SOAP is a mechanism for stateless, XML-based, request/response messaging. It is a communication protocol used to transfer data. SOAP is platform- and language-independent, open standard that can be used by various systems, e.g., Apache Axis or Microsoft .NET Server. A SOAP message is an XML document consisting of an envelope and a mandatory body. To perform the communication between server and client, only the SOAP protocol is required. The integration of a Web service, however, requires WSDL.

■ **Web Service Description Language (WSDL)**

WSDL is the XML-based meta language used to describe a Web service. WSDL is the language that UDDI uses. WSDL is the standard that defines how a SOAP message must be formed to use Web services. WSDL is required because this is the formal description of the contract between a service provider and a client.

Web Service and Intershop 7

Intershop 7 provides a complete infrastructure and various tools to both use Web services as a client in Intershop 7 and to publish business processes as Web services for other applications to use. For example, a pipeline can be published as Web service, with the same pipeline integrating other Web services to solve particular business tasks.

Intershop 7 enables you to do the following:

1. Publish Pipelines as Web Services

Requests to the Web service from a client application then automatically trigger the execution of the pipeline. Using Intershop Studio, you can select and publish Web service pipelines without writing a single line of code.

2. Call External Web Services

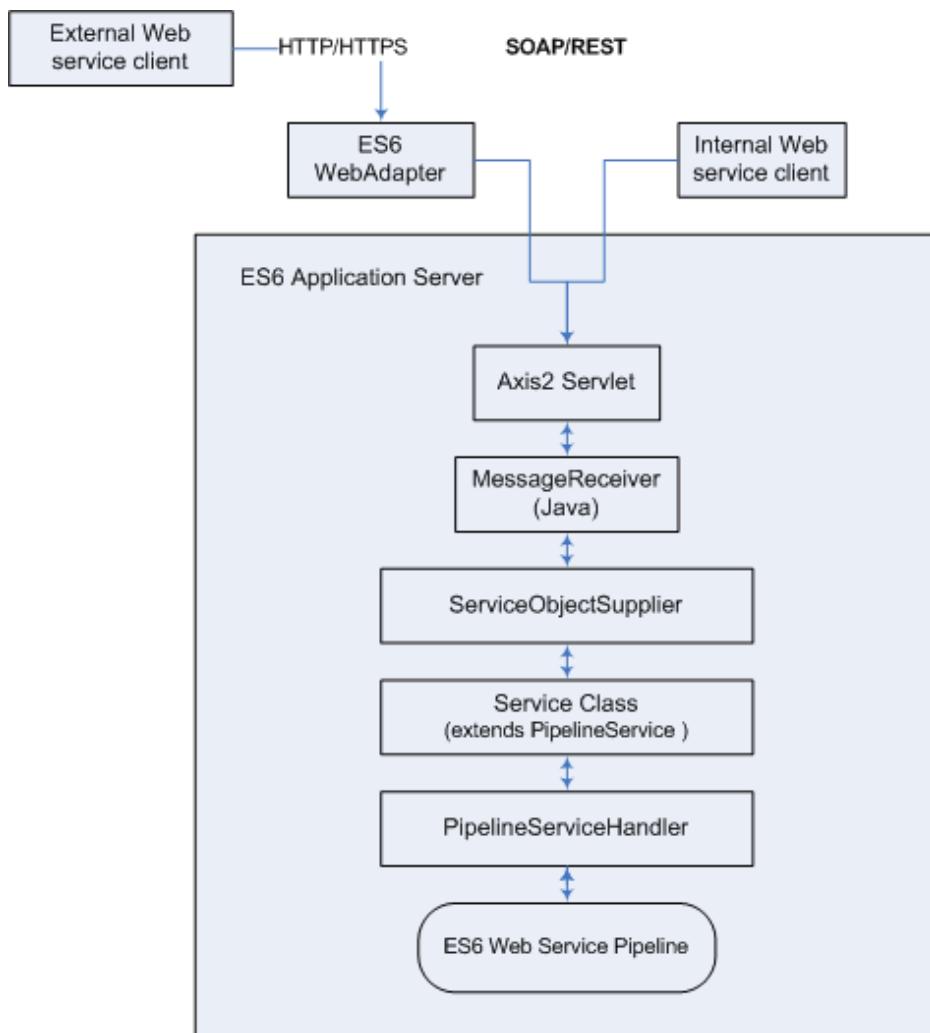
Apart from providing Web services, Intershop 7 applications can easily integrate calls to external Web services. Using Intershop Studio, you can - for example - automatically create the necessary stubs and helper classes to call Web services from pipelets and pipelines.

Technical Infrastructure

The Web service implementation in Intershop 7 is based on the Apache Axis2 tool set.

NOTE: Intershop 7 still includes Apache Axis for compatibility reasons. Intershop, however, recommends to use Apache Axis2 for new Web service projects.

The Web service implementation in Intershop 7 is shown below.

Figure 217. Web Services Integration

At the heart of the Web service integration into Intershop 7 is the Axis2 servlet. Axis2 provides a framework for constructing SOAP processors such as clients, servers, and gateways. In Intershop 7, Axis2 is responsible for dispatching calls to the Web service and generating WSDL files.

Axis2 supports a number of so-called message receivers. Message receivers connect a client's Web service request to a server-side resource that implements a Web service. This way, message receivers allow for implementing Web services on the Java platform using a various technologies like Enterprise Java Beans (EJB) or Java language extensions, e.g., Groovy.

In Intershop 7, Web services are implemented using pipelines. Note that Intershop 7 uses the standard Java message receiver provided by Axis2 instead of implementing a specific pipeline-based message receiver. This way, each pipeline-based Web service is internally represented by an individual Java class, which implements one method for each start node defined in the Web service pipeline.

Upon receiving a request, the Java method - executed by the Axis2 framework - calls the pipeline service handler instance with the arguments delivered by the Web service request. The pipeline service handler instance uses the method's arguments to initialize the pipeline dictionary, executes the requested start node of the Web

service pipeline and determines the return value from the pipeline dictionary after the pipeline execution has finished.

Accessing WSDL Files for Intershop 7 Web Services

Clients can access the WSDL files for Intershop 7 Web services using the following URL:

`http://<host>:<port>/is-bin/INTERSHOP.servlet/WFS/Axis2/<site>/<webservice>?wsdl`

You can test the functionality by checking

`http://<host>:<port>/is-bin/INTERSHOP.servlet/WFS/Axis2/<site>/listServices`

This site includes Web services internally used by Intershop 7.

Figure 218. Example Intershop 7 Web services

Available services

[AuthorInfoService](#)

Service EPR : AuthorInfoService

Service Description : This service provides information about authors and their books (Supported standards: WS-Addressing)

Service Status : Active

Available Operations

- `getAuthorsBibliography`

[EchoService](#)

Service EPR : EchoService

Service Description : This service is a simple echo service that returns the string argument passed as operation parameter. (Supported standards: WS-Addressing, WS-Security)

Service Status : Active

Available Operations

Data Types

Web services typically take a series of arguments and return a value. Because arguments and return values must always be converted from XML to Java (arguments) and back to XML (return values), respectively, only a subset of data types is allowed. The following data types can be sent as parameters and received as results:

NOTE: You can also use arrays based on these data types.

Primitive Types

The following table lists the primitive data types. See the Axis User Guide for details on data type mapping.

Table 106. Primitive Data Types

Java Type	WSDL Type
<code>boolean</code>	<code>xsd:boolean</code>
<code>byte</code>	<code>xsd:byte</code>

Java Type	WSDL Type
short	xsd:short
int	xsd:int
long	xsd:long
float	xsd:float
double	xsd:double
char	xsd:string

Java Object Types

The following table lists the Java Object types. See the [Axis User Guide](#) for details on data type mapping.

Table 107. Java Object Types

Java Type	WSDL Type
java.lang.Boolean	soapenc:boolean
java.lang.Byte	soapenc:byte
java.lang.Short	soapenc:short
java.lang.Integer	soapenc:int
java.lang.Long	soapenc:long
java.lang.Float	soapenc:float
java.lang.Double	soapenc:double
java.lang.String	xsd:string
java.math.BigDecimal	xsd:decimal
java.math.BigInteger	xsd:integer
java.util.Calendar	xsd:dateTime
javax.xml.namespace.QName	xsd:QName

JavaBeans

Java Beans are data containers used to wrap data types not covered by default. JavaBeans implement the `java.io.Serializable` interface, have get and set methods for each attribute, and a public constructor. JavaBeans can be modeled in a Rational Rose model and generated automatically using jGen. The possible combination of data types allows you to model complex data structures.

CAUTION: Do not use any Java-specific collection type that SOAP cannot map to XML (e.g., `java.util.Hashtable`). Use arrays instead.

A client retrieves all necessary information about data types from the WSDL file. This is why it is not possible to implement additional functionality (e.g., parameter checking, content validation, etc.) in a JavaBean.

Preparing a Web Service

Planning the Web Service

Plan your Web service carefully before you create pipelets, data objects and the actual pipeline. Planning a Web service includes the following issues:

■ **Web Service Methods**

All Web service methods need to be defined clearly before starting the implementation, including the exact method signature. For each Web service method, a corresponding pipeline start node with identical name must exist. For example, the `ProductPriceWebService` example has the method `getProductPrice()`. Hence, a pipeline is created (`ProductPriceWebService`) with start node `GetProductPrice`.

■ **Data required by the Web service**

Data required by the Web service (passed as parameter when calling the Web service) are put in the pipeline dictionary before running the pipeline. If your Web service needs data of a type not covered by default, an appropriate JavaBean has to be implemented as data container. See *Data Types* for valid data types.

■ **Data Returned by the Pipeline**

Data returned to the client must be in the pipeline dictionary after pipeline execution. This data is then read from the pipeline dictionary and returned to the client. If your Web service returns data of a type not covered by default, an appropriate JavaBean has to be implemented as data container. See *Data Types* for valid data types.

Implementing Web Service Functionality

In general, implementing a pipeline-based Web service involves the following tasks:

■ **Prepare pipeline elements**

Web service pipelines are standard pipelines in the sense that their functionality is implemented by one or more pipelets and/or by sub-pipelines called. How the pipeline is designed, i.e., which pipelets the pipeline contains and whether or not sub-pipelines are called, depends on the functionality necessary to serve the Web service method and is up to the developer to decide. When preparing pipeline elements, make sure the pipeline reads Web service parameters from the pipeline dictionary and puts the return type into the pipeline dictionary before finishing execution.

NOTE: Make sure to use strict pipelines, i.e., the start parameters and return values to be used by the Web service must be declared with the corresponding start and end node(s). For details, see [here](#).

■ **Implement data objects as JavaBeans**

Data objects that do not belong to range of data type supported out-of-the-box have to be implemented as JavaBeans. These data objects can then be used as arguments for the Web service or as return value.

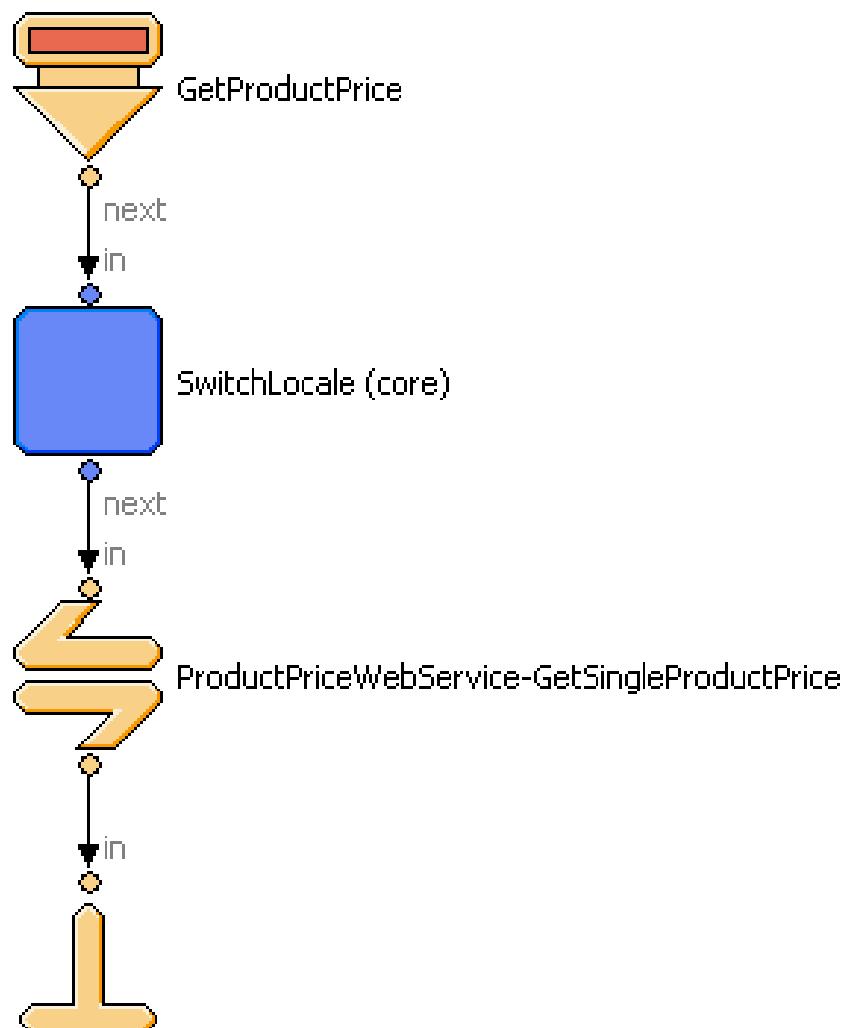
NOTE: When dealing with collections of primitive or complex types as parameters or return values, note that Java collections are not supported. Use arrays instead.

■ Assemble the Web service pipeline

Having prepared all pipeline elements and implemented data objects (if necessary), the actual Web service pipeline can be assembled in the Pipeline Editor. In the example, the `ProductPriceWebService` pipeline basically hands over to a sub-pipeline. The `ProductPriceWebService` pipeline has a `GetProductPrice` start node, which identifies the pipeline as the one serving the Web service method `getProductPrice()`.

NOTE: Make sure to use private pipelines, i.e., with the call mode set "private" at the start node, for creating Web services. For details, see here.

Figure 219. ProductPriceWebService example



NOTE: Make sure to define the required start parameters and return values upon pipeline creation. This data cannot be configured when publishing the Web service pipeline.

Publishing a Web Service Pipeline

This section describes how to publish pipelines as Axis2-based Web services. The development of a Web service is integrated with a single wizard. That is, the wizard guides you completely through the definition and configuration of the Web service.

Basically, the wizard includes the following steps:

- configuring the service implementation and environment, and
- configuring the operations to be made available.

NOTE: The wizard assumes a strict pipeline, i.e., the start parameters and return values to be used by the Web service must be declared with the corresponding start and end node(s) upon pipeline creation, see here.

To invoke the Web service wizard:

1. **In the Package Explorer, drill down to the intended pipeline.**
2. **Right-click the pipeline.**

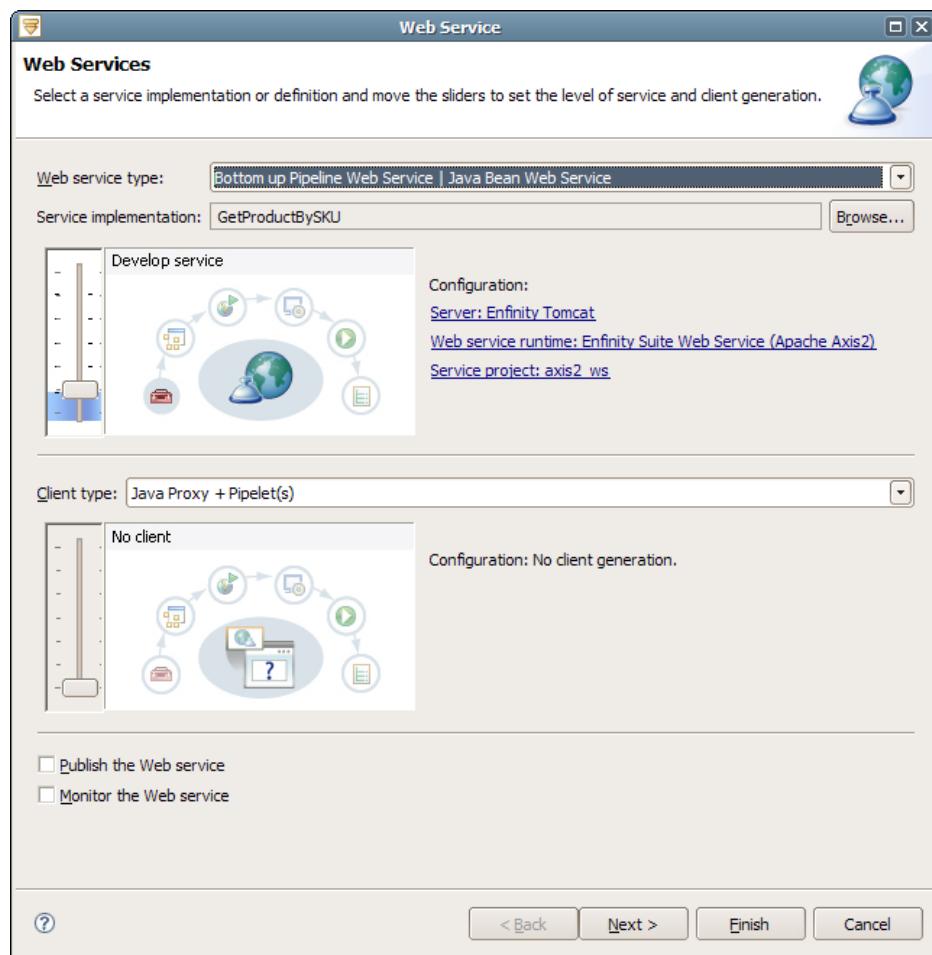
This opens the context menu.

3. **From the context menu, select Web Services | Create Web service.**

This opens the Web Service wizard as shown in *Figure 220, “Specifying basic Web service settings”*.

Configuring the Web Service

On the first wizard page, you specify the Web service implementation, the service type and the service environment.

Figure 220. Specifying basic Web service settings

1. Edit the Web service settings as necessary.

NOTE: When started from the pipeline context menu, the wizard automatically fills all fields with the needed values.

The following options are available:

Table 108. Basic Web service settings

Option	Description
Web Service Type	For Intershop 7 pipelines to be published as Web services, keep the default setting Bottom Up Pipeline Web Service \vert Java Bean Web Service .
Service Implementation	Specifies the Web service implementor (pipeline or Java bean). For Intershop 7 pipelines to be published as Web services, select Pipeline. When the wizard is started from the pipeline context menu, the intended pipeline is already selected.
Development Stage (slider)	Specifies the development stage. To create a pipeline-based Web service, the default stage Develop is sufficient.

Option	Description
Configuration	Includes the following properties: a) Server - specifies the server environment to be used, b) Web Service Runtime - specifies the Web service runtime environment, c) Service Project - specifies the client project environment

NOTE: When creating a pipeline Web service, the Web service client section as well as the options for publishing and monitoring the Web service are not functional.

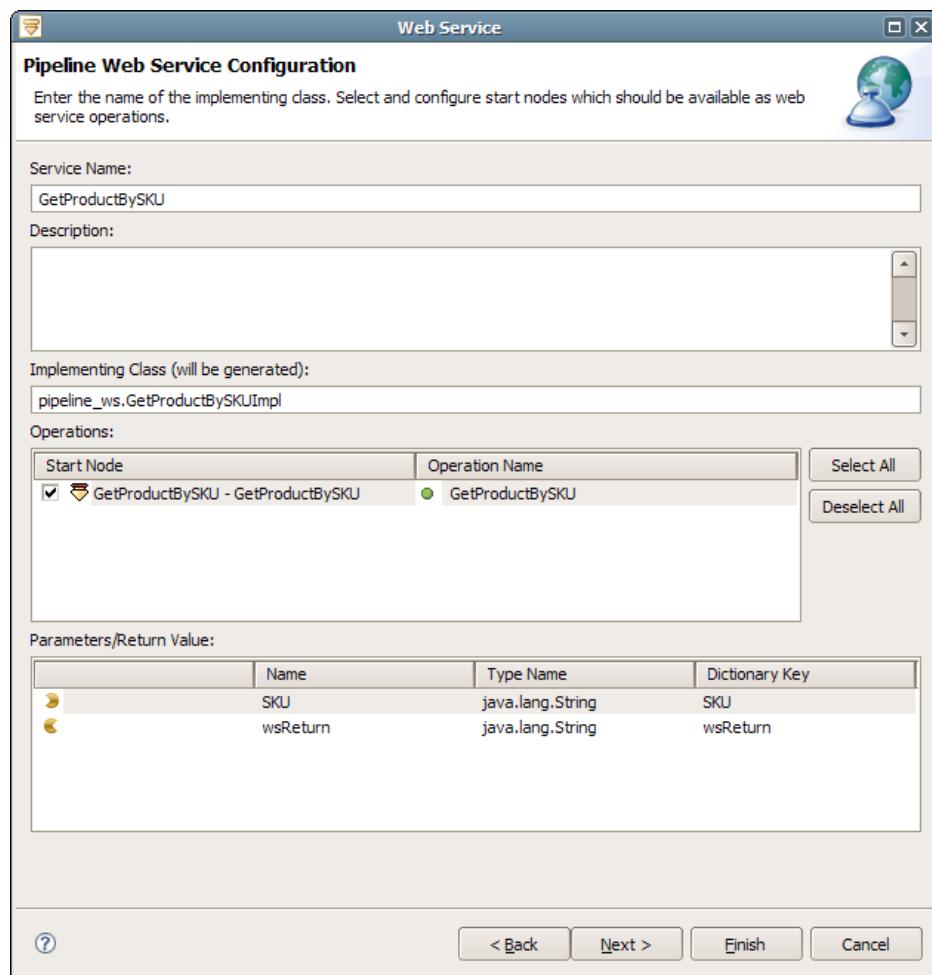
2. Click Next to continue.

Your settings are saved, and the Web Service Operations page is displayed.

Configuring the Web Service Operations

On the Web Service Operations page, you specify the implementing class (which will automatically be generated) and define the pipeline start node(s) to be available as Web service operation(s).

NOTE: Within Intershop 7, Web services are represented by a class file, which is used internally. This class, however, is not involved upon developing the Web service.

Figure 221. Specifying the Web service operations

1. In the Service Name and Description fields, edit the values as necessary.

The service display name is required, the description is optional.

2. In the Implementing Class field, edit the package and class name as necessary.

3. In the Operations section, select the start node(s) to be published as Web service operations.

Select the checkbox(es) for the start node(s) to be included.

NOTE: For each start node, an individual Web service operation will be created.

4. In the Parameters/Return Values section, edit the corresponding dictionary keys as necessary.

For each selected node in the Operations section, the parameters/return values and the corresponding dictionary keys are displayed and edited separately.

5. Click Finish to complete the Web service creation.

Optionally, click Next to specify whether to launch the Web Service Explorer upon completing the wizard.

Accessing the Web Service

Before clients can use the new Web service, you have to:

- build the cartridge containing the Web service (see *Overview of Build Process*)
- register the cartridge (see *Register Cartridges*), and
- assign the cartridge to an app (see *Adding Cartridges To Apps*).

To check whether the new Web service is available, point your browser to the following URL, replacing <site> with the name of the site to which the cartridge containing the Web service has been assigned:

```
http://<host>:<port>/is-bin/INTERSHOP.servlet/WFS/Axis2/<site>/listServices
```

This should return a list as shown in the example below, including your new Web service.

Figure 222. Example Intershop 7 Web services

Available services

AuthorInfoService

Service EPR : AuthorInfoService

Service Description : This service provides information about authors and their books (Supported standards: WS-Addressing)

Service Status : Active
Available Operations

- getAuthorsBibliography

EchoService

Service EPR : EchoService

Service Description : This service is a simple echo service that returns the string argument passed as operation parameter. (Supported standards: WS-Addressing, WS-Security)

Service Status : Active
Available Operations

Creating a Web Service Client

This section describes how to create Axis2-based Web service clients. The development of Axis2-based Web service clients is completely integrated with a single wizard. That is, the wizard guides you through the Java stub class and pipelet generation.

NOTE: The pipelet generation is based on the previously created stub classes.

The generated Java stub classes can be used in custom client applications that invoke Web service methods from Java code, whereas Web service pipelets make it possible to directly integrate Web service method calls into pipelines.

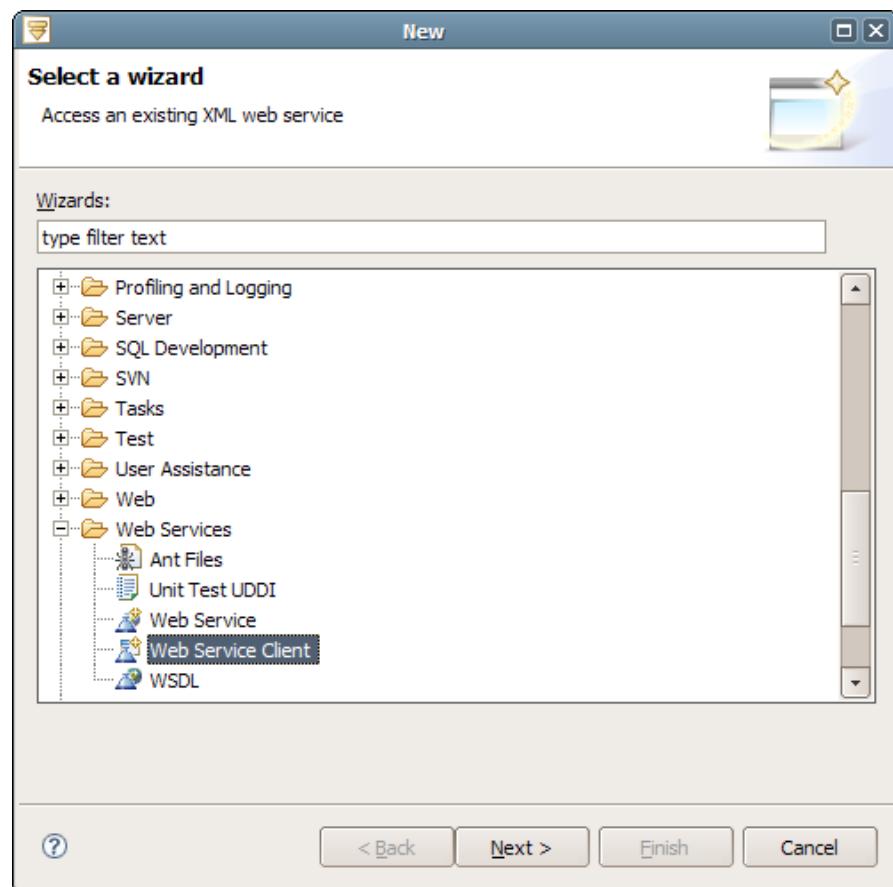
Basically, the wizard includes the following steps:

- specifying the Web service definition location and the client's request scheme type,
- specifying the Java stub class generation options, and
- specifying the pipelet generation options.

To invoke the Web service client wizard:

1. **In the Package Explorer, right-click the cartridge project.**
This opens the context menu.
2. **From the context menu, select New | Other.**
This opens the Wizard Selection window.
3. **In the the Wizard Selection window, expand Web Services and select Web Service Client.**

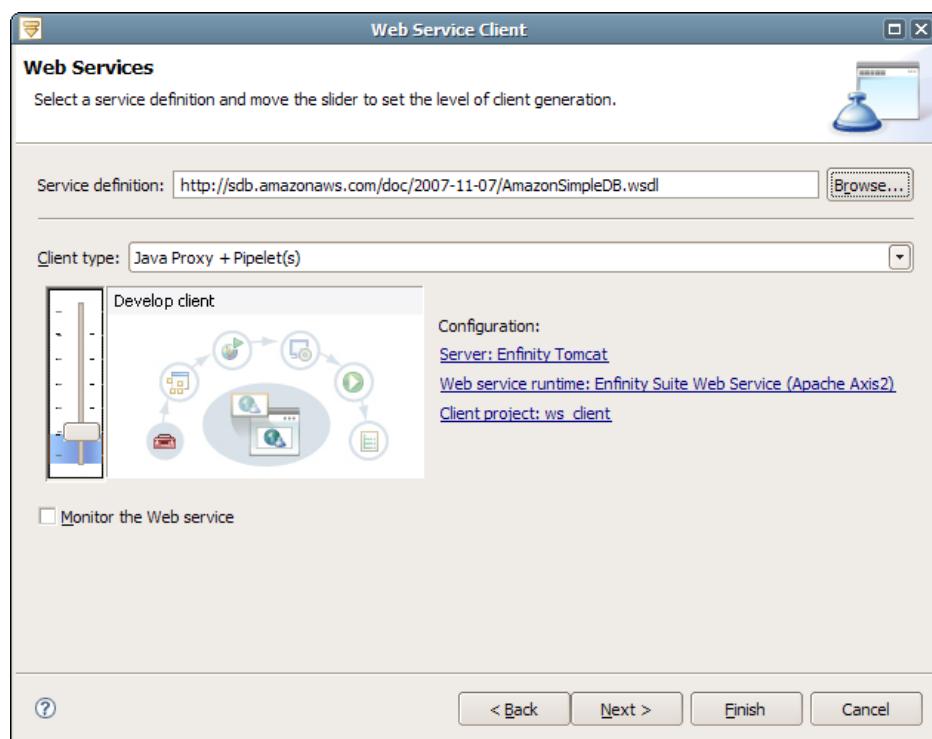
Figure 223. Selecting the Web Service Client wizard



The Web Service Client wizard is displayed.

Configuring the Web Service Client

On the first wizard page, you specify the Web service definition location (which can be a Web URL or a local file), the client's request scheme type and the client environment.

Figure 224. Specifying basic Web service client settings

1. In the Service Definition field, specify the URL of the Web service WSDL file.

Click Browse to open a dialog for browsing and selecting a WSDL file from local resources.

2. In the Client Type field, select the required client type.

Choose either "Java Proxy + Pipelet(s)" to create the corresponding Java stub classes and Intershop 7 pipelets, or "Java Proxy" to create only the Java stub classes.

3. Set the Development Stage slider to "Develop".

To generate Java client classes and Intershop 7 pipelets, the stage Develop is sufficient.

4. If required, edit the configuration as necessary.

Click the option to be edited to open the corresponding detail dialog.

The following options are available:

Table 109. Web service client configuration options

Option	Description
Server	Specifies the server environment to be used.
Web Service Runtime	Specifies the Web service runtime environment (with "Java Proxy + Pipelet(s)" selected as client type, only Axis2 is available).
Client Project	Specifies the client project environment.

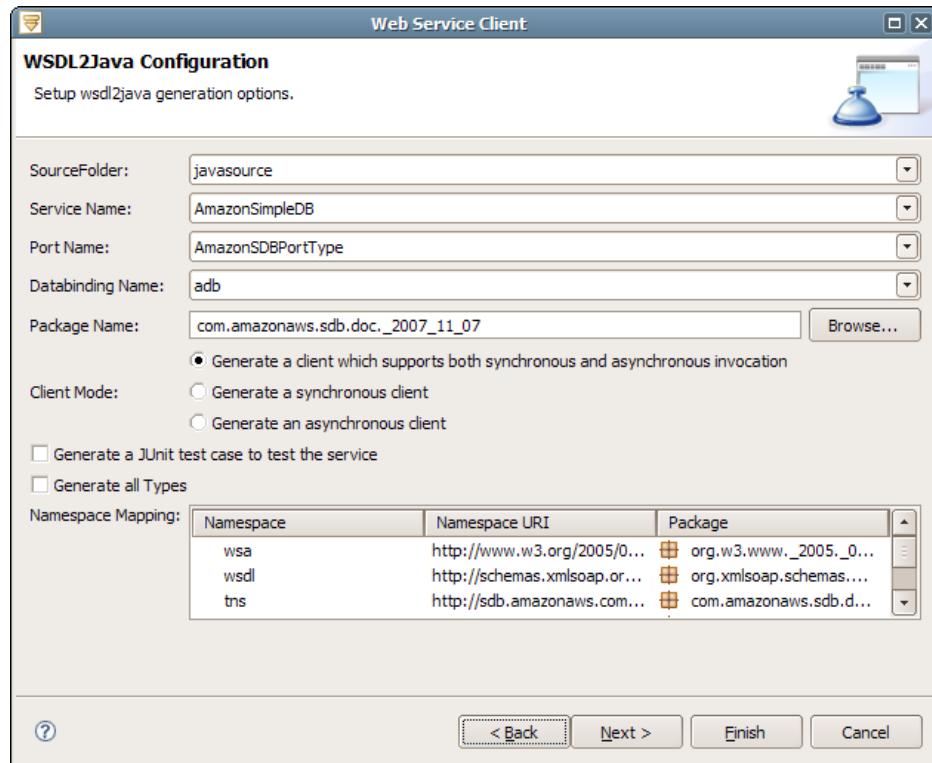
5. Click Next to continue.

Your settings are saved, and the Java Class Configuration page is displayed.

Configuring the Java Stub Class Generation

On the Java Class Configuration page, you specify the Java stub class generation options.

Figure 225. Configuring the Java stub class generation



1. Edit the Java stub class generation options as necessary.

All fields are automatically filled with default values.

The following options are available:

Table 110. Java stub class generation options

Option	Description
Source Folder	Specifies the root location of the Java sources to be generated. (The absolute path is a concatenation of this source directory and the package name.)
Service Name	Specifies the Web service name (taken from the WSDL file).
Port Name	Specifies the Web service port name (taken from the WSDL file).
Databinding Name	Specifies the Axis2 data binding name. Intershop recommends to keep the default value adb (Axis Data Binding).

Option	Description
Package Name	Specifies the location of the stub classes in the source folder.
Client Mode (radio buttons)	Specifies whether the client supports a synchronous, asynchronous or both synchronous and asynchronous (default) request scheme.
Namespace Mapping	Specifies the packages that include the types associated with the namespaces as defined by the WSDL file.
Generate JUnit Test Case	Specifies whether to create a JUnit test case together with the Java stub classes (optional).
Generate All Types	Specifies whether to create all associated types, including those that are defined in the WSDL but remain unused (optional).

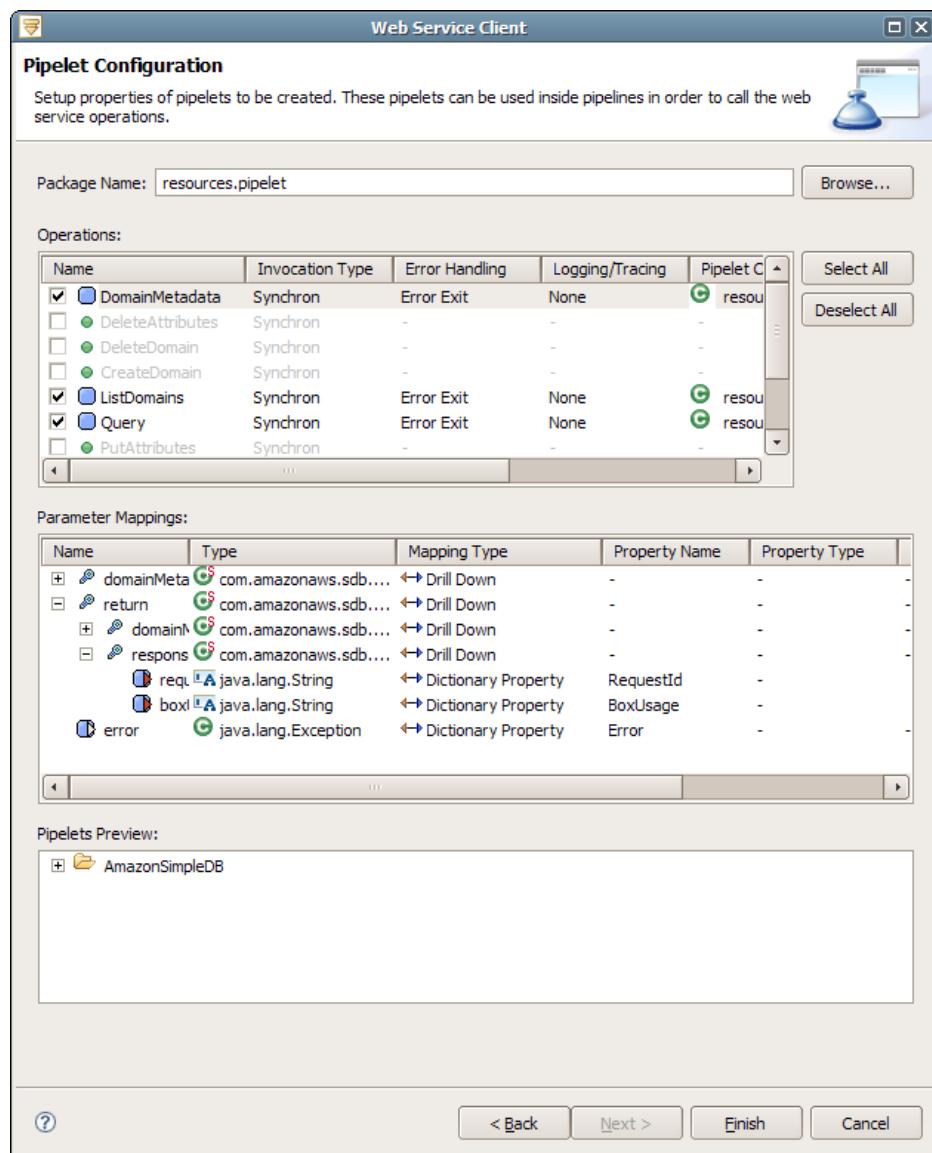
2. Click Next to continue.

Your settings are saved, and the Pipelet Configuration page is displayed.

Configuring the Pipelet Generation

On the Pipelet Configuration page, you specify the setup properties of the pipelets to be created.

NOTE: The pipelet generation is based on the previously created stub classes.

Figure 226. Configuring the pipelet setup properties

- In the Package Name field, specify the location of the pipelet(s) to be generated.**

Click Browse to open a dialog for browsing and selecting a package from local resources.

- In the Operations section, select the Web service methods to be integrated in the pipelet(s).**

Select the checkbox(es) for the method(s) to be included.

NOTE: For each Web service method, an individual pipelet will be generated.

- In the Parameter Mappings section, edit the parameter-dictionary mappings as necessary.**

For each selected method in the Operations section, the parameter mappings and the according dictionary properties are displayed and edited separately.

The Pipelets Preview section displays the pipelets to be generated.

4. Click Finish to complete the Web service client creation.

All corresponding Java stub classes and pipelets are generated as configured in the wizard.

Intershop Studio Reference

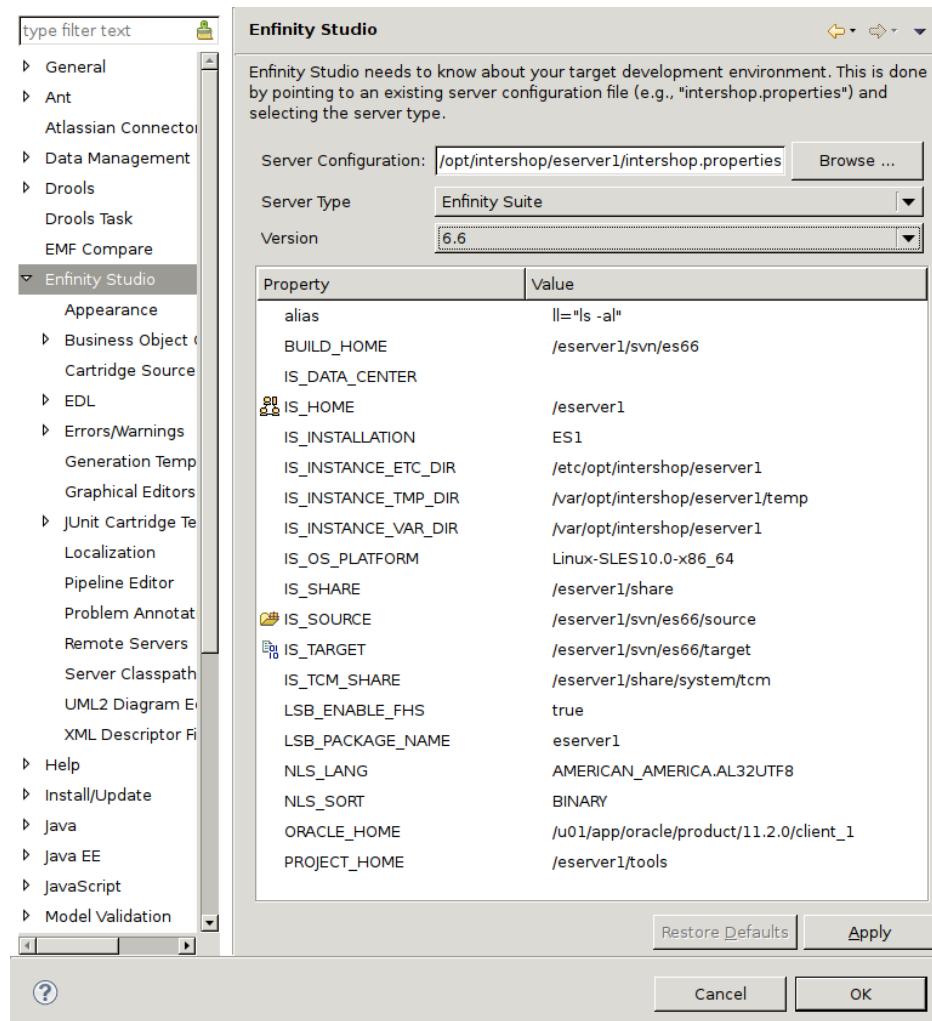
Preferences

Intershop Studio adds a few new settings to the standard set of Eclipse preferences.

Intershop 7 Development Environment

This page is used to specify the installation directory of your Intershop 7 system. Intershop Studio uses this information to locate cartridges already deployed on your system and to find the `intershop.properties` file from which it reads the values for `IS_SOURCE`, `IS_TARGET`, `IS_HOME`, `IS_SHARE` and `PROJECT_HOME` among others. This information is needed to properly set up the development environment for your cartridge projects.

Here is what the Intershop 7 Development Environment preference page looks like:

Figure 227. Intershop 7 Development Environment Preference Page

Appearance Preferences

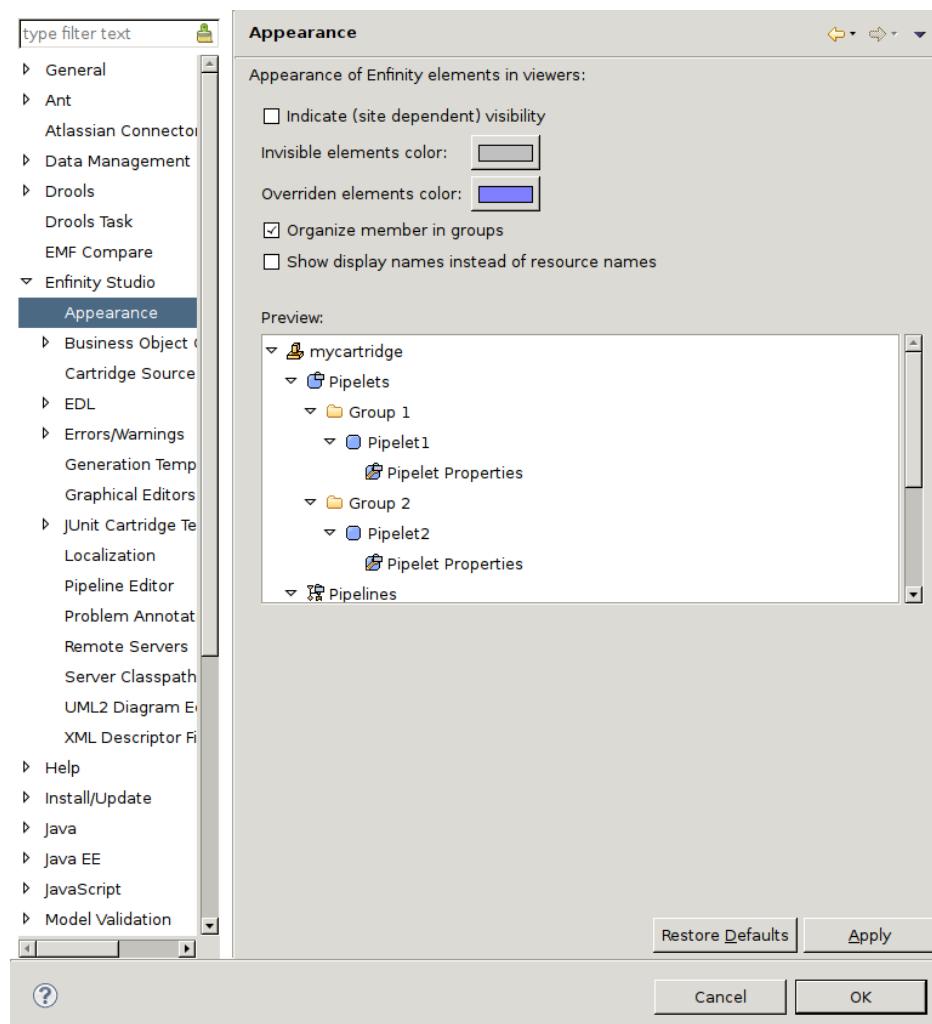
This page is used to define certain aspects of how information is presented in Intershop Studio viewers (such as the Cartridge Explorer or the Outline View). For example, when deselecting "Show Pipeline Members", individual pipeline nodes are not displayed anymore in the Cartridge Explorer or Outline View.

In tree views and element selection dialogs, Intershop 7 elements can be displayed with colored labels, depending on their type. To switch on/off this feature, select the preference General | Appearance | Enable colored labels. To change the colors, edit the preference in General | Colors and Fonts | Intershop 7 Element Labels.

You can also define special colors used to display elements that are invisible or overridden. Overridden elements have a corresponding element which is higher in the lookup hierarchy.

NOTE: For these settings to become effective, you have to define a site context. See here for details.

This is what the Appearance preference page looks like:

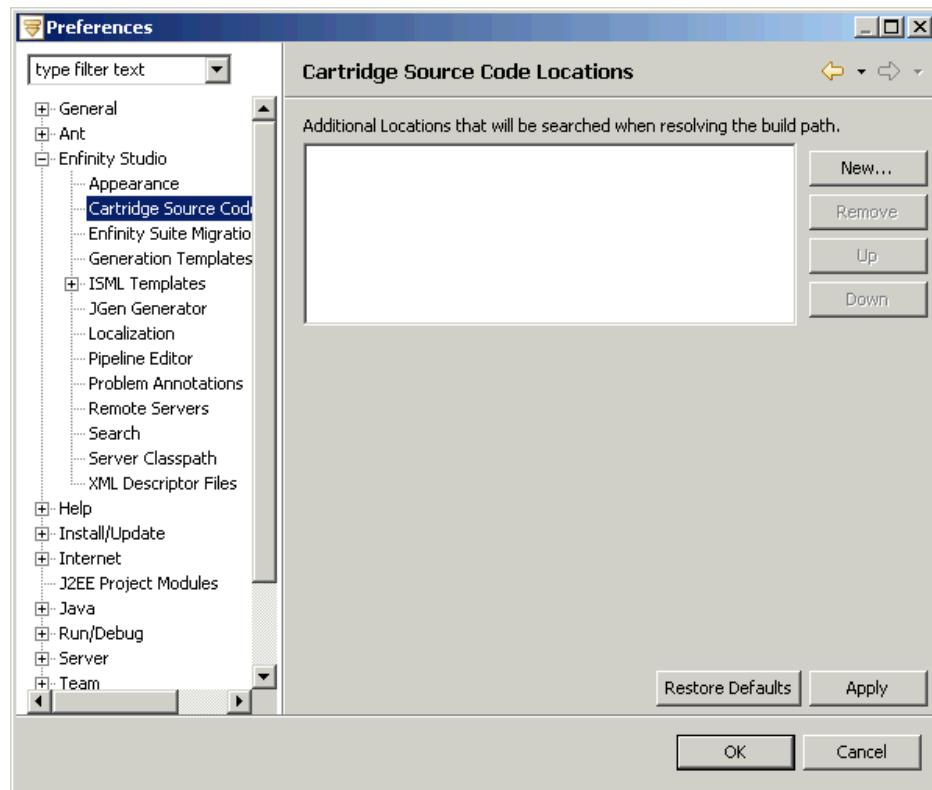
Figure 228. Intershop 7 Development Environment Preference Page

Cartridge Source Code Locations

This page is used to define source code locations for cartridges outside IS_SOURCE. When double-clicking on a *.class file in the Package Explorer, Intershop Studio searches the additional locations and opens the respective Java file (if found) in the Java editor.

For example, suppose you have the source code of cartridge bc_pricing available, stored under c:\temp\source\bc_pricing\javasource. In this case, you would have to provide the path c:\temp\source as cartridge source code location.

Here is what the Additional Source Code Location preference page looks like.

Figure 229. Additional Source Code Location Page

ISML Template Preferences

These page is used to define general settings for the Template Editor.

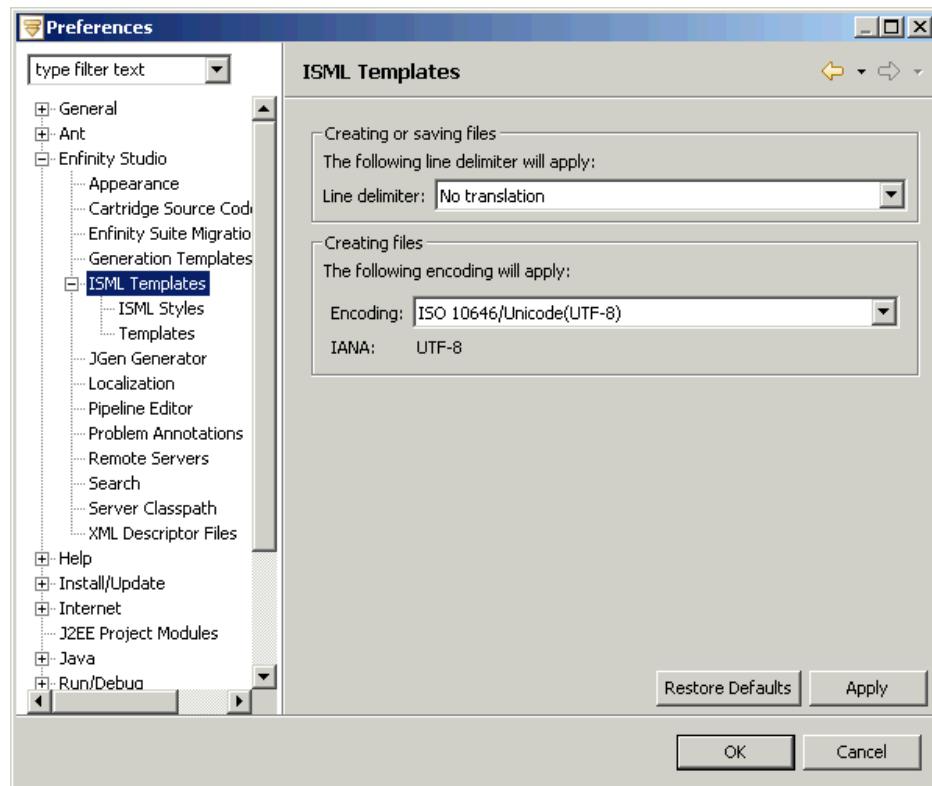
Table 111. Template Editor Options

Option	Description
Line Delimiter	This setting determines whether template files are saved using a line feed (UNIX), a carriage return (MAC) or both a line feed and a carriage return (WINDOWS) at the end of each line. When selecting No Translation (which is the default setting), the encoding appropriate for the current platform is used.
Encoding	This setting determines the character encoding used when saving a template file. The default encoding is UTF-8.
ISML Source	This panel is used to define styles rules for automatic code insertion via content assist. For example, if your template use all upper case for ISML tags, you can force content assist to insert code with all upper case as well. See <i>Styleguide</i> for recommendations regarding the style of ISML templates.
ISML Styles	This panel is used to define details for the syntax highlighting in the Template Editor.

Option	Description
Templates	Via this panel, you can import pre-defined templates to be used in template development, for example, to make available standardized code blocks or comments.

Here is what the Template Preferences page looks like.

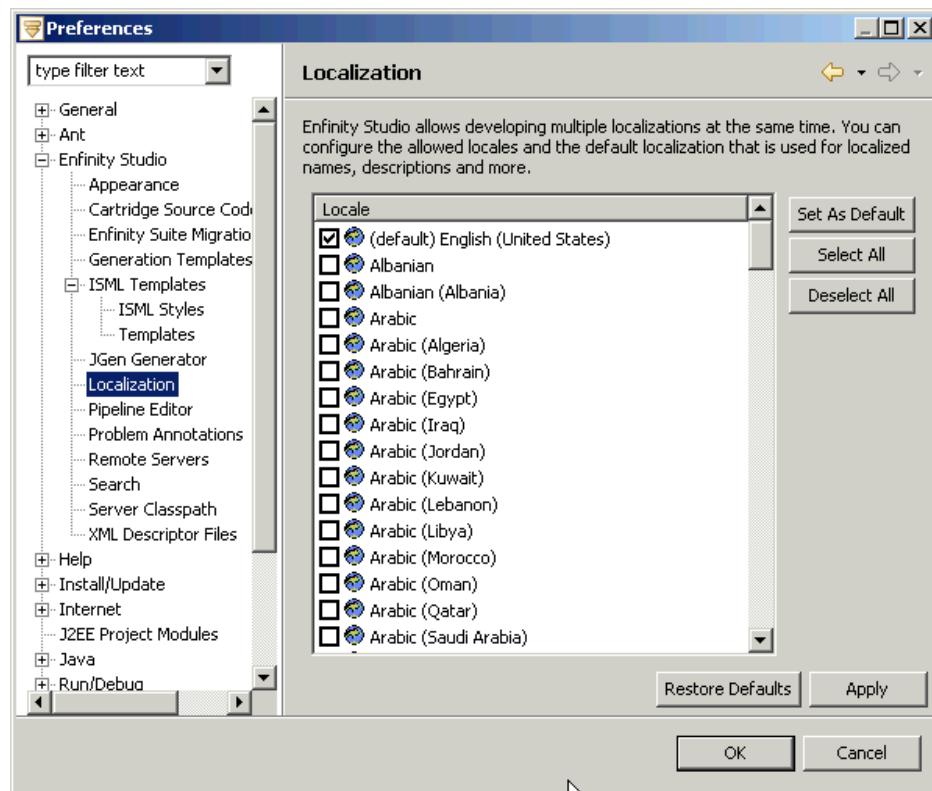
Figure 230. Template Editor Preference Page



Localization Preferences

This page is used to specify the locales your cartridges will support by default. This setting affects, for example, pipelet and pipeline description files or folders available for templates.

Here is what the Localization preference page looks like.

Figure 231. Localization Preference Page

Pipeline Editor Preferences

This page is used to define the following general settings for the Pipeline Editor:

■ General Tab

- Automatic Layout Transitions
Automatically applies a layout to transitions using right-angles.
- Automatically Set Orientation of Nodes

■ Layout Tab

These settings determine the default zoom level, and whether a grid is used in the Pipeline Editor or not. The grid is designed to facilitate pipeline design.

■ Nodes Tab

- Enforce Join Node Usage

This setting determines compatibility of Intershop Studio pipeline representations with the eMC-based Visual Pipeline Manager. If turned off, it is possible to have two or more ingoing transitions for a node, alternatively to using join nodes. Note that the eMC-based Visual Pipeline Manager cannot display such a representation. If turned on, a join node is inserted automatically, which makes the representation compatible with the eMC-based VPM.

■ Show Join Tool in Palette

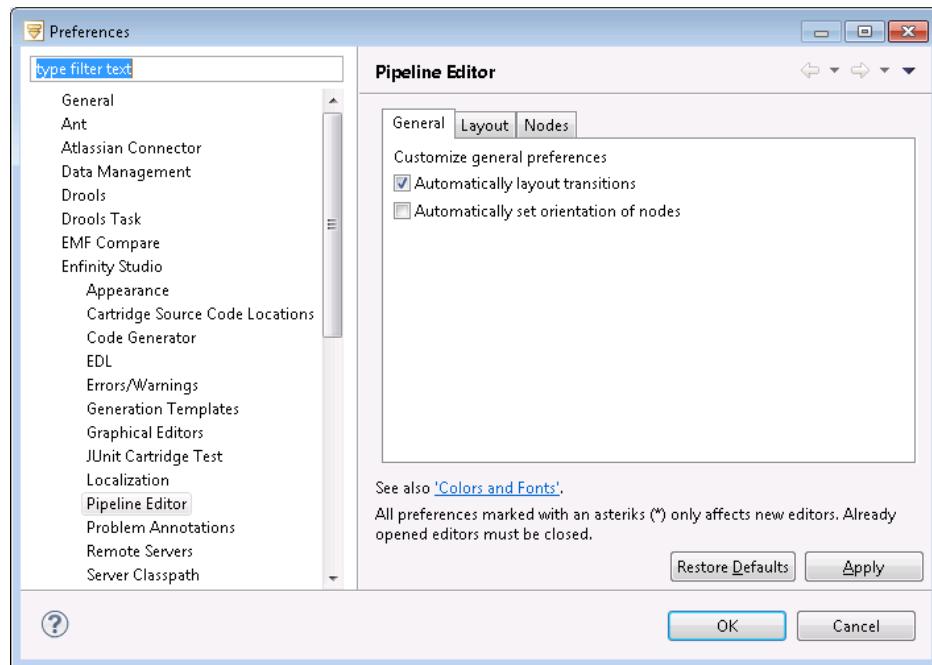
- Show default connector labels

This setting determines whether Intershop Studio displays default labels on connectors (e.g., in, out, do, next).

- Show triangles around nodes on hover

If activated, Intershop Studio displays blue triangles around a node on hover.

Figure 232. Pipeline Editor Preference Page



Problem Annotation Preferences

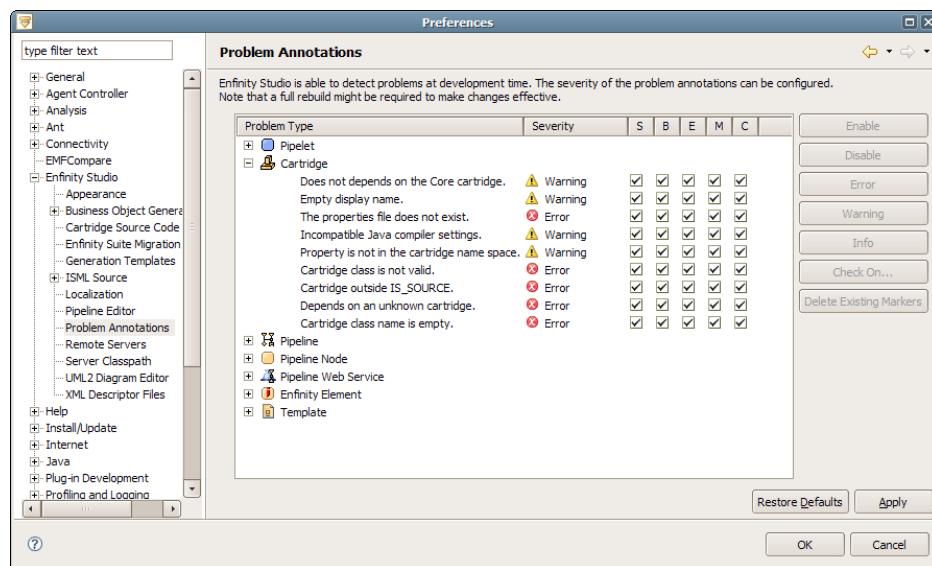
Intershop Studio can check cartridge resources and report errors and warnings in the Problems View. The Problem Annotation page is used to define the types of problems that will be reported when checking resources, and the respective severity levels.

NOTE: Severity levels can be used to define appropriate Problems View filters. See here for details.

Problem checks are available for the following items: pipelets, cartridges, pipelines, pipeline nodes, pipeline Web services, Intershop 7 elements, and templates.

For each possible problem, you can:

- enable or disable the problem check,
- specify a severity level (error, warning, info),
- define a scope, i.e., specify when the resource is to be checked.

Figure 233. Problem Annotation Preference Page

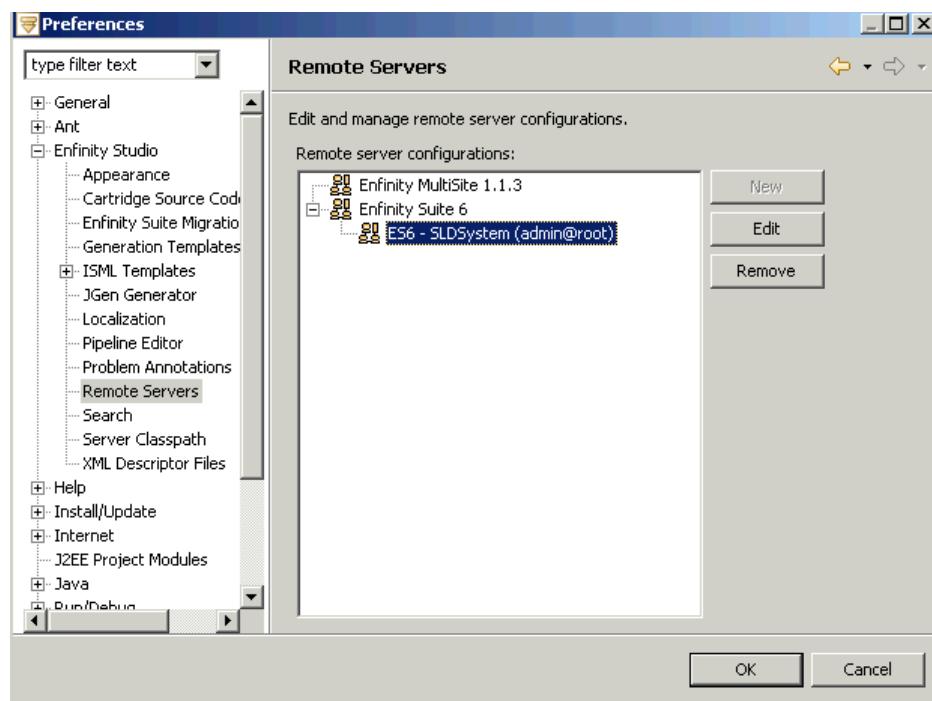
The following table explains the available scope check boxes.

Table 112. Problem Check Scopes

Title	Scope
S	Checks item upon saving.
B	Checks item during automatic or forced builds.
E	Checks item while being edited in its corresponding editor.
M	Checks item when invoked manually (from the context menu).
C	Checks item when invoked from a command line checker application.

Remote Server Preferences

For activities such as pipeline debugging, reload of pipelets and pipelines, or remote editing, Intershop Studio uses connections to remote servers. This page is used to set the necessary connection parameters. See *Remote Server Configuration* for a description of the individual parameters involved.

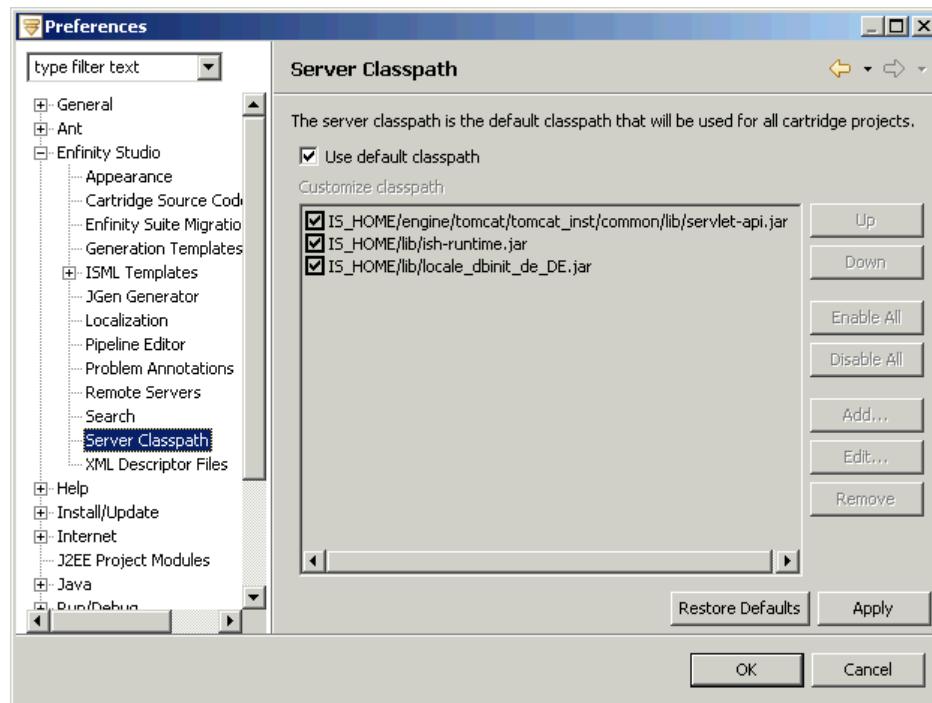
Figure 234. Remote Server Preference Page

Server Classpath Preferences

The server classpath provides the default set of libraries included with the Intershop 7 Server Libraries classpath container, such as EJB container (PowerTier) and servlet engine jar files. The server classpath settings apply to all new cartridge projects. The settings determine the selection of server libraries available to all cartridge projects. However, you can enable or disable individual server libraries individually for each project by modifying the respective classpath container in the project's Java build path.

See *Managing Classpath Settings* for information on classpath concepts in Intershop Studio and *Set Server Classpath* for information on how to set the server classpath.

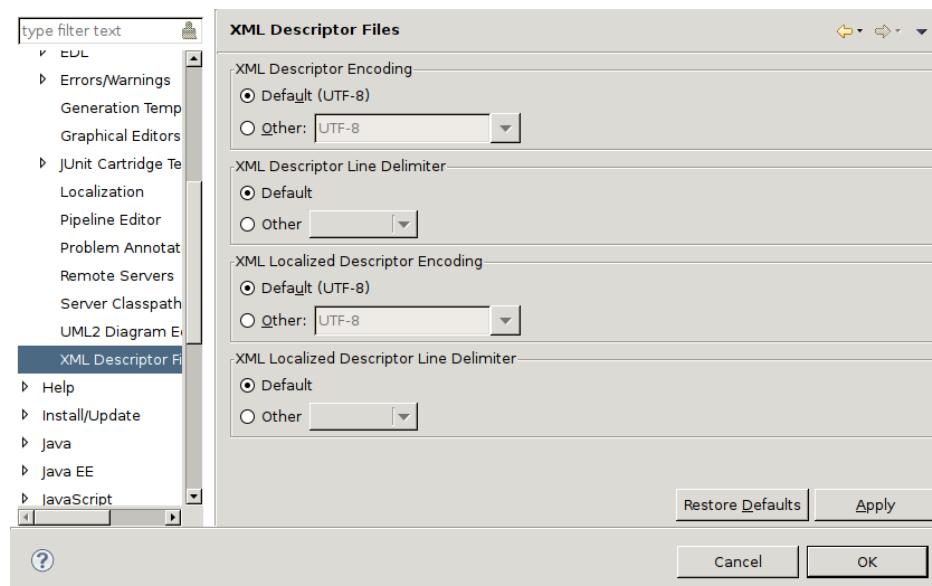
Here is what the Server Classpath preference page looks like.

Figure 235. Server Classpath Preference Page

XML Descriptor Files Preferences

This setting determines the encoding used for XML descriptor files of pipelines and pipelets. The default encoding is UTF-8.

Here is what the XML Descriptor File preference page looks like.

Figure 236. XML Descriptor File Preference Page

Workbench and Perspectives

Workbench, Views and Editors

Intershop Studio opens with a single window called the Workbench. You can open more than one Workbench window at one time. To open another Workbench window, select the Window | New Window menu from the Workbench's menu bar.

The visual components displayed inside the workbench are views and editors. Views are used to navigate a hierarchy of information, e.g. cartridges and their resources. If you double-click a resource in the navigation tree, the tree either expands or, if the resource is editable, the resource opens in an editor. Views are also used to display properties for the active editor. Modifications made in a view are saved immediately. Only one instance of a particular view type (e.g. the Cartridge Explorer) may exist within a Workbench window.

Editors are used to display and modify cartridge resources, e.g. Java classes and pipelines. Modifications made in an editor must be saved manually. Multiple instances of an editor type may exist within a Workbench window.

Intershop Studio uses many of the standard views and editors provided by Eclipse. At the same time, Intershop Studio is equipped with new editors and views specifically designed to facilitate development tasks.

New editors include:

- **Pipeline Editor**

See *Visual Pipeline Editor* for details.

- **Template Editor**

See *Visual Pipeline Editor* for details.

- **EDL model editor**

See for *EDL Model Editor* details.

- **Pagelet Editor**

See for *Pagelet Model Editor* details.

- **Query**

See for *Query Editor* details.

New Intershop 7 views include:

- **Cartridge Explorer**

See *The Cartridge Explorer* for details.

- **Pipeline View, Pipelet View and Template View**

See *Pipelet, Pipeline and Template Views* for details.

- **Pagelet View**

See *Pagelet View* for details.

- **Query View**

See *Query View* for details.

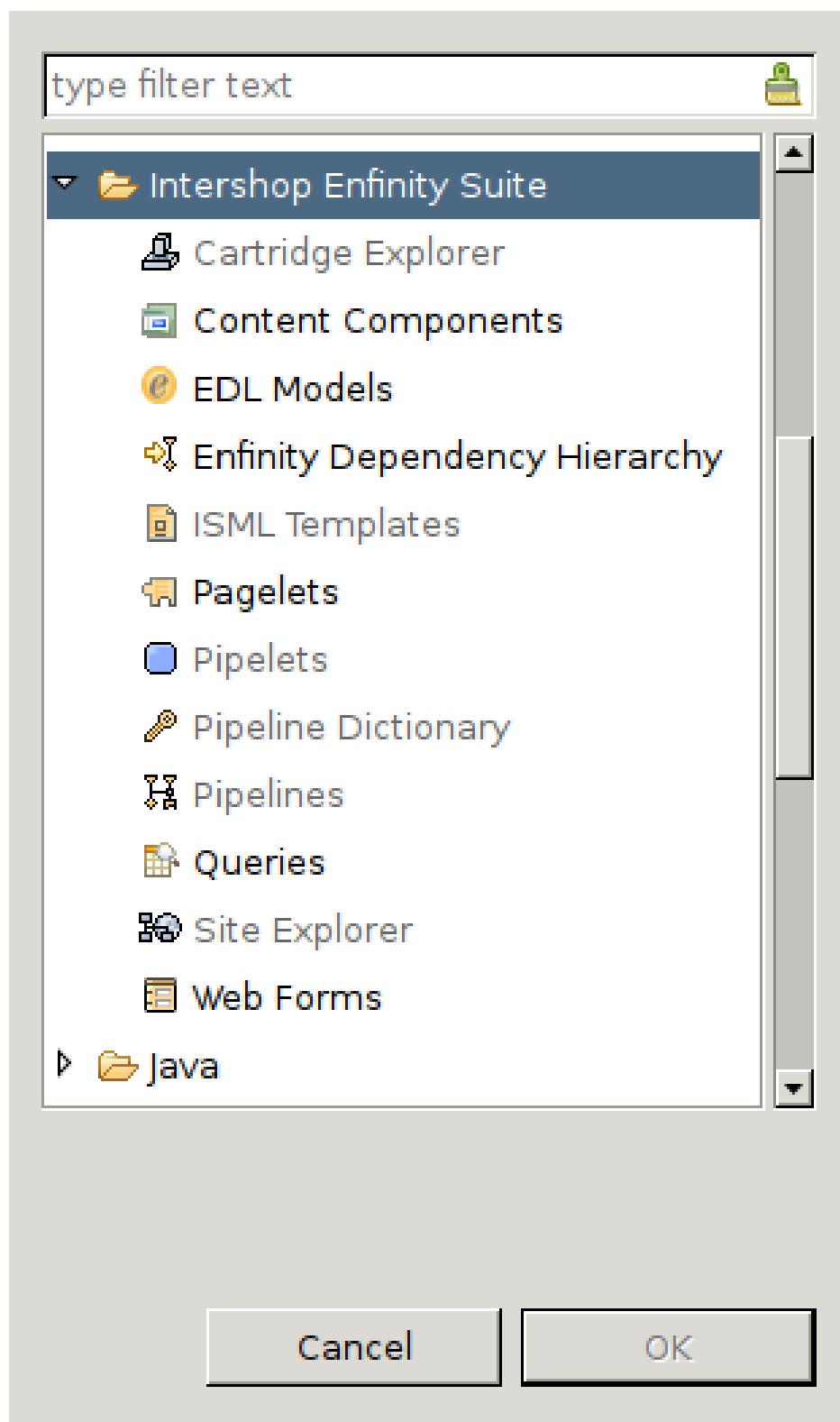
■ Pipeline Dictionary View

See *The Pipeline Dictionary View* for details.

■ Intershop 7 Dependency View

See *Intershop 7 Dependency Hierarchy View* for details.

To launch any of these views, select Window | Show Views | Other from the workbench menu bar and expand the Intershop 7 folder.

Figure 237. Intershop 7 Views

Perspectives

The specific combination of views and editors displayed in the Workbench is called a perspective. Perspectives provide a useful combination of views and editors that are suited to performing a particular set of tasks. Each perspective can be modified individually depending on the task the perspective is used for.

Intershop Studio provides the following perspectives:

- **Cartridge Development Perspective**

This perspective is the default perspective displayed when you start Intershop Studio. It provides views to browse and maintain cartridge resources.

- **Intershop 7 Remote Editing Perspective**

This perspective is invoked when editing templates, pipelines, pipelets or static content resources on a remote server. With respect to the explorers, views and editors it provides, the Intershop 7 Remote Editing Perspective is similar to the Cartridge Development Perspective.

- **Pipeline Development Perspective**

This perspective provides an easy-to-use graphical user interface for pipeline management.

- **ISML Template Development Perspective**

This perspective provides the Template Editor and additional views to support template development.

- **Pipeline Debug Perspective**

This is a specialized perspective for debugging pipelines.

- **Pipeline Comparer Perspective**

This is a specialized perspective for comparing pipelines.

- **Intershop 7 Modeling Perspective**

This perspective provides the EDL model editor and additional views to support Java object modeling.

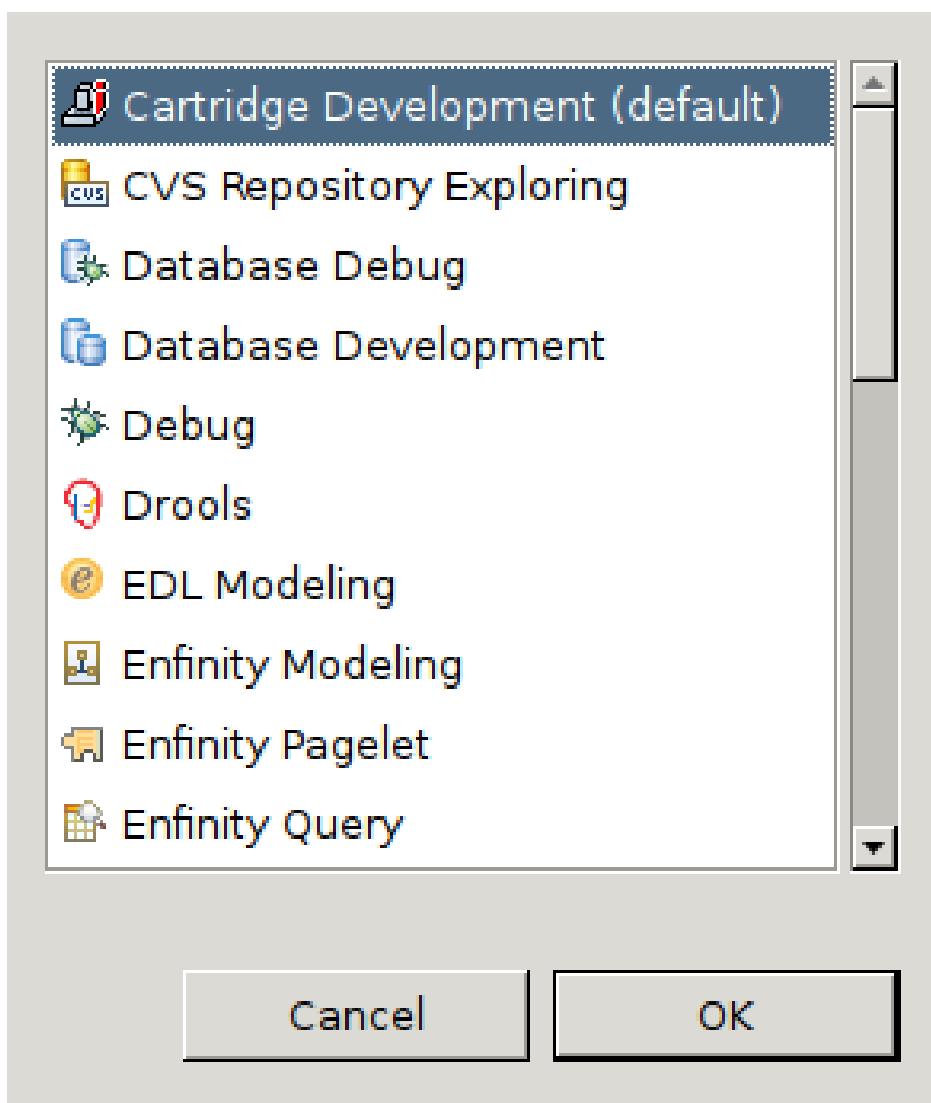
- **Intershop 7 Pagelet Perspective**

This perspective provides the Pagelet Editor and additional views (such as the Pagelet view) to support the creation of pagelet models.

- **Intershop 7 Query Perspective**

This perspective provides the Query Editor to support the creation of query definitions.

To switch from the Cartridge Development Perspective to a different perspective, click on the  icon and select the respective perspective from the window.

Figure 238. Intershop 7 Perspectives

Views

The Cartridge Explorer

The Cartridge Explorer lists all cartridges deployed on your Intershop 7 system. Each cartridge is presented in a navigation tree that can be expanded by double-clicking the name of a cartridge. This way you navigate the entire cartridge and locate the cartridge resources you want to work with.

The Toolbar

The toolbar at the top of the Cartridge Explorer provides additional functions for working with cartridges. Most of the functions are self-explanatory, all others are described in the following table:

Table 113. The Cartridge Explorer Toolbar Icons

Icon	Description
	Displays the properties for the active element in the Properties View. Intershop Studio automatically opens the Properties View if the view is not visible.
	If selected, the resources active in the Explorer view are automatically synchronized with the active editor and vice versa. For example, assuming you have several cartridge resources opened in Editor views, and you select one of them in the Cartridge Explorer, the according Editor view automatically gets the focus.
	Toggles the display of the Information area. The Information area appears at the bottom of the Cartridge Explorer views and displays additional information about the selected resource. For example, for pipelets the Information area displays the description stored in the pipelet descriptor file.
	Toggles the display of the Resource area. The Resource area appears at the bottom of the Cartridge Explorer views and displays files that make up the current resource. For example, for a pipelet the Resource area displays the corresponding *.java and *.xml files.
	Makes the active element the root element of the displayed navigation tree. For ease of navigation all other cartridges are hidden from view.
	Displays the previous root element at the top of the navigation tree.
	Reduces all elements that are currently expanded, showing the root elements only.
	Displays the Cartridge Explorer's standard view showing all cartridges.

The Status Icons

All cartridge resources are represented by icons that provide important information on the status of the resources. The following table describes the displayed icons:

Table 114. Resource Status Icons

Icon	Description
	A gold-colored icon marks source cartridges.
	A blue-colored icon indicates an imported server cartridge.
	The exclamation mark in an yellow triangle indicates that the resource itself or one of its child-elements generated a warning. Double-click the resource to open it in an editor. Use the Problems View to gain more information.

Icon	Description
	information about the generated warning. See <i>The Problems View</i> for more information.
	The white cross in the red square indicates that the resource itself or one of its child elements generated an error. Double-click the resource to open it in an editor. Use the Problems View to gain more information about the generated error. See <i>The Problems View</i> for more information.
	A plain unmarked icon indicates that no warnings or errors were generated for the resource.
	The check mark indicates that the cartridge is registered with Intershop 7 in the cartridgeList.properties file.

The Drop-Down Menu

The drop-down menu on the Cartridge Explorer toolbar provides the following additional functionality:

■ Working Sets

You can use working sets to jointly display cartridges or Java classes relevant for a certain development task and hide all others from view. A list of recently used working sets is also accessible via the drop-down menu allowing you to switch easily between different sets.

■ Filters

By applying filters to the contents displayed in the Explorer, you can show or hide cartridge resources from view.

■ Sorting

Allows you to define the order in which cartridge resources are displayed in the Explorer.

To open the drop-down menu, click the icon. The following table contains a description of the menu commands:

Table 115. Drop-Down Menu Commands

Command	Description
Select Working Set ...	Provides access for creating, editing, and deleting working sets.
Deselect Working Set	Deselects the active working set and returns to the Explorer's standard view with all available cartridges and resources displayed.
Edit Active Working Set ...	Allows you to add or remove components to/from the active working set.
Filters	Allows you to apply filters to the Cartridge Explorer.
Sort By	Defines how the Cartridge Explorer sorts the displayed contents.

Command	Description
Show/Hide Resources Part	Displays an information area at the bottom of the Cartridge Explorer that shows additional information on the resource, e.g. descriptions of pipelets and cartridges.

Drag & Drop Operations

The Cartridge Explorer allows you to drag resources with the mouse and drop them elsewhere in order to:

- **Copy resources between cartridges.**

You may copy an entire pipelet from one cartridge to another by a simple drag & drop operation. Alternatively, you can also use the commands copy and paste accessed through the context menu.

- **Copy pipelet properties.**

Using drag & drop you can copy single properties from one pipelet to the other.

- **Open Resources in an Editor**

Drag & drop resource files from the Explorer to the editor area to open them in an internal editor.

The Package Explorer

The Package Explorer provides a view of the package structure of cartridge projects and allows you to open, close, and delete cartridge projects. By applying filters you can specify what is displayed in the Package Explorer.

The Toolbar

The toolbar at the top of the Package Explorer provides additional functions, most of them are self-explanatory. If you click the  icon, you turn on the automatic synchronization between Explorer and Editor views. For example, assuming you have opened several editors with cartridge resources, and you select one of them in the Package Explorer, the according Editor view automatically gets the focus.

The Status Icons

Icons assigned to each cartridge resource provide important information on the status of the resource. The following table describes the icons:

Table 116. Resource Status Icons

Icon	Description
	Library container
	Individual system library file
	Individual static library file
	Logical package
	Package containing only non-Java resources

Icon	Description
	Package not containing any resources

The Drop-Down Menu

The drop-down menu on the Package Explorer toolbar provides the following additional functionality:

■ Working Sets

You can use working sets to jointly display cartridges or Java classes relevant for a certain development task and hide all others from view. A list of recently used working sets is also accessible via the drop-down menu allowing you to switch easily between different sets.

■ Filters

By applying filters to the contents displayed in the Explorer, you can show or hide cartridge resources from view.

■ Sorting

Allows you to define the order in which cartridge resources are displayed in the Explorer.

To open the drop-down menu, click the  icon. The following table contains a description of important menu commands:

Table 117. Drop-Down Menu Commands

Command	Description
Select Working Set ...	Provides access for creating, editing, and deleting working sets.
Deselect Working Set	Deselects the active working set and returns to the Explorer's standard view with all available cartridges and resources displayed.
Edit Active Working Set ...	Allows you to add or remove components to/from the active working set.
Filters	Allows you to apply filters to the Cartridge Explorer.
Sort By	Defines how the Cartridge Explorer sorts the displayed contents.
Filters	Defines what contents is displayed in the Package Explorer.

Pipelet, Pipeline and Template Views

These views list all pipelets, pipelines or templates that are included with the current cartridge project. Pipelet, Pipeline and Template View are part of the default set of views integrated with the pipeline development perspective.

The views list pipelets and pipelines along with the following properties:

- **Name**
- **Cartridge**
- **Group (for pipelines and pipelets only)**
- **Type (for pipelines only)**
- **Path (for templates only)**

A quick filter is available which can be used to search the pipelet or pipeline lists by any of the properties listed above.

The Toolbar

The toolbar at the top of the list views for pipelets and pipelines provides additional functions, most of them are self-explanatory.

■ **automatic synchronization**

If you click the  icon, you turn on the automatic synchronization between list views and Editor views. For example, assuming you have opened several editors with pipelines and pipelets, and you select one of them in the Pipelet or Pipeline View, the according Editor view automatically gets the focus.

■ **sorting**

If you click the  icon, you can sort the flat list of resources by cartridge (all), by group (pipelets or pipelines), by language (templates) or by path (templates).

The Status Icons

Icons assigned in the view to each pipelet or pipeline provide important information on the status of the resource. The icons used are identical to the icons used in the Cartridge Explorer. See *The Status Icons* for details.

The Drop-Down Menu

The drop-down menu on the view toolbar provides the following additional functionality:

Table 118. Drop-Down Menu Commands

Command	Description
Link With Editor	Links the selected pipelet with the Java editor.
Show Interface	In the pipeline view, toggles the interface area. The interface area displays the pipeline start, end and interaction nodes.
Show Modules	In the template view, toggles the module list.
Show Information Area	Toggles the information area. The information area displays the pipelet or pipeline description.
Show Resources Part	Toggles the resources area. The resources area displays files that make up the current resource.

Command	Description
Show Column	Allows you to add or remove columns from the view.
Scope Type	Allows you to select the scope of the current view. The scope can be the current cartridge project, the current cartridge project and all required cartridges, all cartridges, or all cartridge projects.
Cartridges in Scope	Displays the cartridges which are currently in the scope. Elements of these cartridges are displayed in the current view.
Filters	Allows you to hide or show elements that are invisible or overridden. Overridden elements have a corresponding element which is higher in the lookup hierarchy.

Pagelet View

The pagelet view lists all pagelets defined within Intershop 7 pagelet models. The pagelet view is part of the default set of views integrated with the pagelet perspective.

The view lists pagelets along with the following properties:

- **Name**
- **Cartridge**
- **Pagelet Model**

A quick filter is available which can be used to search the pagelet list by any of the properties listed above.

The Toolbar

The toolbar at the top of the pagelet view provides additional functions, most of them are self-explanatory:

■ automatic synchronization

If you click the  icon, you turn on the automatic synchronization between pagelet view and pagelet editor.

■ sorting

If you click the  icon, you can sort the flat list of pagelets either by pagelet model or by cartridge.

The Status Icons

Icons assigned in the view to each pagelet indicate whether the pagelet serves a page type or a component type.

Table 119. Resource Status Icons

Icon	Description
	This pagelet serves a page type.

Icon	Description
	This pagelet serves a component type.
	This element represents a slot.

The Drop-Down Menu

The drop-down menu on the view toolbar provides the following additional functionality:

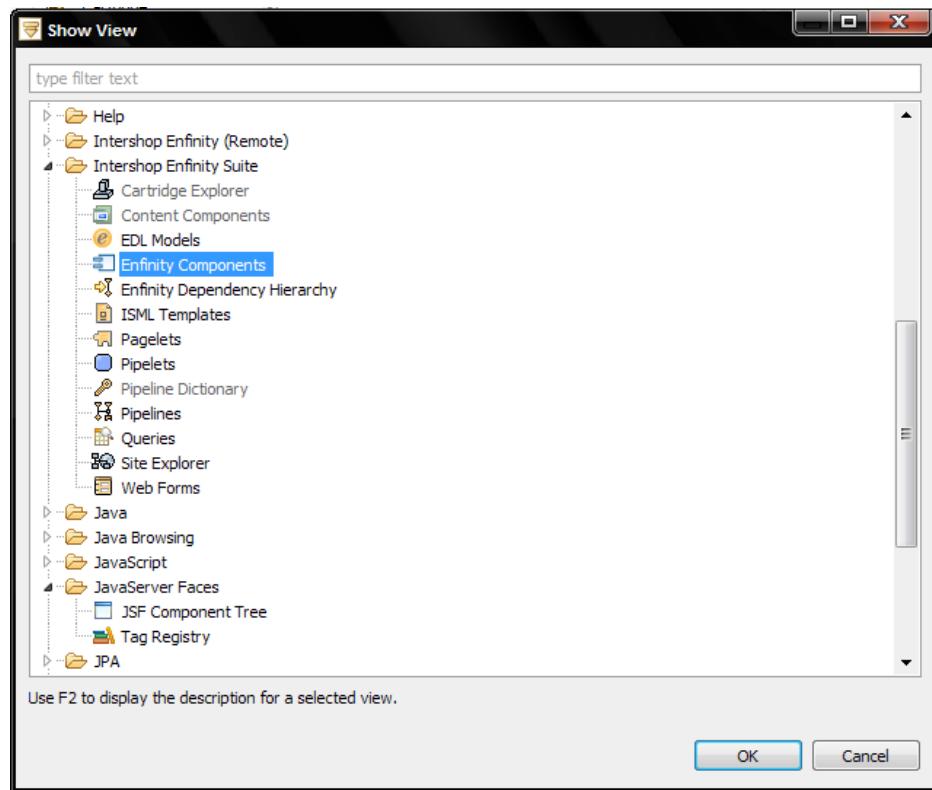
Table 120. Drop-Down Menu Commands

Command	Description
Link With Editor	Links the selected pagelet with the Java editor.
Show Information Area	Toggles the information area. The information area displays the pagelet description.
Show Resources Part	Toggles the resources area. The resources area displays files that make up the current resource.
Show Column	Allows you to add or remove columns from the view.
Scope Type	Allows you to select the scope of the current view. The scope can be the current cartridge project, the current cartridge project and all required cartridges, all cartridges, or all cartridge projects.
Cartridges in Scope	Displays the cartridges which are currently in the scope. Elements of these cartridges are displayed in the current view.
Filters	Allows you to hide or show elements that are invisible or overridden. Overridden elements have a corresponding element which is higher in the lookup hierarchy.

Components View

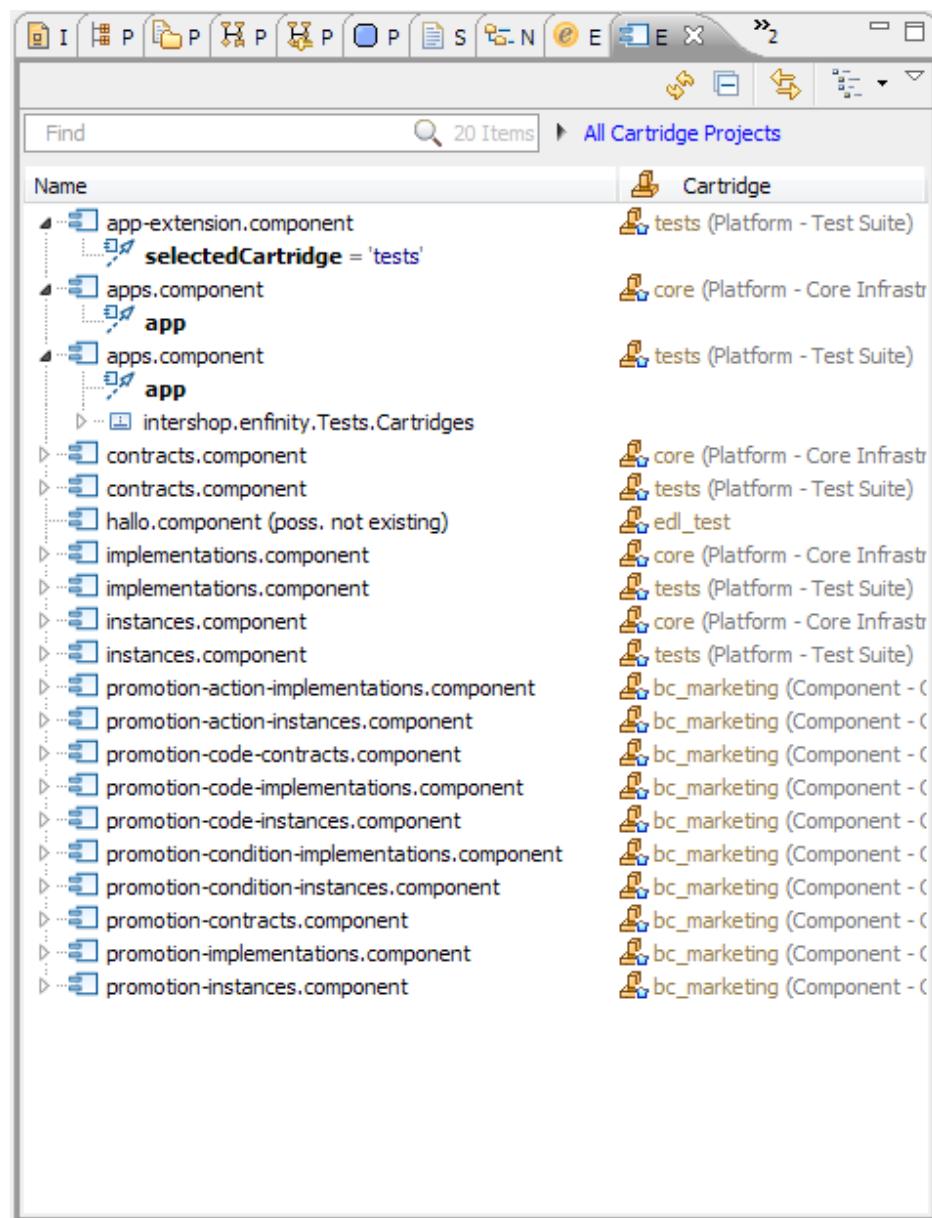
The components view displays components provided by cartridges in the workspace or by your Intershop 7 installation as configured in the Intershop Studio preferences.

To open the Components View, go to Window | Show View | Open other ...

Figure 239. Opening the Components View

The components view allows you to quickly access components, contracts, implementations of contracts, and instances of implementations.

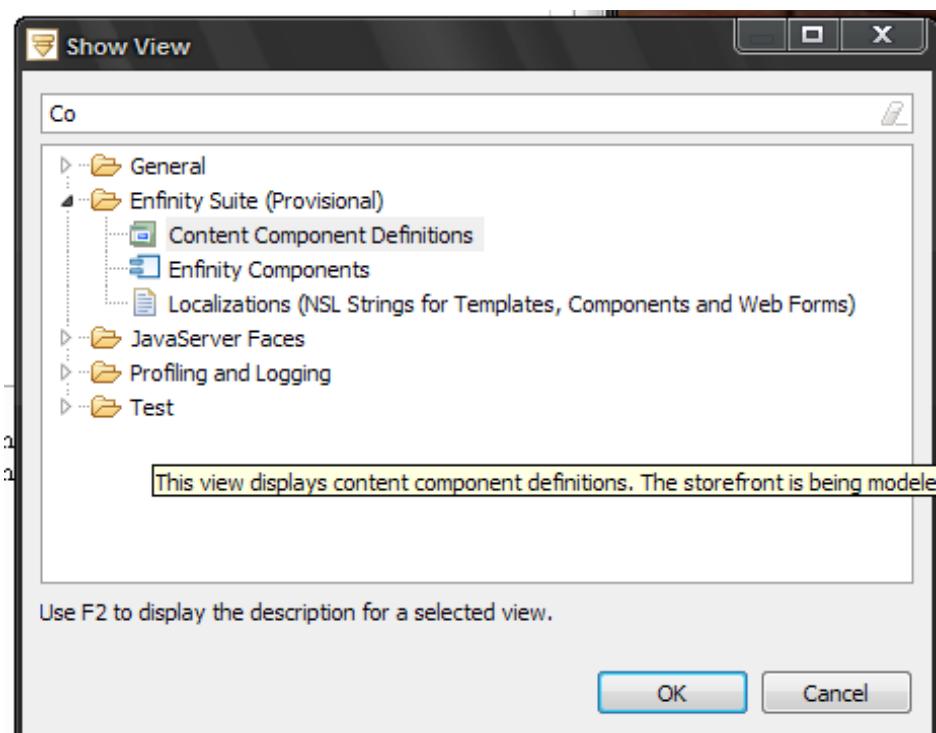
In combination with the properties view more detailed properties of the elements can be shown.

Figure 240. The Components View

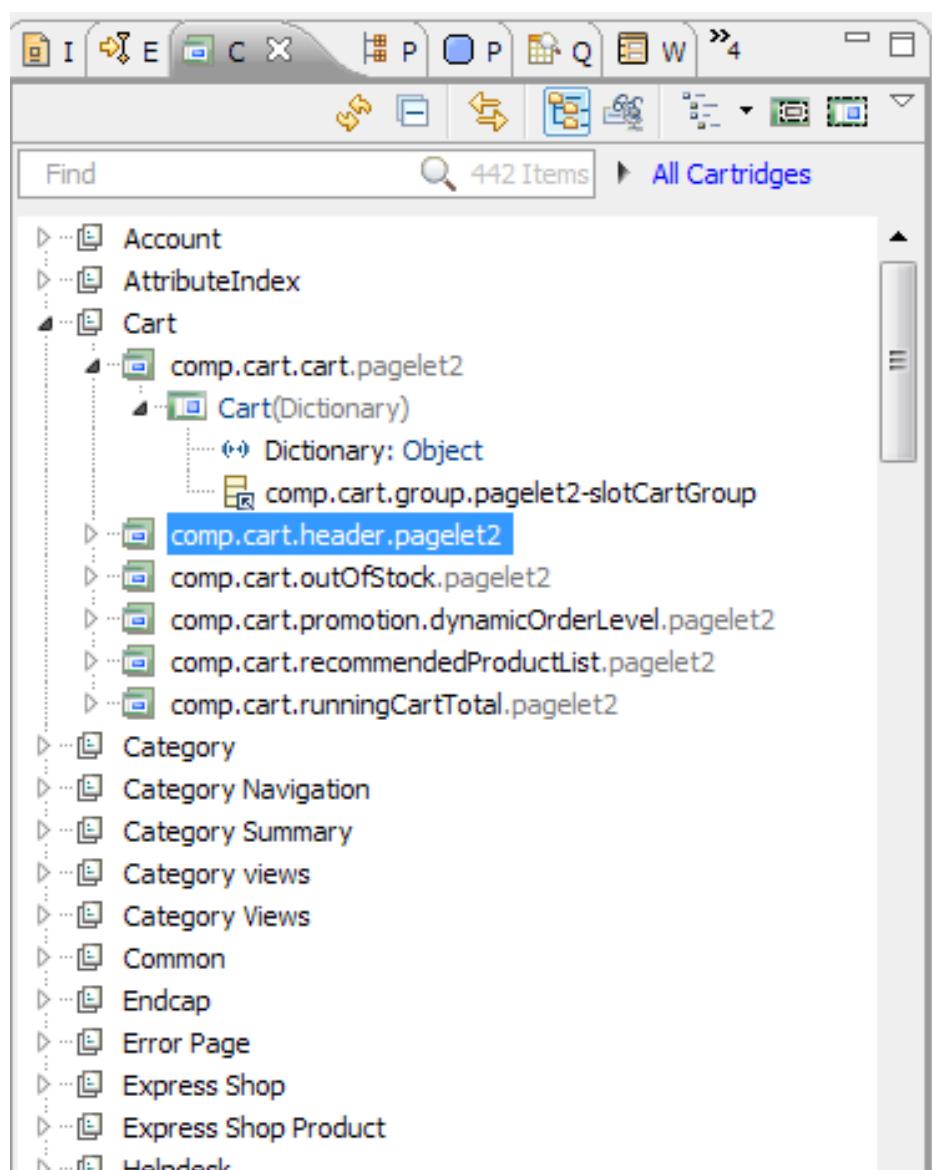
Content Component Definitions View

The content component definitions view displays content components provided by cartridges found in the workspace or by cartridges deployed in your Intershop 7 installation as configured in the Intershop Studio preferences.

To open the content component definitions view, go to Window | Show View | Open other ...

Figure 241. Opening the Content Component Definitions View

The view allows you to quickly access content components, models or content entry points. In combination with the properties view more detailed properties of the elements can be shown.

Figure 242. Content Component Definitions View

Query View

The query view lists all query definitions defined across Intershop 7 cartridges. The query view is part of the default set of views integrated with the query perspective.

The view lists queries by name. A quick filter is available which can be used to search the query list by name.

The Toolbar

The toolbar at the top of the pagelet view provides additional functions, most of them are self-explanatory.

- automatic synchronization

If you click the  icon, you turn on the automatic synchronization between pagelet view and pagelet editor.

- sorting

If you click the  icon, you can sort the flat list of pagelets either by pagelet model or by cartridge.

The Status Icons

Table 121. Resource Status Icons

Icon	Description
	Query definition for a query
	Parameter definitions for a query
	Mandatory parameter
	Optional parameter
	Return mapping definitions for a query
	Default mapping
	ORM mapping
	Constructor mapping
	Template definitions for a query
	Individual template
	Query processor

The Drop-Down Menu

See [The Drop-Down Menu](#).

The Properties View

The Properties View shows the properties of the cartridge resource that is currently selected.

The Toolbar

The following table describes the icons of the toolbar:

Table 122. The Properties View Toolbar Icons

Icon	Description
	Toggles the view of categories.
	Toggles the view of advanced properties.
	Restores the default value of the selected property.

The Problems View

The Problems View displays potential problems of all cartridge projects in form of warnings and error messages. These are listed as tasks that you need to resolve before you can build the cartridge.

By applying a filter you can limit what is displayed in the Problems View. You can filter items according to which resource or group of resources they are associated with, by text string within the Description field, by problem severity, by task priority, or by task status.

Table 123. Resource Status Icons

Icon	Description
	Indicates a warning. Warnings indicate problems that are not critical to the functioning of the cartridge, e.g. code that is not conform to the Intershop Code Style Guide, pipelets with missing descriptions, the usage of deprecated Java methods, etc.
	Indicates an error. An error indicates a critical problem that must be fixed before you can build and run the cartridge.

The Outline View

The Outline View displays a structured outline of the resource that is currently open in the editor area, providing the possibility to drill down to individual elements such as methods of a Java class or pipeline nodes. The contents of the Outline View as well as toolbar options available are editor-specific. See *Outline View for Java Editor*, *Quick Outline View*, *Outline View for Template Editor*, *Outline View for Query Editor*, and *Outline View for EDL Model Editor* for details.

The Pipeline Dictionary View

The Pipeline Dictionary View provides information regarding the Intershop 7 data types and values that currently are in the pipeline dictionary. Click on a node in a pipeline to see

- **the data types and values that are read from the dictionary to process the node.**

Ingoing values are marked by a green arrow, e.g. icon.

- **the data types and values that are put into the dictionary when processing the node.**

Outgoing values are marked by a red arrow, e.g. icon.

NOTE: Click the Link With Editor icon () to track data in the Pipeline Dictionary View.

The Status Icons

Each key in the Pipeline Dictionary is represented by an icon. The following table describes the icons:

Table 124. Resource Status Icons

Icon	Description
	Indicates a simple (primitive) data type.
	Indicates a complex data type.
	Indicates a data type that can be used in a loop.
	The green arrow indicates read access to a parameter. Gold-colored parameters are guaranteed.
	The red arrow indicates write access to a parameter. Blue-colored parameters are optional.

Intershop 7 Dependency Hierarchy View

This view is used to visualize dependencies between a selected resource (e.g., the resource currently active in the editor) and other Intershop 7 resources. The following dependency types can be explored:

■ Overwrites/Overwritten

Resources which the current resource overloads, and resources which overload the current resource.

■ Dependent/Required By

Resources on which the current resource depends, and resources which depend on the current resource.

■ Caller/Callee

Resources which the current resource calls, and resources which call the current resource.

To switch to a different dependency type, select .

To link the content of the Intershop 7 Dependency Hierarchy with the editor, click the .

To display details of dependent elements, toggle the detail area through selecting Show Details from the drop-down menu.

Table 125. Toolbar Icons

Icon	Description
	Displays resources which the current resource calls.
	Displays resources which call the current resource.
	Displays resources which depend on the current resource.
	Displays resources on which the current resource depends.
	Displays resources which override the current resource.

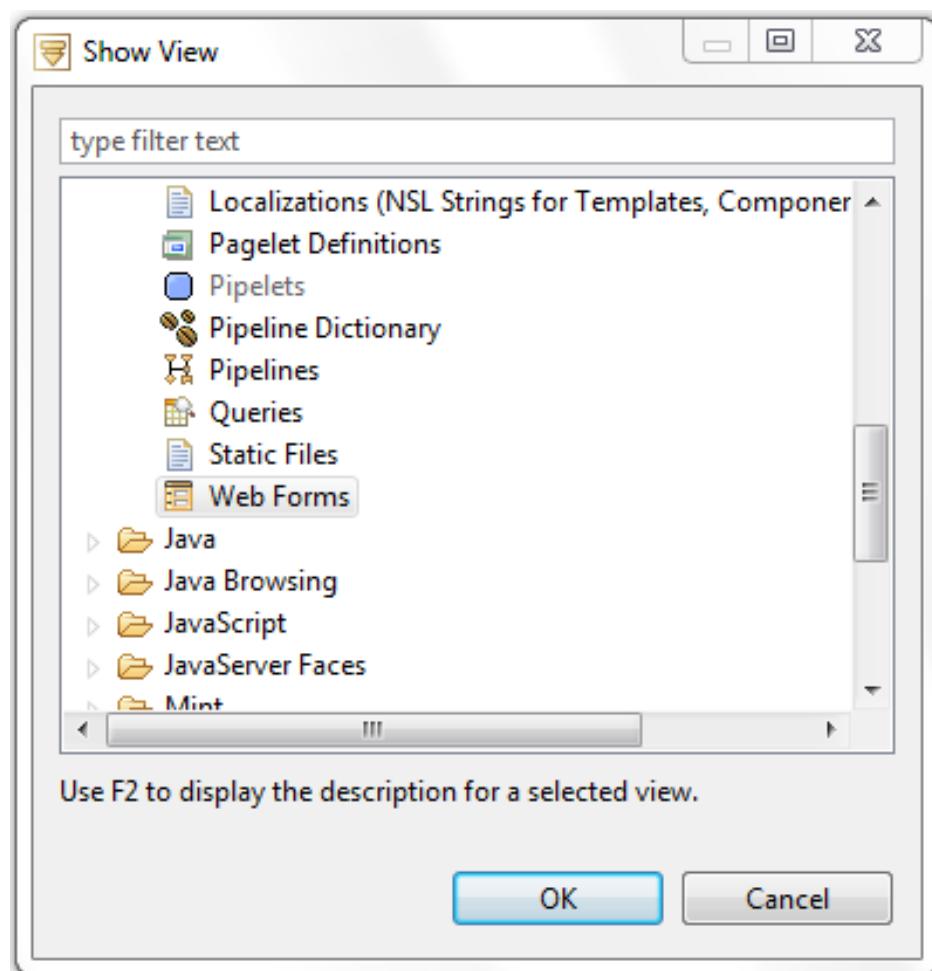
Icon	Description
	Displays resources which the current resource overrides.
	Used to set the focus of the view back to a resources recently viewed.

Webforms View

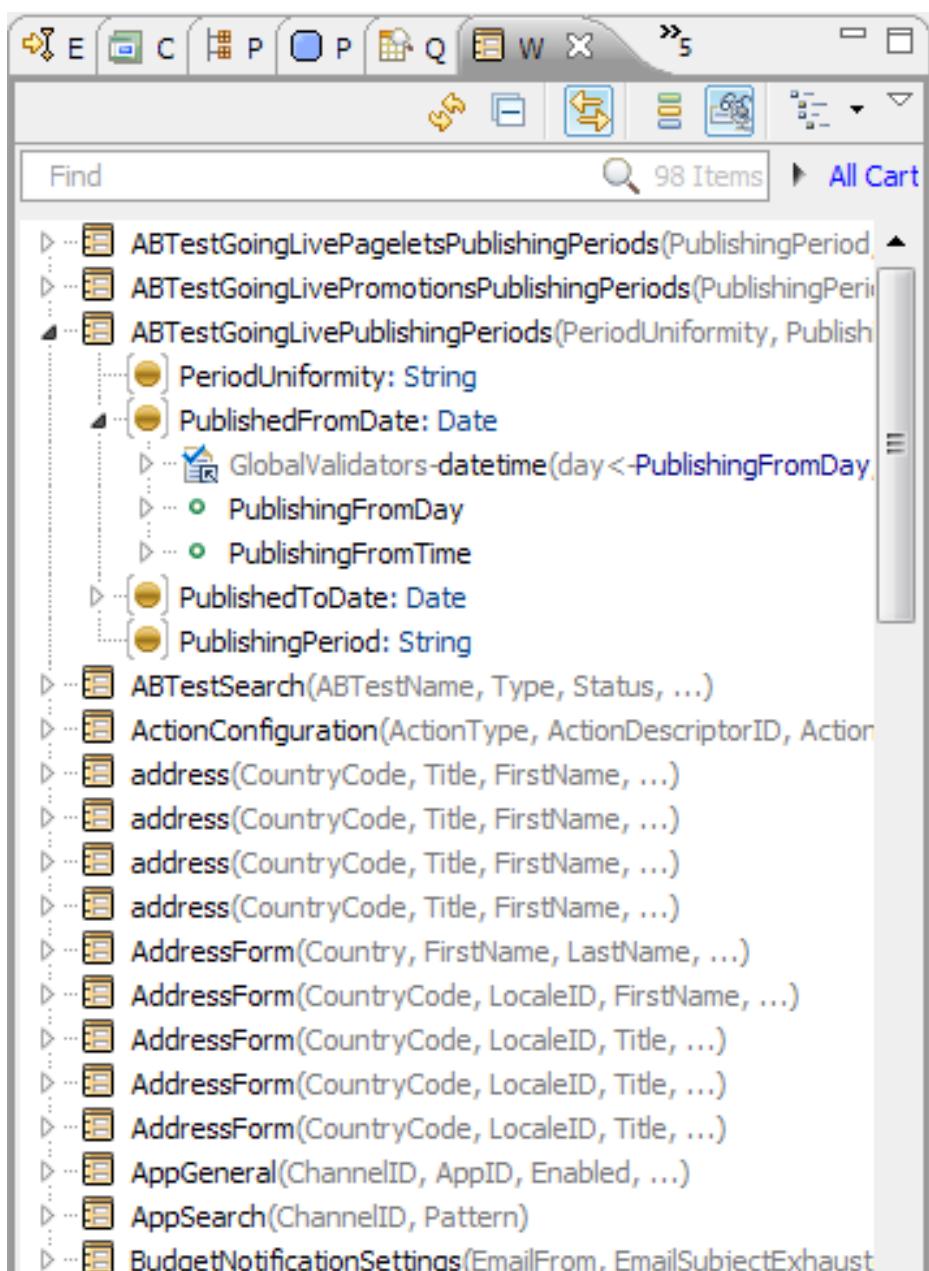
The webforms view lists all web forms found in cartridges in your workspace or cartridges deployed in your Intershop 7 installation as configured in the Intershop Studio preferences.

To open the view, go to Window | Show View | Open other ...

Figure 243. Opening the Web Forms View



The webforms view provides quick access to forms, models, validators, and formatters. In combination with the properties view more detailed properties of the elements can be shown.

Figure 244. Web Forms View

In case the webforms view is not available in your Intershop Studio installation you may have to enable Provisional Intershop Plugins first:

- 1. From the Intershop Studio menu, Open the preferences dialog with Window | Preferences.**
- 2. In the Preferences dialog, select General | Capabilities.**
- 3. Check on the "Provisional Intershop Plugins".**

NOTE: If Intershop Studio complains about insufficient heap size, allocate more memory to Intershop Studio by editing the according properties in <IntershopStudio>\intershopstudio.ini (see Optimize Intershop Studio Performance).

Search

Global Search Capabilities

The Intershop Studio Search Dialog, which is accessed through selecting Search from the Workbench's menu bar or through clicking the  icon, includes the following tabs:

■ File Search

Allows you to search for files or text in the Workbench.

■ Intershop 7 Cartridge Search

Allows you to search in the properties of Intershop 7 cartridge elements.

■ Java Search

Allows you to search for Java elements.

■ Plug-in Search

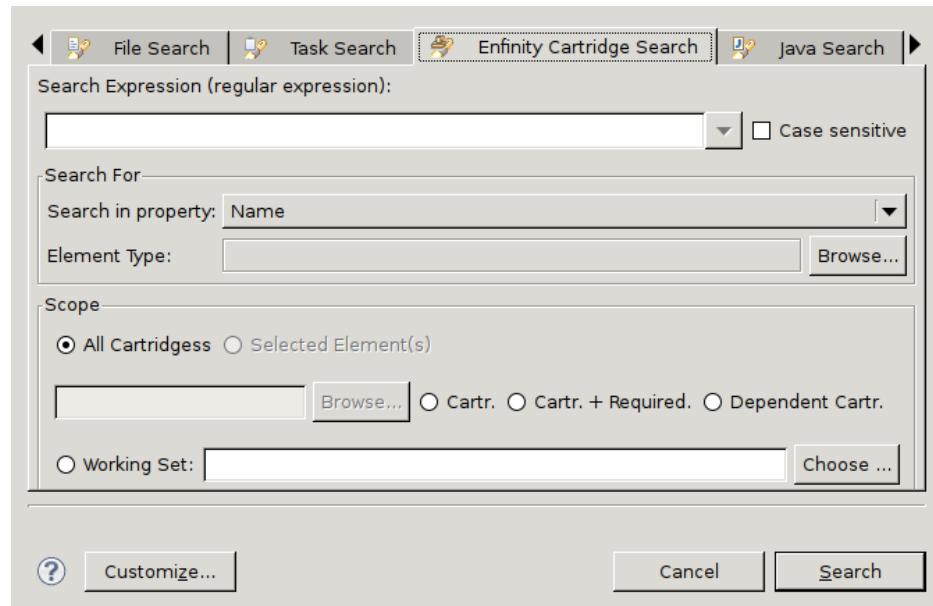
Allows you to search for Eclipse plug-ins or extension points.

NOTE: As the File, Java, and Plug-in are not part of the Intershop Studio features, they are not covered by this documentation set. For information on these search mechanisms, refer to the corresponding plug-in documentation.

Intershop 7 Cartridge Search

The Intershop 7 cartridge search is accessed through selecting Search | Intershop 7 Cartridge from the Workbench's menu bar. Alternatively, you can click the  icon and select the Intershop 7 Cartridge Search tab.

Figure 245. Searching properties of Intershop 7 cartridge elements



This search dialog allows for searching the properties of Intershop 7 cartridge elements. The following table summarizes the available options.

Table 126. Intershop 7 cartridge search options

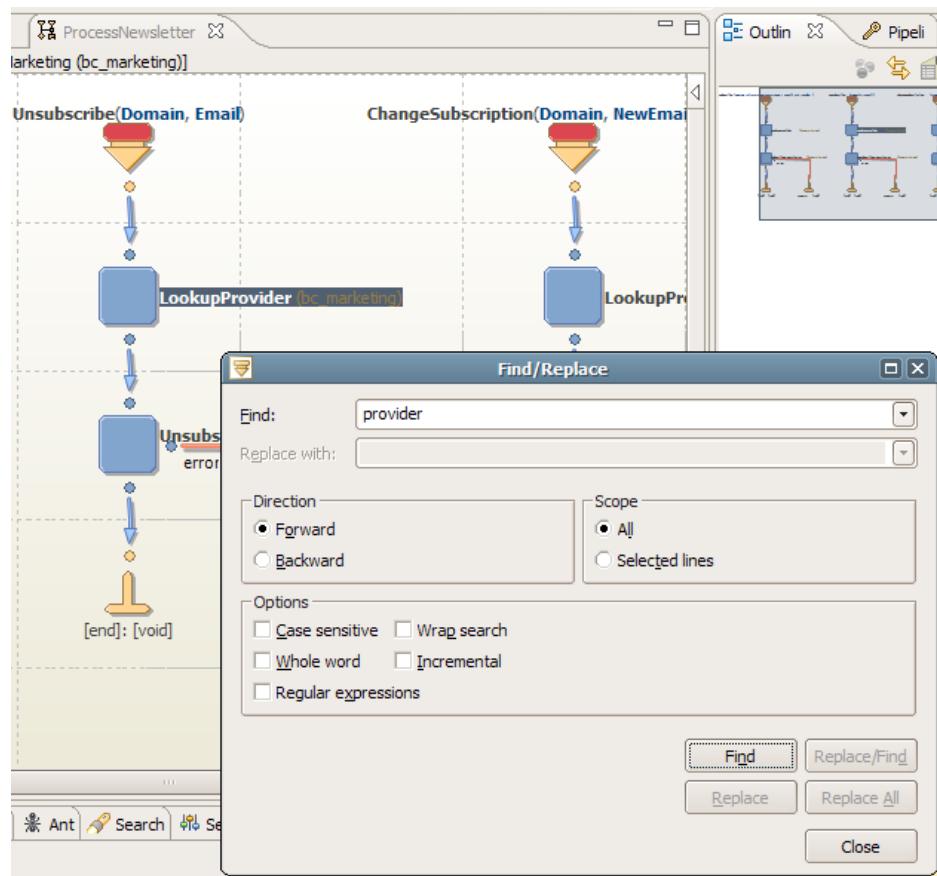
Name	Description
Search Expression	The expression to be searched. Regular expressions and case sensitive searches are supported. From the drop-down list, you can choose to repeat or modify a recent search.
Search in property	Drop-down list that allows for selecting the property type to be searched: name, display name, group, description.
Element Type (Browse)	Opens a dialog to select the Intershop 7 element type to be searched: pagelet, pipelet, pipeline, pipeline Web service, query, static file, template.
All Cartridges	Select this radio button to extend your search to all available cartridges.
Selected Element(s)	Select this radio button to limit your search to either a single cartridge, a cartridge and all required cartridges or dependent cartridges through selecting the corresponding radio button and browsing for the intended cartridge.
Working Set	Select this radio button to limit your search to a specific working set. Click Choose to open a dialog for selecting the working set.

Intershop 7 Element Search

Intershop 7 views provide specific search options for Intershop 7 elements. This enables developers to directly search for Intershop 7 element references, similar elements and elements used.

Finding Elements In Graphical Editors

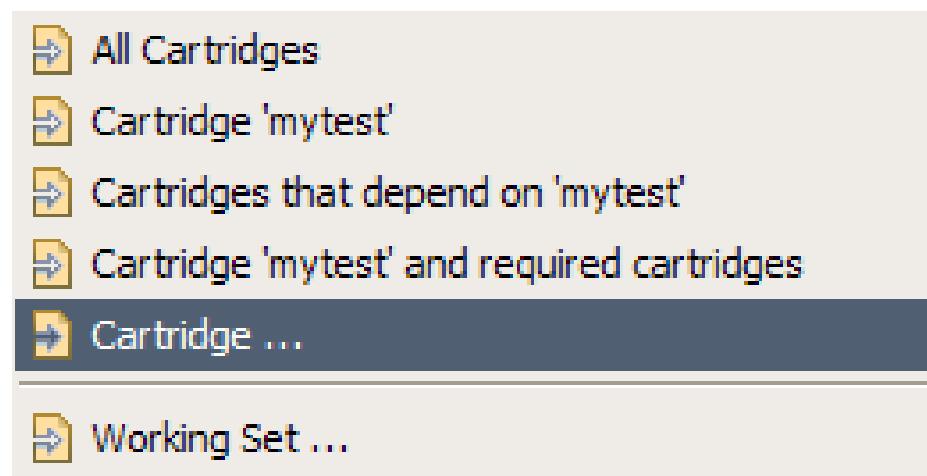
Graphical editors, e.g. the Pipeline Editor, provide a "Find" command. It helps users to find elements by label texts or property values. To invoke this command, use **CTRL+F** or select **Edit | Find/Replace** from the menu.

Figure 246. Find dialog in the Pipeline Editor

Searching Element References

For the elements listed in Intershop 7 views, developers can search for referenced elements. The reference search involves the element itself, i.e., cartridges, pipelines, pipelets, templates, queries or pagelets, as well as its subelements or children.

Upon invoking the reference search through selecting "References" from the item's context menu, a submenu allows you to set the scope for your search.

Figure 247. Setting the scope for reference searches

The following table lists the available options.

Table 127. Search scope options

Name	Description
All Cartridges	Searches across all cartridges in the current workspace.
Cartridge <name>	Searches in the current cartridge only.
Cartridges that depend on <name>	Searches in all cartridges that depend on the current one.
Cartridge <name> and required cartridges	Searches in the current and all required cartridges.
Cartridge ...	Opens a dialog to select a cartridge to be searched.
Working Set ...	Opens a dialog to select a working set to be searched.

Searching Similar Elements

For pipelets and pipelines, including all child elements like dictionary in or out properties or pipeline nodes, as listed in Intershop 7 views, developers can search for similar elements. Depending on the element or subelement selected, the similar elements search can find items with a similar name (pipelets, pipelines or pipeline nodes) or, for example, similar dictionary in or out properties.

Upon invoking the similar elements search through selecting "Similar Elements" from the item's context menu, a submenu allows you to set the scope for your search. The scope options are the same as for the reference search (see *Table 127, "Search scope options"*).

Searching Elements Used

For pagelets, pipelines, pipeline nodes and templates listed in Intershop 7 views, developers can search for other resources used by the current element. This helps to keep track of dependency relations between the individual elements of a project.

NOTE: The delivered search result is another representation of the caller/callee relation as displayed in the Intershop 7 Dependency Hierarchy View.

Upon invoking the elements used search through selecting Elements Used from the item's context menu, a submenu allows you to set the scope for your search. The scope options are the same as for the reference search (see *Table 127, "Search scope options"*).

Remote Server Configuration

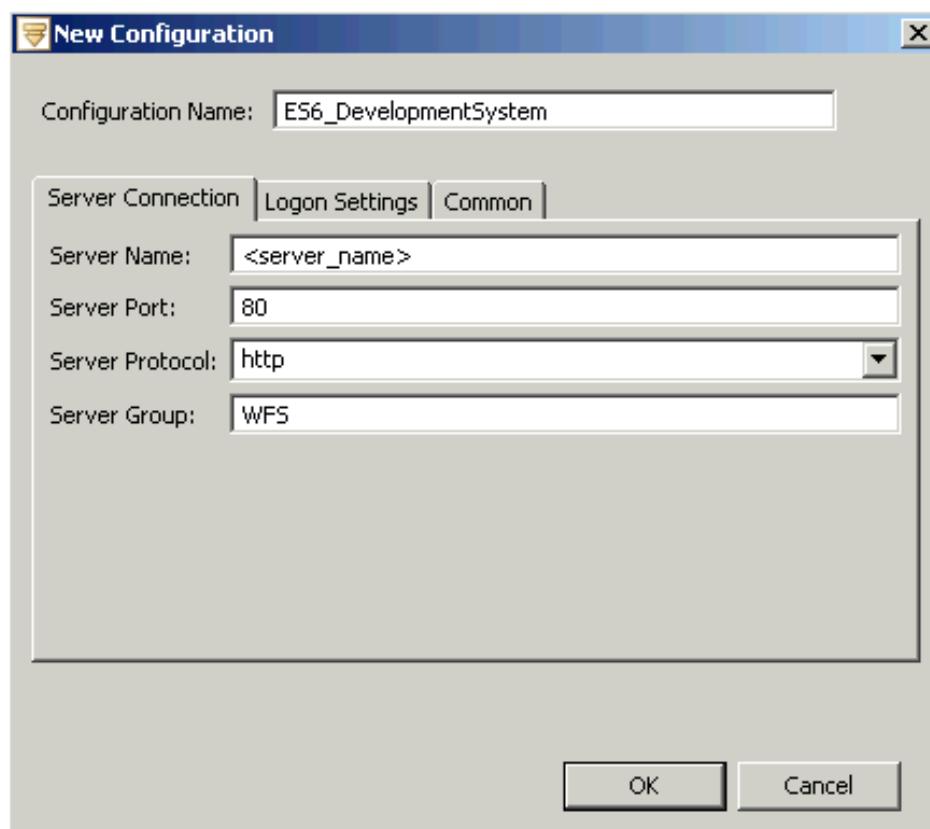
Several activities in Intershop Studio require an active connection to an Intershop 7 Server, such as pipeline debugging, pipeline reload, or pipelet reload. The sections below briefly describe the parameters to be set for remote server connections.

Server Connection Parameters

The following parameters identify the server to connect to.

Table 128. Server Connection Parameters

Parameter	Description
Server Name	Specifies the name of the server that Intershop Studio will connect to. In a distributed (or cluster) installation, you have to provide the name of the server hosting the web adapter.
Server Port	Note that you can use a port different from the standard port 80 for HTTP requests.
Protocol	Specifies the protocol type that Intershop Studio will use to connect to the server. If you require a secure connection to the Intershop 7 server, choose HTTPS. A secure connection is only possible if the web server has been SSL enabled.
Server Group	Specifies the type of server group that Intershop Studio will connect to. Enter WFS to connect to the server group which handles standard storefront requests.

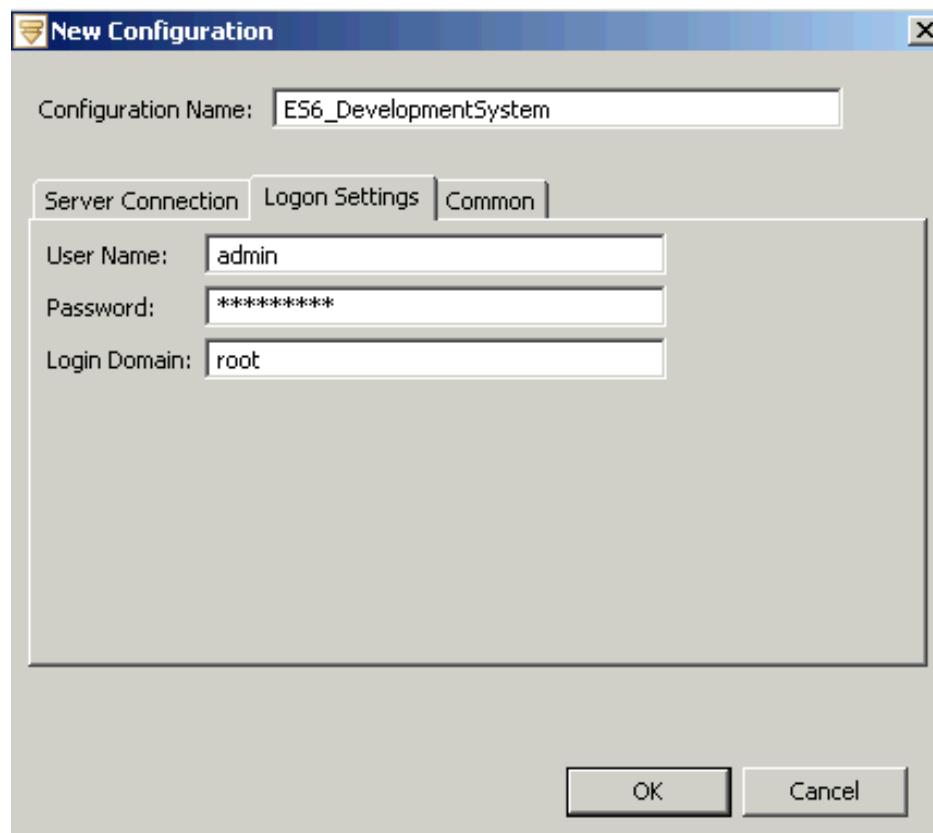
Figure 248. Remote Server Connection Page

Logon Parameters

The following parameters define the credentials for Intershop Studio to log on.

Table 129. Logon Parameters

Parameter	Description
User Name	A valid user name for Intershop Studio to use.
Password	A valid password for Intershop Studio to use.
Domain Name	Defines the domain context to which Intershop Studio logs on. Connect to the <i>root</i> domain, which makes all sites available.

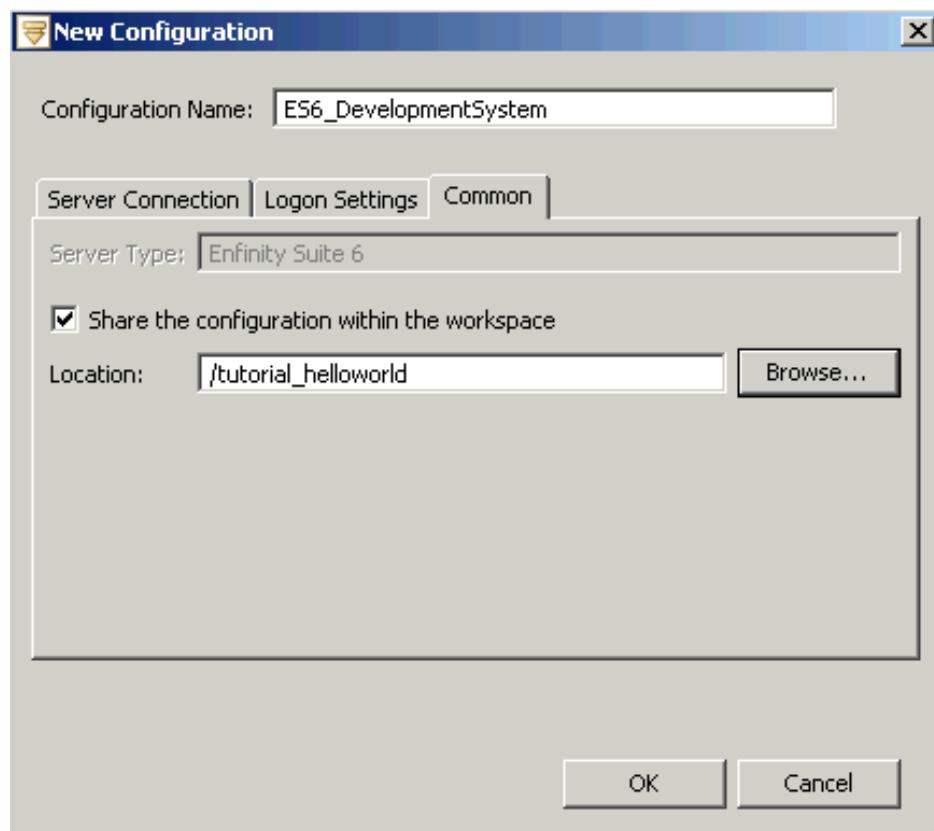
Figure 249. Remote Server Connection Page

Common Parameters

The following additional parameters have to be set:

Table 130. Common Parameters

Parameter	Description
Connection Type	The connection type depends on the version of the Intershop 7 system you are using. Available connection types are displayed in the drop-down menu.
Share Connection With Workspace	The connection information can be shared with and re-used by other Intershop Studio projects. Select the check box and provide the respective project to share the connection information.

Figure 250. Remote Server Connection Page

API Tools

API Tools Introduction

Intershop Studio provides a set of tools that compare the APIs of multiple software artifacts in order to find out changes between different versions. That is, the tools' main intention is to help developers to migrate an Intershop 7 project that is based on a certain version of the platform onto a higher version.

NOTE: The project migration cannot be automated completely, however, these tools help reducing their complexity and thus the time and effort needed. These API tools do not substitute a decent planning and a migration-friendly design during the cartridge development.

Supported Artifacts

Intershop Studio's API tools support the following artifact types:

- Java code
- EDL
- pipelets
- pipelines
- templates
- queries
- pagelets
- Web forms
- localizations
- components
- applications
- static files (e.g., images)

Main Tasks

The main features of Intershop Studio's API tools include:

■ Cartridge API model generation

The tool can generate an API model for a complete cartridge and save it to a file. The cartridge API model includes the APIs of all artifacts of this cartridge

and their change history. The API model also provides information about dependencies to other artifacts. The API model can be exported to a JavaDoc-like HTML document. In addition, the tool can update the API model for a dedicated cartridge with the option to consider split or renamed cartridges passing the old and the new names.

For information about creating the API model, see *Creating API Models*.

For information about exporting the API model, see *Exporting API Documentation*.

For information about updating a cartridge's API model, see *Updating API Model*.

■ **Artifact modification screening**

The tool can identify changes between different artifact versions. The Intershop Studio API view shows in which version of the cartridge the artifact has been touched and states the type of the modification, like, for example, whether an artifact has been added, removed or changed.

For information about displaying the API and any modifications in the "live" view, refer to *Viewing API and API Modifications*.

■ **Change report generation**

The tool can generate reports that document the API changes of cartridges and their artifacts.

For information about creating a change report, see *Exporting Change Reports*.

■ **Migration report generation**

The tool can produce a specific report that shows the artifacts of project cartridges that must be migrated because they depend on Intershop 7 artifacts whose APIs have changed.

For information about creating a migration report, see *Creating Migration Reports*.

■ **Command line support**

The API tools can be invoked by a command line call. This allows for running the tool during the build process, for example, in order to generate the API documentation without human intervention.

For information about the API tools's command line support, see *Command Line API Tools*.

Using API Tools

This section describes the general usage of the Intershop Studio API tools.

Configuring API Tools

The Intershop Studio API tools require some dedicated settings. To define the preferences:

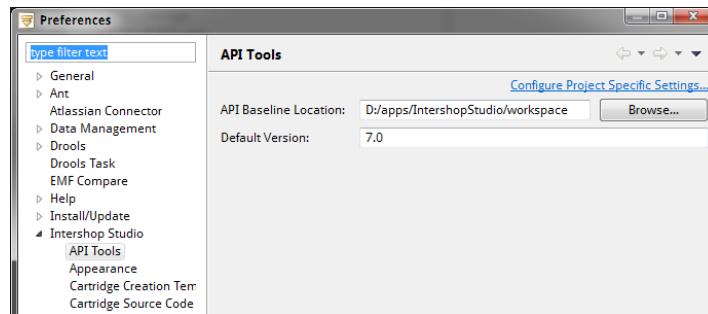
1. Open the Intershop Studio Preferences window.

From the menu, select Window | Preferences.

2. Expand the Intershop Studio tree entry, and select API Tools.

This opens the API Tools preference page.

Figure 251. API Tools preferences



3. Edit the settings as required.

The following preferences are available:

Table 131. API tools preferences

Property	Description
API Baseline Location	Defines the central location where the API model files are stored. The API model kept here is the base for all future API comparisons.
Default Version	Sets a fallback cartridge/product version to be used when no other version information is available upon generating the API model or an API report.

NOTE: You can set project-specific preferences in addition to the global settings.

4. Click Apply to save your settings.

Immediately clicking OK saves the settings and closes the Preferences window.

Creating API Models

To create an initial set of API models as the comparison base or to update the base API model:

1. Open the Intershop Studio file export wizard.

From the menu, select File | Export.

2. Expand the Intershop API tree entry, select Export the API of cartridges, and click Next.

This displays a list of available cartridges.

3. Select the intended cartridges, and click Next.

This displays the Export Type page.

4. Select API Model and click Next.

This displays the Export Properties page.

5. Set the export preferences as necessary.

The following options are available:

Table 132. API model export settings

Preference	Description
Export only changed artifacts	Restricts the API model creation/update to include only modified items of a defined "version range".
Resolve dependencies	Includes dependencies to other cartridges upon creating/updating the API model.
Export artifacts depending on changed items only	Restricts the API model creation/update to include only items that depend on modified items of a defined "version range".

6. Click Finish.

The API model file is saved to the location as set in *Configuring API Tools*.

Exporting API Documentation

To export an API model to a JavaDoc-like HTML set:

1. Open the Intershop Studio file export wizard.

From the menu, select File | Export.

2. Expand the Intershop API tree entry, select Export the API of cartridges, and click Next.

This displays a list of available cartridges.

3. Select the intended cartridges, and click Next.

This displays the Export Type page.

4. Select API Documentation and click Next.

This displays the Export Properties page.

5. Specify the output path.

Specify a correct path or browse for the target location.

6. Click Finish.

The API model is immediately exported as a JavaDoc-like HTML set to the specified location.

Viewing API and API Modifications

Developers can use the Intershop API view to display a "live" API view of the artifacts with which they are currently working. Once created an (initial) API model, the API view immediately highlights modifications.

To toggle the Intershop API view:

1. From the menu, select Window | Show View | Other.

This opens the Show View dialog.

2. Expand the Intershop tree entry, select Intershop API, and click OK.

This displays the Intershop API view.

NOTE: You can move the Intershop API view to any position in the Intershop Studio window that suits you via drag-and-drop.

To view a cartridge's API and API modifications:

- 1. Toggle the Intershop API view as described above.**
- 2. In the Intershop API view, click the Link With Editor icon .**
- 3. Select the intended cartridge in the Package Explorer or the Cartridge Explorer.**

The current API model of the selected cartridge is immediately prepared and displayed in the Intershop API view. Any changes with respect to the API version saved with the baseline API model are highlighted.

The Intershop API view features a number of view options, as outlined below.

Table 133. Intershop API view view options

Function	Icon	Description
Select filters		Opens a dialog for selecting view filters to be applied in the current API view.
Show element version		Switches on or off the display of an artifact's version number in the current API view.
Dependency filter		Allows for activating and defining a version range for a filter that restricts the API view content to items that depend on modified items.

Updating API Model

To update an API model for a single cartridge:

- 1. Open the Intershop Studio file export wizard.**
From the menu, select File | Export.
- 2. Expand the Intershop API tree entry, select Update the API of a cartridge, and click Next.**

This displays the Update API configuration page. The following options are available:

Table 134. Update API settings

Preference	Description
Previous API	Specifies the base API model.
Cartridge	Specifies the cartridge for which the API model is to be updated. In case the original (base) cartridge has been split or renamed, specifies the new cartridge name.
Output path	Specifies the location of the API model to be produced.
Split or rename cartridge	Mark this checkbox in case the original (base) cartridge has been split or renamed. The tool will then try to map the artifact names between the old and the new cartridge version.
Export only changed artifacts	Restricts the API model creation/update to include only modified items of a defined "version range".

- 3. Click Finish.**

The updated cartridge API model file is saved to the location set as output path.

Exporting Change Reports

To export an API change report to a JavaDoc-like HTML set:

- 1. Open the Intershop Studio file export wizard.**
From the menu, select File | Export.
- 2. Expand the Intershop API tree entry, select Export the API of cartridges, and click Next.**
This displays a list of available cartridges.
- 3. Select the intended cartridges, and click Next.**
This displays the Export Type page.
- 4. Select API Changes and click Next.**
This displays the Export Properties page.
- 5. Specify the output path.**
Specify a correct path or browse for the target location.
- 6. Click Finish.**
The API change report is immediately exported as a JavaDoc-like HTML set to the specified location.

Creating Migration Reports

To export a migration-oriented change report that states the modifications of, for example, project-specific artifacts' dependencies:

- 1. Open the Intershop Studio file export wizard.**
From the menu, select File | Export.
- 2. Expand the Intershop API tree entry, select Export the API of cartridges, and click Next.**
This displays a list of available cartridges.
- 3. Select the intended cartridges, and click Next.**
This displays the Export Type page.
- 4. Select Migration Changes and click Next.**
This displays the Export Properties page.
- 5. Specify the output path.**
Specify a correct path or browse for the target location.
- 6. Click Finish.**
The migration change report is immediately exported as a JavaDoc-like HTML set to the specified location.

Command Line API Tools

The API tools are also available as command line tools and may be used without the Intershop Studio GUI, for example, included with build scripts to automatically create an API model upon building a cartridge. Generally, the command

line versions of the API tools are called using *IntershopStudio.exe* using the **-application** switch and the intended parameters. For details, see the following sections.

API Definition Exporter

The API definition exporter can be used for exporting the API of multiple cartridges at the same time. The output may be API models, an API documentation, or a change report. By default, the exporter also analyses the dependencies.

The command line call for the API definition exporter is (all on a single line)

```
IntershopStudio.exe
  -application com.intershop.enfinity.studio.emf.api.APIDefinitionExporter
```

plus the intended parameters. The following table lists the available parameters.

Table 135. API definition exporter parameters

Parameter	Usage	Description
-exporter	required	Specifies the exporter to be used for generating the export output. The available options include docExporter (exports an HTML documentation), diffExporter (generates an HTML change report), modelExporter (exports the API model file), and dependencyExporter (generates an HTML migration report).
-cartridge	required	Specifies the cartridge ID of the cartridge(s) to be processed. Multiple values are separated by comma. Note: Can be omitted if -all_cartridges is used.
-all_cartridges	optional	Forces the tool to export the API definitions of all registered cartridges. Overrides the settings of the -cartridge parameter.
-output_path	optional	Allows to define an output directory other than set in the Intershop Studio workspace preferences (see <i>Configuring API Tools</i>).
-changes_only	optional	Filters for modified items. By passing version limits as arguments, the result can be further restricted to a version range (first argument = lower bound, second argument = upper bound).
-no_dependencies	optional	Forces the tool not to compute the dependencies between the artifacts.
-depends_on_changed	optional	Filters for items that depend on modified items. By passing version limits as arguments, the result can be further restricted to a version range (first argument = lower bound, second argument = upper bound). The parameter is ignored if the dependency analysis is switched off, i.e., -no_dependencies is used.
-data	optional	Defines the path to your workspace, not required if Intershop Studio has set a default workspace.
-nosplash	optional	Hides the Intershop Studio splash screen at startup.
-help	optional	Displays the help text.

API Definition Updater

The API definition updater always works only on one cartridge and is optimized to be used in a build process. It can handle renamed and split cartridges. The output is an API model.

The command line call for the API definition updater is (all on a single line)

```
IntershopStudio.exe
  -application com.intershop.enfinity.studio.emf.api.APIDefinitionUpdater
```

plus the intended parameters. The following table lists the available parameters.

Table 136. API definition updater parameters

Parameter	Usage	Description
-cartridge	required	Specifies the cartridge ID of the cartridge to be processed.
-changes_only	optional	Filters for modified items. By passing version limits as arguments, the result can be further restricted to a version range (first argument = lower bound, second argument = upper bound).
-output_path	optional	Allows to define an output directory other than set in the Intershop Studio workspace preferences (see <i>Configuring API Tools</i>).
-split	optional	Enables the correct artifact path handling upon updating an API model in case a cartridge has been renamed or split.
-api_location	optional	Specifies a file that contains a previous version of the API model. Required if -split is used.
-set_to_deleted	optional	Can be used together with -split to tell the tool to mark the old API model as deleted.
-data	optional	Defines the path to your workspace, not required if Intershop Studio has set a default workspace.
-nosplash	optional	Hides the Intershop Studio splash screen at startup.
-help	optional	Displays the help text.