The DASL Language
Programmer's Guide and Reference Manual

Bob Goldberg

The DASL Language
Programmer's Guide and Reference Manual

Bob Goldberg

SMLI TR-2005-128 January 2005

Abstract:

This report provides an intuitive description of the DASL[1] application modeling language, followed by a formal language specification. DASL was originally developed as part of the Ace research project at Sun Microsystems Laboratories to bridge the gap between high level application modeling languages, such as UML, and the current implementation languages and middleware in which applications are written, such as Java™, J2EE™, and evolving middleware based on web services. The primary goal of the research was to seek a sweet spot between the elegant abstractions possible in UML and the implementations one can express in Java, by creating a complete application modeling language that captures the application's semantics completely and globally. The resultant modeling language is effectively a domain-specific language for a very large domain of distributed business applications. At its core, it is a textual language that, like Java and most 3GLs, is easily read and understood by people. The textual language is tied to a well-defined meta-model that supports a graphical, UML view. The DASL deployment engine converts an application "model" directly into executable code, bypassing hand coding of an implementation "model." DASL vastly simplifies the process of storyboarding, prototyping, and implementing these applications by placing the focus on business semantics, the "what", rather than the implementation details of a particular architecture, the "how."

In essence, DASL technology is a practical realization of the MDA (Model Driven Architecture) vision to "*separate business or application logic from underlying platform technology"* and thus "*insulate business applications from technology evolution*."

Sun microsystems

16 Network Circle
Menlo Park, CA 94025

email address:
bob.goldberg@sun.com

1. DASL stands for *Distributed Application Specification Language*, and is pronounced ***dazzle***.

The DASL Language

Programmer's Guide and Reference Manual

Bob Goldberg

Sun Microsystems, Inc.

First Edition

January 26, 2005

This document reflects the proof-of-concept implementation of the DASL compiler and deployment engines.

Table of Contents

Programmer's Guide

Reference Manual

Acknowledgments

I am very grateful to Bob Sproull for funding our initial work in Sun Labs, and to Jim Mitchell for his unwavering support of the Ace Project within Sun Labs. Designing a new language takes time, and convincing people to use it can take even longer. It is only with Jim's long term support that we have been able to successfully transfer this technology to the commercial side of Sun.

I am also very grateful to John Crupi and the architects in Sun Professional Services for taking the bold and unprecedented step of transferring Sun Labs technology directly into Sun Professional Services.

My deepest appreciation and thanks go to Bruce Daniels, with whom I have shared, through several incarnations, the vision realized by the Ace Project. In addition to his dedication to marketing the project, and his many technical contributions, Bruce was also the first user of our new language to write real world applications. He cheerfully contended with early bugs and misfeatures of the development environment, some of which would periodically erase his programs without warning!

I want to credit and thank my fellow Ace team members, Yury Kamen and Ali Syed, for their many contributions to the the project and language described herein. Credit also goes to the former members of the Ace team, including Peter Yared, Syed Abbas, and Raman Kumar. In a previous incarnation of the project, Ludovic Champenois first suggested a stopgap textual language for bootstrapping purposes that evolved into the BOS part of the language. His contributions, along with those of the other members of the former ExpressApp team, influenced the directions we chose to pursue, as well as those we eschewed.

I value the time and effort Mick Jordan put into thoroughly reading several drafts of this material. His extensive and insightful comments helped me identify the need for describing our novel programming model and corresponding network virtual machinewhich need to be described carefully to fully grasp and understand the language. His other good suggestions are too numerous to list here.

Obviously, the creation and widespread use of the Java Programming Language, its virtual machine, and its extensive API's, provided us with fertile ground on which to build our application specification language. We borrowed heavily from Java for the expression and statement syntax, leveraging the fact that Java syntax is instantly familiar to many programmers, and we used Java and J2EE™ in the deployment engines as a platform on which to execute the resultant application.

I would like to thank James Gosling for calling the attention of the outside world to our work at JavaONE, and for the ideas he shared with us on editing the Java™ expression parse tree as a way of realizing implementation-independent syntactic sugar. We use these ideas to translate user-defined methods in DASL into pure Java code in our current deployment engines.

A thank you goes to Shauna Pickett-Gordon for proofreading an earlier draft and providing useful comments. Jeanie Treichel proofread the final manuscript for English usage and formatting consistency, and provided cheerful support for making it a technical report. Ultimately, the responsibility for all errors and omissions is mine.

Bob Goldberg
Menlo Park, California
January, 2005

Abstract

This report provides an intuitive description of the DASL[1] application modeling language, followed by a formal language specification. DASL was originally developed as part of the Ace research project at Sun Microsystems Laboratories to bridge the gap between high level application modeling languages, such as UML, and the current implementation languages and middleware in which applications are written, such as Java, J2EE, and evolving middleware based on web services. The primary goal of the research was to seek a sweet spot between the elegant abstractions possible in UML and the implementations one can express in Java, by creating a complete application modeling language that captures the application's semantics completely and globally. The resultant modeling language is effectively a domain-specific language for a very large domain of distributed business applications. At its core it is a textual language that, like Java and most 3GLs, is easily read and understood by people. The textual language is tied to a well-defined meta-model that supports a graphical, UML view. The DASL deployment engine converts an application "model" directly into executable code, bypassing hand coding of an implementation model. DASL vastly simplifies the process of storyboarding, prototyping, and implementing these applications by placing the focus on business semantics, the "what", rather than the implementation details of a particular architecture, the "how."

In essence, DASL technology is a practical realization of the MDA (Model Driven Architecture) vision to "*separate business or application logic from underlying platform technology*" and thus "*insulate business applications from technology evolution*".

1 DASL stands for *Distributed Application Specification Language*, and is pronounced ***dazzle***.

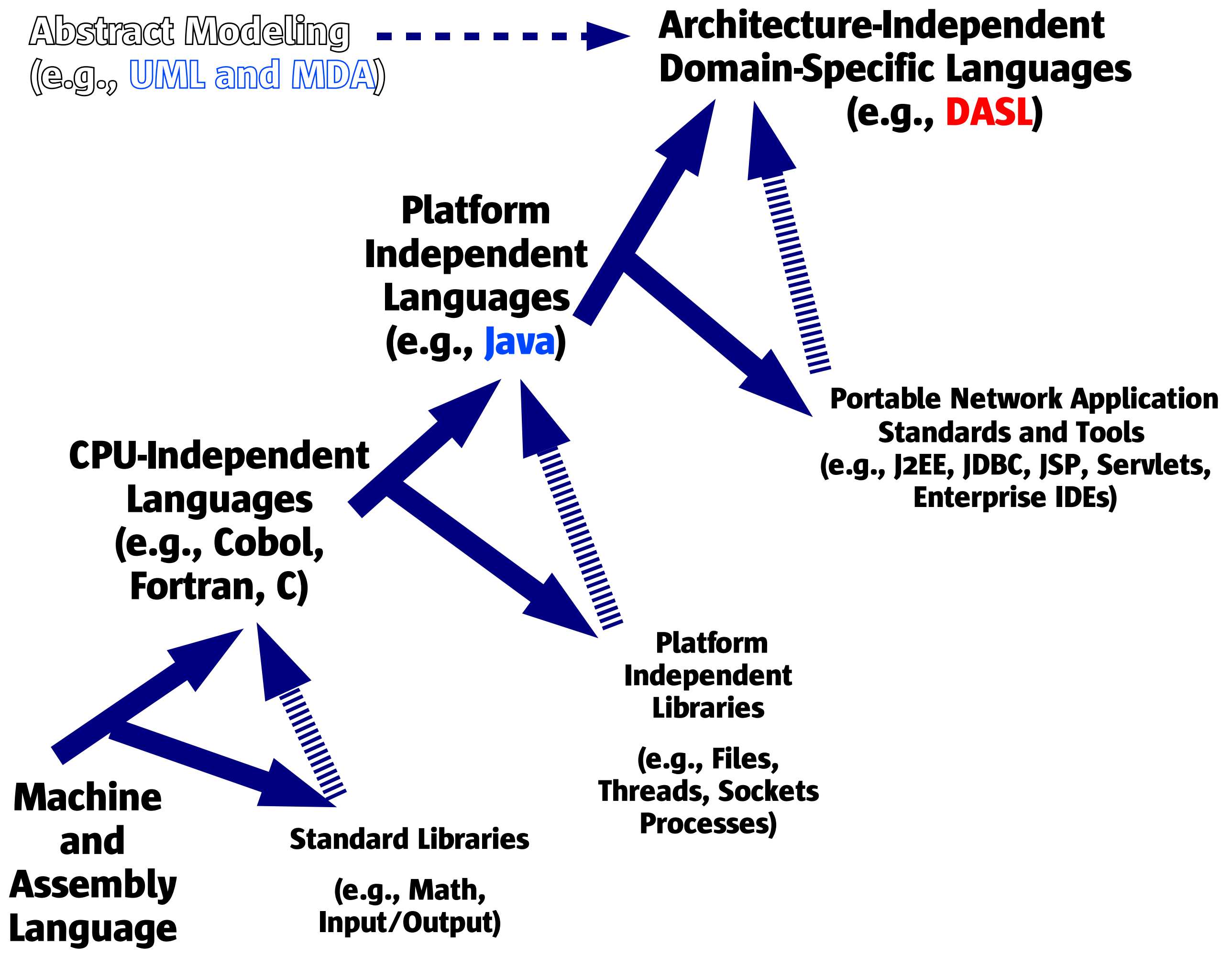
Programmer's Guide

1 Introduction

1.1 Evolution of Application Programming Languages

Sir Isaac Newton wrote that he stood on the shoulders of giants. While our work does not merit the stature of Sir Isaac's, we too stand on the shoulders of language designers before us. A brief history of application programming language advances will help place into perspective the step we have taken.

The first computer applications were written in machine language (refer to the bottom left of the figure below). By entering the numeric instructions defined by the CPU, a specialist could program the computer to do useful tasks. The second generation languages were assembly languages, which relied on an "assembler" program to convert instructions and variables expressed symbolically into numeric instructions and addresses. Assembly languages were less tedious for an application writer to use than purely numeric instructions, but they were still specific to a particular CPU's instruction set.


Abstract Modeling
(e.g., UML and MDA)
Architecture-Independent Domain-Specific Languages
(e.g., DASL)
Platform Independent Languages
(e.g., Java)
Portable Network Application Standards and Tools
(e.g., J2EE, JDBC, JSP, Servlets, Enterprise IDEs)
CPU-Independent Languages
(e.g., Cobol, Fortran, C)
Platform Independent Libraries
(e.g., Files, Threads, Sockets Processes)
Machine and Assembly Language
Standard Libraries
(e.g., Math, Input/Output)

Application programming by domain experts took a giant leap in the development of a 3rd generation of computer languages, such as **COBOL**, **Fortran**, **Algol**, **C**, and **Lisp**. These higher level languages isolate the programmer from the details of the particular CPU, as well as providing a standard interface to common subroutine libraries found on each system. The 3rd generation languages make it possible for domain experts to express calculations and simple input and output at a level that reflects the domain-specific problem they want to solve, without including system-related details. These languages can also be less tightly tied to a particular CPU, so application programs can more easily be shared with other domain experts using different hardware.

In the computing waves that followed, computers were increasingly used for communication, database manipulation, and other non-numeric applications that relied more heavily on services of the operating system, its file system, and separate database management and transaction processing systems. For many years, portability of such applications among operating systems was problematic. To help the programmers deal with platform differences, libraries were developed to manipulate more services than the original 3rd generation languages typically provided, such as files, threads, and processes.

There was also a not-so-successful attempt by software vendors to define 4th generation languages. Most of these so-called 4th generation languages provided direct access to a vendor-specific database system, as well as a mechanism for creating screen-based user interfaces, but their higher level abstraction was computationally incomplete, and the languages were tied to vendor-specific proprietary frameworks.

Addressing the complexity of programming in general, computer science research into languages has provided some interesting alternative programming models, such as the object-oriented approach. During the 1990's, the C++ language became very popular for commercial application programming. Portability among operating systems was still an issue, along with the need for separate compilation for each platform, tedious and error-prone memory management,[2] and code safety[3] issues.

The Java™ programming language is an object-oriented 3rd generation language that includes a programming model that virtualizes the entire computing platform, allowing Java programs that use many advanced operating system services to be compiled once and run anywhere. Java is highly portable, has automatic memory management compared to C++, and provides a high level of code safety.

As with other 3rd generation languages, the Java computational model is a single computer process. Java defines portable ways of *explicitly* programming distributed applications, such as via RMI or J2EE™, but it requires the programmer to be aware of the distribution and write code to interface with the other processes that make up the complete application.

2 C++ memory management lacks automatic garbage detection and collection, and consequently most large programs contain difficult to detect bugs caused by referencing memory that has been freed manually, or freeing memory twice.

3 By "code safety", we have in mind issues such as wild pointers that accidentally clobber memory, as well as vulnerability to viruses and worms when code is imported into a server process and then executed.

1.2 Architecture-Independent Application Modeling

The Ace team chose to take Sun's vision, ***the network is the computer***, to the next level in terms of the evolution of application programming at the enterprise level, by defining a complete modeling language that expresses at a higher level the formal semantics of a business application without requiring the programmer to think about the software or hardware architecture, such as the distribution details. The key observations that led to the definition of an application modeling language were:

a. Most of the coding of distributed business applications (often 95%) deals with the distribution and scalability of the application, and only a small part (perhaps 5%) deals with the application logic.
b. If one could provide a way to express the global semantics of the application precisely and completely enough, the distribution details could be filled in automatically by a deployment engine.

Due to the nature of distributed business applications, we realized that the modeling language must express the desired application semantics in novel ways, including the notion of high level *application transactions*, including atomic, multi-step operations. Furthermore, the model must capture implicitly the degree of synchronization required across a network to implement various constructs. Given these requirements, the compiler for the modeling language must deploy the application by injecting the necessary distribution code to make it work in a variety of distribution architectures and software stacks, including either 2-tier or 3-tier architectures using J2EE, as well as other present and future mechanisms that evolve.

There are many ways in which prior implementation languages failed to capture sufficient semantics to automate distribution. For example, in the C language, a *char* * is often used to represent a string terminated with a zero byte, but it might also refer to single character. Thus, a C program does not have enough declarative semantics to describe how a string (or an array) should be passed to another process.

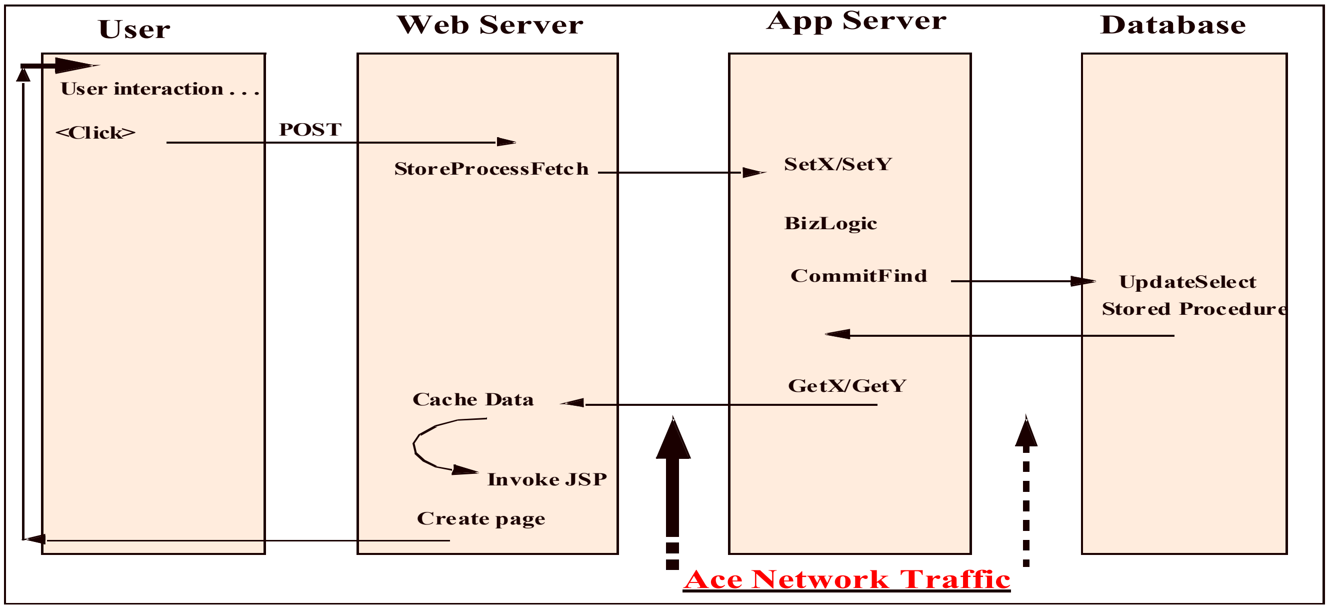
While implementation languages such as Java have corrected these simple problems, other more pernicious ones remain. For example, consider a classic distributed application that retrieves some objects from another process and displays them, allows a user to edit them, and then sends the changes back to the other process, combining the changes with those made by other users. The Java programming language provides the power to implement such an application in an endless variety of ways, but it does not allow the programmer to declare global semantic information about the way the two processes interact, how to store its objects persistently, how to synchronize copies of distributed objects, what happens if the synchronization fails, and how to group a sequence of changes into an atomic, high-level application transaction.

1.3 A New Approach to System Architecture

DASL does not make system architects obsolete. On the contrary, system architects can leverage this technology to encode their expertise of improved design patterns by writing new DASL deployment engines.

Instead of using system architects to design applications one at a time, the DASL approach lets system architects encode their specialized knowledge of distribution patterns by writing deployment engines for the DASL language. Application writing is much faster because 95% of the code is inserted automatically by the deployment engine. The remaining 5%, provided by the application writer, concerns itself with the application semantics rather than distribution semantics.

The figure below illustrates the architecture of an application that is deployed in 3 logical tiers. Because DASL represents the application semantics at a high level, the deployment engines have enough information to optimize various aspects of the deployed application. For example, the figure below illustrates how one of our early deployment engines was able to optimize the number of calls from one process to another so that each user-visible interaction that causes a new page to be displayed in the uesr's browser requires only a single round trip from the user's browser to the web server, then to the application server, then to the database, and then back to the user.



Whenever a system architect designs a new deployment engine for DASL, all existing applications can be redeployed using the new pattern. Instead of spending time refactoring existing applications by hand, the architect can test the new pattern using existing applications. Thus, this new paradigm can be viewed as a productivity tool for the architects themselves.

1.4 Research Results and Artifacts

The primary goal of the Ace research was to seek a sweet spot between the elegant abstractions possible in UML and the implementations one can express in Java. We were seeking a practical realization of the MDA (Model Driven Architecture) vision to "*separate business or application logic from underlying platform technology*" and thus to "*insulate business applications from technology evolution*".

We have developed a formal specification for a language that meets the above requirements, which we call DASL. The DASL specification is an open document available to the software and business development communities.[4]

We have also produced a meta-model and parser for DASL, as well as a compiler that accepts pluggable deployment engines. We have implemented several deployment engines that can deploy DASL applications in a variety of architectures. New deployment patterns may readily be written by knowledgeable architects. The pluggable deployment engine thus encapsulates the expert deployment knowledge of the architect who developed it. We view our research results as a proof of concept for this new way of writing applications.

Defining a language that is both semantically precise and understandable is an art, and while our language is not perfect and still needs refinement, we believe that the DASL language itself is of value to the computing community, as it has achieved the goal of expressing the semantics of a broad domain of executable applications very precisely, yet without any architecture-dependent details. We are hopeful that this language, or some improved variant of it, might someday become the preferred means to write most new business applications.

Finally, our method of injecting the necessary distribution details at deployment time is novel and worthy of note. The synergy of the DASL language and intelligent deployment engines has been very high. In particular, the research project that produced DASL has, to date, produced approximately 30 U. S. patent applications.

1.5 How to Read the Programmer's Guide

This language manual describes the DASL language. It is divided into two sections, a Programmer's Guide and a Reference Manual.

Chapter two of the Programmer's Guide describes the environment in which a DASL application runs. Chapter three is an introductory tutorial to the language. It describes the *HelloWorld* application, and the *RememberMe* application. Chapter four describes a simplified but representative e-commerce application, *PurchaseOrder*.

The remaining chapters comprise the Reference Manual. They describe the DASL language more formally, driven by the language constructs rather than example applications. The BOS and AUS sub-languages are each described, along with the standard DASL query language, available for defining query methods.

Appendix A describes how to invoke the DASL compiler and deployment engines using the command line interface. The complete syntax of the language, expressed in BNF notation, is summarized in Appendix B. Other appendices describe the current deployment engines and some exciting enhancements to the language that are planned.

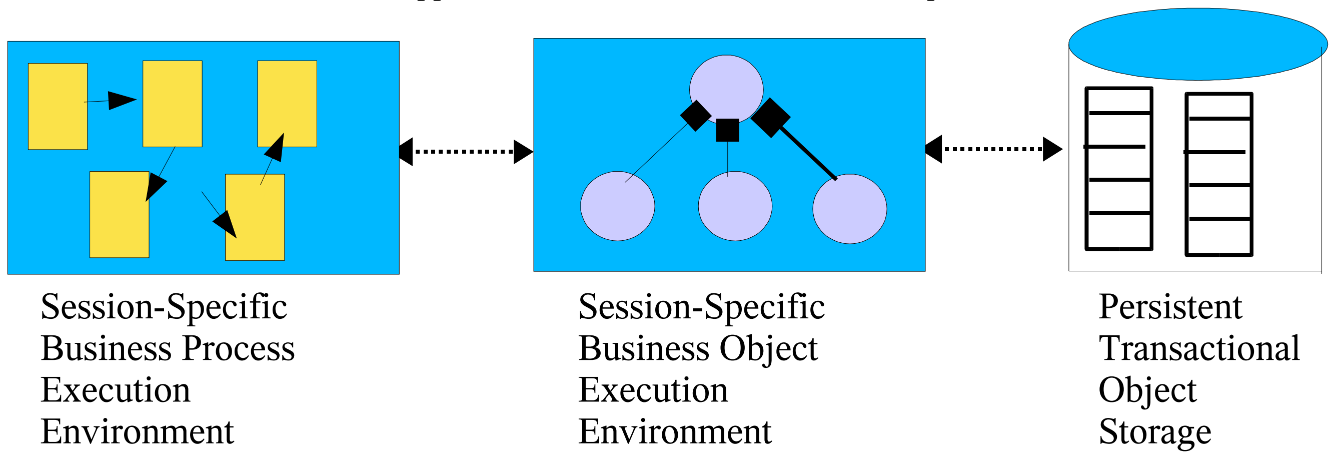
4 If not for history, we might call DASL a 4th generation language. Alas, the terms 4th generation and 5th generation language have been used for technology that never achieved the success of 3rd generation languages.

2 The DASL Programming Model

Every programming language provides a programming model, a level of abstraction above the physical hardware of a computer in which to precisely define the language semantics. The DASL application modeling language goes a little further and provides a level of abstraction above the implementation and distributed deployment of the application on multiple computers. The DASL virtual programming model allows domain logic to be expressed fairly simply, and then guarantees that precise application semantics will be faithfully carried out during the execution of the application, regardless of the way it is deployed.

2.1 The DASL Virtual Programming Model

The schematic diagram below shows the structure of the distributed environment in which every DASL application is defined. The application is described in a disciplined way within this environment, so the DASL compiler has all the semantic information it needs to deploy the application in a variety of distributed configurations. While applications are not required to use all three components of the model, most non-trivial DASL applications make some use of each component.


Session-Specific Business Process Execution Environment
Session-Specific Business Object Execution Environment
Persistent Transactional Object Storage

As illustrated in the figure above, the DASL programming model captures three distinct aspects of the application:

- The center component in the above figure supports the execution of business objects (entities) and relationships, along with their behavior. This component can be thought of as similar to the Java Virtual Machine, though the objects defined at this level correspond to high level business entities, rather than arbitrary objects. The lifetimes of these objects can be declared as either transient (like Java objects) or persistent. The state of these objects is tied to a particular session of the application.
- The component to the left in the preceding figure defines a business process, i.e., a graph of states, connected by transitions from one state to another.[5] States are grouped into

5 As of this writing, the business process is restricted to a single session task, i.e., a graph of states of user interaction,

high-level application transactions (not shown), thus defining the scope and atomicity of changes made to the business objects (in the center component) as the application moves from one state to another.

- The component to the right in the preceding figure supports the persistence of business object instances in a reliable, transactional persistent store. The precise mechanism by which the business objects are stored persistently depends on the deployment; however, regardless of deployment, the state is reliably stored with predictable high level transaction semantics.

These three components may be thought of as distinct execution environments; however, the actual deployment may group them into a single execution environment, or it may partition parts of the three components among some arbitrary number of physical computing environments. Because the code to support the distribution is inserted by the deployment engine, the DASL programmer need not be concerned with the eventual runtime deployment of the application.[6]

2.2 Terms and Concepts

2.2.1 Business Processes and Tasks

A *business process* is a long-lived series of events, represented as a graph of states that are connected to each other by arrows indicating subsequent states that follow from a given state. Business processes have a lifetime that persists beyond any particular process or user session with the application. A reasonable example of a business process is the procedure of hiring a new employee at a large company, requiring the satisfaction of many steps performed by many individuals in the company over a period of days or weeks. There may also be times when several individuals within the company proceed with the process in parallel.

A *business task* is a subset of a business process, tied to a single session between a user and an application program. An example session is bringing up a mail reader and reading one's mail. The task consists of viewing the message headers, selecting a message to view, perhaps replying to messages, and filing them away for later. The entire task takes place within a single session.

The DASL modeling language currently manages business tasks. The specification of business processes is planned as a fairly natural extension to the language.

2.2.2 Application Sessions

An application session begins when an instance of an application is started, and the session ends when the application instance terminates. Sessions are typically started when a user clicks on a link in a web

connected by transitions from one state to another, with scope tied to a particular session of the application.

6 Perhaps you are wondering why we have chosen to complicate the world with the left- and right-hand components. Indeed, wouldn't it be simpler to program using a single VM, such as the Java VM? The short answer is that expressing applications with these additional components provides the DASL compiler with the global intent of the application, so that the deployment engines can safely distribute and partition the application across process boundaries, knowing when to synchronize the data held in each tier, and how to ensure that transactions remain atomic. These points will become clear in the next chapter.

page, or explicitly types into his web browser the URL of the application to be run.[7] Termination of a session is most often a result of application logic initiated by a user gesture, though it can also occur due to hardware and/or software failures of various kinds.

A session itself is not persistent, and typically depends on physically running hardware to maintain its complete state.[8] However, the application may be written to explicitly store some aspects of the logical state of a session, as well as other persistent information, in the persistent storage layer.[9]

For example, the application may choose to store a shopping cart in persistent storage, or alternatively, the application may choose to make the shopping cart a non-persistent session variable, in which case the items in the cart will be forgotten when the session terminates.

2.2.3 Business Object Execution Environment

Business objects that are actively used within a session live in the business object environment during that session. Within a session, the business object execution environment provides

- the Java program execution model,[10]
- a connection to the persistence layer, and
- predefined life cycle operations on persistent objects.

If a session failure occurs, the currently active business objects lose their current state, so any uncommitted changes made to them within the session are lost. Previously committed changes to the business objects are reliably stored in the persistence layer, and will thus be preserved across sessions.

2.2.4 Business Task Execution Environment

The business task part of the application executes in its own environment, which is active only within the scope of a session. This environment is conceptually separate from the business object layer, and also conceptually separate from the persistent storage layer.

Within each session, the business task execution occurs as part of a single transaction.[11] At any given time, that transaction is either ***isolated*** from the business objects and persistent store (during user interaction), or ***synchronized*** with the business objects and persistent store (between user interactions).

During the isolated period (i.e., during user interaction), business objects are available for display and modification of their attributes and relationships. However, business logic may not be invoked at this

7 When business processes are implemented, it will be the case that sessions can also start in response to events.
8 Some deployment architectures may provide failover, but from the point of view of the DASL application, this is indistinguishable from deploying on more reliable hardware that does not have failover capabilities.
9 When business processes are added to DASL, they will automatically persistify the state of the process so that the state lives beyond the session.
10 Currently, some restrictions of the J2EE programming model are imposed by the J2EE deployment engine. In particular, business objects are restricted to a single thread. Deployment engines that do not use J2EE would not be subject to this restriction.
11 The user interface can be written with separate threads to support conceptually non-transactional operations, such as help screens, which do not access data modified in other threads. The current implementation does not support multiple threads with their own distinct transaction.

time.

During the synchronized period (i.e., between user interactions) business object methods and other computation related to business objects may be freely invoked.

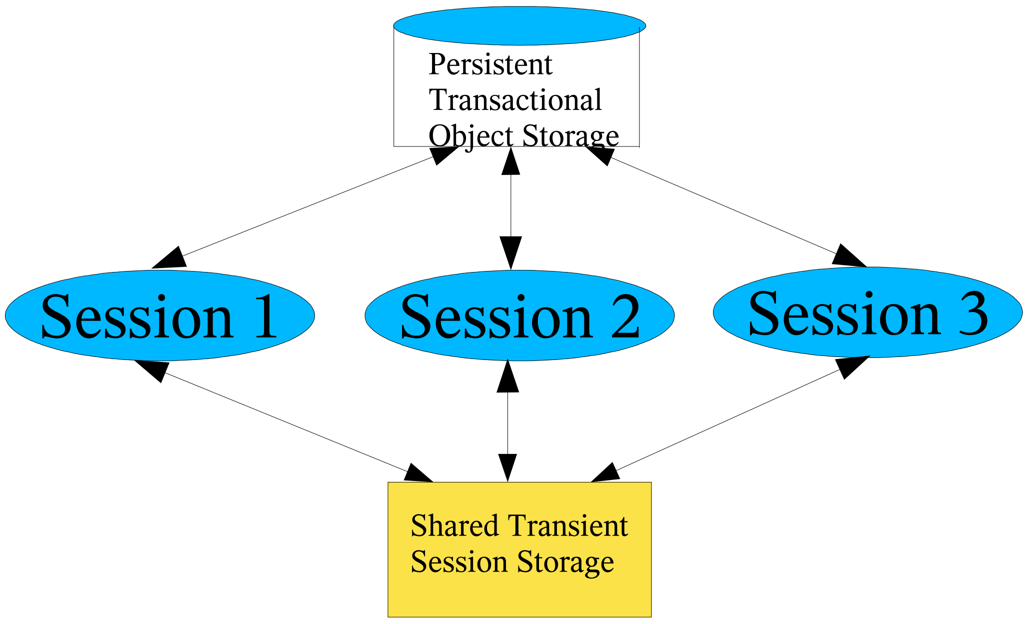
2.2.5 Persistent Store

The persistent store is the only part of the system that exists independently of a session.[12] It is shared by all sessions that are active, and it supports transactional semantics.

For each business object, the persistent store keeps an implicit collection of all instances of that object, called the object's **extent**.

Objects within the persistent store are identified by a visible but immutable value (or tuple of values) known as the object's *primary key*.

2.3 Sessions and Session Interaction


Persistent Transactional Object Storage
Session 1
Session 2
Session 3
Shared Transient Session Storage

The DASL programming model supports multiple sessions of the same or different applications running at the same time. Simultaneous access and update of the persistent store by different sessions is regulated by use of application transactions.[13]

For the most part, each session behaves in an isolated fashion with respect to all others, communicating

12 In the future, business process state will also exist independently of a session.
13 In many of its deployment engines, DASL implements application transactions using optimistic concurrency, by invoking a series of pessimistic DBMS transactions to emulate a single, long-lived application transaction.

only via the persistent object store. However, occasionally it is desirable for application instances in more than one session to share information with each other. To achieve this sharing the application may declare shared variables in the business task.

2.4 Summary

The DASL application execution environment leverages the ubiquitous Java programming language to define the transient environment in which business objects and business logic are activated. But since the full semantics of enterprise applications depends critically on the *persistent store* and the *transactional environment* in which *multiple sessions of the application must cooperate*, the DASL application execution environment integrates the *persistence and transactional* aspects of the application with the transient business object execution environment.

The business task component specifies the interaction between an instance of the application and its user. This choreography is expressed as a set of states and transitions from one state to another. The states capture the semantics of different screens or web pages that make up the application. The transitions capture the semantics attached to the actions that take the user from one screen to the next. Often, such actions cause business objects to be modified and/or business methods to be invoked. Application transactions defined at the business task level ensure consistency of the application's view of the persistent store.

DASL imposes a new discipline on the interaction between business logic and business process. This discipline enables the language to capture the full semantics of the application in a way that is orthogonal to distributed programming, and the deployment engines to implement the application reliably and efficiently in a variety of distributed environments. Thus, domain experts with modest Java programming skills can easily write and deploy DASL applications. Expert programmers can manipulate the DASL specification of a complex application, and understand it in its entirety.

The DASL compiler deploys a complete, working, scalable, and efficient application from the DASL program specification. It adds all the necessary distribution details to ensure that the application semantics are reliably reproduced in the target environment which is selected at compile-time. The DASL compiler also deploys a default presentation, which the application writer may modify.

3 A Quick Tour of the DASL Language

This chapter is a quick tour of the DASL modeling language to get you started writing DASL applications. We tour the language features quickly, without deep-diving into the complete language specification.

3.1 Language Components: BOS, AUS, and OQL

The DASL language provides specialized sub-language components for each of the 3 distinct layers of the DASL execution environment.

- **DASL/BOS** is the business object specification, i.e., the specification of the persistent business objects, relationships, and methods that are manipulated by the application. It is easy to learn if you already know Java.[14] It provides a higher level abstraction than Java, using a Java-like syntax for method bodies and expressions.
- **DASL/AUS** is the application usage specification, i.e., a set of states and transitions that choreograph how the application uses the business objects in one or more BOS. The AUS uses a Java-like declarative syntax, and a Java-like scripting syntax for invoking business object methods during transitions. It also supports the same Java-like syntax as DASL/BOS for defining method bodies.[15]
- **DASL/OQL** implements a clean subset of the OQL query language for querying and manipulating the persistent store, where OQL is a standard defined by the ODMG subgroup of the Object Management Group.[16] By design, OQL is compatible with a subset of the standard relational query language, SQL. Notice that the specification of queries is required only for defining application tasks that specifically require them, such as retrieving a set of objects based on their attributes. The queries required for simple applications are deployed automatically by the DASL compiler.

The subset of DASL/AUS that expresses the application business task as a series of states and transitions leverages UML state transition diagrams, which have been used for many years and are easily understood. In a DASL application, you can think of states as corresponding to application screens, and transitions as what happens when the application moves from one state (or screen) to another when the user presses a button or makes some other gesture.

As a further aid to application writers, the DASL development tools include a graphical editor[17] with which you can create, view, and edit DASL applications graphically. For example, you can create business objects by clicking on a canvas, and then create relationships between them by drawing lines

14 The textual forms of the BOS and AUS are very Java-like, and the constructs that "look like" Java generally "work like" Java as well. In most cases, they should be easily understood by those familiar with Java.

15 The ability to define methods directly in the AUS is not implemented as of this writing. Instead, create a transient business object within the BOS and define the desired method as factory methods of that object.

16 For information about the Object Management Group, ODMG, and OQL, see http://www.omg.org.

17 The graphical editor is currently implemented as a plugin module to NetBeans™.

between the related objects. You can also create states and transitions between them graphically. The graphical editor allows you to visualize an entire BOS or AUS on the screen.

NOTE

> The use of the DASL graphical editor will not be described in this manual. However, we will illustrate programs using the UML diagrams produced by the editor.

3.2 The Presentation Layer

The DASL language is focussed on both ***business logic*** and ***business tasks***. To achieve a standard look and feel, the application writer must ensure that the application flow and logic conforms to the standards. For repeated sequences such as a standard Login screen, a standard library of DASL tasks can be created and invoked from applications.[18]

The DASL language does not have a sub-language for controlling the "beautification" aspects of presentation, such as placement of fields on the screen, or the specific choice of widgets, backgrounds, fonts, colors, or icons. Instead, by selecting among the available presentation deployment engines, the application writer can vary the style and the framework used to implement the default presentation.

For example, a deployment engine that implements a JSP client using the Apache Struts framework is currently available. Eventually, it is our intention that these presentation deployment engines will be modifiable by experienced programmers to produce a default presentation that conforms to a desired corporate style.

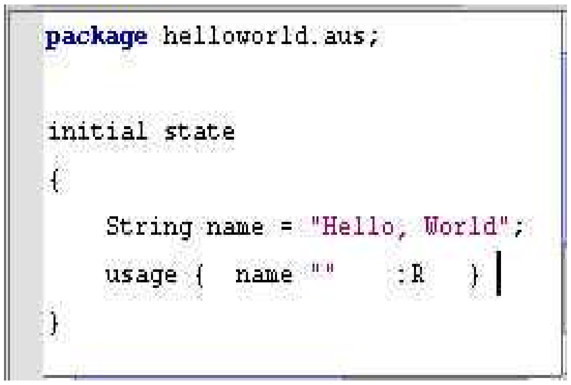
Regardless of whether a standard presentation engine or a custom engine is used, the default deployed presentation can be edited using presentation editing software to "beautify" it (e.g., *Dreamweaver* might be used for editing JSP pages). The rule of thumb is to get the application task logic right before worrying about the beautification, which is "finish work".

3.3 The HelloWorld Application

We will now take a whirlwind tour of the DASL language by examining two small sample programs. The first sample program in many languages prints "Hello, World" To accomplish writing a message in DASL we do not need any business logic or business objects, so no BOS is required. All we need is an AUS with a single state.

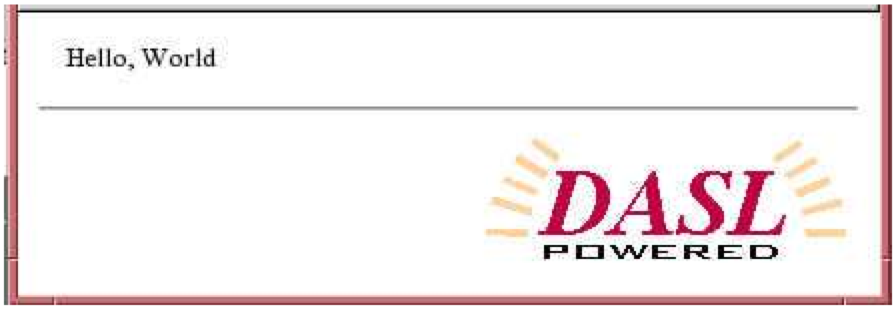
The AUS for HelloWorld that follows is in textual form. Its initial state creates the string to be displayed, and the usage section declares that this string, *name*, is to be presented in read-only form (the user cannot change it) and with no label.

18 Composition of tasks has been defined (see Appendix G) but not yet implemented in DASL. At present it is possible to copy a common AUS task such as a standard Login task into multiple applications and thus reuse it.



```
package helloworld.aus;

initial state
{
    String name = "Hello, World";
    usage {  name ""    :R   }
}
```

When the *HelloWorld* application is deployed, the DASL compiler creates a default presentation (GUI) based on the AUS. The precise look of the presentation depends on the presentation deployment engine chosen at deployment time. When the application is run, it displays the words “Hello, World”.



Hello, World

3.3.1 Exercise 1.1

Type in the HelloWorld AUS as shown above into the file *HelloWorld.aus* using your favorite programming text editor. Make sure the file is created in a folder that is separate from any other DASL applications. Define a DASL application that includes this AUS using the command:[19]

```
        dasl -create HelloWorld.dasl -add HelloWorld.aus
```

Finally, execute the application using the command:

```
        dasl HelloWorld.dasl -execute
```

3.4 The RememberMe Application

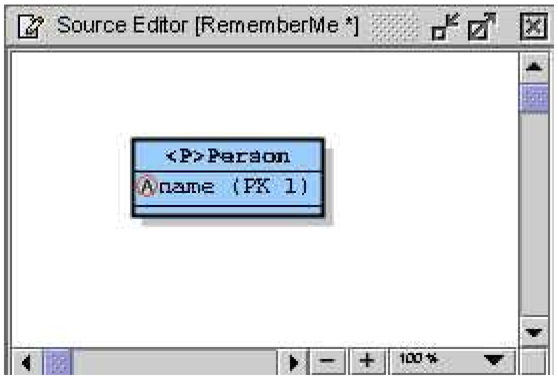
Our next sample application introduces the use of the BOS for defining persistent objects and business logic, and shows how the AUS can manipulate those objects. It also shows that the application can accept input and pass the input to the BOS layer. Because our little application remembers each user across invocations, we call it *RememberMe*.

19 You must have installed DASL on your system for this command to be found.

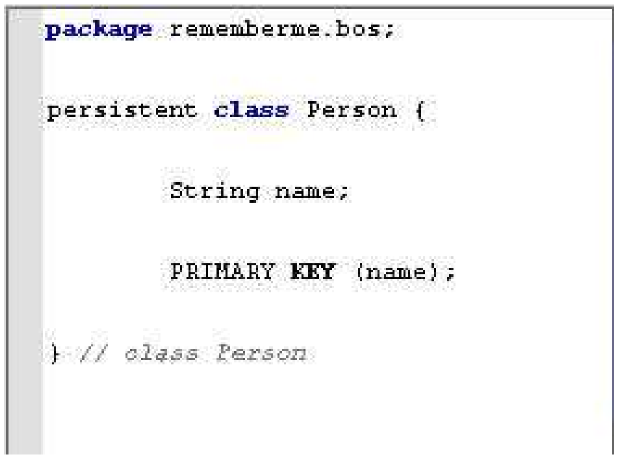
3.4.1 RememberMe BOS

The BOS for the RememberMe application defines the object the application will store persistently, and the business logic associated with that object. It consists of a single business object called *Person*, which stores just the name of a person. For this simple application, we assume that the person's name uniquely identifies the person, so the name is the primary key of the *Person* class.

The DASL toolset includes a UML graphical tool for displaying and editing business objects and their relationships. The tool displays the *Person* business object as shown below. By double clicking the object, more details about its specification can be viewed.



The complete BOS for the RememberMe business objects is shown below. It is stored in the file *RememberMe.bos*.



```
package rememberme.bos;

persistent class Person {

        String name;

        PRIMARY KEY (name);

} // class Person
```

As in Java, the *package* statement defines the namespace in which the objects in this BOS are defined.

The *Person* class is defined with the modifier *persistent*. Persistent objects exist in the persistent store and are transactional. Transient objects exist only as part of a session, and are not transactional.

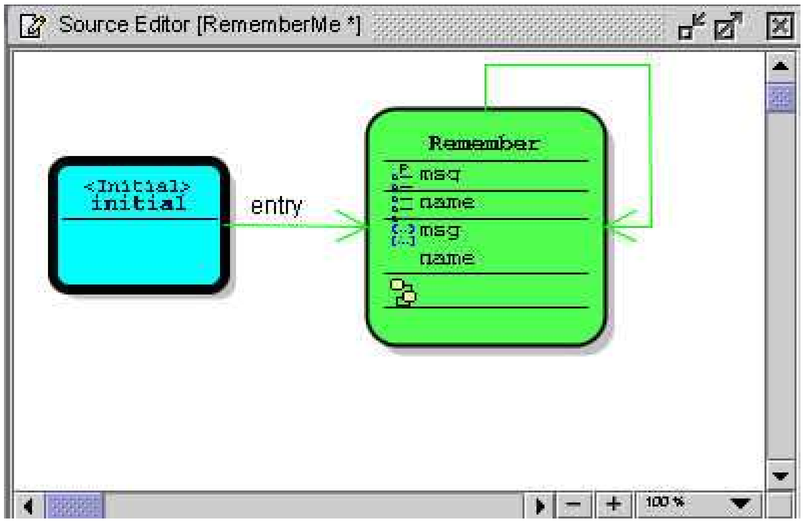
The *Person* class defines one attribute, *name*, whose type is String. Because the class itself is declared persistent, *name* is by default a persistent attribute, i.e., its value will be stored persistently for each instance of this class. Attributes of persistent objects may optionally be declared as transient, in which case they are not stored persistently. Attributes of transient objects are always transient.

The *String* attribute *name* is declared as the *primary key* of the *Person* business object, which means that it uniquely identifies instances of *Person* objects. Thus, there can be at most one *Person* object in the persistent store with a particular name, and an instance of *Person* can be located using *name*.

3.4.2 RememberMe AUS

The RememberMe AUS defines the steps that the the user of the application will follow, and the way the application uses the object(s) in the BOS. An AUS consists of atomic steps, which are called states, and the actions that the user may take at each step, which are called transitions.

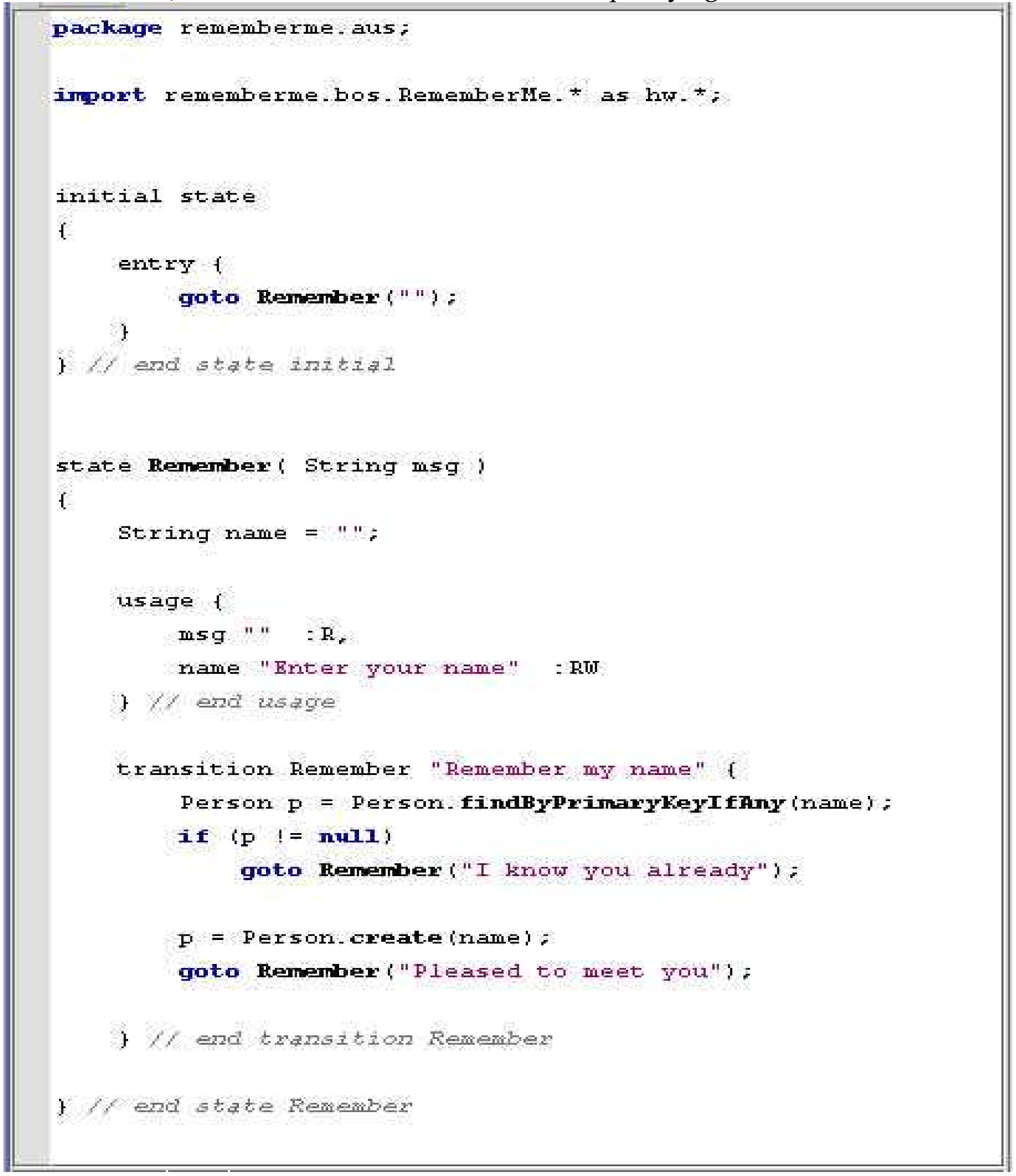
The DASL toolset includes a UML graphical tool for displaying and editing states and transitions. The UML diagram below shows that the application consists of two states: the *initial state*, where an application starts, and a state called *Remember* that displays a status message and allows the user to type in a name. The *Remember* state defines a transition, labeled "Remember", that will use the name typed by the user either to create a new *Person* business object, or else find the instance of the *Person* business object with the specified name in the persistent store.



The complete AUS corresponding to the above UML diagram is stored in the file *RememberMe.aus* and is shown on the next page.

As in the BOS, the *package* statement defines the namespace in which this AUS application will be made available.

The *import* statement defines a BOS (set of objects) that this application will use. The alias *hw*, local to this AUS, can be used to reference the business object classes of the imported BOS from within this application. If the names of those classes do not conflict with variables defined in the AUS or other imported BOS classes, the class names can be used without qualifying them with the alias.



```
package rememberme.aus;

import rememberme.bos.RememberMe.* as hw.*;


initial state
{
    entry {
        goto Remember("");
    }
} // end state initial


state Remember( String msg )
{
    String name = "";

    usage {
        msg ""   :R,
        name "Enter your name"   :RW
    } // end usage

    transition Remember "Remember my name" {
        Person p = Person.findByPrimaryKeyIfAny(name);
        if (p != null)
            goto Remember("I know you already");

        p = Person.create(name);
        goto Remember("Pleased to meet you");

    } // end transition Remember

} // end state Remember
```

One state in an AUS specification must be declared as the initial state. This is the state at which the application starts. The initial state need not be named unless some other state wishes to transition to it. In this example, the initial state contains nothing but a transition to the *Remember* state with an empty message.

The *Remember* state takes a single parameter, *msg*, of type *String*. The interface of a state is similar to the interface of a constructor in Java. It takes parameters, but is not declared to return a value.

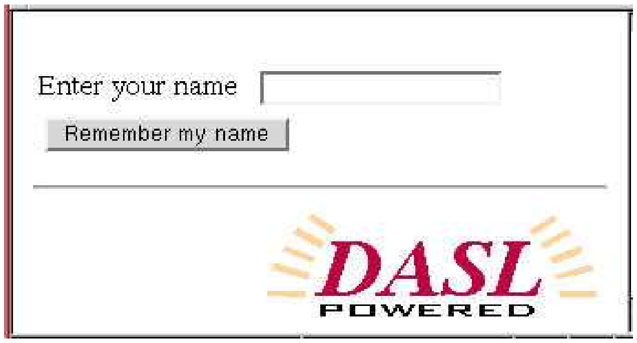
The *Remember* state declares a variable local to the state called *name*, which is initialized to the empty string, "", upon transition to the state. The usage section declares how the state will use the variables *msg* and *name*: *msg* will be presented read-only (no modification allowed), while *name* will be presented so the user can read the value and also write (modify) it. The value of the *msg* variable will be presented in an unlabeled field, and the value of the *name* variable will be presented in a field labeled, "Enter your name".[20]

The *Remember* state contains a single transition labeled "Remember". The transition script invokes the business logic associated with the transition. It uses the persistent class *Person* defined in the BOS to tell if the name has been seen before. It first tries to find an existing *Person* object with the name typed by the user, and if that succeeds, a transition is made to the *Remember* state with *msg* set to "I know you already." If a person with that name is not found, it creates a new *Person* object with the name typed by the user, and a transition is made to the *Remember* state with *msg* set to "Pleased to meet you." In the second case, the next time that same name is entered, an object with that name as its primary key will exist and thus the name will be remembered.

Note that within the AUS, invocation of BOS business logic may occur only in state entry scripts[21] and transition scripts, and is specified by calling business methods and assigning returned values to variables. Changes to business object attributes (in writeable variables) are automatically applied to the business objects prior to execution of the scripts.

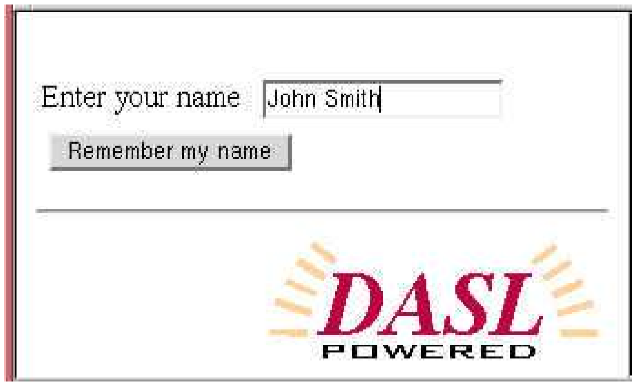
3.4.3 Running RememberMe

The application starts by displaying the following screen, corresponding to the state *Remember* in the AUS.[22]

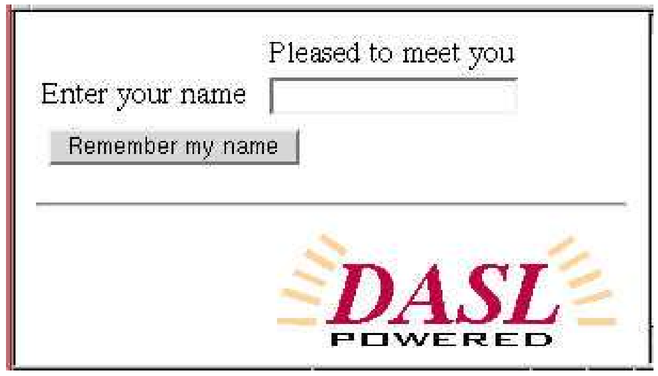


The very top of the screen is a placeholder for the message, which is empty when the application starts. The second line provides an editable text field in which the user may enter a name (an arbitrary single line of text). The user may now enter his or her name, and press the "Remember my name" button.

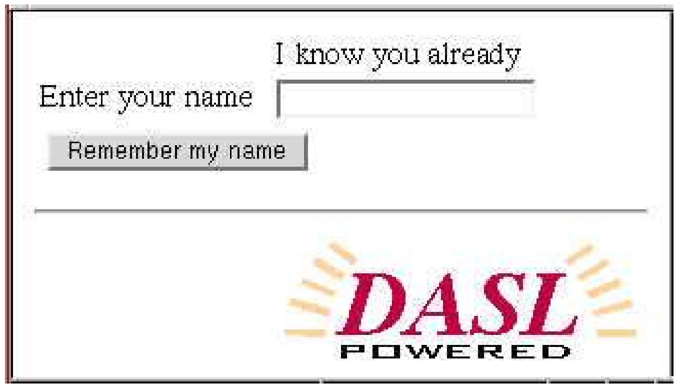
20 This string is actually an I18N label that can be internationalized when the application is deployed.
21 Each state may have a script that gets executed upon state entry, and that performs business logic.
22 The initial state is not displayed when the application runs because it performs a *goto* in its entry script.



The first time a particular name is entered, the application displays the message “Pleased to meet you” and returns to the *Remember* state to accept another name.



The second and subsequent times that the same name is entered, the application finds the name in the persistent store and thus displays the message “I know you already” when it returns to the Remember state to accept another name.



3.4.4 Exercise 1.2

Create the files *RememberMe.bos* and *RememberMe.aus* as shown above and execute the application.

Refer to Appendix A for information on using the DASL line-oriented commands to define and execute an application.[23]

Alternatively, you may wish to try the DASL graphical development environment (a plugin module to SunONE Studio), in which case you can choose to enter the BOS and AUS using either the UML editor, the IDE's textual editor, or a combination of both. Then use the right-mouse-menu command "Execute" to deploy the application.

3.4.5 Exercise 1.3

Modify the RememberMe application to display the greeting message ("Pleased to meet you" or "I know you already") in its own state, with a button labeled "OK" to return to the state *Remember*.

Hint: Create a new state, **Greeting***, with a single String parameter. It will need a transition* OK *to return to the* Remember *state after it displays the message. Modify the* Remember *transition in the* Remember *state to goto ShowMessage(...). Note that the* Remember *state no longer requires a message parameter.*

3.5 The Discipline of Separating User Interaction from Business Logic

You may be used to writing programs that freely invoke calls to business objects from anywhere within the business task, or that define the business task as part of the business objects. However, a close look at the RememberMe application shows that the DASL language imposes the discipline of separating the definition of business tasks (AUS) from the definition of business objects and logic (BOS).

The AUS encapsulates the logic of presentation and flow of the application. Within the AUS, explicit computation using business logic is allowed only during transitions. During a transition you can invoke arbitrary business logic, such as:

- invoking a method on a business object,
- declaring a variable,
- assigning a variable to an expression,
- transferring to another state,
- transferring to another state if certain errors occur, and
- conditionally executing any of the above based on the value of a variable.

In contrast to the AUS, the BOS provides a computationally complete programming environment, similar to 3rd generation languages. In the BOS, the application writer can define methods of objects, both at the instance and class level, and those methods can invoke arbitrary Java classes. However, it is not permitted to specify user interaction within the BOS.

Despite the discipline imposed by DASL to separate computation from task flow, DASL is computationally complete. Simply consider that an arbitrary computation can be performed in a BOS method, and returned as the result of a method call made within the AUS. Within the AUS, it is

23 To obtain the DASL compiler and development environment, visit the Ace Project website: http://sunlabs.eng/projects/ace within Sun, or http://research.sun.com/ace outside of Sun.

possible to conditionally choose the next state based on the result. Therefore, the discipline DASL imposes does not limit the logic that can be expressed within the application.

As an application evolves, what was once considered pure business logic may grow to require user interaction. In such cases, the business logic in the BOS must be refactored into smaller units so that the AUS can invoke it appropriately. While simply inserting the user interaction into the bowels of the business logic may seem easier than refactoring, in the long run the refactoring is preferable because the details of the user interaction can later be modified independently of the business logic.

3.6 Separation of a Business Task from Presentation

In addition to imposing a discipline of separation between business tasks and business logic, DASL also imposes a discipline of separation between a business task and the details of its presentation. Although DASL creates a default presentation, the application writer is free to change the presentation so long as the edited presentation uses the same business logic and has the same states and transitions. Changes to the business logic and task can be made by going back to the BOS and AUS parts of the DASL program, respectively.

DASL encourages refinement of business logic and application task logic prior to beautification of the presentation. Often, the prototyping stage consists solely of modifying the DASL program, with presentation beautification delayed until the AUS and BOS have been determined.

3.7 Automatic Deployment of Life Cycle Business Logic

From the discussion so far, we know that the RememberMe application's business logic is expressed in its BOS. Yet, if you look at the BOS, it contains no user-written methods at all! Where, then, is the business logic expressed?

In a nutshell, the business logic of the RememberMe application is based on the creation and prior existence of *Person* objects, whose life cycle implementation is automatically deployed by DASL. The transition in the *Remember* state first tries to find an existing *Person* object using the method call and assignment:

```
        Person p = Person.findByPrimaryKeyIfAny(name);
```

If the object stored in *p* is not *null* then the *Person* with that name already exists. If the object stored in *p* is null, the transition creates a new *Person* object using the method call and assignment:

```
        Person p = Person.create(name);
```

Thus, the logic of the application relies on automatically deployed life cycle methods *Person.findByPrimaryKeyIfAny()* and *Person.create()*.

3.8 Handling Errors

There is an alternative way that could have been used to detect that a Person already exists, using DASL's error handling mechanism. Instead of looking for an existing *Person* with

Person.findByPrimaryKeyIfAny(), the transition could unconditionally try to create a new *Person* object using *Person.create(name).*

Because the *name* attribute of the *Person* object is declared to be its *primary key*, the persistent store allows only one *Person* object with a given primary key. Attempting to create another *Person* with the same primary key causes an error. Thus, the following transition script could be used:

```
Person p = Person.create(name);
goto Remember("Pleased to meet you");

on error(DuplicatePrimaryKeyException dpke)
    goto Remember("I know you already");
```

The logic of the application is to attempt to create a new *Person* object, using the automatically deployed *Person.create()* method, and handle the error *DuplicatePrimaryKeyException.* Notice that the *on error* statement is a declarative statement saying what to do if the specified error occurs anywhere in the current block of code, either above or below.

Yet another alternative is to use the life cycle method *Person.findByPrimaryKey()* which throws an exception if no such object is found. The transition script in this case becomes:

```
Person p = Person.findByPrimaryKey(name);
goto Remember("I know you already");

on error(NoSuchPrimaryKeyException nspke) {
    p = Person.create(name);
    goto Remember("Pleased to meet you");
}
```

3.9 Further Exercises

3.9.1 Exercise 1.4

Modify the RememberMe application to use the *Person.create()* method together with an *on error* clause to distinguish whether a *Person* with that name has been created already.

3.9.2 Exercise 1.5

DASL does not distinguish primitive Java types from their corresponding class wrapper types, e.g., *long* from *Long*. By convention, the class wrapper type names (e.g., *Long*) are used to declare variables.

Modify the application so that it declares a count value (*Long count*) within the *Person* object, and keeps a count of how many times that name was encountered. When a *Person* is created, set the count to 1 using *p.count = 1L.* When an existing *Person* is found, increment the count by one using *p.count++.*

3.9.3 Exercise 1.6

Further improve the application of exercise 1.4 by including in the message how many times that person has been met, e.g., “I've met you before on 3 occasions, <name>.”

Hint: String s = “I've met you before on “ + p.count + “ occasions, “ + p.name;

3.10 Observations on the Separation of Persistent Business Logic from Tasks

The RememberMe example has introduced the DASL discipline of separating business logic from business task flow. In the exercises, we saw how the logic of creating a new *Person* or finding an existing one go in the business logic specification in the BOS. In fact, even the simplest operations, such as incrementing the count or composing a message, could be considered part of the business logic. In contrast, the sequence of user interactions that make up the business process are described only in the AUS.

Of course, in the “real world” it can become tedious to force all programming other than method calls to be defined in the BOS. Therefore, DASL supports expressions and conditional logic in transition scripts, as we saw in the exercises.

3.11 Object-Relational Mapping Metadata

DASL/ORMAP is a metadata sub-language for defining the object-relational mapping between the persistent business objects and an existing relational database. Because DASL manages the mapping automatically for new applications that do not use existing databases, you can safely ignore the mapping language when defining applications that do not use existing databases.

3.12 “Legacy” Data Sources

One of the challenges of O-R mapping in the real world at the enterprise level is that the underlying data sources for the most critical data are rarely accessible at the relational database level. A second and equally difficult challenge is that accesses to the legacy data often cannot be nested within a short term transaction. No automated O-R mapping scheme can handle the ugliness of legacy data sources in the real world, not even DASL's automated O-R mapping mechanism.

Mapping enterprise legacy data sources into well-behaved business objects, which can then participate in the task/process model of the AUS, is a crucial step that requires programming. One clean pattern for doing so keeps the business logic within business objects that are a *facade* for underlying data access objects, in turn containing the details for translating the legacy data source into business objects.

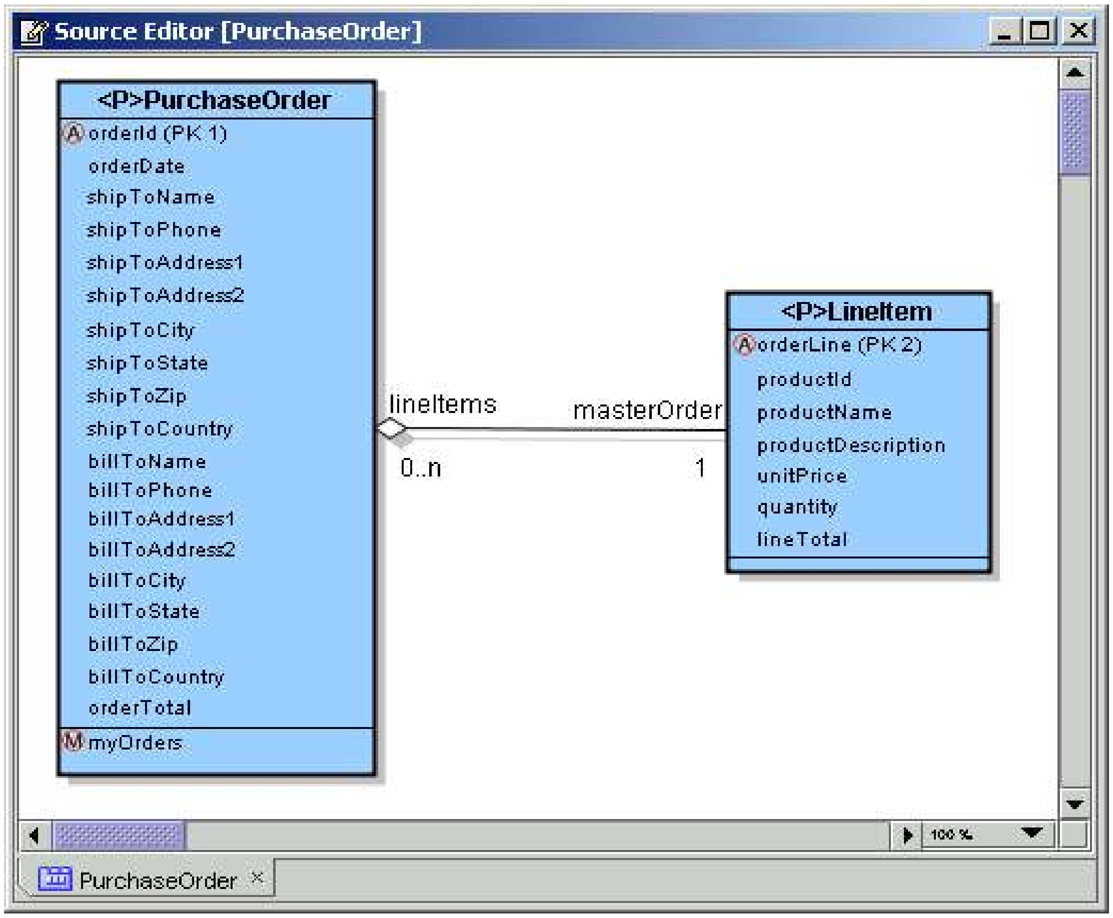
As this manual goes to press, work is proceeding to implement the facade object mechanism within DASL as documented.

4 The PurchaseOrder Application

Our look at the RememberMe example has introduced the DASL disciplined approach to separating business task logic from business object logic. Our next step is to take a look at an easy to understand yet logically complete application called the *PurchaseOrder* application. This application allows purchase orders to be created and modified by the user. This application is a simplified and distilled version of a real world e-commerce application.

4.1 PurchaseOrder Application BOS

The domain model for the PurchaseOrder application consists of a representation of the purchase order as a composite of two persistent object types. This simple domain model can be translated directly into a DASL BOS specification. The UML diagram for the BOS specification is shown below.



4.1.1 Business Objects

The BOS contains two persistent objects: a *PurchaseOrder* object, and a *LineItem* object. A *PurchaseOrder* object contains information about the order as a whole, such as shipping and billing addresses, and it has a one-to-many ownership relationship to *LineItem* objects. A *LineItem* object contains information about a particular item being purchased, including the product identifier, name, description, unit price, and the quantity being ordered. In addition, both objects have some associated business logic and constraints on their attributes. The additional business logic and constraints do not appear directly on the UML diagram, and will be described shortly as they appear in the underlying BOS language.

4.1.2 Relationships

There is a bidirectional relationship between the objects *PurchaseOrder* and *LineItem:* an instance of the *PurchaseOrder* object ***owns***[24] instances of the *LineItem* object, and each *LineItem* instance is owned by an instance of the *PurchaseOrder* object, referred to as *masterOrder*. The *owns* relationship carries the semantics that instances of *LineItem* cannot exist on their own. Rather, they must be part of a *PurchaseOrder*. Furthermore, when a *PurchaseOrder* is deleted, the instances of *LineItem* it owns are also deleted.

DASL provides a second type of relationship between object instances. The ***references***[25] relationship is a bidirectional relationship between two object instances without any ownership semantics. The two objects may be created and deleted independently.

A third kind of relationship, an ***inheritance*** relationship, may be declared between two objects. Unlike the *owns* and *references* relationships, inheritance relationships apply to all instances of the related business objects, not just a particular instance.

Owns and *references* relationships also carry cardinality information. In the *PurchaseOrder* BOS UML diagram, we see that a *PurchaseOrder* can own zero or more instances of the *LineItem* object (represented on the diagram as *0..n*), while in the other direction, a *LineItem* instance must have exactly one associated *PurchaseOrder* object instance (represented on the diagram as *1*), i.e., its *masterOrder*.

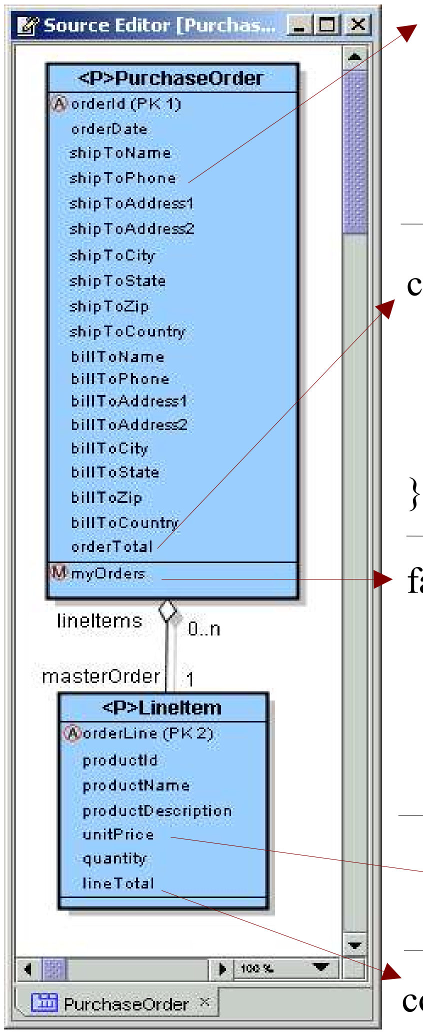
4.1.3 Methods, Computed Attributes, and Constraints

The BOS UML diagram indicates that there is a business method *myOrders()* defined for the *PurchaseOrder* object . As we shall see shortly, there are also some computed attributes associated with these objects, and some constraints on the attributes.

The annotated figure on the next page shows the business methods, computed attributes, and constraints that are defined for the PurchaseOrder Application.

24 The standard UML term for the **owns** relationship is 'aggregation'.
25 The standard UML term for the **references** relationship is 'association'.

- The constraint *phoneNumberFormat* is a regular expression that the phone number string must match.
- The computed attribute *PurchaseOrder.orderTotal* computes the total for the order by iterating over each line item, adding that line's *lineTotal*.
- The factory[26] method *PurchaseOrder.myOrders()* is a method whose implementation is specified as a query that retrieves all the purchase orders from the persistent store that belong to the specified customer.[27]
- The constraint *nonNegative* applies to unitPrice.
- The computed attribute *LineItem.lineTotal* computes the cost of the line item by multiplying its unit price by the quantity.



```
CHECK phoneNumberFormat (
    shipToPhone.length() == 0 ||
    shipToPhone.matches("\\(?(\\d{3}(\\)? ?|[-. ])\\d{3}[-. ]\\d{4}")
);
```

```
computed Decimal orderTotal = {
    BigDecimal total = 0;
    for ( LineItem li: lineItems )
        total += li.lineTotal;
    return total;
};
```

```
factory method myOrders( String billToName )
        returns Collection< PurchaseOrder>
 OQL ( SELECT p
          FROM PurchaseOrder p
          WHERE p.billToName = :billToName )
```

```
CHECK nonNegative (unitPrice >= 0);
```

```
computed Decimal lineTotal = quantity * unitPrice;
```

26 We would have preferred to call them *class* methods instead of *factory* methods, but that was syntactically unworkable. Java calls them *static* methods, but we don't implement them as such so that seemed as though it would lead to confusion. Therefore, we adopted the Objective C term factory method, for want of a better term.
27 This is our first example of the DASL/OQL query language, used for querying the persistent store.

The syntax for declaring *computed attributes* in the BOS is an example of an abstraction that goes beyond the Java Programming Language. The syntax for declaring *method signatures* in the BOS differs from method signatures in Java in several ways:

1. The modifier **method** signals that this is a method declaration.
2. The return type comes after the method parameters, following the **returns** keyword. Omitting that keyword and the type signals that the method does not return a value ("returns void" in Java parlance).
3. The parameter types use the BOS type system, which does not distinguish certain implementation details that Java does.

The body of a method can be specified either as a DASL/OQL query on the persistent object store, as in the *myOrders()* method above, or as DASL/BOS method statements, which are similar to Java method statements. These statements will be described in detail later in this document.

4.1.4 Other Business Logic

In this simple example, the business logic that the application writer needs to specify explicitly in the BOS is limited to the three pieces of explicit procedural logic and the constraints shown in the preceding figure. The rest of the business logic is captured by the cardinality of the relationship between the *PurchaseOrder* and *LineItem* objects, and the semantics of declarations related to the persistent store, such as which objects are persistent and how their primary key is defined.

4.1.5 Complete DASL/BOS Specification for PurchaseOrder

Now that we understand the nature of the business objects in this example, we are ready to look at the textual representation of the DASL language that fully specifies the BOS of the PurchaseOrder Application. This language is an abstraction of Java. Extensions to Java syntax are **highlighted** in the example, and described below.

The BOS for the PurchaseOrder Application consists of two Java classes, one for each business object.

Purchase Order Business Object Specification (textual form)

```
package com.sun.purchaseorder;

/** The PurchaseOrder object is the top-level object */

persistent class PurchaseOrder {
      Long orderId;
      Date orderDate = new Date();
      String shipToName;
      String shipToPhone CHECK phoneNumberFormat (
         shipToPhone.length() == 0 ||
         shipToPhone.matches("\\(?(\\d{3}(\\)? ?|[-. ])\\d{3}[-. ]\\d{4})") );
      String shipToAddress1;
      String shipToAddress2;
      String shipToCity;
      String shipToState;
      String shipToZip;
      String shipToCountry = "USA";

      String billToName;
      String billToPhone CHECK phoneNumberFormat (
         billToPhone.length() == 0 ||
         billToPhone.matches("\\(?(\\d{3}(\\)? ?|[-. ])\\d{3}[-. ]\\d{4})") );
      String billToAddress1;
      String billToAddress2;
      String billToCity;
      String billToState;
      String billToZip;
      String billToCountry = "USA";

      owns LineItem lineItems(0,n) inverse masterOrder(1,1);

      PRIMARY KEY (orderId);

      GROUP phoneIndex TYPE INDEX (billToPhone);

      factory method myOrders( String billToName ) returns List<PurchaseOrder>
      OQL (    SELECT p
               FROM PurchaseOrder p
               WHERE p.billToName = :billToName )

     computed Decimal orderTotal = {
               Decimal total = 0;
               for (LineItem li: lineItems)
                         total += li.lineTotal;
               return total;
      };

} // class PurchaseOrder


/** LineItem objects are owned by PurchaseOrder objects */

persistent class LineItem {
      Long orderLine;
      CaselessString productId;
      String productName;
      String productDescription;
      Decimal unitPrice CHECK nonNegative (unitPrice >= 0);
      Long quantity CHECK nonNegative (quantity >= 0);

      computed Decimal lineTotal = unitPrice * quantity;

      PRIMARY KEY (masterOrder, orderLine);

} // class LineItem
```

4.1.6 Persistent and Transient Objects and Attributes

The modifier *persistent* on a class declaration indicates that instances of the class are to be stored persistently and transactionally. All attributes of persistent classes are, by default, stored persistently and transactionally. The modifier *transient* on an attribute declaration changes that default.

The modifier *transient* on a class declaration indicates that instances of the class are not stored persistently or with transactional semantics. Attributes of transient objects are always transient.

Classes in the BOS that are not modified with either *persistent* or *transient* are made persistent by default.

4.1.7 Basic Types

The DASL type ***CaselessString*** is used for the productId to indicate that this attribute is not case sensitive. In all other respects, *CaselessString* is a Java *String*.

Unlike Java, DASL does not distinguish 'primitive' types from class-based types. For example, *Integer* and *int* are considered the same type. However, DASL does make a distinction between value-based types and identity-based types:

- An **EBT** (Extended Basic Type) is a value-based type, i.e., the identity of an EBT instance is defined by the combined values of all of its attributes. *Integer* and *List* are examples of predefined EBT types. Application-specific EBT types may also be defined. An EBT can be explicitly declared as immutable if its attributes cannot be changed (the predefined scalar EBTs are all immutable, except for iterators).
- **A business object type** is a type for which the identity of instances is defined only by the instance's primary key value, not by any other attribute or relationship values. *PurchaseOrder* and *LineItem* are examples of business object types. Business object types are assumed to be mutable and identity-based.

4.1.8 Attributes

Attributes are object fields whose type is either one of the predefined scalar EBT types, or else a user-defined EBT. Attributes come in four different flavors:

- **persistent** value is specific to an object instance, and is stored persistently.
- **transient** value is specific to an object instance, and is not stored persistently.
- **computed** value is computed dynamically for an object instance, and is thus not stored. A computed attribute is implemented as an instance method with no parameters, but it is accessed using the syntax of an attribute.
- **constant** value is the same across all object instances, and it never changes. The value is thus a class constant.[28] The *PurchaseOrder* application does not use any constant attributes.

Attributes may have associated constraints, specified with the *CHECK* keyword. In this example, the constraint that the *unitPrice* attribute of a *LineItem* must not be negative is expressed by the constraint *notNegative* placed on *LineItem.unitPrice*. Constraints may also be placed on relationships and on the

28 Java uses the keywords *static* and *final* together to signal that a field is a class constant.

entire state of a business object.[29]

4.1.9 Relationships Between Object Instances

Instances of objects may have relationships to other objects. For example, a *PurchaseOrder* owns *LineItem* objects.

Relationships between business objects are declared explicitly, and they can be bidirectional. In contrast, in Java the relationships between objects are treated like any other attribute and are always uni-directional.

For example, in the PurchaseOrder Application, the owns relationship between *PurchaseOrder* and *LineItem* is bidirectional, and is declared in the *PurchaseOrder* class as:

```
owns LineItem lineItems(0,n) inverse masterOrder(1,1);
```

meaning that a *PurchaseOrder* owns zero or more *LineItems*, which can be accessed from the PurchaseOrder by the name *lineItems*, and each *LineItem* is owned by exactly one *PurchaseOrder*, which can be accessed from that *LineItem* by the name *masterOrder*.

In general, relationships may have cardinality 1-to-1, as in the relationship between a *LineItem* and a *Product;* they may be 1-to-many, as in the relationship between a *PurchaseOrder* and its *LineItem* objects; or they may be many-to-many, as in the relationship between *Employee* and *Project* (assuming that an employee may work on more than one project, and a project is worked on by more than one employee).

4.1.10 Primary Key Constraint

Business object may have a *PRIMARY KEY*, which consists of the attributes and/or relationships that uniquely distinguish one instance of the object from all others. Because the primary key is used to identify the object, the key itself is immutable.

For persistent and facade objects, the primary key must be specified before the application is deployed. The deployment engines require it to correctly implement the semantics of the object. Transient objects are not required to have a declared primary key at deployment time.

For example, the *PurchaseOrder* object's primary key is the attribute *orderId*, and the *LineItem*'s primary key is the tuple of the *masterOrder* and the *LineItem's orderLine,* i.e., the line item number relative to the *masterOrder*. Taken together, the *masterOrder* and the *orderLine* uniquely identify a particular instance of a *LineItem*.

To facilitate manipulation of primary keys when the key is a tuple with more than one component, and to make it easeir to represent collections of primary keys, an immutable EBT is implicitly defined for each business object, containing only that object's primary key members. The EBT has the name <businessObjectName>PK. Thus, the automatically defined primary key EBT for *PurchaseOrder* is called ***PurchaseOrderPK.***

29 Constraints imposed between business objects will be supported in a future release of DASL.

In some cases, there is no "natural" primary key for a business object based on its relationships or visible attributes. The standard way to deal with such cases is to declare an attribute *Long id* and make it the primary key.

4.1.11 Automatic Primary Key Management

The PurchaseOrder example relies on automatic primary key management. When the primary key consists of a single numeric attribute, or a numeric attribute and a relationship, a factory method to create new primary keys is automatically provided. The factory method for the PurchaseOrder object has the signature

```
method assignNextPK() returns PurchaseOrderPK
```

This automatic management method may be overridden (or specified for business objects that don't have numeric primary keys) in the BOS.

4.1.12 Comments

Comments are allowed in any of the three styles permitted by Java.[30] These include C-style comments (/* ... */), C++-style comments (// ... end of line), and JavaDoc comments (/** ... */).

4.1.13 Business Methods

Business methods may be included in the BOS for each business object. The body of the method may be specified either in the DASL method language or as a DASL/OQL query.

In the *PurchaseOrder* example, the method *PurchaseOrder.myOrders()* is defined as a DASL/OQL query.

The methods associated with a business object can be:

• **instance** methods	they execute in the context of an instance of this object (i.e., they are invoked via a variable that refers to an instance of the object, and that object is available within the method as the pseudo-parameter called ***this***).
• **factory** methods	they execute in the context of a factory object specific to the class, but independent of any instance. Among other things, they provide the ability to create or locate instances of object, as well as perform any operation that does not logically require an object instance.[31]

30 The DASL IDE module currently preserves only those comments that immediately precede certain major parts of the specification. This shortcoming will eventually be fixed.

31 Java defines the notion of 'static method' to serve the same purpose, but Java ties the concept to syntactic conventions, such as the ability of the compiler to uniquely locate and import the compiled class file for the implementation of the object during compilation. In contrast, factory methods are instance methods of an explicit factory object, and thus work in a distributed environment.

4.1.14 Automatically Deployed Life Cycle Business Methods

In addition to the methods written explicitly by the application designer, a significant number of methods are deployed automatically. These methods cover life cycle events on the business objects, automatic creation of unique primary keys, and retrieval of an object's extent from the persistent store.

For example, the PurchaseOrder object will have several automatically deployed factory methods named *create*:

```
PurchaseOrder create( Long orderId ) {}

PurchaseOrder create( Long orderId, Date orderDate, String shipToName, ... ){}
```

which create new instances of the PurchaseOrder object in the persistent store, as well the instance method remove:

```
void remove() {}
```

which deletes the current instance from the persistent store, and the instance method

```
PurchaseOrderPK assignNextPK();
```

which creates a primary key EBT for a PurchaseOrder object, using the next available primary key value.

4.1.15 DASL Syntactic “Sugar”

The discerning reader may have noticed that the computed attributes *lineTotal* and *orderTotal* appear in **highlighted** font. These attributes use some syntactic sugar provided by the DASL language that does not exist in the Java language.

For example, consider the simple expression used to define *lineTotal*: **(unitPrice * quantity).** If unitPrice and quantity were both *long*s, Java would have no problem with the above expression. However, *unitPrice* is declared as a DASL *Decimal*, which is equivalent in Java to the class *java.util.BigDecimal*, and *quantity* is declared as a DASL *Long*, which is implemented in Java as a *long*. Java does not define the use of the arithmetic operator '*' between a *long* and a *BigDecimal*, and in fact does not define the use of *any* arithmetic operators for *BigDecimal* at all. Instead, Java requires the following tricky expression to multiply the two numbers:

```
this.unitPrice.multiply(new BigDecimal(String.valueOf(this.quantity)))
```

Other DASL extensions occur in the computed attribute *orderTotal*:

```
Decimal total = 0;
for (LineItem li: lineItems)
    total += li.lineTotal;
```

Again, the Java implementation of the type *Decimal* is a class BigDecimal, and it does not allow an instance of the class BigDecimal to be initialized to the numeric literal '0', even though it is 'obvious' what that ought to mean. Java also does not support the '+=” operator applied to BigDecimal objects.

The DASL compiler defines the above initialization and loop to have the same semantics as the following legal Java statements:

```
BigDecimal total = new  BigDecimal(0);
Iterator i = lineItems.iterator();
while (i.hasNext()) {
    LineItem li = (LineItem) i.next();
    total = total.add(li.getLineTotal());
}
```

One further bit of magic to notice in this case is that the computed attribute *lineTotal* is implemented in Java as an instance method of the *LineItem* class (since Java does not directly support the notion of computed attributes). The DASL language allows the direct use of the expected attribute syntax by translating each access to the computed attribute *lineTotal* into a call to *getLineTotal()*.

All DASL extensions to Java provide meaning to constructs (statements and expressions) that would otherwise be flagged by the Java compiler as invalid. The DASL compiler then translates these invalid constructs into legal Java, based on the semantic meaning that it defines for them. Thus, these extended DASL constructs should not cause confusion among those familiar with Java.

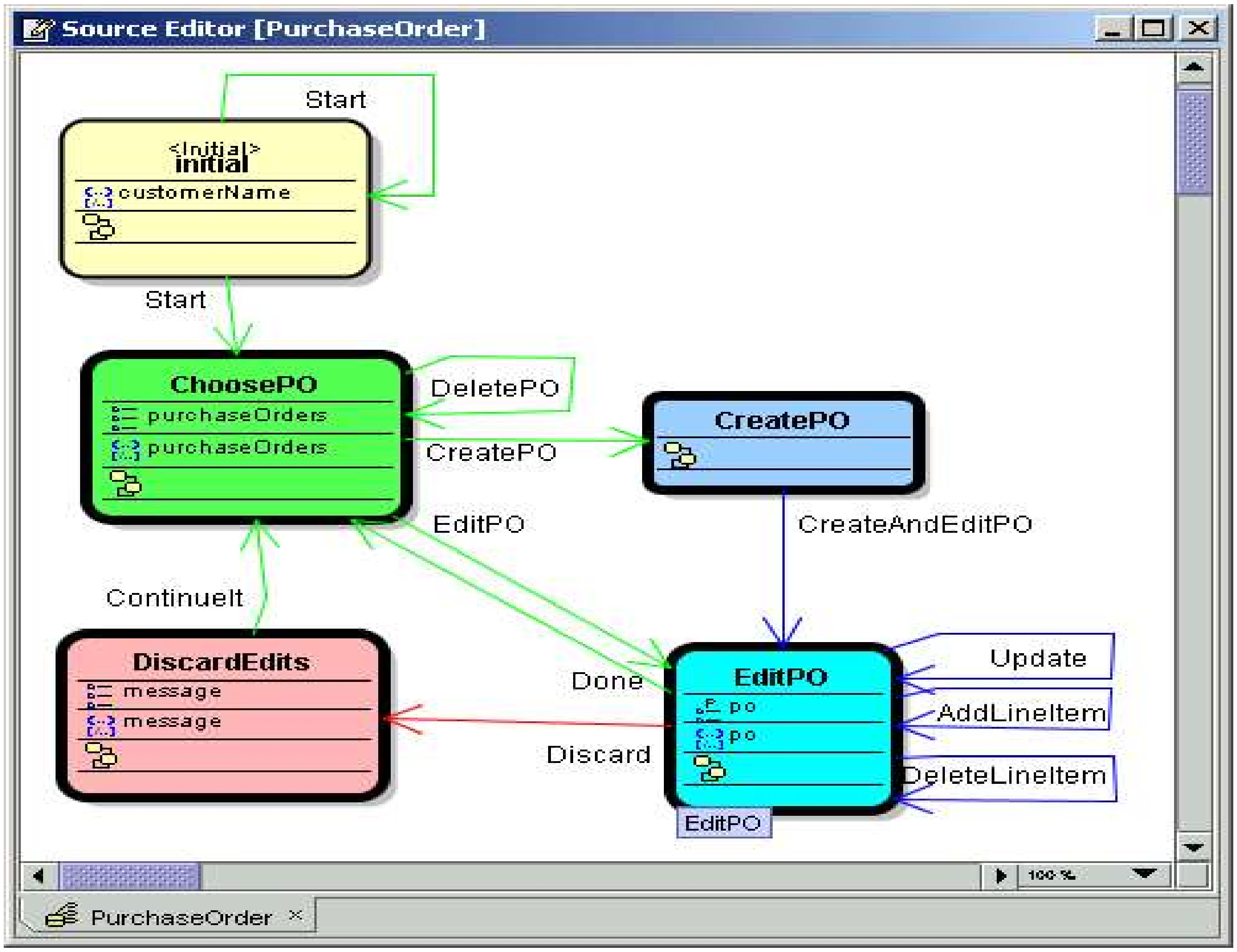
We will look at the full set of enhancements provided by DASL within method bodies in a later section.

4.2 PurchaseOrder Application AUS

The *PurchaseOrder* AUS defines how the two objects in the BOS are used by the application during the interaction between the user and the application, i.e., the business process implemented by the application. As a precursor to defining the AUS, we identify several use cases for our application, as follows:

1. Login: The user enters a name by which he is known to the system. The other use cases are then available.
2. DeletePO: The user deletes an existing purchase order.
3. CreatePO: The user creates a new purchase order, edits it, and saves it.
4. EditPO: The user chooses an existing purchase order and then edits it and saves it.

These use cases can be translated fairly directly into a set of AUS states and transitions, as shown in the UML state transition diagram below.



The four use cases have been implemented using five (5) states representing the screens that the application user will see when the application runs, and using transitions between states representing the navigational actions that the user will take, typically by pressing buttons on the screen.

The first screen is the *initial state Login*, where the user of the application can enter the *customerName* and then press the *Login* button.[32] The *Login* button causes a transition from the *Login* state to the *ChoosePO* state.

Some business logic is executed during the transition between *Login* and *ChoosePO*, part of which is associated directly with the *Login* button and is part of the *Login* state, and part of which is attached to the *ChoosePO* state, and is attached to the *ChoosePO* state. This business logic is not shown in the state transition diagram, and will be described in the next section.

The ChoosePO state retrieves all the purchase orders of the current user when it starts. The *ChoosePO* state then allows the user to do one of several things:

- create a new purchase order and then edit it,
- select and then delete an existing purchase order, or
- select and then edit an existing purchase order.

If the user presses the button to create a purchase order, a transition is made to the CreatePO state. This state creates a new purchase order and then immediately transitions to the *EditPO* state without any user interaction. In the *EditPO* state, the user can edit the order. From the *EditPO*state, the user may choose to apply the edits and return to the *ChoosePO* state, or discard them and return to the *ChoosePO* state. The state DiscardEdits is defined to inform the user that the transaction has been discarded.

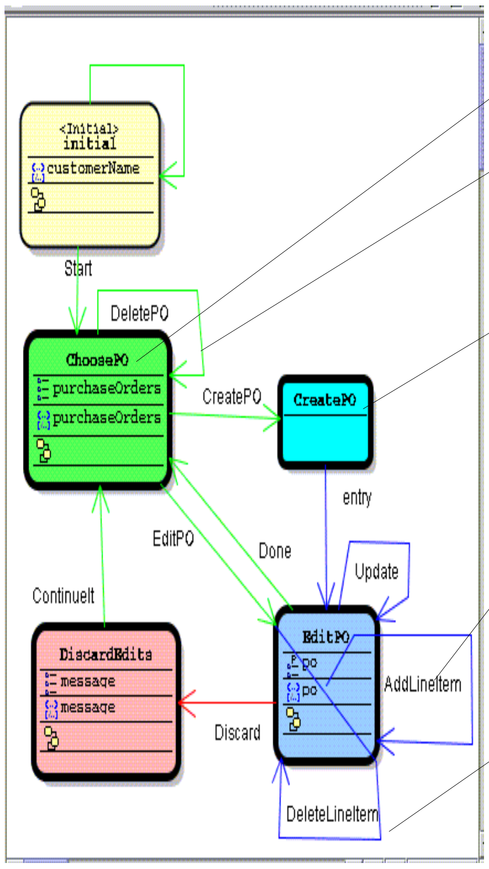
If the user selects one of the purchase orders and presses the delete button, the business logic associated with the *DeletePO* transition deletes the purchase order and then returns to the *ChoosePO* state.

The AUS state diagram on the previous page describes both the use cases and the user-visible steps required to implement them in a concise way. However, it does not show the detailed business logic attached to each transition and to each state (upon entry), and thus does not directly represent how the states and transitions are tied to the business objects in the BOS.

The next section describes how the AUS is connected to the BOS business logic, by invoking business methods upon entry into a state and upon transitions to other states.

32 In a real world application, this step would probably involve a login validation of some kind.

4.2.1 Attaching Business Logic to States and Transitions



```
purchaseOrders = PurchaseOrder.myOrders(customerName);
```

```
PurchaseOrder po = purchaseOrders.getSelectedOne();
PurchaseOrder.remove(po);
goto ChoosePO();
```

```
PurchaseOrderPK pk = PurchaseOrder.assignNextPK();
PurchaseOrder po = new persistent PurchaseOrder(pk);
po.billToName = customerName;
goto EditPO(po);
```

```
LineItemPK pk =  LineItem.assignNextPK(po);
LineItem li = new persistent LineItem(pk);
po.insertIntoLineItems(li);
goto EditPO(po);
```

```
LineItem li = po.lineItems.getSelectedOne();
LineItem.remove(li);
/* The above  implicitly calls po.removeFromLineItems(li)*/
goto EditPO(po);
```

The annotated figure above shows how the application writer defines the business logic to be invoked during entry into a state, and transitions out of a state.

The AUS states are tied to the BOS through explicit business logic. Explicit business logic is specified by attaching a business logic "script", called the *entry script,* to each state, and by attaching another script, called the *transition script,* to each of the transitions out of that state. The scripts may contain business method calls made on the BOS objects, and calls that choose the next state, passing parameters to that state.

The annotations to the state diagram above are as follows:

1. The entry script of the *ChoosePO* state calls the factory method *PurchaseOrder.myOrders (customerName)* to obtain the purchase orders belonging to *customerName*, where *customerName* is the name entered by the user in the initial state.

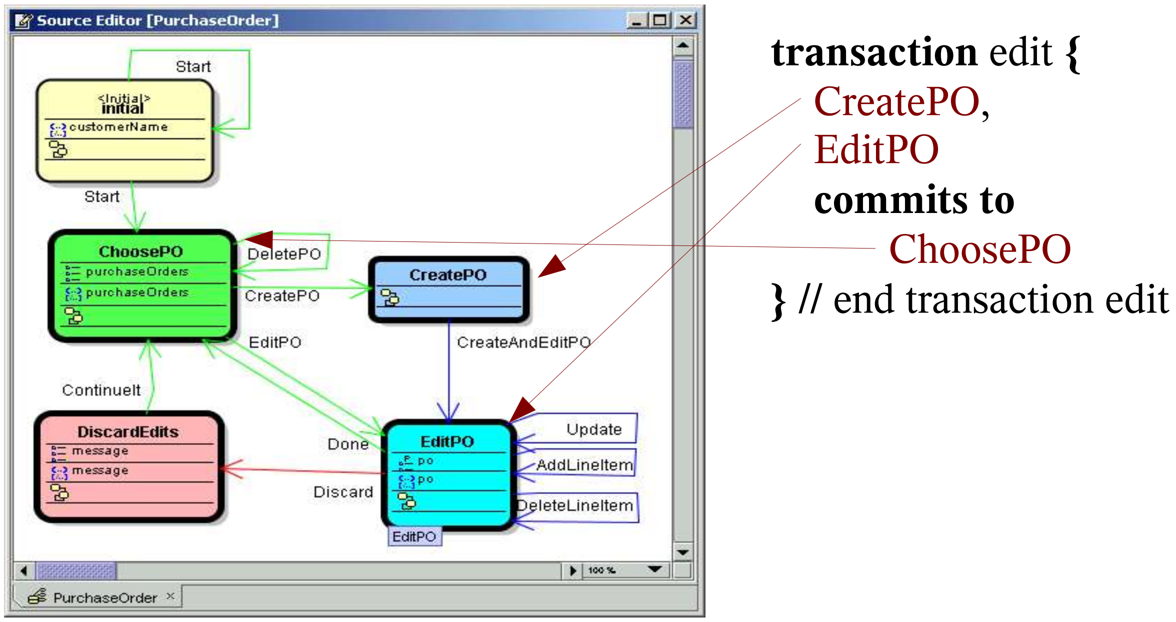
2. The script for the *DeletePO* transition of the *ChoosePO* state obtains the purchase order selected by the user, and calls the automatically deployed factory method *PurchaseOrder.remove()* to delete it from the persistent store. It then "falls out" of the transition script, so that control remains in the *ChoosePO* state.

3. The *CreatePO* state is entered if the user asks to create a new purchase order. It has an entry script that creates a new purchase order,[33] sets the *billToName* attribute to *customerName*, and transitions to the *EditPO* state.

4. Within the *EditPO* state, the script for the transition *AddLineItem* inserts a new (empty) *LineItem* into this *PurchaseOrder*, similarly to the way a new *PurchaseOrder* object is created. Control remains in the *EditPO* state.

5. Within the *EditPO* state, the script for the transition Delete*LineItem* deletes the selected *LineItem* from the persistent store by calling *LineItem.remove(li),* similarly to the way a *PurchaseOrder* is deleted. Because *LineItem* is owned by *PurchaseOrder*, *LineItem.delete(li)* implicitly removes the *LineItem* from the ownership list *po.lineItems* in the *PurchaseOrder* that owns it.

4.2.2 Specifying Application Transactions Declaratively



```
transaction edit {
    CreatePO,
    EditPO
    commits to
        ChoosePO
} // end transaction edit
```

Transactions are specified in a novel and concise declarative form, at the level of application business tasks. For example, the transaction called "edit" is started whenever a transition is made to either of the states CreatePO or EditPO. The transaction is active as long as the application remains in those states. When a transition to ChoosePO occurs, the transaction is committed. When a transition is made to any other state, such as the DiscardEdits state, the transaction is cancelled.[34]

4.2.3 Complete DASL/AUS Specification for PurchaseOrder

Expressed in DASL, the AUS for the PurchaseOrder Application consists of five (5) states.

33 Creating a new PurchaseOrder object requires assigning a unique primary key (PK).

34 Note that the cancel operation may be implemented by the deployment engine as a database transaction rollback, by invoking compensating transactions, or by some other mechanism.

PurchaseOrder Application Usage Specification

```
package com.sun.purchaseorder.aus;

import com.sun.purchaseorder.PurchaseOrder.* as POApplication.*;

String customerName;

initial state Login( )
{
    entry {  customerName = "";  }

    usage {
        customerName        :W;
    } // end usage

    transition Login {
        if (customerName.length() > 0)
             goto ChoosePO();
    } // end transition Login

} // end state initial


state ChoosePO( )
{
    Collection<PurchaseOrder> purchaseOrders;

    entry {  purchaseOrders = PurchaseOrder.myOrders(customerName); }

    usage {
        [purchaseOrders(0,n)]        :R
          orderId    "Order ID",
          orderDate  "Order Date",
          orderTotal "Order Total"
        }
    } // end usage

    transition CreatePO "Create Purchase Order" {
        goto CreatePO();
    } // end transition CreatePO


    transition DeletePO "Delete Purchase Order" {
        Collection<PurchaseOrder> pos = purchaseOrders.getSelectedMany();
        PurchaseOrder.remove(pos);
        goto ChoosePO();
    } // end transition DeletePO

    transition EditPO "Edit Purchase Order" {
        PurchaseOrder po = purchaseOrders.getSelectedOne();
        goto EditPO(po);
    } // end transition EditPO

    transition Logout {
        goto Login();
    }

} // end state ChoosePO
```

```
/** State CreatePO transfers directly to EditPO.  It is here so that creation
 *  and editing of the PurchaseOrder can be made into a single transaction.
 */
state CreatePO( )
{

    entry {
        PurchaseOrderPK pk = PurchaseOrder.assignNextPK();
        PurchaseOrder po = new persistent PurchaseOrder(pk);
        po.setBillToName(customerName);
        goto EditPO(po);
    }

} // end state CreatePO


state EditPO( PurchaseOrder po ) "Edit Purchase Order"
{
    usage {
        po        :RW  {
            orderId "Order ID"        :R,
            orderDate "Date",

            shipToName "Ship To",
            shipToPhone "Phone",
            shipToAddress1 "Address",
            shipToAddress2 "Address2",
            shipToCity "City",
            shipToState "State",
            shipToZip "Zip Code",
            shipToCountry "Country",

            billToName "Bill To",
            billToPhone "Phone",
            billToAddress1 "Address",
            billToAddress2 "Address2",
            billToCity "City",
            billToState "State",
            billToZip "Zip Code",
            billToCountry "Country",

            [lineItems(1,1)] "Line Items" {
                orderLine "Item"        :R ,
                productId "Product ID",
                productName "Product",
                productDescription "Description",
                unitPrice "Unit Price",
                quantity "Quantity",
                lineTotal "Line Total"        :R
            },
            orderTotal "Order Total"        :R
        }
    } // end usage

    transition Update {
        // Synchronize state with business logic, and continue
        continue;
    } // end transition Update

    transition AddLineItem "Add Line Item" {
        LineItemPK pk = LineItem.assignNextPK(po);
        LineItem li = new persistent LineItem(pk);
```

```
          po.lineItems.add(li);
          // Synchronize state with business logic, and continue
          continue;
      } // end transition AddLineItem

      transition DeleteLineItem "Delete Line Item" {
          LineItem li = po.lineItems.getSelectedOne();
          LineItem.remove(li);
          /** The above implicitly removes li from po.lineItems */
          // Synchronize state with business logic, and continue
          continue;
      } // end transition DeleteLineItem

      transition Done "Done Editing Purchase Order" {
          goto ChoosePO();
      } // end transition Done

      transition Discard "Discard Edits" {
          goto DiscardEdits();
      } // end transition Discard

  } // end state EditPO


  state DiscardEdits( )
  {
      String message;
      entry {
          message = "Edits discarded";
      }
      usage {
          message ""        :R ;
      } // end usage

      transition ContinueIt "Continue" {
          goto ChoosePO();
      } // end transition ContinueIt

  } // end state DiscardEdits
```

transaction edit { CreatePO, EditPO **commits to** ChoosePO }

4.2.4 Specifying the Package

The *package* statement specifies the namespace in which this application lives.[35]

4.2.5 Specifying the BOS to Use

The ***import*** statement specifies the BOS to use. The name that follows the optional **as** keyword (in this case, *POApplication*) is the local alias for the imported BOS in this AUS. By default, the local name is the name of the BOS.

The AUS sub-language allows more than one BOS to be referenced, using additional *import*

35 Due to a temporary implementation restriction, the package for an AUS should not be the same as any imported Java package or BOS package that this application uses.

statements.[36]

4.2.6 Session Variables

The variable *String customerName* is declared as a session variable. It is available in all states of this AUS, and its lifetime is the entire application session. Since it does not refer to a persistent business object, the variable is non-transactional.

4.2.7 States

As we saw in the RememberMe application, the AUS specification consists of a set of states. Each state may declare parameters that are passed into it, as well as variables whose scope is the state.

The application starts at the initial state, where the *customerName* is initialized to “” and the user must enter a non-empty name before continuing to the next state, *choosePO*.

4.2.8 State Variables

Each state may have local variables, whose scope is within the state and its transitions. For example, state *ChoosePO* defines *List<PurchaseOrder purchaseOrders>*.

4.2.9 Entry Scripts and Transition Scripts

The purpose of entry scripts and transition scripts is to associate business logic with states and transitions. Entry scripts are invoked upon initial entry to a state (when a **goto** occurs to that state). They are useful for doing initial computation to obtain and set up the state's usage variables. Transition scripts are invoked when a transition occurs. They are useful for doing final computations and invoking business logic based on information retrieved and entered within the state. Both entry and transition scripts occur as part of the state's transaction.

In the *PurchaseOrder* example, the initial state contains an entry script that initializes the session variable *customerName* to “”. The transition *DeletePO* in the *ChoosePO* state invokes business logic to remove (delete) the current purchase order from the persistent store. It then transitions back to the *ChoosePO* state.

4.2.10 DASL Statements Compared to Java

The syntax and semantics of DASL statements that may appear in method bodies, state entry scripts, and transition scripts is fully documented later in this manual. For those familiar with the Java programming language, DASL supports Java statement syntax, with additional syntactic sugar. The following non-Java special features of DASL are used in this example:

- DASL supports a convenient syntax for accessing persistent constructors and factory methods of business objects, based on the object's name in the BOS. This syntax is the same that Java

36 It is an implementation restriction of the currently generated applications that access to multiple BOS's is not supported. In the near future, we expect to provide multiple BOS's, only one of which can be modified. Allowing multiple updatable BOS's in different DBMS servers requires a distributed transaction capability.

uses for accessing static methods of a class. DASL automatically translates this syntax to the appropriate distributed object syntax when it deploys the application in a distributed environment.

- A high-level *on error* statement to catch and handle exceptions without obscuring the important business logic by embedding it inside a try/catch block.

- A syntax for access to selected value(s) of usage variables that are collections, e.g., *purchaseOrders.getSelectedOne()* and *purchaseOrders.getSelectedMany()*.

- An extended switch statement that supports object literals, including *String* literals. For example:

```
switch (s) {
   case "yes":  goto S2(a,b,c);
   case "no":   goto S3(d,e);
   default:     goto S4();
}
```

4.2.11 Restrictions to DASL Statements within Entry and Transition Scripts

The following restrictions are currently imposed on the DASL statements that can occur in entry and transition scripts. Note that these restrictions apply only to scripts in the AUS, not to method bodies defined in business objects. Because complex programming within scripts can easily obscure the logic and intent of the script, these restrictions tend to enforce the discipline of keeping business logic in business objects (methods) where it more likely belongs.

- No loop constructs are currently allowed within scripts. If loops are required, define a business method to compute the result. For example, you can define a transient business object with a factory method, and then invoke that method from a script.
- All exceptions are automatically caught within scripts, resulting in the invocation of a default *on error* if the application does not define an *on error* for the script. The Java-like *try/catch* is not allowed in scripts. In business object methods, both *try/catch* and *on error* are supported, though *on error* is preferred.
- The use of conditional expressions (a ? b : c) is not supported in scripts.

The following restriction applies in the current implementation, and will be lifted in the future:

- General switch statements are not supported in a script. Currently, a special conditional goto statement is supported, as shown in the extended switch statement above, for which the statements in each case must be either a *goto* or *exit* statement.

4.2.12 Summary of DASL Statements Allowed Only in Scripts

The following DASL statements and expressions are specific to AUS scripts:

- Obtaining the selected elements of a collection-valued usage variable using *vbl.getSelectedOne()* or *vbl.getSelectedMany()*.

- Code statements to insert implementation into deployed code (see the next chapter).

- Transition to another state, e.g., *goto ChoosePO()*.

- Continuation to the same state using the *continue* statement.

- Application exit using *exit* or *exit to URL("...")*.

These statements are documented in the next chapter.

4.2.13 Variable Usage

Following the entry script and state variable declarations, the usage of variables within the state is declared. The variables that may be part of the usage include session variables, the declared parameters to the state, and variables declared local to the state.

After the name of the variable, the optional usage mode is specified. If no usage mode is provided, the default mode is *:RW* for top level usages. The common usage modes are:

```
:R        (the variable is read but not changed)
:RW       (the variable is read and changed)
:W        (the variable is entered without being read)
```

If the variable is a business object, then following the usage mode, within brackets {}, the specific object attributes and relationships that will be accessed are described. If no specific attributes and relationships are described, only the primary key of that business object will be used.

If the variable is a collection of some kind, including a relationship with a maximum cardinality of 'n', then the variable is enclosed in square brackets [] in the usage. Following the name of the variable in the square brackets, the selection cardinality may optionally be specified.

For example, in the *ChoosePO* state, the usage for the collection *purchaseOrders* is:

```
[purchaseOrders(0,n)]: R {
        orderId    "Order ID",
        orderDate  "Order Date",
        orderTotal "Order Total"
}
```

The cardinality (0,n) for *purchaseOrders* means the application user may select zero or more purchase orders. The usage mode *:R* means that the collection itself is not directly editable by the user.

The only fields of a *PurchaseOrder* that will be used are *orderId*, *orderDate*, and *orderTotal*. The mode of the outer usage variable, *purchaseOrders*, which is *:R*, is used as the default for the inner usage fields *orderId*, *orderDate*, and *orderTotal*. The default may be overridden by giving an explicit mode for an inner usage field.

The mode of a primary key field cannot be *:W* or *:RW*, since primary keys are immutable. Therefore, if the usage mode of *purchaseOrders* were one of these writable modes, and the mode of *orderId* were not given explicitly, it would default to *:R*

4.2.14 Transactions

The final line of the PurchaseOrder AUS is an explicit transaction definition. It states that upon entering either of the states *CreatePO* or *EditPO*, a transaction is to start; and upon transition to the state *ChoosePO*, that transaction is to be committed. Upon transition to any state other than those mentioned, the transaction is to be rolled back.

For example, if the user creates a new order, edits it, and presses the "Discard Edits" button, a transition is made to state *DiscardEdits*, a state outside this transaction that is not a commit state, so all work done in *CreatePO* and *EditPO* will be undone -- the new order will not be created.

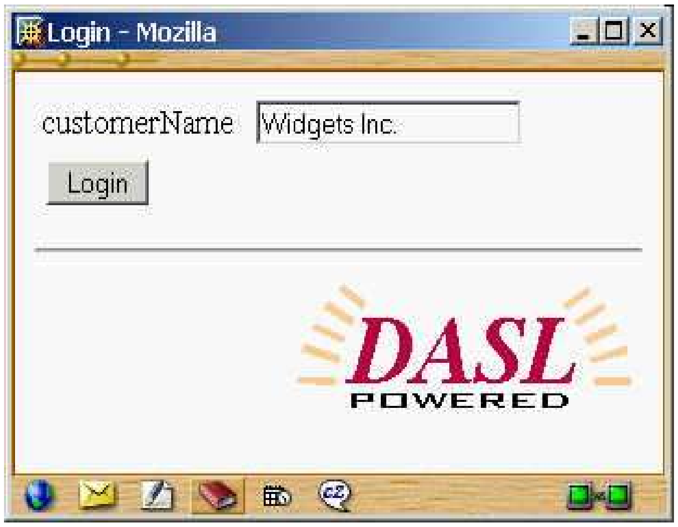
Any state that is not part of an explicit transaction has its own implicit transaction. That transaction begins just prior to execution of the entry script of that state, and commits just after any transition script executes.

4.3 Running the PurchaseOrder Application

We are now ready to take a tour of the deployed application, using the default presentation (GUI screens) that DASL creates. This example can be run from the DASL command line interface as follows:

```
        dasl PurchaseOrder -execute
```

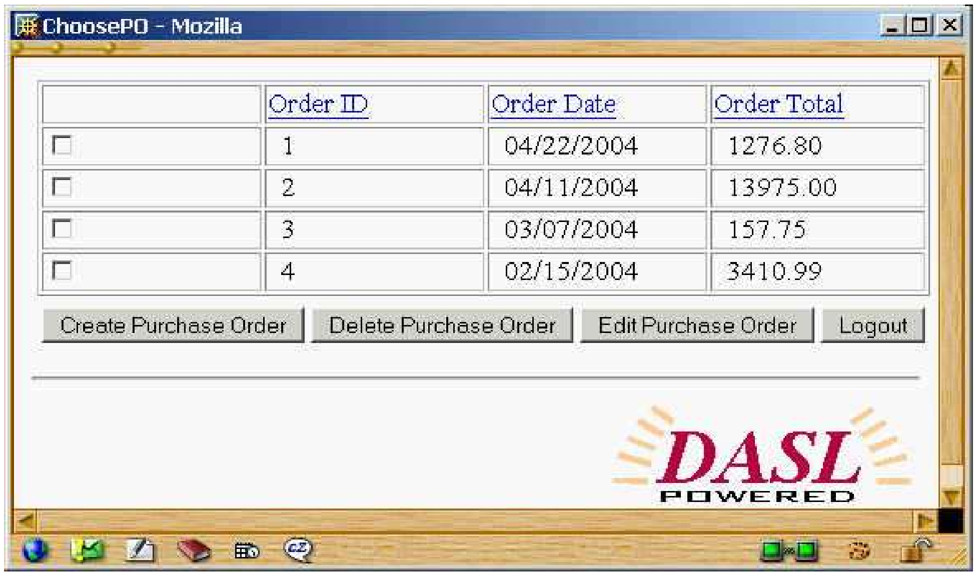
4.3.1 Initial Login Screen


Login - Mozilla

customerName Widgets Inc.

The *PurchaseOrder* application starts up in the initial state, where the user enters a customer name and presses the *Login* button.

4.3.2 ChoosePO Screen


ChoosePO - Mozilla

	Order ID	Order Date	Order Total
☐	1	04/22/2004	1276.80
☐	2	04/11/2004	13975.00
☐	3	03/07/2004	157.75
☐	4	02/15/2004	3410.99

The *ChoosePO* screen, in which all of that customer's purchase orders are displayed, then comes up. The user may either select an existing purchase order to delete or edit, or may press the *Create Purchase Order* button to create a new purchase order. The *Logout* button returns the user to the initial *Login* screen.

For the purpose of this example, assume the user selected order 1 and pressed the *Edit Purchase Order* button.

4.3.3 EditPO Screen

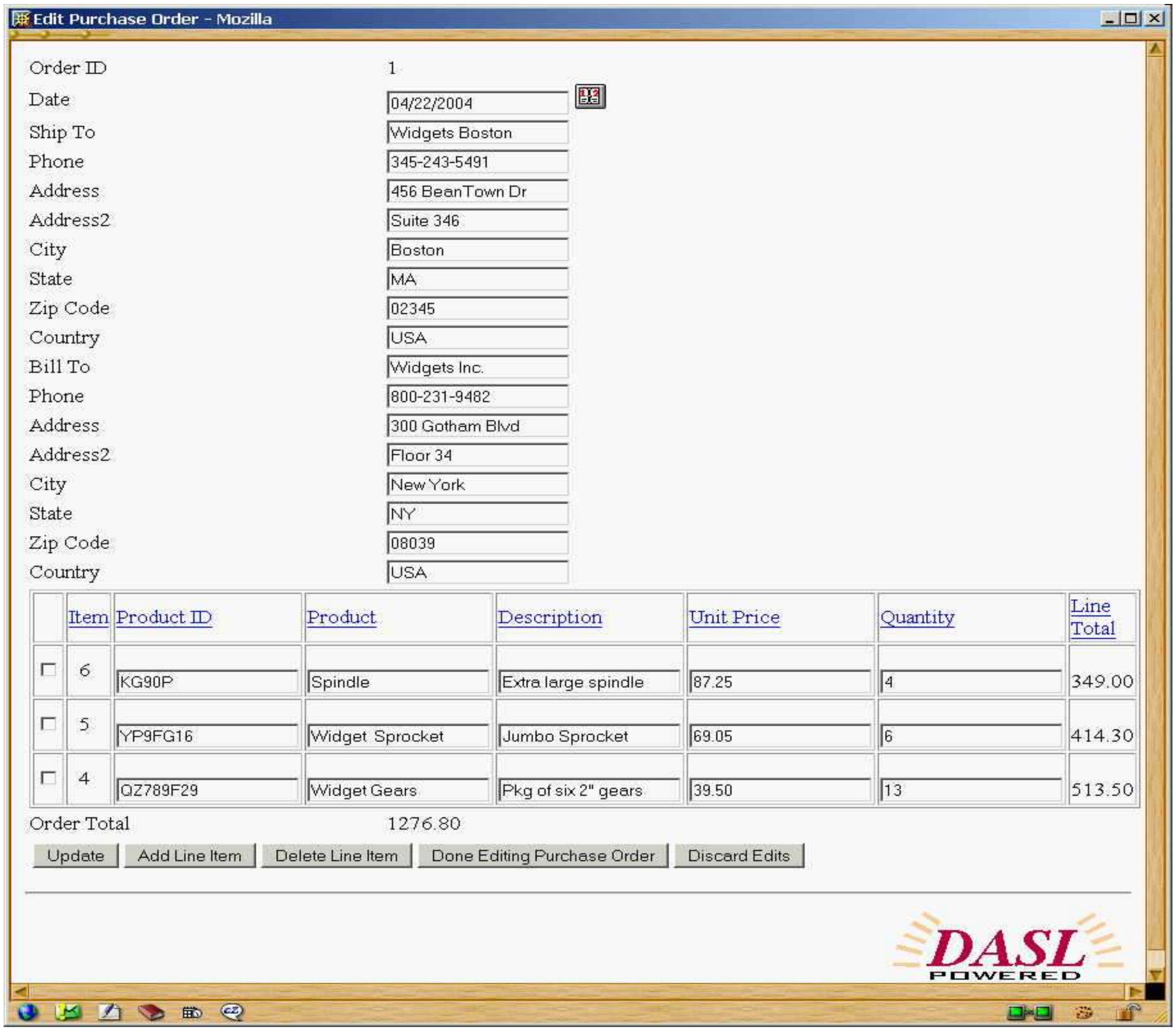
The user now sees a screen on which the fields of PurchaseOrder (such as the ShipTo address) may be edited directly. For convenience, the default presentation provides a calendar date chooser icon for fields of type *Date*. Pressing this icon brings up a standard date chooser.

The existing line items are also directly editable. Line items may be deleted by selecting them and pressing *Delete Line Item,* and new line items may be added by pressing *Add Line Item.*

Pressing the *Update* button causes the changes to be synchronized with the active business objects, where computed attributes are recomputed.[37]

When the user is finished editing and wishes to apply the changes, the *Done Editing Purchase Order* button is pressed. If the user wishes to discard all changes made to this screen, the *Discard Edits* button may be pressed. In both cases, the application transitions to the *ChoosePO* screen above.

37 An upcoming version of DASL will automatically detect such obvious cases as the lineTotal attribute, and compute it dynamically in the browser so that it updates without having to go through a transition.



Order ID	1
Date	04/22/2004
Ship To	Widgets Boston
Phone	345-243-5491
Address	456 BeanTown Dr
Address2	Suite 346
City	Boston
State	MA
Zip Code	02345
Country	USA
Bill To	Widgets Inc.
Phone	800-231-9482
Address	300 Gotham Blvd
Address2	Floor 34
City	New York
State	NY
Zip Code	08039
Country	USA

	Item	Product ID	Product	Description	Unit Price	Quantity	Line Total
☐	6	KG90P	Spindle	Extra large spindle	87.25	4	349.00
☐	5	YP9FG16	Widget Sprocket	Jumbo Sprocket	69.05	6	414.30
☐	4	QZ789F29	Widget Gears	Pkg of six 2" gears	39.50	13	513.50

Order Total 1276.80

4.4 Exercise 3.1 – Run the Application

Run the application, and see what happens when you update the quantity field in a line item while editing a purchase order. *Hint: Try pressing the* Update *button.*

4.5 Exercise 3.2 – Modify the Application Usage

Modify the PurchaseOrder AUS so that it displays the number of LineItems in each PurchaseOrder in the *ChoosePO* state. *Hint: Declare an Integer variable within the ChoosePO state, and set it to pos.size() in the entry script. Modify the usage section of the ChoosePO state to include this variable.*

4.6 Exercise 3.3 – Making PurchaseOrder More Realistic

This exercise is more challenging than the previous ones. In a real e-commerce application, the shipTo and billTo information would not be stored by repeating the name and address attributes. Accordingly, create a new business object class in the PurchaseOrder BOS called *Customer* to hold the name and address of a customer. Include all the attributes relevant to a customer's name and address from the original *PurchaseOrder* class. Now modify the AUS to use the new BOS objects.

Hint: Modify the PurchaseOrder *class to remove the (now redundant)* shipTo... *and* billTo... *attributes and add two relationships to the new* Customer *class to replace them:*

```
        references Customer billTo(1,1) inverse myBilledOrder(0,n);
        references Customer shipTo(1,1) inverse myShippedOrder(0,n);
```

Adjust the AUS so that it creates and/or locates the appropriate billTo *customer during* Login*, and references the attributes of the purchase order's* shipTo *and* billTo *in the usage sections of all states. Add transitions and states that allow a new customer to be created for setting the* billTo *and* shipTo *relationships. Make other adjustments as necessary so that the application works to your satisfaction.*

4.7 Exercise 3.4 – Password Validation

Add password validation to the modified PurchaseOrder application created in exercise 3.3.

Hint: Add a String password *to the* Customer *class. Declare a* String password *local to state Login. Put this local variable in the usage, and declare its access as* :RW*. To prevent the password from being shown on the screen, declare the usage for it as follows:*

```
    password :RW PROPERTIES { display = “password” }
```

In the transition Login, use Customer.findByPrimaryKeyIfAny(name) to locate the customer object, and compare the customer object's password value with the one typed.

> **NOTE:** *In a real application, the password would ideally be stored in some encoded form, and the encoding function would be applied to the entered password before comparing to the encoded stored password.*

4.8 Summary

As we saw in the Purchase Order example, a DASL application is built from two distinct specifications:

- A **Business Object Specification**, or **BOS**, is a set of *business objects* that describe the entities, relationships, business rules, constraints, and procedures of the business. This collection of objects is typically shared across many applications.
- An **Application Usage Specification**, or **AUS**, is a description of the **business tasks** that make up an application. The business tasks are described as a set of *states* and the *transitions* from

one state to another that, taken as a whole, describes how the application interacts with the underlying business objects. The AUS also defines a set of *transactions* that describe atomic changes that the application makes to the state of the underlying business objects.

These two components capture the entire semantics of a correctly working application, including transaction semantics, concurrency considerations, and how each individual attribute is used. There is no need for the application writer to know or be aware of the deployment architecture of the application, or the distribution of the application's functionality within that distributed environment.

Reference Manual

1 Common Language Elements

We are now ready to learn the DASL language more formally. This chapter describes the common elements shared by all of the DASL sub-languages. Subsequent chapters describe the BOS, AUS, and OQL sub-languages.

1.1 BNF Notation Used

The formal syntax of the DASL language will henceforth be described using a mixture of descriptive text together with a notation called BNF.[38] The BNF notation uses two kinds of symbols, as follows:

`<xyz>`	is a non-terminal symbol. It represents a concept and does not appear in an actual program written in the language.
`abc`	is the terminal symbol 'abc', a series of characters that appear directly in programs written in the language.

A language is described using a set of *productions*, or statements that define non-terminal symbols as a sequence of other BNF symbols. A typical BNF production

```
<ifStatement> ::= if ( <expression> ) <block> <elsePart>
```

defines the non-terminal symbol **<ifStatement>** as a sequence of terminal and non-terminal symbols, specifically, the keyword **if** followed by a parenthesized expression and then a block, which is then followed by an **<elsePart**>. Other productions will define **<expression>, <block>,** and **<elsePart>**. Ultimately, every non-terminal can be reduced to a series of terminal symbols in the language.

The following BNF extensions[39] are used to represent frequently occurring cases more succinctly:

<xyz>?	is an optional **<xyz>**
<xyz>*	is zero or more **<xyz>**
<xyz>+	is one or more **<xyz>**
[**<xyz> <pqr>**]	is an optional **<xyz> <pqr>**
<xyz> [, **<xyz>**]*	is one or more **<xyz>** separated by commas
[**<xyz>** ;]*	is zero or more **<xyz>** terminated by semicolons
[**<xyz>** ;]+	is one or more **<xyz>** terminated by semicolons
[**<xyz>** | **<pqr>**]	is either **<xyz>** or **<pqr>** or nothing

For example, using the above abbreviations, we can now express the optional **<elsePart>** in the syntax of an **<ifStatement>** as follows:

38 Backus-Naur Form.
39 We use a slight modification of Niklaus Wirth's extensions to BNF.

```
<ifStatement> ::= if ( <expression> ) <block> [ else <block> ]
```

From the above notation, it can be seen that the following symbols in the BNF grammar have special meaning:

```
::=   <   >   [   ]   |   *   +   ?
```

The following symbols must be quoted if they appear as terminal symbols:

```
'::='  '<'  '>'  '['  ']'  '|'  '/*'  '/**'  '*/'  '//'
```

Note that three of the BNF symbols

```
*   +   ?
```

need not be quoted because in the special notation they are always used directly after the '>' or ']' symbol, such as **<case>***.

1.2 Identifiers and Keywords in DASL

Identifiers in DASL must start with a letter – including an underscore character (_) or a dollar sign ($) – followed by zero or more letters or digits or both. Identifiers in DASL follow the same lexical structure as Java identifiers, with two exceptions:

1. Identifiers and keywords in DASL are not case sensitive.
2. DASL has additional reserved keywords beyond those defined by Java.

1.2.1 Lack of Case Sensitivity

The Java Programming Language is sensitive to the alphabetical case of its terminal symbols. For example, in Java when you declare *Employee employee,* the identifier *Employee* refers to a class, while the identifier *employee* refers to an instance of *Employee*. The two identifiers differ only in capitalization.

However, some other languages are not sensitive to the capitalization of terminal symbols. In particular, the standard query language SQL is insensitive to case. In SQL, you cannot define *Employee* and *employee* separately.[40]

The BOS and AUS sub-languages treat identifiers in a case insensitive way, to prevent collisions of BOS object names and their attributes at the implementation level. For example, declaring a variable or attribute *helloThere* prevents you from declaring another variable or attribute *hellothere* in the same scope. However, within the body of a user defined method, case sensitivity is maintained so that the method can call Java methods, for which identifiers are defined in a case sensitive way. Thus, variables declared within the method, as well as references to business object classes and their fields, must all have the correct case. In particular, if they refer to variables defined in the BOS outside of a

40 SQL allows case sensitivity as a special case when identifiers are quoted with double quotes, e.g., “Employee” and “employee”. However, this convention would not be appropriate for DASL because it conflicts with the use of double

method, they must use the exact same case that was used in the variable declaration.[41]

1.2.2 Reserved Keywords

The following keywords are reserved by DASL, regardless of the capitalization.

- ***abstract***
- ***assert***
- **boolean**
- **break**
- **byte**
- **case**
- **catch**
- **char**
- ***const***
- **CHECK**
- **class**
- **code**
- **computed**
- **constant**
- **constructor**
- **continue**
- **DASL**
- **declare**
- **default**
- **do**
- **double**
- **EBT**
- **else**
- **EMPTY**
- **entry**
- **error**
- **extends**
- **factory**
- **false**
- ***final***
- **finally**
- **float**
- **for**
- **goto**
- **GROUP**
- **hidden**
- **if**
- **immutable**
- **import**
- ***implements***
- **import**
- **include**
- **index**
- **insert**
- **instanceof**
- **int**
- ***interface***
- **inverse**
- **java**
- **long**
- **method**
- ***native***
- **new**
- **null**
- **on**
- **OQL**
- **ORDEREDBY**
- **owns**
- **package**
- **persistent**
- **PRIMARY**
- ***private***
- **properties**
- ***protected***
- ***public***
- **readonly**
- **references**
- **remove**
- **return**
- **returns**
- **shared**
- **short**
- **SQL**
- **state**
- ***static***
- **super**
- **switch**
- **synchronized**
- **this**
- **throw**
- **throws**
- **transaction**
- **transient**
- **transition**
- **true**
- **try**
- **UNIQUE**
- **usage**
- **using**
- ***void***
- ***volatile***
- **when_committed**
- **when_set**
- **while**
- **writeonly**

quotes to denote string literals.

41 Yes, this is an annoyance. A future version of DASL will warn if the wrong case is used in a method body, and correct it automatically. Aren't computers wonderful?

The list includes all keywords reserved by the Java language, including any case in which the keyword appears. Thus, **CLASS, class, and Class** are all reserved by DASL.

The case shown is the preferred case. For style, keywords taken from SQL and OQL are usually shown in upper case. Keywords reserved by Java which are not used directly in DASL are italicized. These keywords are reserved to simplify deployment of the DASL application in Java.

1.3 Package Declaration: <packageDeclaration>

```
package <packageName> ;

  <packageName> ::= <name> [ . <name> ]*

  <name> ::= <identifier>
```

Each DASL specification file contains a package declaration that identifies the namespace in which this component may be referenced externally. The package declaration defines the root package for the application component. For example:

```
package com.mycompany.hr;
```

1.4 Properties: <properties>

Properties consisting of a list of name-value pairs may be associated with many elements in the DASL language. Properties are often used to give hints to the deployment engines, and thus they are closer to compiler directives than metadata.

The general syntax for specifying properties of an element is:

```
PROPERTIES { <property> [, <property>]* }
```

where

```
<property>  ::= <propertyName> = "<propertyValue>"
            ::= <propertyName>
```

The second form sets the value to the empty string "" and is useful for declaring boolean properties, that is, properties that are either present or not present.

1.5 DASL Comments

The DASL language supports 3 kinds of comments:

```
/** DASL Documentation Comment */

/* C-style Comment */

// Single-line Comment
```

Instances of the last kind of comment that occur outside of method bodies are not preserved by the

DASL GUI editor when writing out the file. The GUI editor uses this kind of comment to write its own top-level comments in the output file, such as the date and time the file was written out. When the file is read in again, the "//" comments outside of method bodies are discarded.

1.6 The Type System: <type>

```
          <type>    ::= <className>
                    ::= <EBTName>
                    ::= Decimal [ ( <precision> [ , <scale> [ ,
<roundingMode> ]] )]
                    ::= Collection [ '<' <type> '>' ]
                    ::= List [ '<' <type> '>' ]
                    ::= Set [ '<' <type> '>' ]
                    ::= SortedSet [ '<' <type> '>' ]
                    ::= Map [ '<' <type>, <type> '>' ]
                    ::= <type> '[]'

          <precision> ::= <decimalNumeral>
          <scale> ::= <decimalNumeral>
          <roundingMode> ::= /** See text below */
```

DASL defines a type system for scalar attributes and relationships that is very similar to Java's type system. DASL's types differ from Java in that they do not specify low-level machine implementation details of the type. DASL's type system includes strongly typed collections, which were recently added to Java as of version 1.5. DASL also includes database types (i.e., those found in the Java package java.sql) among its standard types.

The meaning of the optional modifiers for the *Decimal* declaration are described in the section on Decimals below.

There is a subtle difference between BOS and AUS types. In the AUS, a type may be qualified by the local name of the BOS in which a class or EBT is defined. This difference will be explained in the chapter describing the AUS language.

1.6.1 No Distinction Between Primitive and Class Types

Unlike Java, DASL does not distinguish between Java primitive types (e.g., *int*) and the corresponding immutable class types (e.g., *Integer*). The class types are preferred, though DASL translates the built-in type names to class type names as a convenience. The DASL language automatically "boxes" and "unboxes" class types to primitive types, as required within method bodies.

A subtle but important difference exists between the primitive and class-based types when declaring query method return types. If a query method can return a null value, the method must be declared to return a class-based type, such as *Integer*, rather than a primitive type, such as *int*.

1.6.2 Avoidance of Implementation Classes

In DASL, one uses business object class names rather than implementation classes to invoke constructors, such as *new persistent Employee()*, and also to invoke factory methods, such as *PurchaseOrder.myOrders()*. By contrast, in Java one uses the implementation class name, such as *new*

EmployeeBean(), rather than the interface name, such as *new Employee()*. The Java convention requires knowing the implementation class, which is contrary to DASL's goal of architecture independence through specifying the desired semantics, rather than how to implement them.

1.6.3 Strongly Typed Collections

DASL provides strongly typed collections. For example, a method may be declared to return *List<LineItem>*, or even *List<Set<String>>*. The syntax and semantics follow those used in Java as of release 1.5.

1.6.4 EBTs

DASL makes a distinction between value-based types and identity-based types. The **business object types** used in the programmer's guide are types whose identity is defined only by their primary key, but independently of the value of their other attributes and relationships. *PurchaseOrder* and *LineItem* are examples of business object types. Business object types are assumed to be mutable and identity-based.

DASL also supports **EBTs** (Extended Basic Types) whose identity is defined solely by its value. DASL predefines a standard set of EBTs, and it allows applications to define and share their own EBT types. *Integer* and *List* are examples of predefined EBT types.

Instances of EBTs generally have no relationships to other classes.[42]

EBTs may optionally be declared immutable if their value cannot be changed once an instance is created. The predefined *scalar* EBT types are all immutable. Collection-valued EBT types are defined as being mutable, primarily due to programming efficiency considerations.

1.6.5 Predefined Scalar Types

The scalar EBTs that are predefined by DASL are:

- The Java basic types (primitive types and their immutable class equivalents)

• **Boolean**	either true or false
• **Character**	16-bit Unicode 1.1 character
• **Byte**	8-bit integer (signed)
• **Short**	16-bit integer (signed)
• **Integer**	32-bit integer (signed)
• **Long**	64-bit integer (signed)
• **Float**	32-bit floating point (IEEE 754-1985)
• **Double**	64-bit floating point (IEEE 754-1985)

- String types (all but Clob are implemented using java.lang.String)

42 In principal, an EBT class can inherit from other EBT classes, though inheritance is not fully supported by the current deployment engines as of this writing.

• **String**	Basic String type[43]
• **CaselessString**	String that is compared ignoring case
• **Text**	String that is intended to hold more than one line of text, such as a paragraph, a page, or several pages[44]
• **CaselessText**	Text that is compared ignore case
• **Clob**	**C**haracter **L**arge **OB**ject that holds a very large number of characters[45] (accessed using java.sql.Clob)

- ByteString types

• **ByteString**	Strings of bytes (implemented as byte[])
• **Blob**	**B**inary **L**arge **OB**ject that holds a very large number of bytes[46] (accessed using java.sql.Blob)

- Database types (JDBC types for defining database attributes)

• **Decimal**	arbitrary precision fixed point: java.math.BigDecimal
• **Numeric**	arbitrary precision fixed point: java.math.BigDecimal[47]
• **Date**[48]	a calendar date: java.sql.Date
• **Time**	the time of day: java.sql.Time
• **Timestamp**	a date and time on Earth: java.sql.Timestamp

- Additional database types

• **Interval**	DASL-supplied class representing the difference of two Date, Time, or Timestamp values

1.6.6 Decimals As First Class Types

In DASL arithmetic operations may be applied to mixtures of Decimal and predefined numeric types, e.g., *lineTotal = this.unitPrice * this.quantity*. Decimal literals may be assigned to Decimal variables, and mixed with Decimal variables in expressions, e.g. *price = 5.12* and *halfPrice = price / 2,* where both *price* and *halfPrice* are Decimal variables, so the literals “5.12” and “2” are interpreted as

43 The *String* and *CaselessString* types may be used for attributes that are part of a PRIMARY KEY. The limit on the length of a String that is stored persistently depends on the DBMS that is chosen at deployment time. A safe portable limit is 255 characters. Many DBMS vendors support a limit of 4000 characters or more.

44 A reasonable portable limit on the size of a Text or CaselessText is 32,700 characters.

45 The size limit is dependent on the DBMS, and is generally several gigabytes. Be aware that some DBMS vendors do not support this type at all.

46 Previous footnote applies to Blob also.

47 DECIMAL and NUMERIC differ only in how some DBMS systems implement them, according to the ANSI SQL standard. DECIMAL may, in some cases, be implemented using hardware decimal arithmetic, whereas NUMERIC may be implemented using software decimal arithmetic.

48 Note that Java provides another type called 'Date', java.util.Date, that has different semantics from java.sql.Date. Specifically, a java.util.Date is more like Timestamp, in that it represents both a date and a time. The taxonomy of date/time types provided by DASL correspond directly to values stored in most actual databases, and are thus preferred

Decimal literals.[49] Decimal attributes and variables in the BOS and Decimal variables in the AUS may be declared with a specific precision and scale, in which case DASL enforces that precision and scale on assignment. A precision of 0 signifies arbitrary precision, so that Decimal(0,2) specifies arbitrary precision with a scale of 2.

When an assignment is made to an attribute or variable with a declared scale, and the assigned decimal value has a larger scale than the attribute or variable to which it is assigned, rounding must take place. The default round mode used by DASL is called **HALF_EVEN**. Non-default rounding modes can be specified following the precision and scale,[50] e.g.,

```
Decimal(20,2,HALF_UP) salary;
```

The rounding modes supported include all of the rounding modes for Java's BigDecimal. The most common of these are:

- ***HALF_EVEN*** – the number is rounded to the nearest digit, and if it is half-way between two values then it is rounded to the nearest even digit. This kind of rounding is sometimes called “Banker's rounding”. For example, 12.345 rounded to a scale of two digits is 12.34, and 12.355 rounded to a scale of two digits is 12.36.
- ***HALF_UP*** – the number is rounded to the nearest digit, and if it is half-way between two values then it is rounded to the digit above. For example, 12.345 rounded to a scale of 2 digits is 12.35, and 12.355 rounded to a scale of two digits is 12.36.
- ***DOWN*** – the number is truncated (rounded toward zero) by simply discarding the excess digits to the right of the decimal point.

Due to an implementation restriction, literal expressions are treated according to Java rules. For example, the following code snippet:

```
Decimal d = 1 / 2;
```

will assign the value *0* to the variable *d* because in Java, the expression *1 / 2* is evaluated as an int expression, resulting in *0*. To avoid problems with Java capture of literal expressions, create Decimal variables for the constants involved in the expression, thus:[51]

```
Decimal one = 1;
Decimal two = 2;
Decimal d = one / two;
```

over java.util.Date.

49 Alas, the current implementation does not handle expressions of literals as Decimals as expected for Decimals. For example, *price = 1 / 2* results in *price = 0,* because the expression “1 / 2” is treated as an integer division by the Java compiler. To force Decimal arithmetic on literals, use *price = new Decimal(1) / new Decimal(2)* instead.

50 This new syntax is being added to DASL as this manual goes to print.

51 Yes, this is ugly, but unfortunately the little trick we use to translate DASL expressions into Java works by finding invalid Java expressions, and 1 / 2 is perfectly valid Java. Java does not understand that Decimal is a more precise value than int, so it should really be performing all its literal expression evaluation using BigDecimal.

1.6.7 Predefined Collection Types

The collection types that are predefined by DASL may optionally be restricted to contain only objects of a single type, or that inherit from that type. The DASL collection types are

• **Collection ['<' <type> '>']**	A collection with unspecified ordering and uniqueness
• **List ['<' <type> '>']**	An ordered collection, possibly with duplicate items
• **Set ['<' <type> '>']**	An unordered collection, without duplicates
• **SortedSet ['<' <type> '>']**	An ordered collection, without duplicates
• **Map ['<' <type>, <type> '>']**	A mapping from a set of unique keys to an object
• **<type> '[]'**	An array is a list that cannot grow or shrink, whose elements are initialized to zero or null.[52]

1.7 Method Declarations: <methodDeclaration>

The DASL programmer may define methods that operate on the objects defined by the BOS. Both factory and instance methods may be defined. For example, in the PurchaseOrder Application, the method *orderTotal()* which returns the total cost of a *PurchaseOrder* is a programmer-defined instance method. The method *myOrders()* which returns all *PurchaseOrder* instances submitted by a particular customer is a programmer-defined factory method.

Programmer-defined methods may either be stated in terms of the procedural steps required to compute a result or cause a change in the state of an object, or else they may be specified as a query that computes a result. Queries may be specified in either the DASL/OQL query language, or in the SQL query language.

```
<methodModifier>* method <methodName> ( <methodParamDcls>? )
                   [ returns <type> ]
                   [ throws <exceptionList> ]
                   <methodProperties>?
                   <methodImplementation>

<methodModifier>   ::=  hidden
                   ::=  factory

<methodImplementation> ::= { <daslMethodBody> }
                       ::= ( <expression> )
                       ::= OQL ( <OQLQuery> )
                       ::= SQL ( <SQLQuery> )

<daslMethodBody> ::= <daslMethodStatement>*

<methodParamDcls> ::=  <methodParamDcl> [ , <methodParamDcl> ]*

<methodParamDcl> ::=  <type> <parameterName>

<exceptionList> ::=  <qualifiedClassName> [ , <qualifiedClassName> ]*


<qualifiedClassName>  ::=  <className>
                      ::=  <packageName> . <className>
```

52 The Array type is provided for declaring parameters and return types of methods.

```
        <methodProperties> ::=  <properties>
```

The modifier ***factory*** makes the method a factory method. Factory methods are methods defined on a business object class that do not apply to a particular instance of the class, but rather to the class as a whole. The method to create a new object of a given type is usually a factory method, since otherwise it would be impossible to create the first object of that type. A method to find an existing *PurchaseOrder* given its *orderId* is also a factory method.

Instance methods are methods that apply to a specific instance of the class. The method to return the date a particular *PurchaseOrder* was created is an instance method. Syntactically, the absence of the modifier *factory* makes a method an instance method.

Methods that are meant to be called only from within the BOS may be declared using the modifier ***hidden***. Such methods are not available outside the BOS in which they are defined, such as another BOS or an AUS.

The ***throws*** clause of a method declaration declares that the method may throw a particular exception. The semantics of this declaration are those of the Java language.

The method implementation may be specified either as DASL procedural statements, or as a query expressed in DASL's standard object-oriented query language, DASL/OQL, or in standard SQL . The DASL method statements are an extension of those allowed by the Java programming language.[53] DASL/OQL is a subset of the OQL object query language, with some additions.

1.8 Query Method Signatures

Query methods are methods whose body begins with either of the keywords *OQL* or *SQL*. A query method with a signature that returns a collection of some kind will return an empty collection if the corresponding query does not select any objects or values.

```
  method <methodName>( <methodParameters> ) returns <collectionType>
```

For example, in the PurchaseOrder Application, the user defined method myOrders() for the business object *PurchaseOrder* is defined as follows:

```
    factory method myOrders( String billToName ) returns List<PurchaseOrder>
```

If the query selects basic types (e.g., Long, String, Date, etc.) rather than business objects, then these will be returned as a collection of Java objects.

Query methods defined with a signature that returns a single object of some kind will return null if the corresponding query does not select any objects.

```
  method <methodName>( <methodParameters> ) returns <objectType>
```

If the query selects more than one object, an exception will be thrown.

53 In the future, DASL may accept other procedural language syntaxes as well.

Query methods defined with a signature that returns a basic type (e.g., Long, String, Date, etc.) will return null if the corresponding query does not select any values.

```
  method <methodName>( <methodParameters> ) returns <basicType>
```

If the query selects more than one value, an exception will be thrown.

The syntax for the DASL/OQL query language is described fully in Chapter 7.

1.9 Procedural Method Statements

The statements that may be used in a DASL method body are very similar to those that can be used in a Java method, with some added syntactic sugar that is much needed within enterprise applications. These extensions have been defined to work on what would otherwise be invalid Java expressions, so that all statements legal in Java have the same meaning in DASL as in Java.

The scripting statements that may occur in AUS scripts are somewhat limited by comparison. They will be described in the AUS chapter later on.

At this point, we will assume the reader is familiar with Java and simply present the extensions provided by DASL. An appendix of this manual gives the complete syntax of DASL method statements.

1.10 Summary of Statements And Expressions Not Found in Java

The statements and expressions provided by DASL that do not have equivalents in Java are summarized below.

1.10.1 Direct Access to Attribute and Relationship Accessors

DASL allows direct access to attributes and relationship accessors, even if the attributes and accessors are private members, without requiring the JavaBean accessor/mutator method syntax: e.g., *po.billToName* (instead of) *po.getBillToName()* and *po.billToName = x* (instead of) *po.setBillToName (x)*. The appropriate accessor or mutator method will be invoked.

1.10.2 Use of the Business Object (interface) Name to Access a Constructor

DASL allows constructors and factory (static) methods of object classes to be invoked by name, e.g., *new persistent LineItem(...),* and *PurchaseOrder.myOrders("Bob"),* even though the business object class name is usually the name of the interface rather than the implementation class, and even though the actual implementation class is often instantiated in a distributed environment, such as on an application server.

The syntax of constructor expressions and factory method expressions is:

```
 <constructorExpression> ::= new [ persistent ] <businessObjectName> ()
 <factoryMethodExpression> ::= <businessObjectName> . <factoryMethod> (
                                                             <parameters> )
```

1.10.3 Java 5.0 Simple Collection Iteration With Enhanced Operation

DASL supports the Java 5.0 iteration primitive over collections of objects, e.g.,

```
          for ( LineItem li:  po.lineItems ) total += li.lineTotal;
```

Within the iteration block, DASL supports the special prefix operator *remove,* which removes the current iteration element from the collection. For example,

```
         for ( LineItem li: po.lineItems) if (li.quantity == 0) remove li;
```

If a class named remove is in scope, using the remove keyword may cause an error. In this rare case, the special keyword *$remove* may be used instead.

1.10.4 Java 5.0 Strongly Typed Collections

DASL supports Java 5.0 strongly typed collections, both in its type system and in method bodies, e.g.,

```
          List<Integer> purchaseOrderIDs;
```

1.10.5 Java 5.0 Boxing and Unboxing

DASL supports implicit conversion (boxing and unboxing) between primitive Java types (e.g., int) and the corresponding immutable class types (e.g., Integer). e.g.,

purchaseOrderIDs.add(1);

1.10.6 Decimals Declared With Precision, Scale, and Rounding Mode

Decimal variables may be declared with a specific precision, scale, and rounding mode, in which case DASL ensures that values assigned to such variables are converted to the desired precision and scale using the desired rounding mode, if necessary.

1.10.7 Java 5.0 Extended Switch Statement

DASL supports the Java 5.0 extended switch statement that takes class variables as the switch variable, e.g., for String:

```
    switch (result) { case “inserted”: ...; case “deleted”: ... default: ...; }
```

1.10.8 High Level Exception Handling

The *on error* and *on return* statements are a high level alternative to Java's *try/catch* blocks. Unlike Java's *try/catch*, the *on error* block is declarative and not executed as an inline statement. Because the business logic need not be nested inside the error handling, as with *try/catch,* business logic is easier to read.

```
        on error [ <exceptionRestriction> ] <onErrorBlock>

        on return <onReturnBlock>

    <exceptionRestriction> ::= (  <exceptionType> <name> )

    <onErrorBlock>  ::= <statement> | { <statement>* }
    <onReturnBlock> ::= <statement> | { <statement>* }
```

The *on error* statement applies to the entire block in which it occurs lexically, as well as any inner block that does not have its own *on error* for the same exception. An *on error* without an explicit exception covers all exceptions. An *on return* applies to all exit paths from the block in which it occurs (like Java's *try/finally*).

A nested *on error* statement may be declared within `<onErrorBlock>`, just as a nested *try/catch* may be declared within a *try/catch* block in Java.

If none of the exceptions caught by an outer *on error* statement occur, its `<onErrorBlock>` will not be executed. On the other hand, when the block containing an outer *on return* is exited, the statements in <onReturnBlock> will be executed, regardless of whether an error occurred. Thus, an *on return* located inside `<onErrorBlock>` applies only if `<onErrorBlock>` was entered.

1.10.9 Uniform Treatment of Date/Time Constructors

DASL corrects incompatibilities between java.util.Date() and java.sql.Date() that are annoying. It provides default constructors for the java.sql classes Date, Time, and Timestamp that return the current date and time. For example, the syntax *Date d = new Date()* is accepted, even though java.sql.Date has no such constructor. It also provides a constructor with a single String parameter for the java.sql classes Date, Time, and Timestamp, that parses the string and creates the object with the corresponding value, e.g., *Date d = new Date("3/16/1991").*

1.11 A Complete DASL Application

```
         <DaslApplication>  ::=  <BOS>+ <AUS>*
                            ::=  <BOS>* <AUS>+
```

A DASL application consists of at least one BOS or at least one AUS, and possibly other BOSes and/or AUSes. A complete application normally consists of one or more BOS and one or more AUS, where one of the AUS specifications is a main application task, and the others are tasks that are called by the main application task.[54]

1.12 Unit of Compilation and Unit of Execution

In contrast to the Java programming language, where the unit of *compilation* is a class, in DASL the unit of compilation is a DASL application specification, which can include multiple business objects and multiple application states. Thus, it is not possible to ask DASL to compile a single business object within a BOS. This restriction enables DASL to provide some high-level class-related semantics not present in Java, such as dynamic inheritance within a BOS, and two-way ownership relationships between objects.

Also, in contrast to Java, where the unit of *execution* is also a class, in DASL the unit of execution is an application. Again, this restriction enables DASL to provide some high-level semantics concerning the business process, such as application transactions defined via the states that comprise the transaction.

54 As of this writing, tasks are not implemented. If a DASL program contains more one AUS, each one is treated like an independent application, and a "top level" page is created with links to the separate applications. However, this behavior is temporary.

2 DASL/BOS Language Specification

Business objects define the entities, relationships, business rules, and constraints that underlie the application. A business object specification (BOS) is part of a DASL application that specifies a set of related business objects that have a close association with each other.[55] A business object diagram describes object types and their relationships.

2.1 Summary of the Key Elements of a BOS Specification: <BOS>

The BOS sub-language is used to define a set of business object types, their characteristics, attributes, relationships to other objects, and constraints on their values. Most of this information is declarative in nature, rather than procedural.

At the topmost level, a BOS contains the following elements:

- The Package Declaration (see Common Elements)
- Imports of External Objects
- Properties of the BOS
- Business Object Class Declarations
- EBT Class Declarations
- Event Class Declarations

2.1.1 Properties of Elements of the BOS

Properties may be associated with the following elements in the BOS:

- The BOS itself
- Imported Business Objects
- Business Object Classes
- EBTs
- Attributes
- Relationships
- Primary and Unique Keys
- Groups
- Methods
- Constraints

The properties for the BOS itself are placed at the top level, outside of a class declaration. The place where properties are specified for other of these elements is indicated by <properties> in the BNF

55 Those readers familiar with database terminology can think of the BOS objects as residing in the same DBMS schema.

syntax for that element.

2.1.2 Import of Business Objects

Business objects from other BOS compilation units may be referenced by importing them:

```
import [ <package> . ] <bosName> . <className> [ as <alias> ]
                                               [ uuid <uuid> ]
                                               [ <properties> ] ;
```

> **Temporary Restriction**: DASL/BOS currently does not support the ability to import all objects from a BOS using a single statement.

The import statement declares an external business object, i.e., a business object that is defined in another BOS. The *bosName* is the name of the file in which the BOS is stored, without the ".bos" extension.

The *alias* is the name by which this external business object will be known in the current BOS. It must not conflict with any top level object names in the current BOS. If *alias* is not specified explicitly, the class name in the original BOS is the *alias.*

The optional *uuid* is the value of the UUID (Universal Unique Identifier) property associated with the object in the BOS, to disambiguate it from any other business objects that might be defined by the system. A UUID property may be declared explicitly for a class. If not, the UUID property is set to a unique value (for any class that does not define one) the first time the DASL GUI saves a BOS. The uuid may be specified as a safeguard against accidentally importing a class other than the intended one.

2.1.3 Import of Java Objects

Java classes may be imported using:

```
import java [ <package> . ] <className> [ as <alias> ] [ <properties> ] ;
```

The *import java* statement imports an external Java class into the type system. The *alias* is the name by which this external Java class will be known in the current BOS. It must not conflict with any top level object names in the current BOS. If *alias* is not specified explicitly, the Java class name is the *alias.*

2.1.4 Referencing External Java Objects

Within the body of a method, external Java classes that have not been explicitly imported may be referenced using their fully qualified name. Note that this shortcut is allowed only for referencing Java classes. External business objects must be imported to be referenced.

2.2 Business Object Classes, EBTs, and Events

2.2.1 Declaring Business Object Classes

A class declaration defines a business object:

```
<classModifier>* class <className> <inheritanceDeclaration>? {
         <classElement>*
}

<inheritanceDeclaration> ::=  extends <name> [ using <inheritanceImpl> ]

<inheritanceImpl> ::=  STATIC_INHERITANCE | DYNAMIC_INHERITANCE
```

The modifiers for a business object class declaration are:

• **persistent**	The object will be stored persistently. Attributes of the object are also persistent, unless declared to be transient.
• **transient**	The object will be transient, and thus available only for the duration of the session in which it is referenced.
• **facade**	The object will be stored persistently using an implementation that is provided by the application, with some defaults available from DASL if desired. Facade objects are useful for providing what appear to be fine-grained persistent objects that are actually a facade over a legacy persistent store. They are a recently added part of DASL that appears promising for legacy data access as well as providing fine-grained control over the persistent mapping.
• **abstract**	No instances of the business object may be created, i.e., the class is being defined for the purpose of sharing common elements in its subclasses.

Only one of the modifiers *persistent, transient,* or *facade* may be specified. If none of these modifiers is provided, the default is *transient*. The modifier *abstract* also applies to any of these business object types.

2.2.2 Declaring User Defined EBTs

The user may define EBTs explicitly in a BOS. These are then treated by DASL as value-based types just like the predefined DASL EBTs.

```
<ebtModifier>* EBT <EBTName> <inheritanceDeclaration>? {
         <classElement>*
}
```

The modifiers for an EBT declaration are:

• **immutable**	The EBT is known to be immutable, i.e., the value of an instance cannot be changed once the instance is created. Thus, there can be multiple references to an instance of this EBT without concern that changing the EBT through one reference will cause the value seen through other references to change.
• **code**	The EBT must take on one of a set of values stored persistently, where the set of values is assumed not to vary during the execution of an application. Code EBTs are useful for representing state codes, shipment codes, and other enumerated values, where the possible values are not known at deployment time. A code may be thought of as a constraint defined by data stored in the database.[56]
• **abstract**	No instances of the EBT may be created, i.e., the EBT is being defined for the purpose of sharing common elements in its subclasses.

By default, EBTs are assumed to be mutable, not codes, and not abstract.

When an EBT class is declared immutable, DASL is able to optimize the use of the EBT in certain ways. For example, it can create more than one reference to the same instance without concern that the EBT's value might be changed through one reference, and thus unintentionally change the values seen by other references. For example, the EBTs that represent all the DASL scalar types such as Integer and String are immutable.

EBT's that represent collection types are generally not immutable. In particular, all of the DASL collection EBT types are not immutable because their value can be changed through insertion of new elements and deletion of existing elements. Knowing that they are mutable, DASL does not create multiple references to the same instance of the collection types to avoid accidental propagation of changes.

User-defined EBTs that are intended to be used as pure, immutable values should be declared as immutable, to allow DASL to optimize their use by not bothering to copy them when they are assigned. User-defined EBTs that provide mutator methods should not be declared as immutable to avoid accidental propagation of changes.

For example, consider defining an EBT such as Address, to represent the billing or shipping address of a customer. If all you want is to package up the 5 strings as a single immutable value, declare it as an immutable EBT:

```
immutable EBT Address {              // immutable EBT
        String address1;
        String address2;
        String city;
        String state;
        String zipcode;

}
```

56 Codes are not currently supported by any deployment engines.

Using this definition, the only way to create a new address EBT is to supply all the fields at once. It is not possible to selectively update fields once the EBT has been created, e.g., just the street address. If you want to provide a type that can be selectively updated, you can omit the modifier *immutable*. In this case, you can selectively update the EBT, but DASL must generate code to copy the EBT before it is assigned to another variable.

If what you really intend is to have an Address object that is shared among several business objects, such that changing the Address via references from any of the business objects will change it for all objects, then the Address object should not be declared as an EBT. Instead, declare it as a persistent business object with a primary key, and have the other business objects refer to it via a *references* relationship.

2.2.3 Declaring Events

The syntax and semantics for events have not been finalized as of this writing (see the appendix for details).

```
<eventModifier>* event <eventName> <inheritanceDeclaration>? {
        <classElement>*
}
```

The modifiers being considered for an event declaration are:

• **transient**	Events of this class are passed to applications registered for it, but instances are not remembered and preserved indefinitely for future applications that register for this class of event.
• **persistent**	Events of this class are remembered and preserved indefinitely for future applications that register for this class of event.
• **abstract**	No instances of the event may be created, i.e., the event class is being defined for the purpose of sharing common elements in its event subclasses.

2.2.4 Inheritance Hierarchy

Business objects, EBTs, and events may be related in an inheritance hierarchy that includes other entities of the same kind. Inheritance of types provides for specialization or refinement of an object.

The optional ***extends*** clause in a class declaration indicates an inheritance relationship between the class and a base class. The optional ***using*** clause specifies the kind of inheritance to use.

It is often useful to define a business object by refining another business object. For example, in a human resources (HR) database, it may be useful to describe various kinds of people: employees, employees who are managers, and dependents of employees. The relationships between the business object classes can be expressed as follows:

```
    class Person { CaselessString name; String id; ... PRIMARY KEY (id); }

    class Employee extends Person { references Manager manager(1,1)
                                           inverse reports(0,n); }

    class Manager extends Employee { String departmentName; }

    class Dependent extends Person { String relation; }
```

For example, in a company employee database, certain employees may be managers, where managers have additional relationships (such as "direct reports") and attributes. This relationship between the *Employee* and *Manager* can be described by saying that *Manager extends Employee*, and then adding the additional relationships and attributes to the *Manager* object. The *Employee* is the base object and *Manager* is the refined object.

2.2.5 Static and Dynamic Inheritance

Two kinds of inheritance are supported: static inheritance and dynamic inheritance. Static inheritance is what Java provides: the specific class of an object is determined when that instance is created, and it cannot be changed. For example, once a particular employee, “Sam”, is created as an *Employee*, he cannot later become a *Manager*. Clearly this restriction could not be tolerated in a real company HR application!

Dynamic inheritance allows instances of a class to *morph* their type dynamically, after they are created, either up the inheritance tree (to more general types) or down the tree (to more specific types). There is no common root class for all business objects, and furthermore, dynamic inheritance is restricted to objects in a particular BOS. These restrictions ensure that the entire tree of possible morphings of a given class is known at the time the application is compiled.

By default, static inheritance is used unless the following syntax is used:

```
    class A extends B using DYNAMIC_INHERITANCE { ... }
```

The default inheritance is the equivalent of

```
    class A extends B using STATIC_INHERITANCE { ... }
```

In the example of *Manager* and *Employee*, it is best to use dynamic inheritance because it allows particular instances to change or morph between *Manager* and *Employee* dynamically as the application executes. This will happen if an *Employee* is promoted to a *Manager*, for example. If dynamic inheritance is used, *Manager* objects will have an instance method *becomeEmployee*(), and *Employee* objects will have an instance method *becomeManager*().

Restriction: A business object in the current BOS may extend a business object *using DYNAMIC_INHERITANCE* only if the object being extended is defined in the same BOS.

2.3 Attribute Declarations

Objects may have attributes, i.e., named values. For example, the *PurchaseOrder* object has an order id, order date, and shipping and billing information. Attributes may be declared to have any EBT type.[57]

DASL distinguishes 4 kinds of attributes:

• **persistent**	A persistent attribute is stored persistently and retrieved as part of the object. Attributes of persistent objects are persistent by default, unless they are declared to be transient. The attributes of transient objects cannot be persistent.
• **transient**	Infrequently, and usually for reasons of programming efficiency, a persistent business object may have a transient attribute associated with it. The transient attribute is not preserved when the object is stored for later retrieval. Transient attributes can be used to temporarily remember things about the object while it is being used.
• **computed**	Computed attributes are attributes whose value is defined as a computation. A computed attribute is implemented as a member method of the business object with no input parameters, and that returns the attribute value. However, declaring such a method as a computed attribute allows it to be referred to syntactically as an attribute.
• **constant**	Constant attributes are useful for providing unchanging values associated with a class of objects.[58]

2.3.1 Realized Attribute Declarations: *persistent* and *transient*

```
<realizedAttributeModifier>* <bosType> <attributeName>
          [ PRIMARY KEY | UNIQUE KEY ]
          [ = <exprOrReturnBlock> ]
          <attributeConstraintDeclaration>*
          <attributeProperties>? ;

<attributeConstraintDeclaration> ::= <constraintDeclaration>
```

Realized attributes are stored explicitly as part of the object's value.

The modifiers for realized attribute declarations are:

• **hidden**	The attribute is not accessible outside the BOS in which it is defined (i.e., from another BOS or any AUS), except that it may be accessed from an object that inherits from the object that declares it.

57 Currently, arrays and collections are not supported as attributes of business objects. Thus, attributes are currently restricted to scalar EBT types.

58 Constant attributes are usually implemented as *public static final* class members in Java.

• **readonly**	The attribute value may be read but not modified from outside the class. All realized class attributes are modifiable from within the class.
• **writeonly**	The attribute value may be modified but not read from outside the class. All realized class attributes are readable from within the class.
• **transient**	(see above)
• **persistent**	(see above)
• **factory**	The attribute is transient and associated with the class itself, rather than with an instance of the class.

Realized (persistent and transient) attributes may be initialized by following their declaration with an '=' and the initial value. For example, in the *PurchaseOrder* application, the attribute *PurchaseOrder.shipToCountry* is initialized to the literal value "USA", and *PurchaseOrder.orderDate* is initialized to the expression *new Date()*, which creates an EBT *Date* whose value is the current date.[59]

2.3.2 Computed Attributes

```
<computedAttributeModifier>* computed <bosType> <attributeName> =
                             <exprOrReturnBlock>
                      <attributeProperties>? ;
```

Computed attributes are not stored explicitly as part of the object's value.

The modifiers for computed attributes are:

• **hidden**	The attribute is not accessible outside the BOS in which it is defined, such as in another BOS or any AUS.
• **factory**	The attribute is associated with the class itself, rather than with an instance of the class.

Computed attributes must be defined using the '=' followed either by an expression to compute their value or a method body enclosed in *{}* that computes their value. The PurchaseOrder example contains two computed attributes, *orderTotal* and *lineTotal*, defined using the latter syntax.

2.3.3 Constant Attributes

```
<constantAttributeModifier>? constant <bosType> <attributeName> =
                             <constantExpression>
                      <attributeProperties>? ;
```

59 The discerning reader may note that the EBT *Date* is implemented using the Java class java.sql.Date, which has no default public constructor. However, DASL provides a default constructor for the EBT *Date*.

Constant attributes are not stored explicitly as part of the object's value. They have the same value for all instances of the object.

The optional modifier for constant attributes is:

- **hidden** The attribute is not accessible outside the BOS in which it is defined, such as in another BOS or any AUS.

Constant attributes must be defined using the '=' followed by a literal value of the appropriate type, e.g.,

```
constant Float pi = 3.1415926;
```

All constant attributes are factory attributes.

2.4 Relationship Declarations

DASL represents two-way relationships between class instances explicitly. If class *A* has a relationship with class *B*, then class *B* has a corresponding inverse relationship with class *A*. Relationships are declared in the parent class using the syntax:

```
<relationshipModifier>* <relationshipType> <className>
   <forwardAccessorName> ( <forwardCardinalityMin> , <forwardCardinalityMax> )
  [ inverse <inverseAccessorName>? ( <inverseCardinalityMin> ,
                                     <inverseCardinalityMax> ) ]
<relationshipConstraintDeclaration>*
<relationshipProperties>? ;

<relationshipConstraintDeclaration> ::= <constraintDeclaration>
```

The relationship is known by its *forward accessor name* in class *A*, and by its *inverse accessor name* in class *B*. It is sometimes desirable not to store inverse relationships directly, for efficiency reasons. To signal such cases, the inverse accessor name may be omitted from the declaration. However, whether an inverse relationship is implemented or not, it has a cardinality. Cardinality will be discussed shortly.

The class in which the relationship is declared is called the parent class, and the other class is called the child class. It is possible for the parent and child classes to be the same class. In the case of *owns* relationships, the parent class owns instances of the child class. In the case of *references* relationships, the parent and child classes are symmetrical, except that when the inverse accessor is omitted, navigation from the child to the parent cannot be done directly. A query may be used instead.

2.4.1 Relationship Modifiers

```
<relationshipModifier>  ::= [readonly | writeonly]
                        ::= hidden
```

The relationship modifiers are:

• **hidden**	The relationship is hidden outside of the BOS in which it is declared. This modifier affects the external interface of the relationship.
• **readonly**	The relationship value cannot be changed outside of the BOS in which it is declared.
• **writeonly**	The relationship value cannot be obtained outside of the BOS in which it is declared.

It is not allowed to specify both *readonly* and *writeonly*.

2.4.2 Relationship Types

```
<relationshipType>       ::= owns
                         ::= references
```

DASL distinguishes 2 types of relationships between object instances:

• **owns**	The *owns* relationship expresses the notion of a *complex* or *composite object.* It declares the relationship between the owner object (parent) and the owned object (child). The child is logically part of the parent, i.e., a particular instance of the child is contained in one and only one parent instance. Thus, if the parent object is deleted, the child object will also be deleted, and in general, the child cannot exist without the parent.
• **references**	The *references* relationship is a more loose relationship between two objects, in which neither owns the other. It is arbitrary which object is the parent or child, except to identify the forward and inverse direction of the relationship (see below).

2.4.3 Bidirectionality and Accessor Names of Relationships

```
<forwardAccessorName>
<inverseAccessorName>
```

The ***owns*** and ***references*** relationships are bi-directional. The relationship from parent to child is called the *forward* direction of the relationship, and the relationship from child to parent is called the *inverse* relationship.

A relationship has ***accessor names*** in each of the two directions in which it exists between two objects. The accessor names are used when navigating from the parent to the child, and from the child to the parent. Defining the inverse direction of a relationship (and its accessor name) is optional. If the inverse accessor name is not provided, the relationship still exists bidirectionally but is not directly navigable in the inverse direction.

2.4.4 Cardinality of Owns and References Relationships

```
<forwardCardinality> ::= ( <minForward>, <maxForward> )
<inverseCardinality> ::= ( <minInverse>, <maxInverse> )

<minForward> ::= 0 | 1
<maxForward> ::= 1 | n
<minInverse> ::= 0 | 1
<maxInverse> ::= 1 | n
```

Each direction of a relationship has a range of valid cardinalities:

- (0,1) means that there may optionally be one related object
- (1,1) means that there must be one related object
- (0,n) means that there may be zero or more related objects
- (1,n) means that there may be one or more related objects (i.e., at least one)

2.4.5 Persistence of Relationships

Relationships of persistent objects are always persistent, and must refer to other persistent objects. Relationships of transient objects are always transient, and must refer to other transient objects.

2.4.6 Restrictions on Relationships to Imported Business Objects

A business object in the current BOS may *reference* imported business objects, but it may not *own* an imported business object. When an imported business object is referenced, the relationship is unidirectional, without an inverse accessor (because adding an inverse accessor would require changing the imported object's BOS to include it).[60]

2.4.7 Relationship Properties

```
<relationshipProperties> ::= <properties>
```

The deployment engines for business objects look at the following properties of relationships as hints concerning the semantics of the relationship:

• forwardOrderedBy = "col"	The forward relationship is ordered by the named column, or by the primary key if the column string is empty. By default, the relationship has no specified order. Note that if the BOS has the global property "RELATIONSHIPS_ORDERED_BY_DEFAULT" then all relationships are ordered by default. If this property is not set, relationships are returned in an arbitrary order. For example, line items may show up according to some internal database hash code instead

60 These restrictions apply to the current implementation, which treats the deployment of imported BOSes as read-only. They might conceivably be lifted in the future, when the deployment engines are smart enough to regenerate and rebuild imported BOSs.

	of according to the line item number that is part of the primary key.
• forwardUnique = “true”	The forward relationship is to be represented as a set, and therefore must contain unique objects (i.e., no duplicate objects). This restriction is often the case due to the underlying persistent store. Setting this property explicitly guarantees that it will be true regardless of the underlying persistent store's semantics.
• forwardPrefetch = “true”	The implementation should try to prefetch the elements of the relationship when the object is first instantiated. If not set, the implementation should load the elements only when they are accessed. This property is an efficiency hint. If it is not specified, the implementation decides how to prefetch based on the application specification.
• inverseOrderedBy = “col”	Same as forwardOrderedBy, but for the inverse accessor.
• inverseUnique = “true”	Same as forwardUnique, but for the inverse accessor.
• inversePrefetch = “true”	Same as forwardPrefetch, but for the inverse accessor.

2.5 Primary Keys

```
PRIMARY KEY ( <attrOrRel> [, <attrOrRel> ]* ) [ <properties> ] ;

<attrOrRel> ::= <attributeName>
            ::= <relationshipAccessorName>
```

Each persistent object type has a primary key consisting of one or more attributes or relationships of the object. The *primary key* is sometimes also referred to as a persistent *object id.* If a persistent object type has no explicitly defined primary key, the application writer is asked to define one at deployment time.

Primary keys are important in defining persistent objects because they ensure that there is a way to uniquely identify each distinct instance of an object.

If a *<relationshipAccessorName>* is used as part of a primary key declaration, the accessor name must be declared in a class that is defined lexically prior to the primary key declaration that uses the accessor name. It is always possible to rearrange class declarations lexically to obey this restriction, since primary key declarations cannot be defined self-referentially or recursively.

Transient objects are not required to have a defined primary key, as their identity can be ascertained using the address of the transient object in memory. However, it is sometimes useful for them to have primary keys anyway.

2.6 Unique Keys

```
UNIQUE KEY ( <attrOrRel> [, <attrOrRel> ]* ) [ <properties> ] ;
```

A combination of attributes and relationship accessors may be declared as unique keys to prevent the possibility of more than one object having the identical values of that set of attributes and relationship accessors. Similar to the *PRIMARY KEY* declaration, this declaration can be a hint to the persistent store to create an index on the unique key, and it is also a constraint on the values of the key.[61]

2.7 Member Groups

```
GROUP <groupName> TYPE <groupType> ( <member> ,<member>* )
                   [ <properties> ] ;

<groupType> ::= INDEX | <userDefinedGroup> | <internalGroup>[62]

<userDefinedGroup> ::= <name>
```

The group type INDEX requests the an index for the business object, based on the values of a group of its members. More than one group of the same group type may be declared.

User-defined group types are an extensible language mechanism for assigning properties to a set of class instance members (attributes and/or relationships). They can be used to allow the application to describe metadata specific to a deployment engine. For example, a deployment engine may look for a groups of a specific type as an optimization, e.g., to request that certain relationships be cached rather than fetched on demand.

2.8 Constraints

The primary key and unique key constraints have already been introduced, as well as the cardinality constraint on relationships. Explicit constraints may be declared to further limit the values that attributes and relationships of objects may take.

Constraints may be defined at four distinct levels:

- **Attribute constraints**
- **Relationship constraints**
- **Object constraints**
- **Global constraints**

61 As of this writing, the DASL deployment engines rely on the database schema to enforce this constraint. Refer to the workarounds in the appendix.

62 The internal group types PKEY and UKEY are used when a PRIMARY KEY and UNIQUE KEY is declared, respectively. Notice that only one PKEY group may be declared, and the name of this group is always “Pkey”.

2.8.1 Attribute Constraints

Attribute constraints apply directly to a single attribute of the object. An example attribute constraint is that the price of a product must be non-negative:

```
Decimal price CHECK nonNegative ( price >= 0 );
```

Constraints are particularly useful for certain String-based attributes. For example, the following constraint on a "Social Security Number" attribute, ssn, ensures that a number with the correct format is entered:

```
String ssn CHECK ssnSyntax (ssn.matches("^\\d{3}-\\d{2}-\\d{4}$"));
```

Similarly, the following constraint on a 9 digit telephone number ensures that it has one of several reasonable formats:

```
String phone
 CHECK phoneSyntax (phone.matches("\\(?(\\d{3}\\)?[-. ]\\d{3}[-. ]\\d{4})"));
```

2.8.2 Relationship Constraints

Relationship constraints apply directly to a single relationship of the object.

Relationship constraints are stated in the context of the parent object. Care must be taken when referencing accessors to the child object to treat the accessor as a single value or as a collection, depending on its cardinality, and to handle the possibility that the child object is not set, and therefore its accessor returns null.

If the minimum cardinality of a relationship is 1, an implicit constraint exists to enforce that there is at least one such related object, in the direction that the minimum cardinality applies. This constraint is applied at commit time.

In the case of a collection-valued accessor, it may be assumed that the collection is initialized to a non-null value. However, in the case of a relationship whose cardinality is (0,1), the accessor may be null if no related object has been set.

An example relationship constraint is that a child may have at most two parents:

```
references Person parents(0,n) CHECK genetics (parents.size()<=2);
```

2.8.3 Object Constraints

Object constraints apply to an instance of an object. For example, a constraint may be defined to ensure that an employee's date of hire must be after his date of birth:

```
CHECK wasBorn ( hiredate > birthdate );
```

Object constraints are declared within the class definition at the same level as attributes, relationships, and methods.

2.8.4 Global Constraints

Global constraints [not yet implemented] are constraints that apply between instances of objects in a BOS. For example, the constraint that an employee's salary cannot be greater than his manager's salary is a global constraint. Such a constraint must be checked whenever a manager's salary is decreased, an employee's salary is increased, or an employee is reassigned to a new manager:

```
CHECK happyBoss ( Employee.salary <= Employee.manager.salary );
```

2.9 Constraint Declaration Syntax

The keyword *CHECK* introduces a constraint.

```
<constraintDeclaration> ::=  CHECK <constraintName>?
                                   [ WHEN_COMMITTED | WHEN_SET ]
                                   <constraintProperties>?
                                   <constraintImplementation>
```

2.9.1 Constraint Names

```
<constraintName> ::= <name>
```

Constraint names identify a particular constraint, and are primarily useful to document the constraint. The name must be unique within the class in which the constraint is defined.

2.9.2 Constraint Enforcement Timing: WHEN_SET and WHEN_COMMITTED

The application writer can choose whether constraints are to be applied immediately (when the attribute or relationship is set), or alternatively when the transaction commits. For object constraints, it is usually best to delay the constraint checking until the transaction commits.

```
Decimal price CHECK price_non_negative WHEN_SET (price >= 0);

Date death CHECK sane_death WHEN_COMMITTED (death > birth);
```

Unless otherwise specified, constraints are checked *WHEN_SET*.[63]

2.9.3 Constraint Properties

```
<constraintProperties> ::= <properties>
```

User-defined properties may be associated with constraints.

63 Some implementations may not be capable of checking constraints at the time the attribute is set. In this case, the WHEN_SET constraints will be checked before or at commit time.

2.9.4 Constraint Implementation

```
<constraintImplementation> ::=  { <typedMethodBody> }
                           ::=  ( <expression> )
                           ::=  DASL { <typedMethodBody> }
                           ::=  DASL ( <expression> )
                           ::=  OQL ( <OQLExpression> )
                           ::=  SQL ( <SQLExpression> )
```

Constraints may be expressed in any of the following ways:

- as a DASL boolean expression,
- as DASL typed method body that returns a boolean result, or
- as a boolean expression expressed in OQL or SQL.

The keyword *OQL* or *SQL* immediately before the opening parenthesis of the expression introduces a constraint specified as a DASL/OQL or SQL expression, respectively. The absence of a keyword, or the keyword *DASL*, introduces either a typed method body enclosed in curly braces {...} or a boolean expression enclosed in parentheses (...).

2.10 Creating Instances of Business Objects

An instance of a business object of class <name> may be created within a BOS method or AUS script using the syntax:

```
new persistent <businessObjectClassName>(...);

new [transient] <businessObjectClassName>(...);
```

The first form creates a persistent instance of a persistent class. It is not permitted to create a persistent instance of a transient class using the first form.

The second form, with or without the keyword *transient*, creates a transient instance of a persistent or facade class, or a transient instance of a transient class. Omitting the keyword allows the familiar Java syntax to be used for this case.

The signatures of the constructors (...) for a transient class include ( ) and any other constructors declared for that class. The signatures of the constructors for a persistent or facade class require that at least the primary key be specified. Before discussing constructors for persistent and facade objects, we must first consider primary key EBTs.

2.10.1 Primary Key EBTs: <name>PK

Each persistent or facade business object class has an associated primary key EBT, whose name is the persistent object's class name followed by “PK”. The primary key EBT is an immutable EBT representing the values of the primary key of the business object that is implicitly declared for each persistent business object class.

2.10.2 Creation of Unique Primary Keys for Persistent Objects: assignNextPK()

A persistent or facade business object class may optionally define a method with one of the following signatures to control how new primary keys are assigned.

```
  factory method <name>.assignNextPK() returns <name>PK;    // For unowned objects

  factory method <name>.assignNextPK( <ownerClass> ) returns <name>PK;
                                                            // For owned objects
```

For persistent objects only, in the following special cases, the deployment engines supply a useful default implementation of one of the above methods if the class does not define one. For a useful default to be supplied automatically, the persistent object must meet the following requirements:

- The primary key consists of a single attribute of a fixed numeric type, e.g., Integer, Long, or DECIMAL(xxx,0).
- The primary key consists of a single attribute of type String or CaselessString.
- The primary key consists of two members, one of which is a relationship from an owner object that has an assignNextPK() method, and the other of which is a fixed numeric attribute. This is the master/detail pattern, seen in the LineItem object in the PurchaseOrder application.

For an example of when assignNextPK() should be defined explicitly rather than using the default implementation, consider the case that each employee at a company has an employee ID which is the primary key of the Employee object, and the rules for assigning new employee numbers are complex and require obtaining the next number from an external service.

As an example of a case when assignNextPK() cannot be implemented by either the application writer or DASL, consider an application where it is up to the user of the application to enter a unique name, which then becomes the primary key. We saw an example of this case in the RememberMe application earlier.

2.10.3 Creation of a Primary Key Object With the Specified Key Value(s)

```
  factory method <name>.newPrimaryKey(<firstKeyType>, <secondKeyType>, ...)
          returns <name>PK;
or

  new <name>PK(<firstKeyType>, <secondKeyType>, ...)
```

If no default assignNextPK() method exists, or if the primary key of another object (e.g., an owner) is required, the automatically generated factory method <name>.newPrimaryKey(...) may be called to obtain a primary key object with the specified value. Alternatively, a new instance of the primary key EBTmay be created using the *new* keyword. Note that while primary key EBTs are stored as part of persistent or facade objects, they themselves are not persistent objects.

2.10.4 Creation of a Persistent Object Instance

We now have the background to discuss how to create a persistent instance of a business object. The following DASL statements may be used to create a new persistent instance of an object:

```
new persistent <name>( <name>PK )

new persistent <name>( <firstKeyType>, <secondKeyType>, ... )

new persistent <name>( <name>PK, <all scalar attributes and relationships> )

new persistent <name>( <constructorArgs> )
```

The form with all scalar attributes and relationships is used to initialize those attributes and relationships at the time of creation, while the first two forms just set these attributes and relationships to default values. The last form applies to an explicit constructor method that class <name> defines.

Note that the creation of a persistent object can throw the runtime exception

```
public class DuplicatePrimaryKeyException extends DaslAppRuntimeException
```

2.10.5 Creation of a Facade Object Instance

Facade object instances are created the same way as persistent business objects. The facade object implementation determines how the life cycle methods in the previous section are implemented.

2.11 Automatically Deployed Life Cycle Factory Methods

Each persistent class has some factory[64] life cycle methods automatically deployed for it. The automatically deployed factory methods for a business object class named ***<name>*** are described below.

Each facade class must define these same factory life cycle methods for its business object class to have the same compatible API as a persistent business object.

2.11.1 Deletion of an Existing Persistent Object from the Persistent Store

```
method <name>.remove( <name>PK pk );

method <name>.remove( <name> );
```

Note that removing a persistent or facade object can throw the runtime exception

```
public class NoSuchPrimaryKeyException extends DaslAppRuntimeException
```

2.11.2 Finding an Existing Object in the Persistent Store by its Primary Key

```
method <name>.findByPrimaryKey( <name>PK pk ) returns <name>;
```

64 This is a broad use of the term *factory*, in the spirit of Objective C, that is similar to the concept of Java static methods but is not implemented as such.

```
  method <name>.findByPrimaryKey(<firstKeyType>, <secondKeyType>, ...)
         returns <name>;
```

Note that these methods can throw the runtime exception

```
    public class NoSuchPrimaryKeyException extends DaslAppRuntimeException
```

The following methods find an object like *findByPrimaryKey*, but return *null* (instead of throwing an exception) if no such object is found:

```
  method <name>.findByPrimaryKeyIfAny( <name>PK pk ) returns <name>;

  method <name>.findByPrimaryKeyIfAny(<firstKeyType>, <secondKeyType>, ...)
         returns <name>;
```

2.11.3 Finding a Collection of Objects in the Persistent Store

The following method returns all of the primary key objects for objects in the class extent:

```
  method <name>.getAllPrimaryKeys() returns Collection '<' <name>PK '>' ;
```

The following method returns a collection containing all of the objects in the class extent:

```
  method <name>.getAllInstances() returns Collection '<' <name> '>' ;
```

The following method returns a collection containing all of the primary key objects of the objects that exist in the class extent, and that match the condition passed as a *String*, which is the *WHERE* clause of a query.

```
  method <name>.getMatchingPrimaryKeys( String cond )
         returns Collection '<' <name>PK '>' ;
```

The following method returns a collection containing all of the objects that exist in the class extent, and that match the condition passed as a *String*, which is the *WHERE* clause of a query. This is a reasonable way to allow a user to enter the search criteria at runtime:

```
  method <name>.getMatchingInstances( String cond )
         returns Collection '<' <name> '>' ;
```

2.12 Automatically Deployed Instance Methods

Each persistent class has some instance life cycle methods automatically deployed for it. The automatically deployed instance methods for a business object class named ***<name>*** are described below.

Each facade class must define these same factory life cycle methods for its business object class to have the same compatible API as a persistent business object.

2.12.1 Removal of an Object Instance from the Persistent Store

The following instance method removes the object from the persistent store:

```
  method remove() ;
```

Note that the removing a persistent object can throw the runtime exception

```
    public class NoSuchPrimaryKeyException extends DaslAppRuntimeException
```

2.12.2 Obtaining an Object Instance's Primary Key Object

The following instance method returns the object's associated primary key object:

```
  method getPrimaryKey() returns <name>PK ;
```

2.13 Automatically Deployed Relationship Pseudo-Attributes And Methods

Each persistent class has some instance pseudo-attributes automatically deployed for it based on its named relationship accessors that can be navigated from the class. The automatically deployed instance attributes for a business object that has a named relationship accessor ***<accessor>*** to another object of class ***<refclass>*** are described below.

For attributes and relationship accessors of maximum cardinality 1, the following attribute is deployed.

```
    <refclass> <accessor>;
```

This attribute may be assigned values and used in expressions like a normal attribute.

For relationship accessors of maximum cardinality n, the collection-valued member is initialized at creation to an empty collection.

```
    Collection '<' <refclass> '>' <accessor>;
```

The accessor name may be used to obtain the collection, and collection operations on it may be used to modify the collection. In most implementations, the collection is a set. Note that it is not guaranteed that a new value can be assigned to accessor (i.e., replace the entire collection with a new collection).

Because different implementations use different kinds of collections, and therefore support somewhat different methods for accessing elements of the collection, the following instance methods are provided to search and modify the collection (***<uaccessor>*** is the ***<accessor>*** with its first character capitalized):

```
    method lookupIn<uaccessor>( <refclass>PK ) returns <refclass> ;
    method lookupIn<uaccessor>PK( <refclass>PK ) returns <refclass>PK ;

    method lookupIn<uaccessor>( <singlePKattribute> ) returns <refclass> ;
    method lookupIn<uaccessor>PK( <singlePKattribute> ) returns <refclass>PK ;

    method insertInto<uaccessor>( <refclass> ) ;
    method insertInto<uaccessor>PK( <refclass>PK ) ;

    method insertInto<uaccessor>( <refclass> ) ;
    method insertInto<uaccessor>PK( <refclass>PK ) ;
    method removeFrom<uaccessor>( Collection '<' <refclass> '>' ) ;
    method removeFrom<uaccessor>PK( Collection '<' <refclass>PK '>' ) ;

    method removeAll<uaccessor>( ) ;
    method removeAll<uaccessor>PK( ) ;

    method set<uaccessor>( int where, <refclass> ) ;
    method set<uaccessor>PK( int where, <refclass>PK ) ;
```

The following method is defined only when the implementation uses an ordered collection of some kind to represent the relationship. It moves a slice of elements in the collection, without duplication. The ordering changes it makes are not remembered persistently except in the uncommon case that the underlying persistence store is designed to store relationships in an application-specified order.

```
    method moveIn<uaccessor>( int oldIndex, int newIndex, int numberOfElements ) ;
```

A slice of the elements of the ordered collection, from *oldIndex* to *(oldIndex + numberOfElements – 1)* is moved without duplication so that the first element in the slice is placed at *newIndex*. The elements at *newIndex* and beyond are placed after the last element of the moved slice.

The elements of the collection are numbered starting from 0, i.e., 0 is the first element. To move the slice to the beginning of the collection, ahead of any other elements, specify a *newIndex* of 0. To move the slice to the end of the collection, following any other elements, specify a *newIndex* of *collection.size()*. An exception is thrown if the slice does not fall within the existing collection, e.g., if *oldIndex + numberOfElements > collection.size()*.

2.14 Syntax For Accessing Factory Business Methods in the BOS And AUS

The factory for a business object is specified by giving the name of a business object. Methods in the factory may be invoked as pseudo-methods of the business object name.

<name>.<methodName>(...);

In the BOS, if ***<name>*** is an object of the current BOS, its name will always be unambiguous. When invoking an object in another BOS (and thus in another package), the fully qualified package name of the object must be specified.

2.15 Defining Facade Objects

Facade objects are objects that are used just like ordinary persistent objects, and that appear to have the same semantics as ordinary persistent objects, but whose persistence is defined by the facade object class itself. For facade objects to mimic persistent objects, they must present the appearance of persistent objects, including

- persistent attributes and relationships,
- the same transactional semantics as DASL persistent objects,
- the same life cycle methods as DASL persistent objects, and
- the same query capability as DASL persistent objects.

The simplest way to define a facade object is to describe how to map a single instance of the class to and from an external persistent store. In a nutshell, the facade object provides methods that create, retrieve, and remove an object instance during the transaction, and methods that commit and abort changes made to the object during the current transaction to the persistent store.

In some cases, the external persistent store may not be amenable to allowing single instances of a DASL class to be manipulated independently. For example, the external persistent store may be a file containing multiple instances of a DASL class, or multiple instances of heterogeneous fine-grained DASL classes. In such cases, it may be desirable for the facade class to explicitly manage an object cache composed of several classes of fine-grained objects that are mapped to the external persistent store.

DASL facade objects support both the instance-level approach and the class-level approach for mapping between DASL objects and an external persistent store.

2.15.1 Persistent Attributes

Attributes that are going to be stored persistently by the implementation should be declared persistent. It is up to the class implementation to define life cycle, transactional, and query methods so that object instances are retrieved from the persistent store and modifications made to them are stored persistently.

2.15.2 Transient Attributes

Transient attributes are handled within facade objects in the same way as in persistent and transient objects. Accessor and mutator methods are automatically generated for transient objects.

2.15.3 Relationships

Relationships to other facade objects may be declared within a facade object. It is up to the class implementation to define life cycle, transactional, and query methods so that these relationships are retrieved from the appropriate persistent store, and so modifications made to them are stored persistently. Ways in which this may be accomplished will be described below.

2.15.4 Instance State Management: Simple Management of State at the Instance Level

Using the simplest approach, the facade object class provides the implementation for creating, retrieving, and removing (deleting) instances of the class during the current DASL transaction, and methods for modifying the underlying persistent store for each object when committing or aborting the current DASL transaction. DASL provides implementations for the accessors, mutators, and the detailed state and cache management of the objects themselves. This approach is the easiest to define, but provides the least control over optimization of the underlying persistence mechanism.

Using instance state management, a facade object class has the following basic outline:

```
facade class Employee {
    PROPERTIES { STATE_MANAGEMENT = "instance" };
    // Instance state management
    // define transient attributes here.
    // define persistent attributes here.
    // define relationships here.
    // define primary key here.

    /*optional*/ factory method assignNextPK( ) returns EmployeePK
    { ... }        // assigns the next available primary key

    factory method create( EmployeePK pk ) returns Employee
    { ... }        // provisionally creates the object

    factory method retrieve( EmployeePK pk ) returns Employee
    { ... }

    /*optional*/ method removeRelationships( )
    { ... }        // removes relationships between this object & others

    factory method getAllPrimaryKeys() returns Collection <EmployeePK>
    { ... }

    factory method getMatchingPrimaryKeys( String cond )
         returns Collection <EmployeePK>
    { ... }

    method doCommit()
    { ... }        // called when the transaction is being committed

    method doAbort()
    { ... }        // called when the transaction is being aborted
}
```

Instance state management is suitable for accessing an external API that allows individual objects to be retrieved and stored as a whole. For example, it can be used in conjunction with

- File-based storage of individual objects.
- Web services that provide access at the individual object level.
- TP monitors that provide access at the individual object level.
- LDAP access to individual objects.

The facade class must provide implementations for these methods:

- `factory method` **`create`**`( <facadeClass>PK pk ) returns <facadeClass>`
- `factory method` **`retrieve`**`( <facadeClass>PK pk ) returns <facadeClass>`

- `factory method` **`getAllPrimaryKeys`**`()`
  `returns Collection '<' <facadeClass>PK '>'`
- `factory method` **`getMatchingPrimaryKeys`**`( String cond )`
  `returns Collection '<' <facadeClass>PK '>'`

- `method` **`doCommit`**`()`
- `method` **`doAbort`**`()`

The facade class may optionally provide implementations for these methods:

- `factory method` **`assignNextPK`**`( ) returns <facadeClass>PK`
- `method` **`removeRelationships`**`( )`

DASL provides implementations for these methods of the facade class:

- `method` **`getPrimaryKey`**`() returns <facadeClass>PK`
- `factory method` **`remove`**`( <facadeClass>PK pk ) returns boolean`
- `method` **`remove`**`( ) returns boolean`

- `factory method` **`findByPrimaryKeyIfAny`**`( <facadeClass>PK pk )`
  `returns <facadeClass>`
- `factory method` **`findByPrimaryKey`**`( <facadeClass>PK pk )`
  `returns <facadeClass>`

- `factory method` **`getAllInstances`**`()`
  `returns Collection '<' <facadeClass> '>'`
- `factory method` **`getMatchingInstances`**`( String cond )`
  `returns Collection '<' <facadeClass> '>'`

- *all accessor and mutator methods for attributes and relationships*
- *create*() *and remove*() *methods with the primary key fields "flattened"*

The factory *create*() method is called by the persistent constructor for the facade class. It should do whatever is necessary to provisionally create a new object instance during the current DASL transaction. In some cases, the *create*() method will make a call to the underlying persistence mechanism, and in others, the creation may be deferred until the DASL transaction commits.

At a minimum, you must define a *create*() method that takes the primary key object as a parameter. If you define other *create* methods that take the primary key object and other attributes or relationships, the implementation will support persistent constructors with the same parameters as the defined *create* methods.

The factory *retrieve*() method must retrieve an object instance from the persistent store. DASL will handle caching of objects, so this method should not attempt to cache and reuse a previously retrieved copy. Rather, it should always create a new transient instance of the class and populate the attributes and relationship attributes.

If the current object has relationships to other objects, and the relationship information is stored in those other objects, the *removeRelationships*() method must modify those related objects appropriately, i.e., by removing the relationship to the current object, which is being removed.

The *getAllPrimaryKeys*() method must return the primary keys of all objects of this class in the persistent store. DASL uses this method to implement the *getAllInstances*() method.

The *getMatchingPrimaryKeys*() method takes an OQL "WHERE" clause on the attributes and relationships of the object class and must return a collection of primary keys of objects that match the condition. DASL uses this method to implement the *getMatchingInstances*() method.

The *doCommit*() method is called when the DASL transaction is committing, after the constraints have all been checked. It must commit all changes made to the object instance to the persistent store. Changes include creation and deletion of the object instance, modifications made to the object's attributes, and modifications made to the relationships between this object and other objects. The *doCommit*() method should make use of the detailed state information that DASL tracks for the current transaction, which is described below.

Relationships between objects must be handled appropriately in the *doCommit*() method, depending on how the relationship is represented by the underlying persistence mechanism. For example, certain relationships may be represented only in the parent or child object. It is up to the *doCommit*() method to know such details and make the appropriate changes to the underlying persistent store.

The *doAbort*() method is called when the DASL transaction is being aborted. It may safely assume that the current instance of the class will be discarded. The *doAbort*() method may make use of the detailed state information that DASL tracks for the current transaction, which is described below. Specifically, if the *create*() method for this class works by actually making a change to the underlying persisent store, the *doAbort*() method must check to see if the object was created in this transaction, and if so, it must undo any changes to the persistent store made by the *create*() call.

DASL will automatically generate accessor and mutator methods for all attributes. The accessors and mutators for attributes return and set the attribute fields, which are assumed to be initialized after *retrieve*() has been called. The DASL-supplied mutator methods automatically set the detailed state information that tracks whether attributes and relationships have changed.

DASL will automatically generate relationship accessors and mutators based on instances of related objects and based on the primary key object of related objects. The handling of relationships using all approaches is described more fully below.

The implementation of *findByPrimaryKeyIfAny*() that DASL creates keeps a cache of objects already retrieved to prevent accidental object aliases (duplicate copies) from being created within a transaction. It uses this cache in its implementation of the *getAllInstances*() method and the *getMatchingInstances* () method.

2.15.5 Class State Management: Complete State and Cache Management at the Class Level

Using this approach, the facade object provides all aspects of object management at the class level, including fine-grained state and cache management of the objects, and control over the implementation of accessors and mutators. This approach requires that more methods be implemented on the class, and places more burden on the provider of the class to get all aspects of the logic correct. However, it also allows the provider of the class to map coarse-grained external objects into mutiple fine-grained DASL objects of a single or multiple classes. For example, an entire file of objects can be mapped into multiple DASL objects, one per line of the file, and the entire file can be updated as a whole when the transaction commits.

Using class state management, a facade object class has the following basic outline:

```
facade class Employee {
    PROPERTIES { STATE_MANAGEMENT = “class” };

    // Class state management
    // define transient attributes here.
    // define persistent attributes here.
    // define relationships here.
    // define primary key here.

    /*optional*/ factory method assignNextPK( ) returns EmployeePK
    { ... }        // assigns the next available primary key

    factory method create( EmployeePK pk ) returns Employee
    { ... }        // provisionally creates the object

    factory method findByPrimaryKeyIfAny( EmployeePK pk )
         returns Employee
    { ... }        // retrieves object and adds it to the cache

    method remove( )
    { ... }        // marks object for removal (deletion)

    factory method getAllPrimaryKeys() returns Collection <EmployeePK>
    { ... }

    factory method getAllInstances() returns Collection <Employee>
    { ... }

    factory method getMatchingPrimaryKeys( String cond )
         returns Collection <EmployeePK>
    { ... }

    factory method getMatchingInstances( String cond )
         returns Collection <Employee>
    { ... }

    /*optional*/ factory method initializeTransaction()
    { ... }        // called when a new transaction has begun

    factory method doCommit() returns boolean
    { ... }        // called when this instance is being committed

    /*optional*/ factory method afterCommit()
    { ... }        // called after this instance is committed

    factory method doAbort()
    { ... }        // called when this instance is being rolled back

    /*optional*/ factory method afterAbort()
    { ... }        // called after this instance is rolled back
}
```

Class state management provides complete control over object state and object cache management, on a per-facade-class basis, at the expense of requiring the implementation to supply the implementations of many more methods. Specifically, the facade class must provide implementations for these methods:

- `factory method` **`create`**`( <facadeClass>PK pk ) returns <facadeClass>`
- `factory method` **`findByPrimaryKeyIfAny`**`( <facadeClass>PK pk )`
  `returns <facadeClass>`
- `method` **`remove`**`()`

- `factory method` **`getAllPrimaryKeys`**`()`
  `returns Collection '<' <facadeClass>PK '>'`
- `factory method` **`getAllInstances`**`()`
  `returns Collection '<' <facadeClass> '>'`
- `factory method` **`getMatchingPrimaryKeys`**`( String cond )`
  `returns Collection '<' <facadeClass>PK '>'`
- `factory method` **`getMatchingInstances`**`( String cond )`
  `returns Collection '<' <facadeClass> '>'`

- `factory method` **`doCommit`**`() returns boolean`
- `factory method` **`doAbort`**`()`

- *override accessor and mutator methods for attributes and relationships*

The facade class may optionally provide implementations for these methods:

- `factory method` **`assignNextPK`**`( ) returns <facadeClass>PK`
- `factory method` **`initializeTransaction`**`()`
- `factory method` **`afterCommit`**`()`
- `factory method` **`afterAbort`**`()`

DASL provides implementations for these methods of the facade class:

- `method getPrimaryKey() returns <facadeClass>PK`
- `factory method` **`remove`**`( <facadeClass>PK pk ) returns boolean`

- `factory method findByPrimaryKey( <facadeClass>PK pk )`
  `returns <facadeClass>`

- *`default accessor and mutator methods for attributes and relationships`*

DASL generates the same default accessor and mutator methods as it does for instance state management. DASL provides the same object state variables as it does for instance state management. The class may override the default accessor or mutator methods generated by DASL if it wishes.

The factory method *doCommit()* must be defined to save the changes to persistent attributes and relationships. The other transaction methods are optional and are provided so that the implementation may initialize state prior to transactions and clean up state after transactions.

The transaction management methods are as follows:

- **factory method initializeTransaction()**

  This method is called for a facade object class when a new transaction has begun. It is typically used to initialize transaction state, such as the object cache.

- **factory method doCommit() returns boolean**

This method is called on a facade object class when the transaction is being committed. The constraints defined on all objects have already been checked when this method is called. This method typically makes API calls to update all the cached objects of the class in the persistent store.

- **factory method afterCommit()**

This method is called on a facade object class after it has been committed. It is typically used to clean up resources following a transaction, such as the object cache.

- **factory method doAbort()**

This method is called on a facade object class when it is being rolled back. This method may possibly make API calls to invoke compensating transactions in the persistent store.

- **factory method afterAbort()**

This method gets called after all objects have been rolled back. It is typically used to clean up resources following a transaction, such as the object cache.

2.15.6 Automatic Object Instance State Tracking

DASL maintains information about the state of each instance of a facade object, including state information about the object itself, and information about each of its member attributes and relationships. For relationship accessors whose maximum cardinality is greater than one, DASL maintains additional information about the related objects that have been added to the relationship and removed from the relationship.

For each object instance, DASL provides a transient attribute called _objectState that describes the state of the object as a whole during the current transaction:

```
class DASL_ObjectState {
    boolean created;                  // created this transaction
    boolean retrieved;                // retrieved this transaction
    boolean committed;                // committed this transaction
    boolean removed;                  // removed this transaction
    boolean attribute_modified;       // attribute has been modified
    boolean relationship_modified;    // relationship has been modified
}
```

- If the object instance was created during the current transaction, i.e., if one of the *create* () methods has been called to create the object, then *created* is true; otherwise, *created* is false.
- If the object instance was retrieved during the current transaction, i.e., if *retrieve*() was called, then *retrieved* is true; otherwise, *retrieved* is false.
- If the object instance has been updated in the persistent store during the current transaction, i.e., if *update*() has been called on this instance, then *committed* is true;

otherwise, *committed* is false.

- If the object instance has been removed during the current transaction, i.e., if *remove*() was called, then *removed* is true; otherwise, *removed* is false.

2.15.7 Automatic Persistent Member State Tracking

For each persistent attribute or relationship declared as member of a facade object, DASL provides a transient boolean flag to track whether the attribute or relationship has been modified. Consider the persistent attribute:

```
persistent String name;
```

DASL generates a corresponding transient attribute

```
transient boolean name_modified = false;
```

and a default mutator method that tracks the state of the persistent attribute as follows:

```
boolean setName( String name ) {
    this.name = name;
    this.name_modified = true;
}
```

Relationship state is handled similarly. Examples can be found in the following section.

2.15.8 Transient Attributes for Relationships and Delayed Retrieval of Related Objects

An implementation may wish to delay retrieval of the related objects, keeping only the primary key of the related objects on hand. DASL facilitates this approach by automatically defining three attributes for each related object to represent the relationship:

1. a transient attribute in which the *primary key* of the related object may be stored, and

2. a transient attribute in which an *object reference* to the related object may be stored.

3. a boolean flag (as for attributes) to represent that the relationship has been modified.

For example, consider a relationship *abc* to class *Other* whose maximum cardinality is one. There will be three transient attributes to represent this relationship:

```
private OtherPK abcPK;
private Other   abc = null;
private boolean abc_modified = false;
```

Accessor and mutator methods are generated automatically for the relationship, to enforce the following conventions:

- If the transient PK attribute abcPK is null, the relationship is null.

- If the transient PK attribute abcPK is not null and the corresponding transient object reference abc is null, the object reference will be found by calling findByPrimaryKey(), after which it will be cached in the object reference attribute abc.

- If the transient object reference abc is not null, it always corresponds to the transient PK attribute.
- If the transient PK attribute abcPK is set explicitly, the transient object reference abc is set to null.
- If the transient object reference abc is set explicitly, the transient PK attribute abcPK is set using the getPrimaryKey() method of the transient object reference.
- If any changes are made to the relationship, the boolean flag <accessor>_modified is set.

The above rules correspond to the following simplified implementations of the accessor and mutatator methods. As we shall see later, additional logic is used in the mutator methods to keep the object cache and object state up-to-date.

```
method getAbc( )   returns Other {
    if (abc == null && abcPK != null)
         abc = findByPrimaryKey(abcPK);
    return abc;
}

method getAbcPK( ) returns OtherPK {
    return abcPK;
}

method setAbc( Other o ) {
    if (o == null) {
         if (abcPK == null)
             return;
         abcPK = null;
         abc = null;
         abc_modified = true;
    }
    else {
         OtherPK x = o.getPrimaryKey();
         abc = o;
         abcPK = x;
         if (! o.equals(abcPK))
                  abc_modified = true;
    }
}

method setAbcPK(OtherPK opk) {
    if (opk == null) {
         if (abcPK == null)
             return;
         abcPK = opk;
         abc_modified = true;
    }
    else if (! opk.equals(abcPK))
         abc = null;
         abc_modified = true;
    }
}
```

Now consider a relationship *defs* to class *Other* whose maximum cardinality is greater than one. There will be several transient attributes to represent this relationship:

```
List<OtherPK> defsPK        = new TrackedArrayList<OtherPK>();
```

```
List<Other>   defs          = new TrackedArrayList<OtherPK>();
boolean       defs_modified = false;
```

There will be two instances of each of the accessor methods for this relationship:

```
method getDefs() returns Collection<Other>;
method getDefsPK() returns Collection<OtherPK>;

method lookupInDefs( OtherPK opk ) returns Other;
method lookupInDefsPK( OtherPK opk ) returns OtherPK;

...
```

There will be two instances of each mutator method for this relationship:

```
method setDefs( Collection<Other>);
method setDefsPK( Collection<OtherPK> );

method insertIntoDefs( Other o );
method insertIntoDefsPK( OtherPK opk );

...
```

These accessor and mutator methods use the same conventions as the ones for maximum cardinality one relationships. The logic to manage collections is a bit more complex, and is not shown here.

DASL uses a special collection class implementation, TrackedArrayList, to represent relationships to more than one object. TrackedArrayList keeps track of the elements added and removed from the collection, which can then be used to determine how to update the relationship efficiently.

Multiple accessor and mutator methods are supported, and they are not all shown here. For a complete list of the relationship accessor and mutator methods for relationships whose maximum cardinality is greater than one, refer to the section *Automatically Deployed Relationship Pseudo-Attributes And Methods.*

2.15.9 Object Cache Management

To prevent the creation of multiple copies of the same object instance, the facade object should use some kind of cache management scheme. Typically, this scheme is defined by the findByPrimaryKeyIfAny() method using class state management. Using instance state management, DASL generates the findByPrimaryKeyIfAny() method and thus provides the object cache management. When initializing relationship accessors, it is important to check the local object cache before retrieving a possible duplicate object instance. Using instance state management, the generated findByPrimaryKey() method handles cache management automatically.

2.15.10 Example 1: Instance Storage of Flat Objects Using Java Serialization

The facade class uses Java serialization to store one class instance per file, where the file name is derived from the primary key. Notice that there is no logic for caching and finding objects already instantiated, or for preventing the creation of duplicate instances – these tasks are handled by the facade code generator. The class merely implements a way to store transient instances externally at commit time, to re-instantiate them on request, and to find the primary keys of all instances stored externally.

```
facade class PhoneBookEntry {
    PROPERTIES { STATE_MANAGEMENT = "instance" };

    String name;
    String workPhone;
    String homePhone;
    String cellPhone;

    PRIMARY KEY (name);

    constant String NAME_PREFIX = "PhoneBookEntry.";

    factory method create( PhoneBookEntryPK pk ) returns PhoneBookEntry
    {
        PhoneBookEntry pbe = new transient PhoneBookEntry();
         pbe.name = pk.name;
        return pbe;
    }

    factory method retrieve( PhoneBookEntryPK pk ) returns PhoneBookEntry
    {
         // retrieve instance from a file
         PhoneBookEntry pbe;
         FileInputStream fis;
         fis = new FileInputStream(NAME_PREFIX + pk.name);
         ObjectInputStream ois = new ObjectInputStream(fis);
         pbe = (PhoneBookEntry) ois.readObject();
         ois.close();
         return pbe;

         on error (FileNotFoundException fnfe)
             throw new DaslAppRuntimeException(fnfe);
         on error (IOException ioe)
             throw new DaslAppRuntimeException(ioe);
    }

    factory method getAllPrimaryKeys( ) returns Collection <PhoneBookEntryPK>
    {
         // Find files whose names are PhoneBookEntry.<name>
         // and return the list of PKs based on it!
         Collection<PhoneBookInstancePK> extent =
                  new ArrayList<PhoneBookInstancePK>();
         File dir = new File(new File("").getAbsolutePath());
         if (! dir.exists())
             throw new DaslAppRuntimeException("Current directory not found!");
         for (File f: dir.listFiles()) {
             String filename = f.getFilename();
             if (! filename.startsWith(NAME_PREFIX))
                  continue;
             int len = NAME_PREFIX.length();
             if (len > filename.length())
                  continue; // NAME_PREFIX with empty extension
             // make sure file exists and is not a directory and is readable
             if (! f.exists() || ! f.isFile() || ! f.canRead())
                  continue;
             String name = filename.substring(len);
             // Have a valid name, so add a pkey to the extent.
             extent.add( new PhoneBookInstancePK(name) );
         }
         return extent;
    }

    factory method getMatchingPrimaryKeys( String cond )
```

```
            returns Collection <PhoneBookEntryPK>
    {
         throw new DaslAppRuntimeException("PhoneBookEntry: " +
                                  "getMatchingPrimaryKeys not implemented");
    }

    method doCommit()
    {
         // store instance in a file; use a temporary file for safety
         File desired = new File(NAME_PREFIX + name);
         File temp = File.createTempFile(NAME_PREFIX,"_" +
                           name,desired.getAbsoluteFile().getParentFile());

         // Write to temp file
         FileOutputStream fos;
         fos = new FileOutputStream(temp);
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(this);
         oos.close();

         // Rename temp to desired name
         boolean success;
         if (desired.exists()) {
             success = desired.delete();
             if (!success)
                  throw new DaslAppRuntimeException("Failed to delete prior " +
                                  "version of " + desired + ".");
         }
         success = temp.renameTo(desired);
         if (!success)
             throw new DaslAppRuntimeException("Failed to rename " + temp +
                                                 " to " + desired + ".");
         temp.delete(); // ignore failure to delete temp file
         on error (FileNotFoundException fnfe)
             throw new DaslAppRuntimeException(fnfe);
         on error (IOException ioe)
             throw new DaslAppRuntimeException(ioe);
    }

    method doAbort()
    {
         // no need to do anything, because we don't store to file when we
         // do a create or anywhere else except doCommit().
    }

}
```

2.15.11 Example 2: Class Extent Storage of Flat Objects Using Java Serialization

The facade class uses Java serialization to store all class instances in a single file. This simple implementation maintains the entire extent (collection) of all objects of the class in memory, reading it in initially from a single file, and storing it to the file upon commit. Unlike the instance storage example, the doCommit() method is a factory method, since all changes to objects are written out in one operation.

```
facade class PhoneBookEntry {
    PROPERTIES { STATE_MANAGEMENT = "class" };

    String name;
    String workPhone;
    String homePhone;
    String cellPhone;

    PRIMARY KEY (name);

    constant String EXTENT_NAME = "PhoneBookEntry.extent";

    transient java.util.HashMap /*<PhoneBookEntryPK,PhoneBookEntry>*/ extent;

    factory method create( PhoneBookEntryPK pk ) returns PhoneBookEntry
    {
        // Create a transient object and add it to the extent
        PhoneBookEntry pbe = new transient PhoneBookEntry();
        pbe.name = pk.name;
        extent.put(pk,pbe);
        return pbe;
    }

    factory method findByPrimaryKeyIfAny( PhoneBookEntryPK pk )
        returns PhoneBookEntry
    {
        // In this implementation, we keep all known objects in the cache.
        initializeExtent();
        PhoneBookEntry pbe = extent.get(pk);
        return pbe;
    }

    method remove( )
    {
        // DASL marks this object as removed -- we don't need to do more here.
    }

    factory method getAllPrimaryKeys( ) returns Collection <PhoneBookEntryPK>
    {
        initializeExtent();
        return extent.keySet();
    }

    factory method getAllInstances( ) returns Collection <PhoneBookEntry>
    {
        initializeExtent();
        return extent.values();
    }

    factory method getMatchingPrimaryKeys( String cond )
            returns Collection <PhoneBookEntryPK>
    {
        throw new DaslAppRuntimeException("PhoneBookEntry: " +
                              "getMatchingPrimaryKeys not implemented");
    }

    factory method getMatchingInstances( String cond )
            returns Collection <PhoneBookEntry>
    {
        throw new DaslAppRuntimeException("PhoneBookEntry: " +
                              "getMatchingInstances not implemented");
    }


    hidden factory method initializeExtent( )
    {
        // read in all entries the first time this method is called
        if (extent != null)
```

```
            return;

        // Sanity check on directory
        File dir = new File(new File("").getAbsolutePath());
        if (! dir.exists())
            throw new DaslAppRuntimeException("Current directory not found!");

        // Does extent file exist?
        File extentFile = new File(dir,EXTENT_NAME);
        if (! extentFile.exists()) {
            extent = new java.util.HashMap<PhoneBookEntryPK,PhoneBookEntry>();
            return;
        }

        FileInputStream fis = new FileInputStream(extentFile.toString());
        ObjectInputStream ois = new ObjectInputStream(fis);
        extent = (HashMap<PhoneBookEntryPK,PhoneBookEntry>) ois.readObject();
        ois.close();

        on error (FileNotFoundException fnfe)
            throw new DaslAppRuntimeException(fnfe);
        on error (IOException ioe)
            throw new DaslAppRuntimeException(ioe);
        on error (ClassNotFoundException cnfe)
            throw new DaslAppRuntimeException(cnfe);
    }


    factory method doCommit() returns boolean
    {
        // get rid of all removed instances now.
        for (PhoneBookEntry pbe: extent.values()) {
            if (pbe._objectState.removed)
                extent.remove(pbe.getPrimaryKey());
        }

        // store extent in a file; use a temporary file for safety
          File desired = new File(EXTENT_NAME);
          File temp = File.createTempFile(EXTENT_NAME,"",
                            desired.getAbsoluteFile().getParentFile());

        FileOutputStream fos = new FileOutputStream(temp);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(extent);
        oos.close();
        extent = null;

         FileOutputStream fos;
         fos = new FileOutputStream(temp);
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(this);
         oos.close();

         // Rename temp to desired name
         boolean success;
         if (desired.exists()) {
             success = desired.delete();
             if (!success)
                  throw new DaslAppRuntimeException("Failed to delete prior " +
                                   "version of " + desired + ".");
         }
         success = temp.renameTo(desired);
         if (!success)
```

```
                throw new DaslAppRuntimeException("Failed to rename " + temp +
                                                  " to " + desired + ".");
            temp.delete(); // ignore failure to delete temp file
            return true;

            on error (FileNotFoundException fnfe)
                throw new DaslAppRuntimeException(fnfe);
            on error (IOException ioe)
                throw new DaslAppRuntimeException(ioe);
        }

        method doAbort()
        {
            // no need to do anything, because we don't store to file when we
            // do a create or anywhere else except doCommit().
        }
    }
```

2.15.12 Example 3: Instance Storage of Objects with Relationships

The facade class uses Java serialization to store one class instance per file, where the file name is derived from the primary key. Notice that there is no logic for navigating relationships – these tasks are handled by the facade code generator.

Coming Soon: PurchaseOrders and LineItems stored separately, one per file.

2.15.13 Example 4: Class Storage of Objects with Relationships

The facade class uses Java serialization to store all class instances of each class in a single file.

Coming Soon: All PurchaseOrders in a file,
and all of its LineItems stored in another file.

2.15.14 Example 5: Coarse-Grained Class Storage of Objects with Relationships

The facade class uses Java serialization to store one coarse-grained object per file, consisting of a purchase order and all of its line items.

Coming Soon: One PurchaseOrder and all of its LineItems stored in a file.

3 DASL/AUS Language Specification

The application usage specification (AUS) describes a set of logical screens (*states)* within the application, and the actions (*transitions*) that cause the application to move from one state to another. It also describes the transactions (atomic units of work) within the application, to ensure that the underlying persistent data are updated in a consistent way.

An AUS is conveniently viewed as an annotated UML state diagram that represents the flow of interaction between the application and the user. Transitions from one state to the other, represented as arcs in the state diagram, denote that the application invokes underlying business logic.

A script associated with each transition contains method calls to the back end of the application. The calls are typically to business objects in the Business Object Specification (BOS), but they can also reference external Java libraries and frameworks, either directly or indirectly via business object methods defined as part of the BOS.

For concreteness, one can think of the AUS as representing a series of web pages that constitute the business application; however, the AUS does not represent any particular presentation or layout information that one would normally associate with web page design. Rather, it represents the logical flow of information between the user and the back end of the application. The scripts associated with the transitions are an integral part of the AUS, but they are not directly visible in the state diagram.

3.1 Application Sessions

DASL/AUS defines two kinds of sessions: interactive sessions and server sessions. The type of session that an AUS specification will produce is determined by the structure of the AUS itself. As of this writing, only interactive sessions have been implemented. For the remainder of this document, the term “session” refers to interactive session.

3.1.1 Interactive Application Sessions

An interactive application session begins when the application is invoked, typically by pointing a browser to a URL or clicking on a link to that URL. An interactive session ends when the application performs an *exit* statement in a script, or when a fatal error occurs. The *exit* statement may optionally provide a URL to display, possibly the URL of another application.

3.1.2 Stateless Server Sessions [future]

A stateless server session has only a single active state, so the point at which it begins or ends is irrelevant. It may be started automatically when the machine on which it is deployed is booted, or it may be started as the result of an event that it handles. Stateless server sessions are useful for handling discrete events that have no session context.

3.1.3 Stateful Sever Sessions [future]

A stateful server session starts when an interactive application session sends it an event. The stateful server session responds by sending a reply that includes a unique identifier that identifies the session context. Stateful server sessions are useful for handling events that occur within some context, which can be represented by distinct AUS states and session variables. Such interactions between a client and a stateful server are sometimes called “dialogs”.

3.2 Summary of the Key Elements of an AUS Specification: <AUS>

The AUS sub-language is used to choreograph a business task or process by defining a set of states and transitions between them, and by grouping the states into application-level transactions. Most of this information is declarative in nature, rather than procedural.

At the topmost level, an **AUS** consists of the following elements:

- Package declaration (see Common Elements)
- Imports of BOS Objects
- Properties of the AUS
- Session Variables (scoped to entire session)
- Shared Variables (scoped to all user sessions)
- State Declarations (initial state and other states)
- Transaction Declarations

3.3 Imports into the AUS

3.3.1 Import Of Business Objects

An AUS usually imports the business objects from at least one BOS:

```
import [ <package> . ] <bosName> . <className> . '*' [ as <alias> . '*' ] ;
```

NOTE: DASL/AUS does not support importing a single business object class, as is permitted in the Java programming language's *import* statement. This restriction reflects differences between Java and DASL regarding units of compilation and deployment in the two languages.

The optional *alias* is the local name for referring to *bosName*. If it is not given explicitly, the *alias* will be the same as the *bosName*. In either case, the *alias* must be unique in the current BOS.

3.3.2 Import of Java Objects

Currently, the import statement in the AUS does not support direct importing of Java classes. However, Java classes may be referenced explicitly, as described below.

3.3.3 Referencing External Java Classes

In declarations where a type is required, Java classes can be specified as types using the syntax:

```
declare <packageName> . <javaClassName>
```

For example,

```
declare java.text.SimpleDateFormat df;
```

declares a variable *df* of type java.text.SimpleDateFormat, and

```
state editor ( declare java.lang.StringBuffer buf ) { ... }
```

declares a state with a java.lang.StringBuffer parameter.

3.4 Properties of Elements of the AUS

Properties were introduced in the Common Elements chapter. In the AUS, properties may be associated with the following elements:

- The AUS itself
- Declared Variables
- States
- State Parameters
- State entry scripts
- State transition scripts
- State variable usages
- Transactions

The properties for the AUS itself are placed at the top level, outside of a state declaration. The place where properties are specified for other elements is indicated by <...properties> in the BNF syntax for that element.

3.5 State Declarations

The AUS defines a set of states. A state may be thought of as encapsulating a phase of execution of the application during which user interaction occurs. When a state transition occurs to a given state, parameters are pass to it, similar to the way the constructor for a Java class accepts parameter variables. The state defines user interaction and controls execution until the next state is entered.

Each state in an AUS describes an (optional) entry script, a set of objects and attributes that are used by the state, how those objects and attributes are used (e.g., whether they are modified by the application), and the transitions to other states. The key elements of a state include:

• **State Name**	The name of the state.
• **State Parameters**	Variables passed into the state.

• **State Label**	The external name of the state (subject to I18N translation).
• **State Properties**	Optional properties that apply to the entire state.
• **State Variables**	Variables declared local to the state.
• **State Entry Script**	Business logic executed upon entry to the state.
• **State Variable Usage**	Declaration of the variables and data affected by the state.
• **Transitions**	Transitions to other states, and associated business logic.
• **Included States**	Other states that are logically included in this state.

3.5.1 State Declaration Syntax

```
initial state [<stateName> ( ) ] <label>?
{
    <stateElement>*
}

state <stateName> ( <stateParameterDeclarations> ) <label>?
{
    <stateElement>*
}
```

```
<stateElement> ::=  <stateProperties>
               ::=  <stateVariableDeclaration>
               ::=  <entryScript>
               ::=  <usageSpecification>
               ::=  <transition>
               ::=  <stateInclude>
```

Every AUS has an initial state, the state in which the business task represented by the AUS begins.

In the PurchaseOrder application described in Chapter 4, the initial state allows the user to type in his name. When the Login button is pressed, the Login transition determines if a name has been entered. If so, the transition is made to the ChoosePO state. If not, the initial state is re-entered.

3.5.2 State Name

The state name must be unique among the topmost elements of the AUS, including other state names, session variable names, and the aliases of imported business objects. Initial states are not required to have a name, but they need one if there are to be any transitions back to the initial state.

3.5.3 State Parameter Declarations

State parameters are formal parameters, as described in the *Common Elements* chapter. The syntax is the same as for method parameters.

3.5.4 State Label

A state label is a string literal that is used as a label for the state in the default presentation. It is subject to translation by the I18N internationalization code that is created during deployment of the application.

3.5.5 State Variable Declarations

```
        <stateVariableDeclaration> ::= <variableDeclaration>
```

The declaration and scope of variables in the AUS is described more fully below. The syntax of <variableDeclaration> and <scriptExpression> are also described below.

3.5.6 State Entry Script: <entryScript>

```
        entry { <scriptStatement>* }
```

The (optional) state entry script describes the business methods to invoke to obtain the initial values of objects referenced in the state's usage declarations. For example, in the PurchaseOrder Application shown above, the ChoosePO state has an entry script that retrieves all purchase orders belonging to the current user, by calling the factory business method *myOrders(userName)* defined for the business object *PurchaseOrder*. Only those attributes and relationships of each purchase order that are included in the usage declaration need be retrieved.

The entry script may contain a transition statement (*goto* or *exit*). If the transition statement is executed, the state does not produce a visible screen. Such states are sometimes useful for defining business logic that must be executed in a separate transaction.

A state may have only one entry script.

3.5.7 Variable Usage Specification: <usageSpecification>

```
        usage {
            <variableUsage> [ , <variableUsage> ]*
        }

  <variableUsage> ::= <usageVariable> <usageLabel>? [ : <modes> ]
                      <properties>? [ { <memberUsage> } ]


  <usageVariable> ::= <variableName>
                      ::= '[' <collectionVariableName> ']'
                      ::= '[' <collectionVariableName>
                           ( <selectCardinalityMin> , <selectCardinalityMax> ) ']'

  <selectCardinalityMin> ::= 0 | 1
  <selectCardinalityMax> ::= 0 | 1 | n

  <memberUsage>    ::= { <variableUsage> [, <variableUsage> ]* }
                   ::= { }
```

The variable usage declarations within a state describe how variables are navigated, displayed, and modified by the application's user while the state is active. These values need to be retrieved when the state is entered, and later resynchronized with the business logic layer when the state is exited. Note that the navigation might be for the purpose of displaying the value within the state, or it might be for the purpose of presentation programming logic while in the state.

Each usage variable may have a label or name by which it will be displayed in the application. The label is a string literal. The deployed code treats this string as an I18N internationalized string, i.e., the label is used as an entry in the *i18n* localization table. The default English localization is the string itself.

The top-level or *outer* usage variables must be variables that are known to the state. Thus, they can be parameters passed to the state, variables declared within the state, or any other variables available and in scope of the state (e.g., session or shared variables available throughout the session).

If an outer or inner usage variable is a business object, the specific attributes and relationship accessors of that object that are required by the application may be described using a nested set of {}'s describing the member usage of that object's attributes and relationship accessors.

Associated with each variable or navigable member is a mode indicator that describes how this member is used from within the state. Modes are described below.

Square brackets around a usage variable indicate that this value is a collection, and that the nested variable usage that follows applies to each element of the collection. The conventions and semantic flags used to describe the way values are used are described below.

3.5.8 Modes of Usage Variables

```
<modes> ::=  <modeLetter>+

<modeLetter> ::= R | W | C | I
```

Each usage variable navigated to from an outer variable may have an explicit mode describing how the variable is to be used by the application. Some variables are only to be read from the persistent store, while others are read and written (modified) by the user of the application. In addition, the mode can declare the level of read and write "consistency" of variables that the application requires during transactions, as other transactions may change the values independently.[65] Finally, most variables are made visible to the user of the application, but occasionally a variable is used internally and not intended to be visible to the user.

The possible mode letters and their meanings are:

Mode Letter	Meaning
R	variable's value is read from the persistent store
W	variable's value is written to the persistent store
C	consistency is enforced (must be used with R and/or W)[66]
I	variable is invisible (not displayed)

Note that the above mode letters may be combined, e.g., **RW**, **RI**, **RC**, **RWC**, etc. The mode letters must be placed together, with no intervening space.

Modes of usage variables inside the *{}*'s are the same as for the outer indicator, unless a specific mode is given for the inner usage variable, in which case it is used.

65 Read and write consistency will be described more fully later.

66 Omitting the **C** mode allows DASL to choose an implementation that may not preserve read and/or write consistency rigorously. Currently, write consistency is always enforced in all implementations, and read consistency is not enforced by any implementation.

Modes of relationship accessors, such as *lineItems*, refer to the ability to modify the relationship, not the object itself.

3.5.9 Selection Cardinalities of Collection-Valued Usage Variables

The optional cardinality specified in parentheses *()* in *[lineItems(0,n)]* specifies the number of *lineItems* that may be selected at any given time. In the above example, zero or more *lineItems* may optionally be selected. In general, the possible selection cardinalities of a collection-valued usage variable are:

- (0,0) – disable selection of elements (i.e., no elements may be selected)
- (0,1) – zero or one elements may be selected
- (1,1) – exactly one element must be selected
- (0,n) – zero or more elements may be selected
- (1,n) – at least one element must be selected.

The following pseudo-methods may be called within the transition script to obtain the selected elements.

- **<usageVariable>.getSelectedOne()**

  returns a single selected element of a collection.

- **<usageVariable>.getSelectedMany()**

  returns all the selected elements of a collection. To allow more than one element to be selected, the cardinality of the collection must be specified as (0,n) or (1,n) in the state's variable usage.

If the maximum cardinality is 1, *getSelectedOne()* may be used to return the selected element.[67] If the minimum cardinality is 0, *getSelectedOne()* will fail if no element has been selected.

If the maximum cardinality is n, *getSelectedMany()* must be used to return the set of elements that have been selected.[68] If the minimum cardinality is 0, *getSelectedMany()* will return an empty collection if no elements have been selected.

67 The user interface in this case is logically equivalent to a radio button.
68 The user interface in this case is logically equivalent to a checkbox.

3.5.10 Variables Used Only in Scripts Need Not Be Declared as Usage Variables

Because the entry and transition scripts execute as part of the business object layer, and need not be passed to the client for examination or modification, the variables in these scripts need not be included in the state's usage declaration.

3.5.11 Example Variable Usage Section

Consider the variable usage of the EditPO state in the PurchaseOrder Application:

```
usage {
  po: RW {
        orderId "Order ID": R,
        orderDate "Date",

        shipToName "Ship To",
         ...
        billToName "Bill To",
        ...
        [lineItems(0,n)] "Line Items": RW {
                 orderLine "Item": R,
                 productId "Product ID": R,
                 productName "Product",
                 productDescription "Description",
                 unitPrice "Unit Price",
                 quantity "Quantity"
        }
   }
}
```

This particular variable usage starts with a single outer *variable*, the PurchaseOrder ***po***, which was passed into the state. In general, more than one outer variable may be used, though in this example there is only one.

Because *po* refers to a business object, we must describe the precise attributes and relationships that the application intends to navigate and present to the user. For example, the attributes *orderId*, *orderDate*, *shipToName*, *billToName*, and the one-to-many relationship *lineItems* are all being used. The *orderId* attribute is read-only and thus cannot be modified by the user, while the other members can be modified.

The *po* has a relationship accessor called *lineItems* that is a collection. The square brackets *[]* around *lineItems* are required to make it explicit that *lineItems* returns a collection, rather than a scalar value. Zero or more *lineItems* may be selected, and they have mode RW except for *orderLine* and *productId*, which have mode R.

3.5.12 Usage Variable Properties Currently Used by the Deployment Engines

The deployment engines look at the following properties of usage variables as hints concerning the semantics of the variable's usage:

• chooseFrom = “<variableName>”	Allow the variable to be set to one of the elements of <variableName>, which must be a collection of values that are assignable

	to the variable.
• display = “password”	Treat the variable as a password, and thus obscure it as it is being typed.
• display = “html”	Interpret the variable as HTML-formatted text.
• currency = “USD”	Interpret the Decimal variable as a quantity of US dollars, or some other currency code. DASL uses the standard Java names for currencies defined by class Currency.

For example, the variable usage

```
usage {
     password  "Password"    :W  PROPERTIES { display = "password" }
}
```

will allow the user to input a password string, without displaying the characters typed, and the variable usage

```
usage

  customer PROPERTIES { chooseFrom = "allCustomers" } {
       id,
       name,
       company
   }
}
```

will display the id, name, and company of every customer in the collection allCustomers and allow the user to choose one of the customers.

3.5.13 Transitions Out of the State

Transitions into a state occur as the result of executing a goto statement, which specifies the expressions to be passed as parameters to the new state. A transition from state A to state B causes state A to cease, and state B to become the currently active state. Transitions typically occur as a result of a user gesture, such as pressing a button on the GUI.

Each state defines transitions to other states. For each transition, a script is provided to determine the business logic to execute for the transition, and the parameters to pass to the next state. Alternatively, the script may exit from the application, optionally displaying a specific URL when it exits.

The key elements of a transition include:

• **Transition Name**	The name of the transition.
• **Transition Label**	The external name of the transition (subject to I18N)

- **Transition Script**
  - Business logic executed during a transition out of the state
  - Conditional transition logic (optional)
  - The target state invocation(s), including parameter values to pass to the new state

- Error handling logic (optional)

3.5.14 Transition Declaration Syntax

```
transition <transitionName> <transitionLabel>? <transitionProperties>?
{ <scriptStatement>* }
```

A transition declares a way to move from the current state to a new state (the target state), updating and synchronizing locally changed business object variables, optionally executing business logic in the BOS layer, and possibly using programming logic to choose the target state.

During transitions, all changed business objects are automatically synchronized with the business objects, i.e., the changed values at the AUS layer are conceptually propagated to the active business objects in the BOS layer. Thus, when the first statement in a script is reached, the AUS and BOS layers are synchronized.

The transition logic is described via a script containing:

- statements, such as assignments and calls to business object methods,
- script-local variables to hold the results of business object method calls, and
- transitions to new states, including the parameters passed from the current state into the new state.

The ability to invoke specific business object methods is very important because in most real world applications, the business object methods encapsulate significant behavior about the objects, such as how to hire a new employee or what steps to take when creating a new purchase order.

The business object method calls may navigate from a known object in the state of origin to a new value or object, via relationship accessor methods. They may also call factory methods associated with a business object class, such as *PurchaseOrder.findByPrimaryKey(pk)* to find a persistent object.

In the most simple case, a transition has a single destination state. However, transitions may optionally have more than one destination state, where the choice of destination state is determined by the result of calling a business object method.

From the declarative information in the transition, DASL derives the set of possible states to which this transition might be made, and the specific actual parameter values to pass into each of those states.

If execution "falls out" of a transition script without executing any transfer statements (goto's or exit's), an implicit continue statement is performed (see below).

In some cases, it is useful to define a state that does not allow any user interaction. Such as state may be defined by including a *goto* in the entry script. If the *goto* in the entry script is unconditional, there is no reason to include usage variables or transitions in the state.

The script statements are described below.

3.5.15 Transition Properties Currently Used by the Deployment Engines

The deployment engines look at the following properties of a transition as hints concerning the semantics of the usage:

• refresh = "nnn"	Invokes the transition automatically every nnn seconds, where nnn is a decimal number. This property may be used to create a transition that automatically refreshes a page.

For example,the transition

```
transition Refresh ""    properties { refresh = "20" }
{
    allOrders = PurchaseOrder.getAllInstances();
    continue;
}
```

will recompute the collection of all purchase orders every 20 seconds, and stay in the current state.

The transition

```
transition Done "Done"    properties { refresh = "30" }
{
    goto DoneEditing();
}
```

will go to the *DoneEditing* state automatically after 30 seconds.

3.5.16 Including a Nested State: <stateInclude>

```
include <stateCall> [ when ( <booleanScriptExpression> ) ] ;

include URL ( <stringLiteral> ) [ when ( <booleanScriptExpression> ) ] ;


<stateCall> ::= <stateName> ( [ <scriptExpression> [, <scriptExpression> ]* ] )
```

A state may include other states, either unconditionally or when a boolean expression evaluates to *true*. Script expressions are described below. Included states are merged into the including state at its end, in the order in which they are included.

The entry script of the including state is performed first, and then the entry scripts of the included states are performed, in order. The variable usages of the including state and the included states are combined and, by default, presented in the same order as the entry scripts are executed.

Included states are useful for presenting the same logical substate on multiple pages. For example, in an application that has a common menu for all pages, the common menu is easily implemented as a state that is included in all the main states of the application.

3.6 Variable Declarations and Their Scope and Lifetimes

Within an AUS specification, variables may be declared with a variety of scopes and lifetimes. As in the Java programming language, variables that refer to objects are merely references to those objects.

The objects themselves may have a lifetime beyond that of the variable that references them if there exists another reference to that same object that remains in scope.

3.6.1 AUS Variable Declarations: <variableDeclaration>

The states and transitions use the BOS type system to describe the types of objects and values. All variable declarations share the common syntax

```
<variableDeclaration>   ::=  <normalDeclaration>
                        ::=  declare <qualifiedDeclaration>


<normalDeclaration> ::=  <ausType> <variableName> [ = <scriptExpression> ]
                                   <variableProperties>? ;

<qualifiedDeclaration> ::=  <qualifiedClassName> <variableName> [ =
                            <scriptExpression> ] <variableProperties>? ;

<ausType> ::=  [ <localName> . ] <className>
          ::=  [ <localName> . ] <EBTName>
          ::=  Collection [ '<' <ausType> '>' ]
          ::=  List [ '<' <ausType> '>' ]
          ::=  Set [ '<' <ausType> '>' ]
          ::=  SortedSet [ '<' <ausType> '>' ]
          ::=  Map [ '<' <ausType> '>' ]
          ::=  <ausType> '[]'
```

where *<ausType>* is basically one of the attribute or object types in a BOS, or a collection of such types, with an optional local name specified in the import statement.

When declaring variables at the session level, outside of a state, the optional keyword *shared* may be used to signify that the variable is shared among all sessions (see below).

The optional keyword *declare* may be used to introduce a type not otherwise known at the BOS level, such as an arbitrary Java class. The necessity for the *declare* keyword is a temporary implementation restriction that will be lifted in the future.

3.6.2 Shared Variables

Shared variables are declared in the AUS outside of any state, using the scope keyword *shared*. They have a scope of all the states in the application, and a lifetime beyond the entire application session. They are shared among all of the other sessions of this same application.[69]

Shared variables are transient (non-transactional), but they can refer to business objects in the persistent store, which are transactional. For example, an application might use a shared variable to cache frequently accessed read-only objects, to save the cost of retrieving those objects during each session. Shared variables are also convenient for accumulating information between different user

69 Due to implementation limitations of some deployment platforms, it is not recommended that applications depend on shared variables being shared by all sessions. In particular, some application servers may scale by creating separate processes, and then provide a separate instance of each shared variable for each process. Using shared variables for optimization purposes, such as a cache, should work despite these implementation limitations.

sessions, such as the number of concurrent users. Shared variables may be referenced in anywhere within the AUS.

3.6.3 Session Variables

Session variables are declared in the AUS outside of any state. They have a scope of all the states in the application, and a lifetime equal to the entire application session. However, unlike shared variables, they are private to a particular session and thus not directly accessible outside the session.

Session variables are convenient for accumulating information for the particular session. For example, they can be used to maintain a shopping cart or user login information. Session variables may be referenced anywhere within the AUS.

3.6.4 State-Local Variables

Variables declared within a state have a scope within the state, and a lifetime that begins when the state is entered and ends upon transition out of the state, typically upon execution of a *goto* statement in one of the state's transitions. State-local variables may be referenced in usage declarations, as well as in entry and transition scripts.

3.6.5 Script-Local Variables

Variables declared within a script have a scope within the script, and a lifetime that begins when the script is entered and ends when execution falls out of the script, or when a script performs a *goto* or *exit* statement.

3.7 The AUS Scripting Language

Business logic in the form of a script may be associated with entry to a state, or to transition out of the state. The script contains calls to appropriate methods on business objects, as well as the next logical state in the business process.

The primary purpose of the scripts is to invoke business logic that is defined at the level of business objects, and to select the next state to which the application needs to transition. Instance methods are invoked via variables that refer to particular objects. Factory methods are invoked by using the name of the business object, e.g., *PurchaseOrder.myOrders(userName).*

Although the scripting language could support all the control structures of methods, rather than just the ability to do assignments and invoke business object methods, it is felt that placing arbitrarily complex programming logic in a transition script detracts from the clarity of the application specification. Because any business object method may be invoked, and because business object methods have access to the full Java language, the current restriction does not limit the code that may be invoked in a transition.

3.7.1 Scripting Language Statement Syntax: <scriptStatement>

The statements that are supported in entry and transition scripts are as follows:

```
<scriptStatement> ::=  <scriptProperties>
                  ::=  <variableDeclaration>
                  ::=  <scriptExpression> = <scriptExpression>
                  ::=  <scriptExpression>
                  ::=  if ( <scriptExpression> ) <scriptStmtOrBlock>
                         [ else <scriptStmtOrBlock> ]
                  ::=  <transferStatement>
                  ::=  throw <scriptExpression>
                  ::=  on error [ ( <exceptionVariableDeclaration> ) ]
                          <scriptStmtOrBlock>
                  ::=  on return <scriptStmtOrBlock>
                  ::=  CODE [ ( <language> ) ] { <codeBody> }
```

Most script statements are similar to the statements that may occur in the body of a method, including declarations, assignment statements, and conditional statements (if-else). The *if* and *else* may be a script statement or a block of script statements, enclosed in curly brackets:

```
<scriptStmtOrBlock> ::=  <scriptStatement>
                    ::=  { <scriptStatement>* }
```

Unlike method bodies, scripts may contain statements that transfer control to another state.

```
<transferStatement> ::=  goto <stateCall>
                    ::=  continue
                    ::=  exit [ to URL ( <stringLiteral> ) ]
                    ::=  switch ( <scriptExpression> ) { <scriptSwitchCase>* }
```

The *goto* statement invokes a new state, performs its entry script, and lets the user interact according to that state's definition. When invoking another state, parameters must be passed into the state:

```
<stateCall> ::= <stateName> ( [ <scriptExpression> [ , <scriptExpression> * ] )
```

The *continue* statement goes back to the current state, preserving any modifications that have been made by the user, waiting for a user gesture that triggers a transition. If the current state included other states, those states are still included unless they were included conditionally and the condition no longer applies. The continue statement does not execute the entry script of a state again, but if a previously not included state is now included, and that state's entry script has not yet been executed, it is now executed.

The *exit* statement terminates the application, rolling back any pending transactions, and then optionally displaying a URL on the user's screen.

The *switch* statement is a conditional transfer to one of several states, based on the value of an expression.[70] A common usage pattern is to use a String variable whose value is returned by a business method.

```
<scriptSwitchCase> ::=  <switchLabel> <transferStatement>
```

70 In a future release of DASL, the switch statement will be generalized so that <scriptSwitchCase> can be an arbitrary list of script statements.

```
<switchLabel> ::=  case <constantExpression> :
              ::=  default :
```

The *on error* and *on return* statements are described under comment language elements.

The CODE statement allows arbitrary deployment-level code to be inserted into a transition. Because different deployment engines may produce different scripting languages, the CODE statement may be qualified with the name of a language that the deployment engine uses. If the language is specified, an error message will be produced by the deployment engine if it encounters a language it does not know how to generate.

```
<language> ::=  <stringLiteral>

<codeBody> ::=  /** any sequence of characters with balanced {}'s */
```

3.7.2 Scripting Language Expression Syntax: <scriptExpression>

Script expressions are similar to Java and C++ expressions. They include literals, component selectors, method calls, parenthesized expressions, and combinations of these using prefix and infix operators.

```
<scriptExpression> ::=  <literal>
                   ::=  <rootVariable> <componentSelector>*
                   ::=  ( <scriptExpression> )
                   ::=  <prefixOp> <scriptExpression>
                   ::=  <scriptExpression> <infixOp> <scriptExpression>
                   ::=  <usageVariable> . <selectionMethod>

<rootVariable> ::=  [ <localName> . ] <className>
               ::=  [ <localName> . ] <EBTName>
               ::=  <variableName>

<componentSelector> ::=  . <componentName>
                    ::=  '[' <scriptExpression> [ , <scriptExpression> ]* ']'
                    ::=  ( [ <scriptExpression> [ , <scriptExpression> ]* ] )

<selectionMethod> ::=  getSelectedOne()
                  ::=  getSelectedMany()
```

3.7.3 Scripting Language Literal Syntax: <literal>

Some example String, numeric, boolean,and object literals (respectively) are: *"blue"*, *10*, *true*, and *null*.

```
<literal> ::=  <integerLiteral>
          ::=  <floatingPointLiteral>
          ::=  <booleanLiteral>
          ::=  <characterLiteral>
          ::=  <stringLiteral>
          ::=  null

<integerLiteral> ::=  <decimalIntegerLiteral>
                 ::=  <hexIntegerLiteral>
                 ::=  <octalIntegerLiteral>

<decimalIntegerLiteral> ::=  <decimalNumeral> <integerTypeSuffix>*
```

```
<hexIntegerLiteral> ::=  <hexNumeral> <integerTypeSuffix>

<octalIntegerLiteral> ::=  <octalNumeral> <integerTypeSuffix>


<integerTypeSuffix> ::=  u
                    ::=  L

<decimalNumeral> ::=  <digit>+

<hexNumeral> ::=  0 x <hexDigit>+
             ::=  0 X <hexDigit>+

<hexDigit> ::=  <digit>
           ::=  a | b | c | d | e | f
           ::=  A | B | C | D | E | F

<octalNumeral> ::=  0 <octalDigit>*

<octalDigit> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7


<floatingPointLiteral> ::=  <digit>+ . <digit>+ <exponentPart>
                            <floatTypeSuffix>
                       ::=  . <digit>+ <exponentPart> <floatTypeSuffix>
                       ::=  <digit>+ <exponentPart> <floatTypeSuffix>
                       ::=  <digit>+ <exponentPart> <floatTypeSuffix>

<exponentPart> ::=  <exponentIndicator> <signedInteger>

<exponentIndicator> ::=  e | E

<signedInteger> ::=  <sign> <digit>+

<sign> ::=  + | -

<floatTypeSuffix> ::=  f | F | d | D

<booleanLiteral> ::=  true | false

<characterLiteral> ::=  ' <unicodeCharacter> '
                   ::=  ' <stringLiteralEscape> '

<stringLiteralEscape> ::=  \ <hexNumeral>
                      ::=  \ <stringCharacter>

<stringLiteral> ::=  " <stringCharacter>* "

<stringCharacter> ::=  /** character except double quote and backslash */
                  ::=  <stringLiteralEscape>

<unicodeCharacter> ::=  /** any UNICODE character */

<commentStringCharacter> ::=  /** character except '*' '/' */

<nonNewlineStringCharacter> ::=  /** character except newline */

<nullLiteral> ::=  null
```

3.8 Syntax For Accessing Factory Methods in the AUS

A factory method on a business object may be invoked from an AUS transition using a standard syntax, regardless of the selected deployment architecture. The deployment engines translate this syntax into the appropriate call. The syntax is:

```
[<bosName> . ] <businessObjectName> . <methodName> ( <methodActualParameters> )
```

For example, in the PurchaseOrder Application, the object PurchaseOrder defines a method *myOrders* to retrieve all orders for a customer. This method may be invoked from a script using:

```
PurchaseOrder.myOrders(billToName)
```

In the AUS, if there is only one BOS being referenced, or if more than one BOS is referenced but the name of the business object is found in only one BOS, the simple business object class name may be used unambiguously. If there are multiple BOSes associated with an AUS and an object with a given name is found in more than one BOS, the local name of the BOS must be used to qualify the ambiguous object name. For example, in the PurchaseOrder application's AUS, the disambiguated call would appear as follows:

```
purchaseOrders = POApplication.PurchaseOrder.myOrders(customerName);
```

3.9 Application Transactions

Transactions are declared at the same level as states. The key elements of a transaction include:

• **Transaction Name**	The name of the transaction (for documentation purposes).
• **Comprising States**	The set of states comprising the transaction.
• **Committing States**	The set of states which commit the transaction.
• **Cancelling States**	[by elimination] all other states cancel the transaction.

3.9.1 Transaction Declaration Syntax

```
transaction <transactionName> { <includedStates> [, <includedStates>]*
                                commits to <commitState> [, <commitState>]*
                              }
      <commitState> ::=  <stateName>
                    ::=  exit
```

Several states and transitions may be grouped into a single transaction, meaning that the set of edits represented by those states either take place in full and without error, or else do not occur. These application transactions have the same ACID properties usually associated with database transactions.

Although application transactions are often implemented using database transactions, they do not necessarily correspond directly to database transactions. They are defined on a higher level, according to application semantics.

Database transactions are often too consumptive of resources and hold on to too many locks to be used directly by highly scalable applications. Even when the underlying DBMS provides some notion of optimistic concurrency, that notion may not correspond with the application's notion, and in particular, the application often has isolation requirements that differ for each datum that is used, while a generic DBMS optimistic concurrency mechanism does not.

3.9.2 Explicit Transactions

A DASL application transaction specification contains information about the transaction boundaries of the application. It consists of a set of states that encompass a distinct business transaction, along with the states outside that transaction.

Consider the following example of a transaction declaration:

```
transaction Tx { A, B, C commits to D, E }
```

The following rules apply to explicitly declared transactions:

- A transition from a state outside the transaction (states other than *A, B,* or *C*) to one of the transaction states (*A, B,* or *C*) begins the transaction.
- A transition from a state within the transaction (*A, B,* or *C*) to another state within the transaction (*A, B,* or *C*) continues the transaction.
- A transition from a state within the transaction (*A, B,* or *C*) to a *commit state* outside the transaction (*D* or *E*) signals that the transaction should be committed.
- A transition from a state within the transaction (*A, B,* or *C*) to a state outside the transaction that is not listed as a *commit state* signals that the transaction should be rolled back (aborted).

For example, in the PurchaseOrder Application, there is a transaction called *edit* that encompasses states *CreatePO* and *EditPO.* The transaction begins upon any transition to either of these two states. The *edit* transaction commits to state *ChoosePO*, which means that any transition to *ChoosePO* from within the transaction will cause the transaction to commit. By inference, any transition to a state other than *CreatePO*, *EditPO*, or *ChoosePO* from within the transaction will cause the transaction to roll back.

As a special case, if a state is included in the transaction and that same state is also in the *commits to* states of the transaction, a transition from within the transaction to that state will cause a commit and a new transaction to start. This special case is provided to avoid the need to define a new state outside the transaction that redirects to a state inside the transaction. An example of this special case is:

```
transaction Tx2 { MakeChanges commits to MakeChanges }
```

3.9.3 Implicit Transactions

Each state that is not grouped within a transaction is assumed by default to be in its own transaction. Thus, if state N is not one of the states within a declared transaction (e.g., *A, B,* or *C* above), then the following implicit transaction is assumed:

```
        transaction Tx_N { N commits to * }
```

where '*' indicates all states, including state N.

Following the rules for explicit transactions, it is the case for implicit single state transactions that:

- The transaction begins upon entry to the state.
- The transaction commits after the transition script out of the state executes.

3.9.4 Summary of Transactional Behavior During Transitions

Application-level transactions start, commit, and abort as a result of transitions.

Consider a transition from state A to state B. If B is part of the same transaction as A, then A's transaction continues. If B is one of the commit states for A's transaction then A's transaction is committed, and then B's transaction begins. If B is neither part of A's transaction nor one of the commit states of A's transaction then A's transaction is rolled back, and then B's transaction begins.

3.10 Synchronization of Application State with Server and Database States

The transaction semantics described above require synchronization at the start of the transaction, and also at the end of the transaction. However, in distributed application architectures, there are times when the various tiers are not completely synchronized with each other during a transaction.

Consider a multi-tier architecture consisting of a web browser interacting with a web server, which in turn communicates with an application server containing business objects, which in turn communicates with a DBMS server that manipulates a database. During the execution of the application, there will be times when the browser screen does not reflect the latest changes to the underlying database, and most probably there will even be times when the web server and application server do not reflect the latest changes to the database.

Therefore, to fully understand the way an application works, the following rules of synchronization should be understood:

- All business objects changed by user interaction within a state are resynchronized with the business object layer prior to execution of the transition script.
- If a transition is made to a *commit state*, the changed business objects will be synchronized with the database; otherwise, they may or may not be synchronized with the database, depending on the implementation.

Whenever a state is entered as part of a new transaction, the values it obtains from its parameters and the values it obtains in its entry script are synchronized with the database upon entry.

While the application remains in that state, and any other states that make up the same transaction, the application is isolated from the database and may become unsynchronized with changes other sessions make to the database.

When a transition occurs to a state that commits the current transaction, the application and database states are resynchronized by sending changes made in the application to the database, and detecting changes made to the database by other applications. Conflicting changes are generally detected at this time, though they may be detected earlier, depending on the implementation.

3.10.1 Read Isolation Between Transactions; "Collision" Detection For :C Mode

Currently, DASL relies on the underlying deployment architecture to provide varying degrees of read isolation and optimistic or pessimistic collision detection when more than one transaction changes the same data. None of the current architectures provide the precise semantics that DASL needs for consistent read mode (:C). Thus, if a value is read in a transaction and then changed by another session, that will not cause the reading transaction to fail unless it also modified the value. This lack of strict adherence to the DASL requirements will be addressed in future deployment engines.

3.10.2 Relaxing Isolation for Increased Scalability

Strict isolation between sessions can cause severe scalability problems due to resource contention. Therefore, DASL provides various ways to "relax" the isolation level of the persistent store, according to the logic of the application.

For example, collection variables within the AUS may be used to store collections computed by query from the BOS. It is the choice of the application whether to recompute the collection upon entry to a state, or alternatively, to reuse the previously computed collection (which may become out-of-date). The semantics of the application dictate whether the possibility of having a slightly out-of-date collection is acceptable.

3.10.3 References to Business Objects Kept in Session Variables Across Transactions

When a reference to a business object is kept in a session or shared variable across a transaction, the reference is preserved as a primary key to the desired object, and the object's attributes are refreshed from the business object layer when the new transaction first examines the object.

4 DASL/OQL Language Specification

To enable applications to be specified regardless of the underlying DBMS or object-relational mapping mechanism, we define a standard object-oriented query language in business object methods defined using queries. Currently, that language is compatible in syntax with a subset of the standard object-oriented query language, OQL, and also provides a superset of capabilities. The primary restriction is that DASL does not allow arbitrary business object methods to be invoked within the query.

DASL/OQL supports EJBQL syntax (EJBQL is the query language defined as part of the J2EE EJB specification). Our implementation tries to accept as much of EJBQL syntax as it can, except for some changes that were required to remove implementation details from the query (so it makes sense for any architecture and software stack). Specifically, compared to EJBQL, DASL/OQL:

- refers to an extent of a class (the collection of all instances of that class) using the business object name, rather than the EJB implementation name, which usually ends with "Bean," e.g., *Employee e* instead of *EmployeeBean e* .
- uses named parameters (e.g., :*name*, :*addr*) instead of ordinal positions (e.g., *?1*, *?2*) because we feel it is more readable. For the EJBQL diehards, we do accept numbered parameters as an option, and we also allow the syntax ?*name* to mean the same as :*name*.
- does not require the syntax *Object(e)* to select an entire object, but we accept it optionally.

4.1 Overview

This chapter presents an overview of DASL/OQL and the use of SQL queries. The syntax of DASL/OQL may be found in the appendix.

An <OQLQuery> has the form:

```
SELECT e
FROM Employee e
WHERE e.name = 'Joe' AND e.manager.name = 'Bob'
```

Parameters of the method containing the query are referenced in the query using the syntax

```
: <formal>
```

or

```
? <formal>
```

e.g., in method foo(String what), the query may refer to the method parameter as

```
:what
```

The syntax in the appendix is presented in what we consider the canonical or standard form. We allow variations of this form, for compatibility with other query languages such as EJBQL, JDOQL, and SQL.

An <SQLquery> undergoes translation from BOS member names to the underlying table column names. Eventually, navigational chains such as *e.manager.name* will also be translated in SQL queries, but currently they are not, so instead you need to do the appropriate joins, e.g.,

```
SELECT e
FROM Employee e, Manager m
WHERE e.name = 'Joe' AND m.name = 'Bob' AND e.manager = m.id
```

If multicolumn primary keys are being used, refer to the SQL ddl (<applicationName>.sql) file produced during generation of the application to discover the column names containing the foreign keys (e.g., e.manager in the example above).

4.2 Categories of Queries

A general query language such as SQL (e.g., SQL92) includes sub-languages for data definition (SQL DDL) as well as data manipulation (SQL DML). Data manipulation includes not only retrieval of data (SQL SELECT statement) but also modification of data (SQL INSERT, UPDATE, and DELETE statements).

When creating an application using the DASL sub-language(s), the data definition (DDL) step occurs at the level of defining persistent business objects, and mapping those objects to new or existing underlying databases. The mapping of business objects to new databases is performed automatically, and the mapping to existing databases is performed using the ORMAP language.[71]

At the present time, the architectures and software stacks for which DASL deploys implementations support only SELECT-type queries. We believe that there is an important subset of applications that are best expressed using queries to modify data. At some point, we intend to add support for data modification queries into DASL/OQL. If necessary, we will implement them by bypassing part of the architecture, going directly to the JDBC connection to the underlying database.

The queries DASL supports fall into the following categories:

4.2.1 Query to Retrieve Objects

select <alias> from <businessObject> <alias> where <expression>

For example,

```
select po from PurchaseOrder po where po.billToName = :customerName
```

The named parameter :*customerName* refers to a parameter to the method signature in which this query is defined. If the method signature returns a single object rather than a collection, the method will raise an exception if more than one object is returned by the query.

4.2.2 Query to Retrieve An Attribute Value

select <attribute> from <businessObject> <alias> [where <expression>]

71 The ORMAP language is a future enhancement of DASL.

For example,

```
          select po.orderDate from PurchaseOrder po
          where po.orderId = :desiredOrderId
```

The query returns the *orderDate* of the *PurchaseOrder* whose *orderId* matches the value passed in the method parameter *desiredOrderId*.

4.2.3 Query to Navigate Relationships

select <accessor> from <businessObject> <alias> [where <expression>]

For example,

```
          select emp.manager.name from Employee emp
          where emp.name = 'Jones'
```

The query selects the name of the manager of the employee named “Jones”, by navigating from the employee to its corresponding manager object via the relationship accessor *manager*.

4.2.4 Query to Retrieve An Arbitrary Non-Object Value

select <expression> from <businessObject> <alias> [where <expression>]

For example,

```
          select MAX(po.orderId) from PurchaseOrder po
```

The query returns the maximum orderId in use. If the method signature returns a single value rather than a collection, the method will raise an exception if more than one value is returned by the query.

In both cases above, the query may return a single object or value, or a collection of objects or values.

4.3 DASL/OQL Query Syntax

Please refer to Appendix B for the syntax of DASL/OQL.

4.4 Example Queries in DASL/OQL

Consider the BOS for the PurchaseOrder Application we've been using. We will define some example query methods using the two objects in this BOS and describe what they do.

4.4.1 Query That Returns a Collection Of Objects

```
        factory method myOrders(String billToName)
                        returns Collection<PurchaseOrder>
        OQL ( SELECT p
              FROM PurchaseOrder p
              WHERE p.billToName = :billToName )
```

This query returns all *PurchaseOrder* objects with the specified *billToName* value.

4.4.2 Query That Returns a Single Object

```
        factory method oneOfMyOrders(Long orderId) returns PurchaseOrder
        OQL ( SELECT p
              FROM PurchaseOrder p
              WHERE p.orderId = :orderId )
```

This query returns the *PurchaseOrder* object with the specified primary key value.

4.4.3 Query That Returns a Collection of Objects Using Relationship Navigation

```
        factory method firstLineItem( String billToName )
                        returns Collection<LineItem>
        OQL ( SELECT li
              FROM LineItem li
              WHERE li.orderLine = 1
                   AND li.masterOrder.billToName = :billToName )
```

This query returns the first *LineItems* of all *PurchaseOrder* objects with the specified *billToName* value. Notice that it uses the inverse relationship *masterOrder* to compare the *billToName* attribute of the *PurchaseOrder* that owns the *LineItem*.

4.4.4 Query That Returns a Collection Of EBT Values

```
        factory method ids() returns Collection<Long>
        OQL ( SELECT p.orderId
              FROM PurchaseOrder p )
```

This query returns all *PurchaseOrder orderIds* as a collection of *Long* values.

4.4.5 Queries That Return a Single EBT Value

```
        factory method nextId() returns Long
        SQL ( SELECT MAX(p.orderId) + 1
              FROM PurchaseOrder p )
```

This query returns the next available *orderId*. Notice that this query is stated as an SQL query, rather than as an OQL query. Some OQL implementations do not yet support using the *MAX* function.

4.4.6 Query on an Owned Object That Returns a Single EBT

```
        factory method nextLineNumber(Long orderId) returns Long
        SQL ( SELECT MAX(li.orderLine) + 1
              FROM LineItem li
              WHERE li.masterOrder.orderId = :orderId )
```

This query returns the next unused *LineItem* number in the *PurchaseOrder* with the specified primary key. Again, notice that this query is stated as an SQL query, rather than as an OQL query. Some OQL implementations do not yet support using the *MAX* function.

4.4.7 Query That Returns Owned Objects Selectively

```
       factory method expensiveLineItems( Decimal price, String billToName )
            returns Collection<LineItem>
       OQL (  SELECT li
              FROM LineItem li
              WHERE li.masterOrder.billToName = :billToName
                 AND li.unitPrice > :price )
```

This query returns only the “expensive” *LineItems* of certain *PurchaseOrders*, i.e., the *LineItems* whose unit price exceeds the specified price for the *PurchaseOrders* with the specified *billToName* value.

APPENDIX A: DASL Command Line Interface

a. Installation of the DASL Development Environment

Prior to using the commands described in this appendix, you must install the DASL development environment. The directions for doing so are distributed with the environment itself. You should also modify your system path environment variable to include the folder containing the dasl command. If you installed the full DASL environment, including NetBeans or Sun Java Studio, you can find out the path to the dasl command by clicking on Help -> About -> Detail Tab and look at the value of the "User Dir" field at the bottom of the screen. Then add the following to the system path:

```
<UserDir>/modules/bin
```

b. Syntax of the *dasl* command[72]

```
dasl <globalOption>* [ <option>* <applicationName> <option>* ]
```

The complete list of options for the *dasl* command is displayed using the command:

```
dasl -help
```

c. Commands for Creating a New DASL Application File

The easiest way to set up a DASL program for compilation is to create an application folder (directory) to contain the application-related files. In this folder, create another folder called "src" to contain the DASL source files for your application. Create the DASL files (e.g., RememberMe.bos and RememberMe.aus) that make up your application source in this directory.

For example, to create an application called RememberMe using Unix shell commands, do the following:

```
% mkdir RememberMe
% cd RememberMe
% mkdir src
% cd src
% vi RememberMe.bos  ... enter the BOS for RememberMe
% vi RememberMe.aus  ... enter the AUS for RememberMe
% cd ..
% dasl -create RememberMe -add src/RememberMe.bos -add src/RememberMe.aus
```

In this example, we first create the application directory, *RememberMe*, and then create a subdirectory called *src* below the application directory. We then create the *.bos* and *.aus* files in the *src* subdirectory (it would also work to create those files anywhere in or below the application directory). The ***dasl*** command is then used while connected to the application directory to mark the application directory as a DASL application (it creates the file *RememberMe.dasl*, which is a parameter file for the entire

72 This syntax works on Windows only if the cygwin package is installed. It works on Solaris and MacOS X as shown.

application). Finally, the *.bos* and *.aus* files are added to the application. Once the RememberMe.dasl file is created, the above steps need not be repeated.

Additional *.aus* and *.bos* files may be added or removed from the *RememberMe* application using the commands:

```
dasl RememberMe -add <file>        adds the file to the application.

dasl RememberMe -remove <file>     removes the file from the application.

dasl RememberMe -list              lists the files in the application.
```

d. Basic Commands for Executing a DASL Application

The following *dasl* command is used to deploy the application named *RememberMe* (defined by the file *RememberMe.dasl*) using the default architecture choices:

```
dasl RememberMe -execute           creates, builds, and runs an application
                                   using the current or default parameters.

dasl RememberMe -run               runs a previously executed or deployed
                                   application.
```

e. Commands for Individual Deployment Steps

The steps performed by *-execute* may be performed separately using the following dasl commands:

```
dasl RememberMe -verify     checks the DASL files for syntax errors.

dasl RememberMe -generate   does -verify, then produces the static part of the
                            application intended for deployment.

dasl RememberMe -compile    compiles the static part of the application.

dasl RememberMe -build      builds a previously generated application
                            (jars up the compiled files, etc.).

dasl RememberMe -deploy     does -build, then deploys a previously generated
                            application (into a web or application server).

dasl RememberMe -target <t> runs the specified build target (ant).
```

f. Deployment Options

The default deployment architecture choices may be modified using the following *dasl command options*. The command ***dasl -architectures*** or ***dasl -a*** lists all the possible deployment choices.

```
-architecture <arch>                           target architecture
                                               (default: 2tier).

-webserver <ws>                                target Web Server for 2tier
                                               (default: tomcat).

-appserver <as>                                target App Server for 3tier
                                               (default: sunone).
```

`-persistence <mapping>`	*server side object implementation (default for 2tier: jdo).*
`-presentation <pres>`	*client-side presentation framework (default: struts).*
`-dbms <dbms>`	*target DBMS (default: pointbase).*

g. Setting Generator-specific Deployment Properties

The command ***dasl -architectures*** or ***dasl -a*** lists all the possible deployment properties for all generators.

`-properties`	*lists the generation properties used for the current command.*
`-readproperties <file>`	*reads generation properties from <file>.*
`-set <property>`	*sets <property> with value of empty string.*
`-set <property>=<value>`	*sets <property> to the specified value, which must be enclosed in double quotes.*
`-unset <property>`	*unsets a previously set property.*

h. Deployment Debugging Options

`-verbose`	*print verbose output during the deployment steps.*
`-dasldebugging`	*enable DASL debugging.*
`-nodasldebugging`	*disable DASL debugging (default).*
`-dasllinemapping`	*enable DASL <--> Java line mapping.*
`-nodasllinemapping`	*disable DASL <--> Java line mapping (default).*

i. Deployment Option Pass-through

Some experimental deployment engines require options that are not supported by the dasl command directly. These options can be passed through to the deployment engine using the syntax:

`-option <option>`	*passes <option> to the deployment engine. Be sure to include the leading '-' in <option>.*

j. Commands For Importing and Related Utilities

```
dasl -createausfrombos <bosFile>   creates a default AUS based on a BOS that
                                   contains states to create, retrieve, update,
                                   and delete each persistent object in the
                                   BOS.

dasl -importddl <sqlFile>          imports SQL DDL statements from an .sql file
       [ -as <bosFile> ]           into a BOS. If the bosFile is not given,
                                   the ddlFile will be used with ".bos"
                                   appended to it.

dasl -importuml <umlFile>          imports UML statements in an .xml file
       [ -as <bosFile> ]           into a BOS.  If the bosFile is not given,
                                   the umlFile will be used with ".bos"
                                   appended to it.
```

APPENDIX B: Complete BNF of the DASL Language

BNF Notation Used

The special meta-symbols for BNF grammar are:

```
::=   @   <   >   [   ]   |   *   +   ?
```

The following terminal symbols in the grammar must be quoted:

```
'::='  '@'  '<'  '>'  '['  ']'  '|'  '/*'  '/**'  '*/'  '//'
```

The terminal symbols '*', '+', and '?' need not be quoted unless they follow ']' or '>' with no intervening space.

```
<xyz> is a non-terminal symbol named 'xyz'.
<xyz>? is zero or one <xyz>
<xyz>* is zero or more <xyz>
<xyz>+ is one or more <xyz>
[ <xyz> <pqr> ] is an optional <xyz> <pqr>
<xyz> [ , <xyz> ]* is one or more <xyz> separated by commas
[ <xyz> ; ]* is zero or more <xyz> terminated by semicolons
[ <xyz> ; ]+ is one or more <xyz> terminated by semicolons
[ <xyz> | <pqr> ] is either <xyz> or <pqr> or nothing
```

Spaces between non-alphanumeric symbols in the BNF notation are optional.

Comments in the BNF grammar may be denoted by /* ... */ or /** ... */ or by a line that starts with //, which will not be read into the parser, and thus not preserved by the GUI editor.

NOTE

The BNF grammar that follows is available as an HTML file that can be navigated via links from the use of a non-terminal symbol to its definition.

DASL Program

```
/** Annotated BNF for the DASL Language. */

// The root productions are (alphabetically):
//
//     <DaslProgram>
// ___________________________________________

<DaslProgram> ::=  <BOS>+ <AUS>*
              ::=  <BOS>* <AUS>+
```

BOS

```
/** Annotated BNF for the DASL BOS Language (.bos file) */

// The root productions are (alphabetically):
//
//     <BOS>
// ___________________________________________

<BOS> ::=  <documentedBosElement>*

<documentedBosElement> ::=  <daslComment>* <bosElement>

<bosElement> ::=  <packageDeclaration>
             ::=  <bosProperties> ;
             ::=  <bosImport>
             ::=  <classDeclaration>
             ::=  <ebtDeclaration>
             ::=  <eventDeclaration>

/* Imports */

<bosImport> ::=  <bosClassImport>
            ::=  <javaClassImport>

/** class, EBT, and eventDeclarations */

<classDeclaration> ::=  <classModifier>* class <className>
                        <inheritanceDeclaration>? { <documentedClassElement>*
                        }

<ebtDeclaration> ::=  <ebtModifier>* EBT <EBTName> <inheritanceDeclaration>? {
                      <documentedClassElement>* }

<eventDeclaration> ::=  <eventModifier> event <eventName>
                        <inheritanceDeclaration>? { <documentedClassElement>*
                        }

<classModifier> ::=  abstract
                ::=  persistent
                ::=  transient
                ::=  facade
                ::=  factory

<ebtModifier> ::=  code
              ::=  immutable

<eventModifier> ::=  abstract
                ::=  persistent
                ::=  transient

<inheritanceDeclaration> ::=  extends <name> [ using <inheritanceImpl> ]

<inheritanceImpl> ::=  STATIC_INHERITANCE
                  ::=  DYNAMIC_INHERITANCE
```

```
/** NOTE: <classDeclaration> resembles a Java class declaration with ANSI
SQL clauses added. Each <classDeclaration> declares an object class. Transient
classes are not persistifiable. Persistent classes are always persistifiable,
using automated persistence.  Facade classes are always persistifiable, using
some combination of class-specified persistence and automated persistence
ranging from complete database mapping using JDBC and automatic state and cache
management, to all of these mechanisms specified explicitly by the class.

Persistent classes are represented in the meta-model as PBO's.
Facade classes are represented in the meta-model as FBO's.
Transient classes are represented in the meta-model as BO's.
EBT's are represented in the meta-model as EBT's.
Events are represented in the meta-model as Event's.

If inheritance is used, the inheritance implementation of the derived class can
be selected. That implementation is said to be the implementation of the
derived class. The implementation mechanism of root classes is always based on
Java inheritance, i.e., root classes are always implemented as Java
classes. Once another mechanism is introduced, the implementation of a further
derived class cannot be Java.
*/

<documentedClassElement> ::=  <daslComment>* <classElement>

<classElement> ::=  <classProperties> ;
               ::=  <memberDeclaration> <memberProperties>? ;
               ::=  <methodDeclaration>
               ::=  <constructorDeclaration>
               ::=  <keyDeclaration> <keyProperties>? ;
               ::=  <groupDeclaration> <groupProperties>? ;
               ::=  <classConstraintDeclaration> ;

<memberDeclaration> ::=  <realizedAttributeModifier>* <realizedAttribute>
                    ::=  <computedAttributeModifier>* <computedAttribute>
                    ::=  <constantAttributeModifier>* <constantAttribute>
                    ::=  <relationshipModifier>* <relationshipType>
                         <relationship>

/** Attributes */

/** NOTE: A <realizedAttribute> that isn't preceded by 'transient' or
 *        'persistent' is persistent if the class is persistent, and
 *        transient if the class is transient.  The keyword 'facade' is not
 *        not relevant to attributes because they are either persistent or
 *        not, and the difference between facade and persistent occurs at
 *        the class level, not at the individual attribute level.
 */

<realizedAttributeModifier> ::=  hidden
                            ::=  [ readonly | writeonly ]+
                            ::=  transient
                            ::=  persistent
                            ::=  factory

<realizedAttribute> ::=  <className> <attributeName> [ PRIMARY KEY | UNIQUE
                         KEY ] [ = <exprOrReturnBlock> ]
                         <attributeConstraintDeclaration>*

<computedAttributeModifier> ::=  hidden
                            ::=  factory

<computedAttribute> ::=  computed <className> <attributeName> =
                          <exprOrReturnBlock>

<constantAttributeModifier> ::=  hidden

<constantAttribute> ::=  constant <className> <attributeName> =
                          <constantExpression>

/** Relationships */

<relationshipModifier> ::=  hidden
                       ::=  [ readonly | writeonly ]+

<relationshipType> ::=  owns
                   ::=  references

<relationship> ::=  <className> <forwardName> ( <forwardCardinalityMin> ,
                    <forwardCardinalityMax> ) [ inverse <inverseName>? (
                    <inverseCardinalityMin> , <inverseCardinalityMax> ) ]
                    <relationshipConstraintDeclaration>*

<relationshipAccessorName> ::=  <forwardName>
                           ::=  <inverseName>

/** NOTE: In a <relationship>, forwardName is how you get from the
parent object (the one whose class the relationship is declared) to the child
object. The inverseName is how you get from the child to the parent. The
cardinality and kind of relationship are determined by the keyword, e.g., owns
means an owns relationship, and references means a references
relationship. (1,n) after the accessor means that the accessor has a minimum
cardinality of 1 and maximum greater than 1.

NOTE: The inverseName is optional. If it isn't given, direct navigation from
the child to the parent may not be possible without a query.

NOTE: The following relationship modifiers may be specified as properties, e.g.
          PROPERTIES { forwardOrderedBy }

   PropertyName             Meaning
   -----------------        ----------------------------------------------------
   forwardOrderedBy         The column on which the forward relationship is
                            ordered (primary key if no column is specified)
   inverseOrderedBy         The column on which the inverse relationship is
                            ordered (primary key if no column is specified)
   forwardUnique            The forward relationship is unique
   inverseUnique            The inverse relationship is unique
   forwardPrefetch          The forward relationship should be prefetched
   inversePrefetch          The inverse relationship should be prefetched

   (the following apply to bootstrapped models only)

   parentExternal           The parent is an external class
   childExternal            The child is an external class
*/

<forwardCardinalityMin> ::=  0
                        ::=  1

<inverseCardinalityMin> ::=  0
                        ::=  1

<forwardCardinalityMax> ::=  1
                        ::=  n
```

```
<inverseCardinalityMax> ::=  1
                        ::=  n

/** Methods */

<methodDeclaration> ::=  <methodModifier>* method <methodName> (
                         <methodParamDcls>? ) [ returns <bosType> ] [ throws
                         <exceptionList> ] <methodProperties>?
                         <methodImplementation>

<constructorDeclaration> ::=  <constructorModifier>* constructor <className> (
                              <methodParamDcls> ) [ throws <exceptionList> ]
                              <methodProperties>? <methodImplementation>

<methodModifier> ::=  hidden
                 ::=  factory

<constructorModifier> ::=  hidden

<methodParamDcls> ::=  <methodParamDcl> [ , <methodParamDcl> ]*

<methodParamDcl> ::=  <bosType> <parameterName>

<bosType> ::=  <className>
          ::=  <EBTName>
          ::=  Collection [ '<' <bosType> '>' ]
          ::=  List [ '<' <bosType> '>' ]
          ::=  Set [ '<' <bosType> '>' ]
          ::=  SortedSet [ '<' <bosType> '>' ]
          ::=  Map [ '<' <bosType> '>' ]
          ::=  <bosType> '[]'

<exceptionList> ::=  <qualifiedClassName> [ , <qualifiedClassName> ]*

<methodImplementation> ::=  { <daslMethodBody> }
                       ::=  ( <expression> )
                       ::=  DASL { <daslMethodBody> }
                       ::=  DASL ( <expression> )
                       ::=  OQL ( <OQLQuery> )
                       ::=  SQL ( <SQLQuery> )

<daslMethodBody> ::=  <daslMethodStatement>*

<exprOrReturnBlock> ::=  <expression>
                    ::=  { <typedMethodBody> }

<typedMethodBody> ::=  <daslMethodBody> return <expression> ;

<keyDeclaration> ::=  PRIMARY KEY ( <attrOrRelList> )
                 ::=  UNIQUE KEY ( <attrOrRelList> )

<groupDeclaration> ::=  GROUP <groupName> TYPE <groupType> ( <attrOrRelList> )

<groupType> ::=  LOCK
            ::=  INDEX
            ::=  PKEY
            ::=  UKEY
            ::=  <identifier>

<attrOrRelList> ::=  <attributeOrRelationship> [ , <attributeOrRelationship> ]*
```

```
<attributeOrRelationship> ::=  <attributeName>
                          ::=  <relationshipAccessorName>

<constraintDeclaration> ::=  CHECK <constraintName>? [ WHEN_COMMITTED |
                             WHEN_SET ] <constraintProperties>?
                             <constraintImplementation>

<classConstraintDeclaration> ::=  <constraintDeclaration>

<attributeConstraintDeclaration> ::=  <constraintDeclaration>

<relationshipConstraintDeclaration> ::=  <constraintDeclaration>

<constraintImplementation> ::=  { <typedMethodBody> }
                           ::=  ( <expression> )
                           ::=  DASL { <typedMethodBody> }
                           ::=  DASL ( <expression> )
                           ::=  OQL ( <OQLExpression> )
                           ::=  SQL ( <SQLExpression> )

/** End of BNF for the DASL/BOS Language (.bos file) */
```

AUS

```
/** Annotated BNF for the DASL/AUS Language (.aus file) */

// The root productions are (alphabetically):
//
//     <AUS>
// ___________________________________________

<AUS> ::=  <documentedAusElement>*

<documentedAusElement> ::=  <daslDocComment>* <ausElement>

<ausElement> ::=  <packageDeclaration>
             ::=  <ausProperties> ;
             ::=  <ausImport>
             ::=  <sessionVariableDeclaration>
             ::=  <sharedVariableDeclaration>
             ::=  <stateDeclaration>
             ::=  <transactionDeclaration>

<ausImport> ::=  <bosPackageImport>

<sessionVariableDeclaration> ::=  <variableDeclaration>

<sharedVariableDeclaration> ::=  shared <variableDeclaration>

<variableDeclaration> ::=  <normalDeclaration>
                      ::=  declare <qualifiedDeclaration>

<normalDeclaration> ::=  <ausType> <variableName> [ = <scriptExpression> ]
                         <variableProperties>? ;

<qualifiedDeclaration> ::=  <qualifiedClassName> <variableName> [ =
                            <scriptExpression> ] <variableProperties>? ;

<stateDeclaration> ::=  state <stateName> ( <stateParamDcls> ) <stateLabel>? {
                        <documentedStateElement>* }
                   ::=  initial state <stateLabel>? { <documentedStateElement>*
                        }
                   ::=  initial state <stateName> ( ) <stateLabel>? {
                        <documentedStateElement>* }

<stateParamDcls> ::=  <stateParamDcl> [ , <stateParamDcl> ]*

<stateParamDcl> ::=  <ausType> <parameterName>

<ausType> ::=  [ <localName> . ] <className>
          ::=  [ <localName> . ] <EBTName>
          ::=  Collection [ '<' <ausType> '>' ]
          ::=  List [ '<' <ausType> '>' ]
          ::=  Set [ '<' <ausType> '>' ]
          ::=  SortedSet [ '<' <ausType> '>' ]
          ::=  Map [ '<' <ausType> '>' ]
          ::=  <ausType> '[]'
```

```
<stateLabel> ::=  <stringLiteral>

<documentedStateElement> ::=  <daslDocComment>* <stateElement>

<stateElement> ::=  <stateProperties>
               ::=  <stateVariableDeclaration>
               ::=  <entryScript>
               ::=  <usageSpecification>
               ::=  <transition>
               ::=  <stateInclude>

<stateVariableDeclaration> ::=  <variableDeclaration>

<entryScript> ::=  entry <scriptBlock>

<usageSpecification> ::=  usage { <documentedVariableUsage> [ ,
                          <documentedVariableUsage> ]* }
                     ::=  usage { }

<transition> ::=  transition <transitionName> <transitionLabel>?
                  <transitionProperties>? <scriptBlock>

<transitionLabel> ::=  <stringLiteral>

<stateInclude> ::=  include <stateCall> [ when ( <booleanScriptExpression> ) ]
               ::=  include URL ( <stringLiteral> ) [ when (
                    <booleanScriptExpression> ) ]

<stateCall> ::=  <stateName> ( [ <scriptExpression> [ , <scriptExpression> ]* ]
                 )

/*****************************************************************
 *  ********* Blocks and Statements Within Scripts ****************
 ******************************************************************/

<scriptBlock> ::=  { <documentedScriptStatement>* }

<documentedScriptStatement> ::=  <daslDocComment>* <scriptStatement>

<scriptStmtOrBlock> ::=  <scriptStatement>
                    ::=  <scriptBlock>

<scriptStatement> ::=  <scriptProperties>
                  ::=  <transferStatement>
                  ::=  on error <exceptionRestriction>? <scriptStmtOrBlock>
                  ::=  on return <scriptStmtOrBlock>
                  ::=  <scriptVariableDeclaration>
                  ::=  if ( <booleanScriptExpression> ) <scriptStmtOrBlock> [
                       else <scriptStmtOrBlock> ]
                  ::=  CODE [ ( <language> ) ] { <codeBody> }
                  ::=  <scriptAssignmentStatement>
                  ::=  <scriptExpression>
                  ::=  throw <scriptExpression>

<transferStatement> ::=  goto <stateCall>
                    ::=  continue
                    ::=  exit [ to URL ( <stringLiteral> ) ]
                    ::=  switch ( <scriptExpression> ) { <scriptSwitchCase>*

<scriptSwitchCase> ::=  <switchLabel> <transferStatement>
```

```
<scriptVariableDeclaration> ::=  <variableDeclaration>

<language> ::=  <stringLiteral>

<codeBody> ::=  /** any sequence of characters with balanced {}'s */

<scriptAssignmentStatement> ::=  <scriptExpression> = <scriptExpression>

/***************************************************************************
 ************************** Variable Usage ******************************
 ***************************************************************************/

<documentedVariableUsage> ::=  <daslDocComment>* <variableUsage>

<variableUsage> ::=  <usageVariable> <usageLabel>? [ : <modes> ]
                     <variableUsageProperties>? [ { <memberUsage> } ]

<usageVariable> ::=  <variableName>
                ::=  '[' <collectionVariableName> ']'
                ::=  '[' <collectionVariableName> ( <selectCardinalityMin> ,
                     <selectCardinalityMax> ) ']'

<selectCardinalityMin> ::=  0
                       ::=  1

<selectCardinalityMax> ::=  0
                       ::=  1
                       ::=  n

<memberUsage> ::=  { <documentedVariableUsage> [ , <documentedVariableUsage> ]*
                   }
              ::=  { }

/** Mode letters must be placed together, with no intervening space.
 *  The order in which they are placed does not matter.
 */

<modes> ::=  <readMode> <writeMode> <consistentMode> <invisibleMode>

<readMode> ::=  R

<writeMode> ::=  W

<consistentMode> ::=  C

<invisibleMode> ::=  I

<usageLabel> ::=  <stringLiteral>

/***************************************************************************
 *************************** Transactions *******************************
 ***************************************************************************/

<transactionDeclaration> ::=  transaction <transactionName> {
                              <transactionProperties>? <stateOrNestedTx> [ ,
                              <stateOrNestedTx> ]* commits to <commitState> [
                              , <commitState> ]* }

<stateOrNestedTx> ::=  <stateName>
                  ::=  <transactionDeclaration>
```

```
<commitState> ::=  <stateName>
              ::=  exit

/***************************************************************************
 ************************ Script Expressions *****************************
 ***************************************************************************/

<scriptExpression> ::=  <literal>
                   ::=  <rootVariable> <componentSelector>*
                   ::=  ( <scriptExpression> )
                   ::=  <prefixOp> <scriptExpression>
                   ::=  <scriptExpression> <infixOp> <scriptExpression>
                   ::=  <usageVariable> . <selectionMethod>

<booleanScriptExpression> ::=  <scriptExpression>

<rootVariable> ::=  [ <localName> . ] <className>
               ::=  [ <localName> . ] <EBTName>
               ::=  <variableName>
               ::=  request

<componentSelector> ::=  . <componentName>
                    ::=  '[' <scriptExpression> [ , <scriptExpression> ]* ']'
                    ::=  ( [ <scriptExpression> [ , <scriptExpression> ]* ] )

<selectionMethod> ::=  getSelectedOne ( )
                  ::=  getSelectedMany ( )

/** End of BNF for the DASL/AUS Language (.aus file) */
```

Common

```
/** Annotated BNF for the Common part of the DASL Language. */

/** Common Expression Elements */

// The root productions are (alphabetically):
//
//     <attributeName>
//     <ausName>
//     <ausProperties>
//     <bosClassImport>
//     <bosPackageImport>
//     <bosProperties>
//     <classProperties>
//     <collectionVariableName>
//     <componentName>
//     <constraintName>
//     <constraintProperties>
//     <daslComment>
//     <EBTName>
//     <eventName>
//     <forwardName>
//     <groupName>
//     <groupProperties>
//     <inverseName>
//     <javaClassImport>
//     <javaPackageImport>
//     <keyProperties>
//     <memberProperties>
//     <methodName>
//     <methodProperties>
//     <name>
//     <packageDeclaration>
//     <scriptProperties>
//     <stateName>
//     <stateProperties>
//     <transactionName>
//     <transactionProperties>
//     <transitionName>
//     <transitionProperties>
//     <type>
//     <variableName>
//     <variableProperties>
//     <variableUsageProperties>
// __________________________________________

<packageDeclaration> ::=  package <packageName> ;

/**  NOTE: <packageName> is the desired name of the deployment package to
 *         generate, e.g.   package com.sun.xa.bo.memorymodel;
 *         Each BOS and each AUS must have its own unique package name.
 */

<packageName> ::=  <identifier> [ . <identifier> ]*

<bosPackageName> ::=  <packageName> . <bosName>
```

```
/** Comments */

<daslComment> ::=  <daslDocComment>
              ::=  <cStyleComment>
              ::=  <ephemeralComment>

<daslDocComment> ::=  '/**' <commentStringCharacter>* '*/'

<cStyleComment> ::=  '/*' <commentStringCharacter>* '*/'

<ephemeralComment> ::=  '//' <nonNewlineStringCharacter>*

/** Imports */

<bosClassImport> ::=  import <bosPackageName> . <className> [ as <localName> ]
                       [ UUID <uuidName> ] <importProperties>? ;

<bosPackageImport> ::=  import <bosPackageName> . * [ as <localName> . * ] [
                        UUID <uuidName> ] <importProperties>? ;

<javaClassImport> ::=  import java <packageName> . <javaClassName> [ as
                       <localName> ] <importProperties>? ;

<javaPackageImport> ::=  import java <packageName> . * [ as <localName> ]
                         <importProperties>? ;

/** Properties */

<properties> ::=  properties { <property> [ , <property> ]* }

<property> ::=  <propertyName> = <stringLiteral>
           ::=  <propertyName>

<bosProperties> ::=  <properties>

<ausProperties> ::=  <properties>

<classProperties> ::=  <properties>

<memberProperties> ::=  <properties>

<variableProperties> ::=  <properties>

<variableUsageProperties> ::=  <properties>

<methodProperties> ::=  <properties>

<keyProperties> ::=  <properties>

<groupProperties> ::=  <properties>

<constraintProperties> ::=  <properties>

<importProperties> ::=  <properties>

<stateProperties> ::=  <properties>

<scriptProperties> ::=  <properties>

<transitionProperties> ::=  <properties>
```

```
<transactionProperties> ::=  <properties>

/** Names */

<uuidName> ::=  <identifier>

<attributeName> ::=  <identifier>

<methodName> ::=  <identifier>

<forwardName> ::=  <identifier>

<inverseName> ::=  <identifier>

<bosName> ::=  <identifier>

<ausName> ::=  <identifier>

<className> ::=  <identifier>

<variableName> ::=  <identifier>

<collectionVariableName> ::=  <identifier>

<componentName> ::=  <identifier>

<name> ::=  <identifier>

<javaClassName> ::=  <identifier>

<EBTName> ::=  <identifier>

<eventName> ::=  <identifier>

<localName> ::=  <identifier>

<parameterName> ::=  <identifier>

<propertyName> ::=  <identifier>

<groupName> ::=  <identifier>

<constraintName> ::=  <identifier>

<stateName> ::=  <identifier>

<transitionName> ::=  <identifier>

<transactionName> ::=  <identifier>

/************************************************************************
 ************************* IDENTIFIERS *********************************
 ************************************************************************/

/** NOTE: Java and SQL reserved words may not be used as identifiers.
 *  The use of unreserved SQL keywords is discouraged but allowed.
 */

<identifier> ::=  <identifierStart> <identifierRest>*
```

```
<identifierStart> ::=  <alpha>
                  ::=  _
                  ::=  $

<identifierRest> ::=  <alpha>
                 ::=  _
                 ::=  <digit>

<alpha> ::=  A
        ::=  B
        ::=  C
        ::=  D
        ::=  E
        ::=  F
        ::=  G
        ::=  H
        ::=  I
        ::=  J
        ::=  K
        ::=  L
        ::=  M
        ::=  N
        ::=  O
        ::=  P
        ::=  Q
        ::=  R
        ::=  S
        ::=  T
        ::=  U
        ::=  V
        ::=  W
        ::=  X
        ::=  Y
        ::=  Z

<digit> ::=  0
        ::=  1
        ::=  2
        ::=  3
        ::=  4
        ::=  5
        ::=  6
        ::=  7
        ::=  8
        ::=  9

<qualifiedClassName> ::=  <className>
                     ::=  <packageName> . <className>

<qualifiedIdentifier> ::=  <identifier> [ . <identifier> ]*

<type> ::=  <bosType>
       ::=  <ausType>

/*******************************************************************
 *  ********* Blocks and Statements Within Methods *****************
 *******************************************************************/

<block> ::=  { <daslMethodStatement>* }
```

```
<daslMethodStatement> ::=  <localVariableDeclarationStatement>
                      ::=  [ <identifier> : ] <statement>

<localVariableDeclarationStatement> ::=  [ final ] <typeExpression>
                                         <variableDeclarators> ;

<statement> ::=  <block>
            ::=  if ( <booleanExpression> ) <statement> [ else <statement> ]
            ::=  <forLoop>
            ::=  while ( <booleanExpression> ) <statement>
            ::=  do <statement> while ( <booleanExpression> ) ;
            ::=  try <block> <catches>
            ::=  on error <exceptionRestriction>? <statement>
            ::=  on return <statement>
            ::=  switch ( <expression> ) { <switchCase>* }
            ::=  synchronized ( <expression> ) <block>
            ::=  return <expression>? ;
            ::=  throw <expression> ;
            ::=  break <identifier>? ;
            ::=  continue <identifier>? ;
            ::=  assert <expression> [ : <expression> ] ;
            ::=  ;
            ::=  <expressionStatement>
            ::=  <identifier> : <statement>

<forLoop> ::=  <arithmeticFor>
          ::=  <collectionFor>

<arithmeticFor> ::=  for ( <forInit> ; <expression>? ; <forUpdate> )
                     <statement>

<collectionFor> ::=  for ( <forInit> : <expression>? ) <collectionStatement>

<collectionStatement> ::=  <statement>
                      ::=  <removeStatement>
                      ::=  <insertStatement>

<removeStatement> ::=  <remove> <identifier>

<remove> ::=  remove
         ::=  $remove

<insertStatement> ::=  <insert> <identifier>

<insert> ::=  insert
         ::=  $insert

<catches> ::=  <catchClause> <catches>
          ::=  <finallyPart>

<catchClause> ::=  catch <exceptionRestriction> <block>

<finallyPart> ::=  finally <block>
              ::=

<exceptionRestriction> ::=  ( <qualifiedClassName> <parameterName> )

<switchCase> ::=  <switchLabel> <daslMethodStatement>*

<switchLabel> ::=  case <constantExpression> :
              ::=  default :
```

```
<forInit> ::=  <statementExpression> <moreStatementExpressions>
          ::=  [ final ] <typeExpression> <variableDeclarators>

<moreStatementExpressions> ::=  [ , <statementExpression> ]*

<forUpdate> ::=  <statementExpression> <moreStatementExpressions>

<variableDeclarators> ::=  <variableDeclarator> [ , <variableDeclarator> ]*

<variableDeclarator> ::=  <identifier> <variableDeclaratorRest>

<variableDeclaratorRest> ::=  <arrayBrackets>? [ = <variableInitializer> ]

/*******************************************************************
 *********************** Literals **********************************
 *******************************************************************/

<literal> ::=  <integerLiteral>
          ::=  <floatingPointLiteral>
          ::=  <booleanLiteral>
          ::=  <characterLiteral>
          ::=  <stringLiteral>
          ::=  <nullLiteral>

<integerLiteral> ::=  <decimalIntegerLiteral>
                 ::=  <hexIntegerLiteral>
                 ::=  <octalIntegerLiteral>

<decimalIntegerLiteral> ::=  <decimalNumeral> <integerTypeSuffix>*

<hexIntegerLiteral> ::=  <hexNumeral> <integerTypeSuffix>

<octalIntegerLiteral> ::=  <octalNumeral> <integerTypeSuffix>

<integerTypeSuffix> ::=  u
                    ::=  L

<decimalNumeral> ::=  <digit>+

<hexNumeral> ::=  0 x <hexDigit>+
             ::=  0 X <hexDigit>+

<hexDigit> ::=  <digit>
           ::=  a
           ::=  b
           ::=  c
           ::=  d
           ::=  e
           ::=  f
           ::=  A
           ::=  B
           ::=  C
           ::=  D
           ::=  E
           ::=  F

<octalNumeral> ::=  0 <octalDigit>*

<octalDigit> ::=  0
             ::=  1
```

```
               ::=  2
               ::=  3
               ::=  4
               ::=  5
               ::=  6
               ::=  7

<floatingPointLiteral> ::=  <digit>+ . <digit>+ <exponentPart>
                            <floatTypeSuffix>
                       ::=  . <digit>+ <exponentPart> <floatTypeSuffix>
                       ::=  <digit>+ <exponentPart> <floatTypeSuffix>
                       ::=  <digit>+ <exponentPart> <floatTypeSuffix>

<exponentPart> ::=  <exponentIndicator> <signedInteger>

<exponentIndicator> ::=  e
                    ::=  E

<signedInteger> ::=  <sign> <digit>+

<sign> ::=  +
       ::=  -

<floatTypeSuffix> ::=  f
                  ::=  F
                  ::=  d
                  ::=  D

<booleanLiteral> ::=  true
                 ::=  false

<characterLiteral> ::=  ' <unicodeCharacter> '
                   ::=  ' <stringLiteralEscape> '

<stringLiteralEscape> ::=  \ <hexNumeral>
                      ::=  \ <stringCharacter>

<stringLiteral> ::=  " <stringCharacter>* "

<stringCharacter> ::=  /** character except double quote and backslash */
                  ::=  <stringLiteralEscape>

<unicodeCharacter> ::=  /** any UNICODE character */

<commentStringCharacter> ::=  /** character except '*' '/' */

<nonNewlineStringCharacter> ::=  /** character except newline */

<nullLiteral> ::=  null

/****************************************************************************
 ****************************  Expressions **********************************
 ****************************************************************************/

<expressionStatement> ::=  <expression>

<expression> ::=  <expression1> <expressionRest>?

<booleanExpression> ::=  <expression>

<expressionRest> ::=  [ <assignmentOperator> <expression1> ]
```

```
<assignmentOperator> ::=  =
                     ::=  + =
                     ::=  - =
                     ::=  * =
                     ::=  / =
                     ::=  & =
                     ::=  '|='
                     ::=  ^ =
                     ::=  % =
                     ::=  '<<='
                     ::=  '>>='
                     ::=  '>>>='

<typeExpression> ::=  <identifier> [ . <identifier> ]* <elementType>?
                      <typeSelector>* <arrayBrackets>?
                 ::=  <primitiveType>

<simpleTypeExpression> ::=  <identifier> [ . <identifier> ]* <arrayBrackets>?

<statementExpression> ::=  <expression>

<constantExpression> ::=  <expression>

<expression1> ::=  <expression2> <expression1Rest>?

<expression1Rest> ::=  [ ? <expression> : <expression1> ]

<expression2> ::=  <expression3> <expression2Rest>?

<expression2Rest> ::=  [ <infixOp> <expression3> ]*
                  ::=  <expression3> instanceof <typeExpression>

<infixOp> ::=  '||'
          ::=  & &
          ::=  '|'
          ::=  ^
          ::=  &
          ::=  = =
          ::=  ! =
          ::=  '<'
          ::=  '>'
          ::=  '<='
          ::=  '>='
          ::=  '<<'
          ::=  '>>'
          ::=  '>>>'
          ::=  +
          ::=  -
          ::=  *
          ::=  /
          ::=  %

<expression3> ::=  <prefixOp> <expression3>
              ::=  <expression> <expression3>
              ::=  <simpleTypeExpression> <expression3>
              ::=  <primary> <selector>* <postfixOp>*

<primary> ::=  ( <expression> )
          ::=  this <arguments>?
          ::=  super <superSuffix>
          ::=  <literal>
          ::=  new [ persistent ] <arrayCreator>
          ::=  <identifier> [ . <identifier> ]* <subscriptExpr>?
          ::=  <primitiveType> <arrayBrackets>? . Class

<subscriptExpr> ::=  '[' <arrayBrackets>? . class ']'
                ::=  '[' <expression> ']'
                ::=  '[' <arguments> ']'
                ::=  '[]' . <subscriptPseudoField>

<subscriptPseudoField> ::=  class
                       ::=  this
                       ::=  super <arguments>
                       ::=  new <innerArrayCreator>

<prefixOp> ::=  + +
           ::=  - -
           ::=  !
           ::=  ~
           ::=  +
           ::=  -

<postfixOp> ::=  + +
            ::=  - -

<selector> ::=  . <identifier> <arguments>?
           ::=  . this
           ::=  . super <superSuffix>
           ::=  . new [ persistent ] <innerArrayCreator>
           ::=  '[' <expression> ']'

<typeSelector> ::=  . <identifier> <elementType>?

<superSuffix> ::=  <arguments>
              ::=  . <identifier> <arguments>?

<primitiveType> ::=  byte
                ::=  short
                ::=  char
                ::=  int
                ::=  long
                ::=  float
                ::=  double
                ::=  boolean

<arguments> ::=  ( [ <expression> [ , <expression> ]* ] )

<elementType> ::=  '<' <typeExpression> [ , <typeExpression> ]* '>'

<arrayBrackets> ::=  '[]'

<arrayCreator> ::=  <qualifiedIdentifier> <elementType>? <arrayCreatorRest>
               ::=  <qualifiedIdentifier> <elementType>? <arguments>

<innerArrayCreator> ::=  <identifier> <elementType>? <arguments>

<arrayCreatorRest> ::=  '[' ']' <arrayBrackets>? <arrayInitializer>
                   ::=  '[' <expression> ']' [ '[' <expression> ']' ]*
                        <arrayBrackets>?
```

```
<arrayInitializer> ::=  { [ <variableInitializer> [ , <variableInitializer> ]* ]
                          [ , ] }

<variableInitializer> ::=  <arrayInitializer>
                      ::=  <expression>

/** End of BNF for the Common Elements of DASL */
```

OQL

```
/** Annotated BNF for the DASL/OQL Query Language */

/* NOTE:
 * The syntax is presented in what we consider the canonical or standard form.
 * We allow variations of this form, for compatibility with other query
 * languages such as EJBQL, JDOQL, and SQL.
 *
 * SQL queries undergo translation from BOS member names to the underlying
 * table column names.  Eventually, navigational chains such as e.manager.id
 * will also be translated in SQL queries, but currently they are not, so you
 * need to do the appropriate joins.
 *
 */

// The root productions are (alphabetically):
//
//     <OQLExpression>
//     <OQLQuery>
//     <SQLExpression>
//     <SQLQuery>
// _______________________________________________

<SQLExpression> ::=  /** SQL Expression */

<SQLQuery> ::=  /** SQL Query */

<OQLQuery> ::=  <selectClause> <fromClause> <whereClause>?

<OQLExpression> ::=  <valueExpression>

<selectClause> ::=  SELECT [ DISTINCT ] <singleValuedExpression>

<singleValuedExpression> ::=  <rangeVariable>
                         ::=  <identificationVariable>
                         ::=  <queryParameter>

<fromClause> ::=  FROM <sourceCollection> [ , <sourceCollection> ]*

<sourceCollection> ::=  <businessObjectClass> [ AS ] <rangeVariable>
                   ::=  IN ( <rangeVariable> . <relationshipAccessorName> ) [
                        AS ] <identificationVariable>

<whereClause> ::=  WHERE <conditionalExpression>

<conditionalExpression> ::=  <conjunctiveExpression> [ OR
                             <conjunctiveExpression> ]
                        ::=  ( <conditionalExpression> )

<conjunctiveExpression> ::=  <comparisonExpression> [ AND
                             <comparisonExpression> ]
                        ::=  ( <conjunctiveExpression> )

<comparisonExpression> ::=  <valueExpression> = <valueExpression>
                       ::=  <valueExpression> '<' <valueExpression>
```

```
                        ::=  <valueExpression> '<=' <valueExpression>
                        ::=  <valueExpression> '>' <valueExpression>
                        ::=  <valueExpression> '>=' <valueExpression>
                        ::=  <valueExpression> '<>' <valueExpression>
                        ::=  <valueExpression> BETWEEN <valueExpression> AND
                             <valueExpression>
                        ::=  <valueExpression> LIKE <patternExpression> [
                             ESCAPE <escapeCharacter> ]
                        ::=  <valueExpression> NOT LIKE <patternExpression> [
                             ESCAPE <escapeCharacter> ]
                        ::=  <valueExpression> [ NOT ] IN ( <valueExpression> [
                             , <valueExpression> ]* )
                        ::=  <valueExpression> IS NULL
                        ::=  <valueExpression> IS NOT NULL

<valueExpression> ::=  <singleValuedExpression> . <attributeName>
                  ::=  <valueExpression> . <attributeName>

<patternExpression> ::=  <OQLstringLiteral>
                    ::=  <queryParameter>

<OQLstringLiteral> ::=  ' <OQLstringCharacter>* '

<OQLstringCharacter> ::=  /** character except double quote and backslash */

<escapeCharacter> ::=  <characterLiteral>

<queryParameter> ::=  : <parameterName>
                 ::=  ? <parameterName>

/*
 * NOTE: The above syntax does not allow infix numeric operators such as +, -,
 * *, or /.  Such operators are passed to the implementation, so if they are
 * supported they will work.  Currently, the EJBQL specification severely
 * limits the use of infix operators, but that may improve with time.
 */

<businessObjectClass> ::=  <className>

<rangeVariable> ::=  <identifier>

<identificationVariable> ::=  <identifier>

/** End of BNF for the DASL/OQL Query Language */
```

APPENDIX C: Known Limitations and Workarounds

This appendix will rapidly become out-of-date, but is presented to help the aspiring DASL programmer understand some of the limitations of the current implementation and how to work around them.

i. "Gaps" in Automatic Primary Key Assignment

Most implementations of automatic primary key assignment will leave "gaps" or unused keys when a transaction creates an object and then rolls back the creation by not committing. It is difficult to avoid such gaps unless the application goes to the trouble of remembering them and reassigning them later. The underlying reason is that there is a tradeoff between the desire to prevent gaps and scalability, i.e., the ability for more than one transaction (user session) to be allowed to create an object at the same time.

If you must prevent gaps, you will need to implement your own service that assigns and tracks primary keys or ids for objects. This can be implemented as a method in an object in a BOS, or externally in a Java class. However, it is usually best to write your application assuming that gaps may occur, and thus not rely on every id being used.

ii. Query Methods Cannot Access Class Members

Currently, class members are not available to be used as parameters to the query in query methods. To work around the problem, create a procedural method (e.g., *expensiveItems* below) that calls the query method (*doActualQuery* below), passing it the class variable:

```
method expensiveItems( Decimal price ) returns Collection<LineItem>
{
  doActualQuery(price,id)
}

method doActualQuery( Decimal price, Long id )
SQL (SELECT l FROM LineItems l WHERE l.id = :id AND l.unitPrice > :price)
```

iii. Attribute Null Values are Not Handled Well for Java Primitive Types

When used as attribute types, the following types are mapped into the corresponding Java primitive types, which have no representation for null (unknown) values:

- Boolean
- Short
- Integer
- Long
- Float
- Double

For example, a *Long* attribute in a business object becomes a *long* attribute in the corresponding Java object. There is consequently no way to specify that the attribute's value is null (i.e., unknown). When such an object is created, DASL always sets this attribute to zero, even if no value has been set by the application.

The application designer, and in particular, the author of the BOS for the application, must work around this shortcoming for now. For example, an out-of-range value (such as 0 or -2) can be used to signal null. A related future enhancement to DASL is proposed in a later appendix.

iv. Cannot Import Java Utility Classes Explicitly into the AUS

Importing of Java classes into the BOS is supported, but in the AUS it is not supported. However, at times a Java class is needed in a declaration or expression. Until the AUS import statement supports Java classes, there are several possible workarounds:

1. Put a Java import statement into a BOS that this AUS imports. That Java class is then available in expressions, e.g.

   In the BOS: *import java java.util.ArrayList;*

   In the AUS: *List<String> strings = new ArrayList<String>()*;

2. For declaring variables, the *declare* keyword can be used to declare a fully qualified Java class, e.g., *declare java.util.ArrayList a;*

3. Use the fully qualified name of the class. For example, in an *on error* or *throw* statement, either using suggestion 1 above, or use the fully name of the exception class:

```
on error (com.sun.ace.runtime.common.NoSuchPrimaryKeyException e) {
    throw new com.sun.ace.runtime.common.DaslAppRuntimeException(
       e.getMessage());
}
```

v. Usage Variables Cannot Be Collection<EBT>

The client deployement engines currently do not support usage variables in the AUS that are collections of an EBT type, e.g., List<String>. You can get around this limitation in several ways.

1. For List<String> in particular, allow entry of a string with multiple entries, and convert the string to the desired list of Strings during the transition. The Mail application in the appendix does this for recipients of the email.

2. Create a transient class in the BOS that contains the desired type, e.g., class MyString { String s; } and use a List<MyString> in the AUS.

Note that collections of String can be used in the chooseFrom in a usage, but not as the usage variable itself.

vi. Issues Due to Non-Transparent Boxing and Unboxing of Primitive Types

The current deployment generators are based on Java 1.4 and thus do not always handle conversions between primitive types and class-based types transparently, e.g., *int* <=> *Integer*. Because DASL does not distinguish these as separate types, variables you declare in a DASL program outside of a method body as either the primitive or the class will be deployed as the deployment engine chooses, typically as primitive types. Such variables include class attributes, method return types, method formal parameters, state formal parameters, and variables declared in the AUS at all lexical levels (shared, session, state, script, etc.).

If boxing and unboxing worked transparently, the programmer would not have to know how the deployment engine chose to implement such types. But until the deployment generators are updated to Java 5.0, certain cases are not handled transparently. Specifically, the following constructs will fail:

- boxing method invocations on primitives, e.g.: *int i; i.toString();*
- boxing of method parameters: *method foo(int i) {...} foo(new Integer(i));*

The results may be especially confusing because the DASL source may contain what looks like a class-based variable declaration (e.g., *Integer*), but the deployed code uses a primitive type (e.g., *int*). If you get deployment compilation errors with the constructs above, changing your source from int to Integer will not fix the problem. Instead, change the implementation to assume that the variables are the primitive types, e.g.

- *Integer i; i.toString(); ==> new Integer(i).toString();*
- *method foo(Integer i) {...} foo(new Integer(3)); ==> foo(3);*

vii. The Scripting Language (AUS) Is Incomplete

Certain Java statements and expressions are not yet supported in the scripting language of the AUS. As of this writing, strongly typed constructors and list subscripting are not supported. The CODE escape mechanism can be used to work around such limitations by forcing DASL to pass statements into the deployed Java code. For example, a strongly typed list can be constructed and subscripted in an AUS script as follows:

```
List<Customer> clist;
Customer c;
CODE {
        clist = new ArrayList<Customer>(custs);
        c = (Customer) clist[0];
}
```

viii. Java Code Is Not Deployed Automatically

Sometimes, DASL expressions call Java code that is written as part of an application. In such cases, the Java source code is not automatically deployed during the build process. The workaround is that the user must deploy the DASL application, get errors when the Java classes are not found, and then manually copy the Java source files that were written as part of the appliction into the generated directory, in the appropriate directories (folders). After the Java sources have been placed, the user must redeploy the application. The newly placed Java sources will be compiled and packaged into a *jar* file during the second deployment operation.

APPENDIX D: Production Deployment Of Applications

This appendix gives a brief overview of how to deploy applications that have been generated using the current (as of this writing) 2-tier and 3-tier deployment engines. The information in this appendix is subject to becoming out-of-date very quickly as the deployment engines are improved and new ones are developed. However, this information is currently required to deploy a DASL application on a system other than the one the developer used, and thus is crucial for creating production applications until there are DASL tools that automate remote deployment.

1 Default User and Password For 2-Tier (Tomcat)

- default user: **admin**
- default password: **admin123**
- default port: **8080**

To deploy to another machine, on the 3rd page of the deployment GUI, specify installation on the desired host and port, e.g., dasl.sfbay:8080.

2 Default User and Password For 3-Tier (SunOne App Server)

- default user: **admin**
- default password: **adminadmin**

3 Creating a Pointbase Database (e.g., for remote deployment)

The default deployment creates a local Pointbase user name and a local Pointbase schema that are the application name in all lower case. It then creates the database tables in the schema for the app automatically.

To deploy the app somewhere else, you need to set up the database user and schema and then run the DDL file for the app to create the database tables, and the populate script to populate the database tables.

To set up a remote Pointbase database, run the Pointbase GUI, which is usually in $IAS_HOME/pointbase/client_tools/PB_console.sh or something similar for your operating system. You have to first connect as pbpublic and create the user and schema, both of which are the name of the application in all lower case. You then connect as the user (app name).

- Start Pointbase console e.g.: **$IAS_HOME/pointbase/client_tools/PB_Console.sh**

- Create user and schema for the application (if necessary) by connecting to the desired remote machine as PBPUBLIC

```
Driver:  com.pointbase.jdbc.jdbcUniversalDriver
URL: jdbc:pointbase://dasl.sfbay:9092/ace
User: PBPUBLIC
Password: pbpublic


DBA->create user <appname>
password <appname>

DBA->Create a schema named <appname>

DBA->disconnect from database
DBA->connect to database
```

- Create tables by connecting to the desired remote machine as <appname>:

```
Driver:  com.pointbase.jdbc.jdbcUniversalDriver
URL: jdbc:pointbase://dasl.sfbay:9092/ace
User: <appname>
Password: <appname>

Run the <appname>.sql script to create the tables.
Populate tables with data as required.
```

4 Simple Way to Deploy a DASL Application Remotely (2-Tier Tomcat)

```
Point your browser at URL:  http://remotehost:8080

Click on "Tomcat Manager"

        User Name:     admin
        Password:      admin123

If application is in the list displayed, click to stop it and remove it.

At bottom of screen, click:  Upload WAR file <Browse>

The file you want to install an be found relative to the directory in which
your DASL project (.dasl file) was built on your local machine.  If this
directory is <lclDir>, then the file you need to upload is:

     <lclDir>/generated/<appName>/<architecture>/build/deploy/<appName>.war

If this file is not reachable directly by some path from the remote host,
you will need to get it there somehow and then navigate to the place you
put it on remotehost.

Press <Install> in Tomcat Manager.


If database schema changed since the last time this application was
installed on remotehost, or if the database has not been created,
see "Creating a Pointbase Database" above.

Start the application in the Tomcat Manager.
```

APPENDIX E: Miscellaneous Proposed Extensions

a. Generalize Switch Statements in Scripts

The only form of switch statement currently allowed in a script is one whose cases are transition statements. This enhancement adds full switch statement syntax to the scripting language.

```
<conditionalStatement> ::= <ifStatement>[73]
                       ::= <switchStatement>

<switchStatement> ::= switch ( <expression> ) { <cases> [ <defaultCase> ] }

<cases> ::= <case>*

<case> ::= case <literal> : <scriptStatement>* break;

<defaultCase> ::= default: <scriptStatement>*
```

b. Loops in Scripts

This enhancement adds loop constructs, so that collections may be iterated. The *foreach* loop is similar to that in Java 5.0, with the addition of a *remove* statement that removes the current element from the collection. A for statement as existed in pre-5.0 Java could also be added for completeness.

```
<loopStatement> ::= <foreachLoop>
                ::= <whileLoop>

<foreachLoop> ::= for ( <type> <forIterationVariable> : <collection>
                    <foreachScriptBlock>

<forScriptBlock> ::= <foreachStatement>
                 ::= { <foreachStatement>* }

<foreachStatement> ::= <scriptStatement>
                   ::= remove <forIterationVariable> ;
                   ::= $remove <forIterationVariable> ;

<whileLoop> ::= while ( <booleanExpression> ) <scriptStmtOrBlock>
```

c. Allow <usageVariable>.setSelected() in Scripts

Some applications may wish to set the default selected elements of a <usageVariable>, so that those elements come up in the browser already selected, and thus a subsequent call to getSelectedOne() or getSelectedMany() with no further user action will return the default selected elements. The setSelected() pseudo-method may be called in an entry or transition script to achieve this result.

73 <ifStatement> is already implemented as a script statement.

If c is a collection-valued <usageVariable>, then

- c.setSelected(int) – sets the selected element to the element whose index is the specified int. This form is valid only for ordered collections. Any currently selected elements are deselected.
- c.setSelected(Object) - sets the selected element to the element whose value is the specified object. An error occurs if the maximum selection cardinality is one and the specified object occurs more than once in c. Any currently selected elements are deselected.
- c.setSelected(Collection) – sets all the objects contained in the collection as the selected elements. An error occurs if the maximum selection cardinality is one and the collection contains more than one element. Any currently selected elements are deselected.

d. User-Defined Methods as Part of the AUS

Allow the user to define methods as part of the AUS, using the same syntax as in the BOS. The methods could be defined at several scopes:

- session scope (outside of a state) – method getDesiredOrders() in the example below.
- state scope (inside a state) -- method addWild() in the example below:

```
state QueryByExample( String user, PurchaseOrders pos ) {
    ...

    method addWild( String s ) { return “%” + s + “%”; }

    transition find {
        pos = getDesiredOrders(user,addWild(keyword),total);
        goto QueryByExample(user,pos);
    }
}


method getDesiredOrders(String user, String keyword, Decimal total)
    returns Collection<Order>
OQL ( SELECT Order o
      FROM Order
      WHERE customer.name = :user
          AND description LIKE :keyword
          AND orderTotal >= :total
    )
```

`Factory methods would not be allowed.`

e. Better Handling of NULL Values for Boolean and Numeric Attributes

This enhancement applies both to the BOS and the AUS.

When used as attribute types, the following types are mapped into either the corresponding Java primitive types, which have no representation for null (unknown) values, or into the corresponding Java class types, which do have a representation for null, depending on whether the attribute is declared to allow a null value:

- Boolean
- Short
- Integer
- Long
- Float
- Double

If the attribute is declared so it cannot have null values, e.g., via CHECK (foo != null) or CHECK SQL (not null)

- The attribute is declared in the object as the corresponding Java primitive type.
- The attribute is initialized to zero or false.

If the attribute is declared so it is allowed to have null values

- The attribute is declared in the object using the Java class type, rather than the primitive type.
- The attribute is initialized to null by default.
- Two set methods are provided, e.g., setX(long) and setX(Long).
- The get method returns the Java class type: Long getX().
- Null values of the attribute are mapped to DBMS NULL, and vice versa.

ISSUES:

1. Do the implementations of JDO or EJB CMP handle these conventions? Regardless, these are the "official" conventions which the BOS and AUS sub-languages define.
2. DASL will make the use of these fields fairly transparent to the programmer. If the user codes:

```
long x = foo.x;
```

then a runtime exception should occur if foo.x is null. If the user codes:

```
    foo.x = x;
```

then there is no problem because foo.setX(x) handles the case that x is being changed from null to non-null. If the user codes:

```
Long y;
foo.x = y;
```

there is similarly no problem, as both foo.x and y have type *Long*.

f. Allow Import of Java Classes into the AUS

Currently, Java classes must be imported into a BOS, and that BOS imported into an AUS, for the Java class to become available in the AUS. As a convenience, allow Java classes to be imported directly into the AUS.

g. Allow Import of Entire BOSes and Java Packages into the BOS

Currently, business objects in other BOSes must be imported into a BOS one at a time. As a convenience, allow an entire BOS to be imported into the BOS.

APPENDIX F: Proposed Asynchronous Events

In DASL/AUS, the transition is typically driven by user gestures (e.g., clicking a button) in a GUI interface. The purpose of DASL events is to allow asynchronous messages to be sent and processed between applications, including non-interactive service applications that have no GUI interface.

DASL defines events at a higher level than Java, so that DASL events can be implemented using any of several Java mechanisms. For example, JMS (Java messaging service) could be used, or the Swing event mechanism could be used.

The proposed DASL event mechanism allows transitions to be driven by external events.

The event mechanism described below is preliminary and not yet final. Events are declared, much like BOS classes, and can be handled to cause transitions in AUS states. Each event received may have attributes; the transition script associated with an event may use those attributes as part of its business logic.

Your suggestions concerning high level events are welcome. Keep in mind that the goal is simplicity and powerfulness of declarative specification, rather than completeness. Existing event mechanisms should be thought of as the “assembly language” into which DASL events are “compiled.”

a. Declaring, Causing, and Handling Events

Events are class-like objects that are declared within a BOS, but unlike business objects and EBTs, they have their own special semantics. Events are declared in the BOS, similarly to classes and EBTs:

```
event <eventName> [ extends [ <packageName> . ] <eventName> ] {
        <classElement>*
}
```

Events may be imported into other BOS or AUS files. Their full name includes the BOS package name.

Events are *caused* within a BOS method or AUS transition script or method using the **cause** statement:

```
cause new <eventName> ( <eventConstructorArgs> )
```

This statement is similar to a throw statement, except that the event is propagated to any running AUS application that handles this particular event, not necessarily just the current process.

Events may be handled within the AUS in 3 scopes: session, state, or script:

```
package foo.aus;

import foo.bos.Foo.*;

on event ( <eventName> <eventVariable> ) {
```

```
                <scriptStatement>*
        } // session-scope event handling

        state Display() {
                on event ( <eventName> <eventVariable> ) {
                        <scriptStatement>*
                } // state-scope event handling

                transition T {
                        on event ( <eventName> <eventVariable> ) {
                                <scriptStatement>*
                        } // transition-scope event handling
                        ...
                }
        }
```

Like an **on error** or **on return** statement, the lexical presence of an **on event** statement causes that handler to be invoked to handle the event within its scope. Events that are not caught by a handler somewhere up the lexical scope are ignored.

When an AUS application starts, it registers as a consumer of all its event classes. If it contains a generic event handler, i.e., an **on event (Event e)** statement, then it registers for all events.

An AUS that has no transitions is deployed as a service application that is always running. Such applications may either be session-based, in which a separate identified session is started for each client and the session keeps state, or as sessionless. These two kinds of applications might be implemented under EJB using stateful and stateless session beans, for example.

b. Example Use Of Events

The following example shows how events might be used to notify every interactive user of an application when any of them post a message. Essentially, this is an instant messaging application controlled by an IM service.

The BOS (IM.bos):

```
        package im.bos;

        event SendMessage {
                String t;
                constructor SendMessage( String t ) {
                        this.t = t;
                }
        }

        event ReceiveMessage {
                String t;
                constructor ReceiveMessage( String t ) {
                        this.t = t;
                }
        }
```

The AUS service application (IMSRV.aus):

```
package imserver.aus;
import im.bos.IM;

on event ( SendMessage sm ) {
        cause new ReceiveMessage(sm.t);
}
```

The AUS interactive application (IM.aus):

```
package imclient.aus;
import im.bos.IM;

initial state {
        entry { goto Display(“”); }
}

state Display( String msg ) {
        on event ( ReceiveMessage rm ) {
                goto Display(rm.t);
        }
        String notification = “”;
        usage {
                msg “” :R,
                notification :W
        }
        transition Send {
                if (notification.length() > 0)
                        cause new SendMessage(notification);
                continue;
        }
        transition Exit {
                exit;
        }
}
```

Based solely on the placement of the **on event** declarations and **cause** statements, and the absence of interactive transitions in the service application, DASL implements the above application as follows:

- The interactive application starts in the Display state with a blank message.
- If the user enters some text in the notification field and presses the *Send* button, the *SendMessage* message is sent to all applications that have an *on event* handler, i.e., only the service application (server) receives the message.
- If either the current user or some other user sends a message, the service application handles it by causing the event *ReceiveMessage*. This event is sent to all applications that handle it, i.e., all instances of the interactive AUS application shown above. If the application receives the *ReceiveMessage* while in the *Display* state, it re-enters the *Display* state with the new new message displayed.
- The effect is that any of the current interactive users can “post” a new message, and all interactive users see it.

Exercise 1: Modify the above application so that the old messages are retained, and the newly posted message is appended to the end of the combined message. Hint: To ensure that the message is not lost due to a service application crash, declare a persistent class in im.bos and create or reuse an instance of that class to hold the combined message.

APPENDIX G: Proposed Composition of Tasks

Currently, the AUS sub-language specifies a business task as a single set of states with transitions between those states. In programming terms, the control flow between states is a [conditional] branch from the current state to another state (possibly the same state), as opposed to, say, a stack-oriented procedure call mechanism.

The purpose of these enhancements is to enable composition of existing business tasks to form new ones, analogous to the way that procedural programming languages provide methods which can be composed to create other methods. Composition also addresses issues of scaling, reuse and sharing of application business tasks.

This proposal generalizes the AUS sub-language to meet the following goals:

1. Allow business tasks to be composed from existing business tasks.
2. Allow a business task to "scale" to many states without requiring the entire task to be viewed as a single diagram.
3. Provide sharing of common tasks, such as handling of login and account creation, among different applications.
4. Allow “choosers” to be defined that allow navigation capabilities among a set of states, possibly including a recursive type of browsing as in a file chooser. Such choosers can be used for setting relationships between objects, among other things.
5. Allow creation of independent threads of user interaction, for example when a user brings up a series of help screens.

a. Task Blocks

An explicit **task** block is required to enclose the states and transactions of a business task. In the case of tasks that return values, the **task** block includes the parameters and return type of the task.

```
task <taskName> <optionalParameters> <optionalReturnType>
{
    <statesAndTransitions>
}
```

Under this proposal, every AUS becomes a task. Existing AUSes become tasks without parameters or return types. An example of creating top-level AUS tasks, and reusable task units, is shown below:

Example of Old Top-Level AUS	Proposed New Top-Level AUS	Proposed New Typed Task With Return Type

package com.sun.nametool;
 	
import com.sun.hr;
 	
initial state {
    transition T {
        goto S2(...);
    }
}
 	
/* Must currently put all
states and transactions
associated with Login here.
*/
 	
state S3(...) {
    transition done {
        exit;
    }
}
 	
transaction T { ... }
// end NameTool
	package com.sun.nametool;
	
	task NameTool {
	
	  import bos com.sun.hr;
	  import aus com.sun.common.Login;
	
	  initial state {
	    transition T {
	        goto S2(...);
	    }
	  }
	
	  state S2(...) {
	    Employee e =
	        call Login("NameTool");
	    if (e == null) **goto** S2(...);
	    goto S3(e);
	  }
	
	  state S3(Employee e) {
	    transition done {
	        exit;
	    }
	  }
	
	  transaction T { ... }
	
	} // end NameTool
		package com.sun.common;
		
		task Login(String app)
		               returns Employee
		{
		  import bos com.sun.hr;
		
		  initial state {
		    transition T {
		        goto S2(...);
		    }
		  }
		
		  state S2(...) {
		    Employee e;
		    transition done {
		        return e;
		    }
		  }
		
		  transaction T { ... }
		
		} // end Login

b. Tasks

The purpose of *tasks* is to define reusable business task sequences consisting of a series of states, that return a predefined set of values to the caller business task. A task is thus akin to a method in a programming language, in that it executes a series of actions and then returns values to its caller. Like methods, tasks may be invoked from more than one state in an application, and from more than one application.

The table above shows an example of the proposed new syntax for top-level AUSes, and an example of the syntax for defining and invoking (sub)tasks designed to be called. As before, a task is represented in an AUS file, and an AUS without parameters starts executing at the initial state of its main task. As before, transitions may be defined to other states in the main task. In addition, a task my be invoked as a subtask, passing in parameters to the subtask, and the task thus invoked may return values to the calling task. Tasks invoked as subtasks start execution in their initial state, and return to the calling task when the *return* statement is executed in a script.

For syntactic uniformity and consistency, and to make the name and optional parameters of the task explicit, the new syntax encloses the states of a task in a named task block. The task name must match the name of the AUS file in which it is stored.

In the example in the table above, there is a task called "Login" (in the third column) that takes as a parameter the name of an application, and that returns an Employee object. This task is invoked from state S2 in the NameTool task (in the second column).

The parameters to a task have the same scope and lifetimes as session variables.

The invocation of a task is made from a task state in the calling application. The task state may be transitioned to in the same way as any state, and it may define a transition to any other state(s). However, the content of the task state is defined by invoking a task, passing the required parameters.

The task proceeds to execute in the same way as an application, following its state diagram. In particular, it may invoke tasks using its own task states. It continues to execute until one of its transitions returns a value to the caller instead of transitioning to a local state in the task.

The value returned by a task must match the declared return type of the task. At this point, execution resumes by executing the transition script of the calling task state, with the returned value available for use.

Transactions defined inside tasks are nested inside the scope of transactions in the calling task, if any.

c. Control Flow Statements in Transition Scripts

- ***goto*** *S(...)* causes the application to exit the current state and goto to a new instance of state S, passing it the specified parameters. This is not a stack-based call, but rather represents leaving the current state and entering the new one. This new syntax replaces the current syntax: *return new S(...)* which was based on the implicit loop above.

- ***return*** *expr* within a task means that the value of the expression *expr* is returned to the calling application's task state (the state that invoked this task). Use of *return* is valid only inside a transition in a state of a task, and the type of the expression *expr* must match the declared return type of the task.

- ***continue*** causes control to remain within the current state, without creating a new state instance, and without executing the state's entry script again. All state variables retain their current values. An example using this statement will be presented later in this document.

- ***exit*** causes the main application to exit. It may occur in a transition of a task or the main application.

- ***exit to URL****("<url>")* causes the main application to exit and the specified URL to be shown. It may occur in a transition of a task or the main application.

Given the control flow semantics of these transition statements, it is now possible to provide reusable tasks that are invoked much like subroutines in a programming language.

Interestingly, the control flow provided by the *goto* statement is a bit difficult to express in Java, as Java does not provide any conditional chaining control flow mechanism in which one method can invoke another, passing parameters to the invoked method, and not have it return to the caller. However, the concept is familiar, since it is essentially a finite state machine control mechanism.

d. BNF for Task Additions to the AUS Sub-language

```
<AUS> ::= [ <packageDeclaration> ]

          task <taskName> [ ( <stateParameterDeclarations> ) ]
                              [ returns <ausType> ]
          { <ausElement>* }

<ausElement>  ::= ...
              ::= <ausImport>

<taskName> ::= <name>

<ausImport> ::= import bos <bosPackageName> . <bosName> [ as <alias> ] ;
            ::= import aus <ausPackageName> . <taskName> [ as <alias> ] ;

<stateElement> ::= ...
               ::= <taskReturn> ;

<taskReturn>  ::= return <scriptExpression> ;

            NOTE: The <scriptExpression> must have the declared <ausType>
                  for this task.

<scriptExpression> ::= ...
                   ::= <taskInvocation>

            NOTE: The predeclared type task is the type returned by fork.
                  The type returned by call is declared by the task.
                  A <taskCall> may be used without an assignment, in which
                  case the result (if any) will not be available to the caller.

<taskInvocation> ::= call <taskName> ( [<variableName>[,<variableName>]*] )
                 ::= fork <taskName> ( [<variableName>[,<variableName>]*] )
```

e. Calling Tasks Directly (Login)

The example in the table above shows how the common **Login** task is invoked.

```
import aus com.sun.common.Login;

state S2(...) {
    Employee e = call Login("NameTool");
    transition {
      switch (e) {
        case null: goto S2(...);
        default: goto S3(e);
      }
    }
}
```

f. Modal Browser Tasks: Setting Relationships by Navigation

Within an application, it is often desirable to allow the user to browse for the required data to be entered into a form. For simple attribute values such as a calendar date, the browser may be a GUI widget that is specific to the type of the attribute being set. However, for relationship values, it is often necessary to allow the user to search for the desired object.

For example, when setting the value of the manager attribute of an employee, the browser might allow using a query to look up the employee id of an existing employee to be the current employee's manager.

```
import aus com.foo.employeechooser;

state A(Employee e) {
  e: R {
        firstName: R,
        lastName: R,
        manager choose with employeechooser(e): RW {
                id: R,
                lastName: R,
                firstName: R
        }
  }
  ...
}
```

We extend the variable usage section of a state to describe how related objects may be chosen. The BNF is as follows:

```
<variableUsage> ::= <usageVariable> <optionalLabel>
                        <optionalBrowser>
                        <optionalModes>
                      [ <variableUsagePropertiesSpec> ] <optionalMembers>

<optionalBrowser> ::=
                  ::= choose with <taskCall>
                  ::= choose from <variableCollection>
                  ::= choose subset from <variableCollection>
```

- **choose with:** The first optional browser syntax, *choose with*, specifies the invocation of a task to choose the desired object (e.g., an employee's manager). This statement is translated into a button on the screen next to the manager that invokes the employeechooser task. The resultant employee returned by the chooser task is assigned to the manager field (e.g., *e.manager* in the example above). The type and cardinality of the chosen field must match the return type and cardinality of the task.
- **choose from:** The second syntax, *choose from*, specifies a collection containing the possible values for the relationship. This statement is translated into a pull-down menu displaying the fields of the relationship that are displayed (e.g., *id, lastName,* and *firstName* in the example above). If no fields are being displayed, the primary key fields are displayed in the pull-down menu. Each element of the specified <variableCollection> must match the type and cardinality of the <usageVariable> field. For example, if the <usageVariable> field is a List<*Employee*>, the <variableCollection> must be a collection of *List<Employee>*, e.g., *List<List<Employee>>*.
- **choose subset from:** The third syntax, *choose subset from,* specifies a collection containing the possible scalar values of a collection-valued <usageVariable>. This statement is translated into a pull-down menu displaying the fields of the relationship that are displayed, as in the *choose*

from statement. The user may select any subset of the displayed menu to be the result of the choice. This browser is valid only for <usageVariable> variables that are collections. The <variableCollection> must be of the same type and cardinality as the <usageVariable>.

g. Independent Non-Transactional Application Threads (e.g., HELP)

This enhancement builds on the task concept to provide the ability to create separate execution threads that proceed independently of the application that starts them. In the language of GUI designers, such applications execute in non-modal windows. In the language of database programmers, such applications have their own, independent session and transaction environment.

A simple application for which this enhancement is useful is providing online help for an application. Pressing the "help" button brings up a new window, from which the user can navigate to the relevant sections of the manual. While looking at the displayed help information, the user can continue to interact with the original application. Ideally, invocation of the help windows should not change the state of the original window at all.

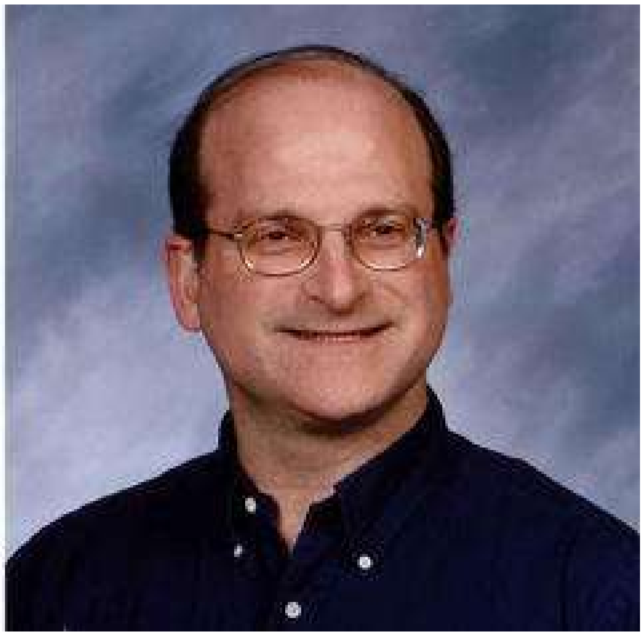
Here is a simple example of how a generic help system might be specified so that a non-modal help thread can be invoked from an application when the user presses a button.

```
import aus com.bar.generic.HelpApp;

...

state A( List<PurchaseOrder> pos ) {
    [pos] {
         orderid,
         name
    }
    transition NewOrder { ... }

    transition Done { ... }

    transition Help {
            fork HelpApp("A");
            continue A;
         }
}
```

The transition "Help" causes a button labeled "Help" to appear on the screen, and that button can be pressed to bring up a window in which the user may navigate to the desired documentation. However, because the transition uses a *fork* statement, pressing the Help button causes the task HelpApp to be invoked in an independent, non-modal window. Because the transition ends with a *continue* statement, the state *A* is not affected or disturbed by pressing "Help", so the window for state *A* remains.

Alphabetical Index

About the Author



Bob Goldberg, Ph.D.
Senior Computer Scientist
Sun Client Solutions

Dr. Goldberg is a Senior Computer Scientist, and the chief designer of the DASL language. He joined Sun in 1995 to develop a database connectivity strategy for Sun's CORBA effort, which led him to his current work on architecture-independent application specification languages. He holds multiple U.S. patents related to relational and object-oriented DBMS technology, distributed application specification, and automatic deployment of applications based on executable specification languages.

Before joining Sun in 1995, Dr. Goldberg worked on complex object retrieval in object-relational DBMS systems at Oracle. Prior to that, he designed and implemented the first commercially used implementation of a SQL3 Relational DBMS engine, which he implemented “from scratch” in the MAINSAIL Programming Language. Prior to that, in the mid-1980's, he added transparent distributed computing to the MAINSAIL language, including a (pre-CORBA) mechanism for remote invocation of objects, and the design of the language's cooperative multithreading mechanism.

Dr. Goldberg has been working with computers and computer languages since 1973 at the National Institutes of Health, where he developed what may have been the world's first general purpose computerized spreadsheet program and statistical analysis language. He holds a degree in Physics from the University of Maryland, and a Ph.D. in Computer Science from Rutgers University, where he did research in artificial intelligence, experimented with early machine independent programming languages, and built a prototype distributed text editor to demonstrate concepts in his thesis. Throughout his career, his passion has been to create innovative ways to enable people to interact with computers in the language of their problem domain.