

Creating ObjectSwitch Applications

ObjectSwitch 4.0.1

© 2002, 2001 Kabira Technologies Inc. All rights reserved.

Without written consent of Kabira Technologies Inc, except as allowed by license, this document may not be reproduced, transmitted, translated, or stored in a retrieval system in any form or by any means, whether electronic, manual, mechanical, or otherwise.

Trademarks

ObjectSwitch is a registered trademark of Kabira Technologies Inc.

Rational Rose, Rose 98i, and Rose 2000 are registered trademarks of Rational Software Corporation.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Windows NT is a registered trademark of Microsoft Corporation.

Informix is a registered trademark of Informix Software, Inc.

Oracle is a registered trademark of Oracle Corporation

Sybase is a registered trademark of Sybase, Inc.

Any other trademarks are the property of their respective owners.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

This product includes software written by the PHP Development Team.

Feedback

This document is intended to address *your* needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other ObjectSwitch documentation, please send e-mail to pubs@kabira.com. Thank you!

Feedback about this document should include the reference DSE-DG-16.

Created on: June 17, 2002

About this book

This book tells you how to develop ObjectSwitch applications. It provides introductory and reference materials, and some guidelines on how to model effectively. Before reading this book, you should already be familiar with object-oriented software concepts. The book is mainly for developers of ObjectSwitch applications, and consists of the following sections.

Introduction A more detailed overview of this book, and the steps and tools that you use to develop ObjectSwitch applications. Introduces the sample application and some conventions used in this document.

Building components Describes the environment and tools that you use to build ObjectSwitch applications.

IDLos and Action language These two chapters describe the static and dynamic modeling languages for building ObjectSwitch applications.

This book is part of a series of ObjectSwitch manuals. Refer also to the following manuals:

- *Advanced ObjectSwitch Modeling*
- *Deploying and Managing ObjectSwitch Applications*

For more information about using adapters in your applications, see the documentation for each individual adapter factory.

Contents

Introduction	1
<i>ObjectSwitch components, 1</i>	
<i>UML and IDLos, 4</i>	
<i>Terminology, 5</i>	
Creating ObjectSwitch models	7
Models	8
<i>Object oriented, 10</i>	
<i>Component oriented, 10</i>	
<i>Richly typed, 10</i>	
<i>Adaptable, 11</i>	
<i>Importing and exporting IDLos, 11</i>	
<i>An example, 12</i>	
Entities	15
<i>Visual Design Center, 17</i>	
<i>IDLos, 18</i>	
<i>Entity properties, 19</i>	
Local entities	21
<i>Visual Design Center, 22</i>	
<i>IDLos, 22</i>	
Attribute	23
<i>Visual Design Center, 24</i>	
<i>IDLos, 25</i>	
<i>Read-only attributes, 26</i>	
Operation	26
<i>Visual Design Center, 27</i>	
<i>IDLos, 30</i>	
<i>Operation properties, 31</i>	
Key	36
<i>Visual Design Center, 37</i>	
<i>IDLos, 38</i>	
Relationship	38
<i>Visual Design Center, 39</i>	

<i>IDLos</i> , 40	
<i>Associative relationships</i> , 41	
Entity trigger	42
<i>Visual Design Center</i> , 45	
<i>IDLos</i> , 45	
Attribute trigger	46
<i>Visual Design Center</i> , 47	
<i>IDLos</i> , 47	
Role trigger	48
<i>Visual Design Center</i> , 49	
<i>IDLos</i> , 51	
Module	53
<i>Visual Design Center</i> , 53	
<i>IDLos</i> , 54	
Package	54
<i>Visual Design Center</i> , 56	
<i>IDLos</i> , 56	
<i>Example</i> , 57	
<i>Package properties</i> , 58	
Interface	59
<i>Visual Design Center</i> , 63	
<i>IDLos</i> , 64	
<i>Interface properties</i> , 64	
Local Interface	66
<i>Visual Design Center</i> , 67	
<i>IDLos</i> , 68	
State machine	68
<i>Visual Design Center</i> , 69	
<i>IDLos</i> , 70	
State	71
<i>Visual Design Center</i> , 72	
<i>IDLos</i> , 73	
Signal	73
Transition	74
Inheritance	75
<i>Entity inheritance</i> , 75	
<i>Operations</i> , 79	
<i>Interface inheritance</i> , 86	
Namespaces	88
<i>Modules</i> , 89	
<i>Model elements defining namespaces</i> , 89	

<i>Scoped names, 89</i>	
<i>Ordering and forward declarations in IDLoS, 91</i>	
A big example	93
<i>Visual Design Center, 93</i>	
<i>IDLoS, 95</i>	
Action language	99
Overview	99
<i>Why is there an action language?, 99</i>	
<i>Action language, 100</i>	
<i>What is action language like?, 100</i>	
Some basic features of the action language	100
<i>Variables, 101</i>	
<i>Manipulating data, 104</i>	
<i>Implicit conversion between string and numeric types, 108</i>	
<i>Name spaces, 110</i>	
<i>Data types, 110</i>	
Control structures	110
<i>Loops, 110</i>	
<i>Branches, 111</i>	
Manipulating objects	111
<i>Creating objects, 112</i>	
<i>Deleting objects, 112</i>	
<i>Singletons, 112</i>	
<i>Object references, 113</i>	
<i>Operation and signal parameters, 114</i>	
<i>Accessing operations and attributes, 115</i>	
<i>Handling relationships, 115</i>	
Building ObjectSwitch components	119
The ObjectSwitch component	120
<i>What is a component specification?, 121</i>	
<i>Defining a component specification, 121</i>	
<i>Graphics vs Text to build a component specification, 123</i>	
Creating a project	123
<i>Project properties, 125</i>	
Working with model sources	128
Defining a component	129
<i>Component properties, 130</i>	
Putting packages in a component	131
Importing another component	133

Adding adapters	135
<i>Adapter properties, 136</i>	
Adding model elements to adapters	136
<i>Model element properties, 138</i>	
<i>Putting relationships and roles into adapters, 138</i>	
<i>Putting attributes into adapters, 138</i>	
Saving a component specification	138
Building the component	139
<i>What can you build?, 139</i>	
<i>What is auditing?, 141</i>	
<i>What you get after you build, 141</i>	

Accessing ObjectSwitch through PHP	143
Overview	145
Data types	146
<i>Basic types, 146</i>	
<i>Boolean, 147</i>	
<i>Enum, 147</i>	
<i>Object references, 147</i>	
<i>Arrays and sequences, 147</i>	
<i>Unsupported types, 148</i>	
PHP script language	149
<i>PHP Syntax, 149</i>	
<i>Using the extension, 149</i>	
<i>Error handling, 150</i>	
PHP4 Extension	151
os_connect	152
<i>Error Conditions, 152</i>	
os_create	153
<i>Example, 153</i>	
<i>Error Conditions, 154</i>	
os_delete	155
os_disconnect	156
os_extent	157
os_get_attr	159
os_invoke	161
os_relate	164
os_role	165
os_set_attr	167
os_unrelate	169

Web Server—Apache	170
Command Line Utility	171
<i>Usage, 171</i>	
<i>Example, 171</i>	
Execute PHP in action language	172
Transactions	175
<i>Web browser or command line transactionality, 175</i>	
<i>Action language callout transactionality, 176</i>	
A PHP example	177
<i>The model, 178</i>	
<i>Example scripts, 180</i>	
Lexical and syntactic fundamentals	185
<i>Character set, 185</i>	
<i>Tokens, 186</i>	
<i>White space, 189</i>	
<i>Comments, 189</i>	
<i>Preprocessing directives, 190</i>	
ObjectSwitch types	191
any	194
<i>Semantics, 194</i>	
<i>Visual Design Center syntax, 194</i>	
<i>IDLos syntax, 195</i>	
<i>Action language syntax, 196</i>	
<i>Example, 196</i>	
<i>General Information, 196</i>	
array	198
<i>Semantics, 198</i>	
<i>Visual Design Center syntax, 198</i>	
<i>IDLos syntax, 198</i>	
<i>Action language syntax, 198</i>	
<i>Example, 198</i>	
<i>General Information, 198</i>	
boolean	199
<i>Semantics, 199</i>	
<i>Visual Design Center syntax, 199</i>	
<i>IDLos syntax, 199</i>	
<i>Action language syntax, 200</i>	
<i>Example, 200</i>	
bounded sequence	201
<i>Semantics, 201</i>	

<i>Visual Design Center syntax, 201</i>	
<i>IDLos syntax, 201</i>	
<i>Action language syntax, 201</i>	
<i>Example, 201</i>	
<i>General Information, 201</i>	
bounded string	202
<i>Semantics, 202</i>	
<i>Visual Design Center syntax, 202</i>	
<i>IDLos syntax, 202</i>	
<i>Action language syntax, 202</i>	
<i>Example, 202</i>	
bounded wstring	203
<i>Semantics, 203</i>	
<i>Visual Design Center syntax, 203</i>	
<i>IDLos syntax, 203</i>	
<i>Action language syntax, 203</i>	
<i>Example, 203</i>	
char	204
<i>Semantics, 204</i>	
<i>Visual Design Center syntax, 204</i>	
<i>IDLos syntax, 204</i>	
<i>Action language syntax, 205</i>	
<i>Example, 205</i>	
const	206
<i>Semantics, 206</i>	
<i>Visual Design Center syntax, 206</i>	
<i>IDLos syntax, 206</i>	
<i>Example, 207</i>	
<i>General Information, 207</i>	
context	208
<i>Semantics, 208</i>	
double	209
<i>Semantics, 209</i>	
<i>Visual Design Center syntax, 209</i>	
<i>IDLos syntax, 209</i>	
<i>Action language syntax, 210</i>	
<i>Example, 210</i>	
entity	211
<i>Semantics, 211</i>	
<i>Visual Design Center syntax, 211</i>	
<i>IDLos syntax, 211</i>	
<i>Example, 211</i>	
<i>General Information, 212</i>	

enum	213
<i>Semantics, 213</i>	
<i>Visual Design Center syntax, 213</i>	
<i>IDLos syntax, 213</i>	
<i>Example, 213</i>	
<i>General Information, 213</i>	
exception	214
<i>Semantics, 214</i>	
<i>Visual Design Center syntax, 214</i>	
<i>IDLos syntax, 214</i>	
<i>Example, 214</i>	
<i>General Information, 214</i>	
extern	216
<i>Semantics, 216</i>	
<i>Visual Design Center syntax, 216</i>	
<i>Action language syntax, 216</i>	
<i>Example, 216</i>	
fixed	217
<i>Semantics, 217</i>	
float	218
<i>Semantics, 218</i>	
<i>Visual Design Center syntax, 218</i>	
<i>IDLos syntax, 218</i>	
<i>Action language syntax, 218</i>	
<i>Example, 219</i>	
interface	220
<i>Semantics, 220</i>	
<i>Visual Design Center syntax, 220</i>	
<i>IDLos syntax, 220</i>	
<i>Example, 221</i>	
<i>General Information, 221</i>	
long	222
<i>Semantics, 222</i>	
<i>Visual Design Center syntax, 222</i>	
<i>IDLos syntax, 222</i>	
<i>Action language syntax, 222</i>	
<i>Example, 223</i>	
long double	224
<i>Semantics, 224</i>	
long long	225
<i>Semantics, 225</i>	
<i>Visual Design Center syntax, 225</i>	

<i>IDLos syntax, 225</i>	
<i>Action language syntax, 226</i>	
<i>Example, 226</i>	
native	227
<i>Semantics, 227</i>	
<i>Visual Design Center syntax, 227</i>	
Object	228
<i>Semantics, 228</i>	
<i>Visual Design Center syntax, 228</i>	
<i>IDLos syntax, 228</i>	
<i>Action language syntax, 228</i>	
<i>Example, 228</i>	
<i>General Information, 229</i>	
octet	230
<i>Semantics, 230</i>	
<i>Visual Design Center syntax, 230</i>	
<i>IDLos syntax, 230</i>	
<i>Action language syntax, 230</i>	
<i>Example, 231</i>	
pipe	232
<i>Semantics, 232</i>	
sequence	233
<i>Semantics, 233</i>	
<i>Visual Design Center syntax, 233</i>	
<i>IDLos syntax, 233</i>	
<i>Example, 233</i>	
<i>General Information, 233</i>	
short	234
<i>Semantics, 234</i>	
<i>Visual Design Center syntax, 234</i>	
<i>IDLos syntax, 234</i>	
<i>Action language syntax, 235</i>	
<i>Example, 235</i>	
string	236
<i>Semantics, 236</i>	
<i>Visual Design Center syntax, 236</i>	
<i>IDLos syntax, 236</i>	
<i>Action language syntax, 236</i>	
<i>Example, 236</i>	
<i>General Information, 236</i>	
struct	237
<i>Semantics, 237</i>	

<i>Visual Design Center syntax, 237</i>	
<i>IDLos syntax, 237</i>	
<i>Action language syntax, 237</i>	
<i>Example, 237</i>	
<i>General Information, 238</i>	
typedef	239
<i>Semantics, 239</i>	
<i>Visual Design Center syntax, 239</i>	
<i>IDLos syntax, 239</i>	
<i>Example, 239</i>	
<i>General Information, 239</i>	
union	240
<i>Semantics, 240</i>	
<i>Visual Design Center syntax, 240</i>	
<i>IDLos syntax, 240</i>	
<i>Action language syntax, 240</i>	
<i>Example, 241</i>	
<i>General Information, 242</i>	
unsigned long	244
<i>Semantics, 244</i>	
<i>Visual Design Center syntax, 244</i>	
<i>IDLos syntax, 244</i>	
<i>Action language syntax, 244</i>	
<i>Example, 245</i>	
unsigned long long	246
<i>Semantics, 246</i>	
<i>Visual Design Center syntax, 246</i>	
<i>IDLos syntax, 246</i>	
<i>Action language syntax, 247</i>	
<i>Example, 247</i>	
unsigned short	248
<i>Semantics, 248</i>	
<i>Visual Design Center syntax, 248</i>	
<i>IDLos syntax, 248</i>	
<i>Action language syntax, 249</i>	
<i>Example, 249</i>	
void	250
<i>Semantics, 250</i>	
<i>Visual Design Center syntax, 250</i>	
<i>IDLos syntax, 250</i>	
<i>Example, 250</i>	
wchar	251
<i>Semantics, 251</i>	

<i>Visual Design Center syntax, 251</i>	
<i>IDLos syntax, 251</i>	
<i>Action language syntax, 251</i>	
<i>Example, 252</i>	
wstring	253
<i>Semantics, 253</i>	
<i>Visual Design Center syntax, 253</i>	
<i>IDLos syntax, 253</i>	
<i>Action language syntax, 254</i>	
<i>Example, 254</i>	
IDLos Reference	255
Files and engines in IDLos	257
<i>Files, 257</i>	
<i>Engines, 257</i>	
action	259
attribute	260
const	261
enum	263
entity	264
exception	266
expose	267
interface	268
key	270
local entity	272
module	274
operation	275
package	277
relationship / role	279
signal	281
stateset	282
struct	283
transition	284
trigger	285
typedef	287
Complete IDLos grammar	289
Action language reference	297

<i>About the notation, 297</i>	
<i>Some common elements, 298</i>	
break	300
cardinality	301
continue	303
create	304
create singleton	306
declare	308
delete	310
empty	311
Exceptions	313
extern	317
for	318
for ... in	320
if	
else	
else if	322
in	324
isnull	325
relate	326
return	328
select	330
select...using	332
<i>Syntax, 332</i>	
<i>Description, 332</i>	
<i>Warnings, 332</i>	
<i>Examples, 332</i>	
<i>Locking examples, 333</i>	
self	334
spawn	335
Transactions	341
<i>IDLos Constraints, 341</i>	
Types	343
unrelate	345
while	347
Complete action language grammar	349
Build specification reference	355

adapter	357
component	358
group	359
import	360
Macros	361
Properties	363
source	366
swbuild	367
Complete build grammar	368
Complete build specification example	370

PHP reference	375
----------------------	------------

About the notation, 375

os_connect	376
-------------------	------------

Error Conditions, 376

os_create	377
------------------	------------

Example, 377

Error Conditions, 378

os_delete	379
------------------	------------

os_disconnect	380
----------------------	------------

os_extent	381
------------------	------------

os_get_attr	383
--------------------	------------

os_invoke	385
------------------	------------

os_relate	388
------------------	------------

os_role	389
----------------	------------

os_set_attr	391
--------------------	------------

os_unrelate	394
--------------------	------------

Part One: Creating ObjectSwitch Applications

This chapter introduces tools and terminology used in modeling and building an ObjectSwitch component. ObjectSwitch components are built directly from object-oriented models without a traditional coding stage. This makes them quick to develop and easy to maintain.

In this chapter you are introduced to:

- modeling and building components
- the terminology

Before reading this book, you should already be familiar with the concepts presented in the *Overview of ObjectSwitch*. You should also be familiar with object-oriented design concepts.

ObjectSwitch components

An ObjectSwitch component is an executable model that can be deployed and reused. When you build a component, everything needed for deployment on an ObjectSwitch node and for reuse in another component is wrapped into a single file.

Components let you create modular applications. You can model a related set of types and services in a single component, then access those types and services from any number of components. In addition, you can wrap interfaces to non-ObjectSwitch services—for example, applications implemented in Java or accessed through CORBA—into components, and reuse those components in other components.

Although each component is deployed separately, you don't need to think about deployment when modeling a component. Dependencies between components are specified at build time, and are stored with the dependent component. When you load a dependent component on an ObjectSwitch node, the components upon which it depends are loaded automatically.

To build an ObjectSwitch component you will:

- model your application logic in IDLos or UML with Action Language
- create a component specification describing how to build the model
- build the component using the Design Center server

Modeling a component You model ObjectSwitch components using one of two methods:

- graphical modeling using *Rational Rose*
- a text editor such as *vi*

In Rational Rose you model using the Unified Modeling Language (UML). Using a text editor, you model using the ObjectSwitch interface definition language, IDLos. IDLos is a superset of IDL with entities and relationships added, and is UML compliant. You can think of it as a textual version of UML.

The two methods of creating component models are equivalent, allowing for round-trip development. Models can be created or modified textually, then imported into Rational Rose to create a graphical representation of the model that you can work with using Rose's graphical tools. Models that you create or modify in Rose can be exported to IDLos, then modified with a text editor.

Component specification The description of how to build an ObjectSwitch component is called a *component specification*. You use the same tools to create a component specification that you use to model. In *Rational Rose* you use drop-down menus and drag-and-drop icons to create a component specification. Textual component specifications are written in a text editor using a simple build

specification language. The two methods are equivalent. A component specification created in *Rational Rose* can be exported to create a textual component specification.

When specifying how you want the component to be built, you can import other components upon which your new component depends. You need to import any component that defines types or operations used in the component you are building.

Building a component Once your model and component specification are complete, you can build the component. Again, you can do this from within *Rational Rose* or textually at the command line. With either method, the component specifications are sent to an ObjectSwitch Design Center server to generate the deployable and reusable component.

The components are built on the platform where the Design Center server is running. You can model, specify, and build a component while working on one platform that will be deployed on another, simply by connecting to a Design Center server that is running on the deployment platform.

UML and IDLos

Throughout these books, models are expressed using both UML and IDLos. Figure 1 summarizes some basic elements of a UML class diagram.

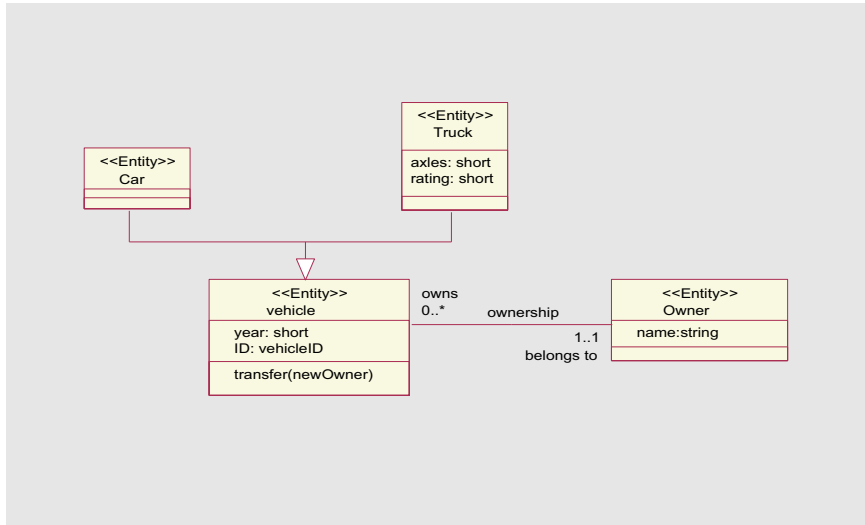


Figure 1: Basic features of the UML

Figure 1 shows four entities:

- zero or more Vehicles are owned by exactly one Owner. In IDLos this relationship would be represented as:

```

relationship ownership
{
    role Owner owns 0..* Vehicle;
};

```

- Vehicle has two attributes and an operation. The vehicle entity in IDLos would appear:

```

entity Vehicle
{
    attribute short year;
    attribute VehicleID ID;
    void transfer(in Owner newOwner);
};

```

- Car and Truck are two kinds of Vehicle, and Truck has two additional attributes beyond those inherited from Vehicle.

There are many good books on UML; this manual does not attempt to teach the notation. The UML chapter in *Advanced ObjectSwitch Modeling* describes in detail how IDLoS is represented in UML.

Terminology

Object-oriented analysis and design uses a number of different terms to represent similar concepts, varying from one methodology to another. ObjectSwitch draws on a number of different traditions, so it is necessary to clarify the terminology that we use.

Term	Meaning
Attribute	Is a piece of data defined as part of an entity. Some traditions refer to this as a <i>data member</i> .
Component	The Design Center builds your application into one or more components, each containing one or more <i>engines</i> . You use the Engine Control Center to deploy a component on an ObjectSwitch node.
Engine	A single executable process. In the Design Center, you map your application model onto one or more engines, and then the Design Center builds components containing those engines. When you deploy a component, the Engine Control Center installs its engines onto an ObjectSwitch node.
Entity	An uninstantiated object. Other methodologies may refer to this as a <i>class</i> or as an <i>object</i> .
Event	A data structure used to communicate information between objects. Events may be synchronous or asynchronous.
Interface	Indicates what part of an entity is exposed outside of a package. An entity may have many interfaces, but each interface may have only one entity. Other traditions have a broader definition.
Node	A managed area of shared memory that contains object data, the Event Distribution Bus, and other data structures.

Term	Meaning
Object	An instance of an entity.
Operation	Is an executable part of an entity, known in other traditions as a <i>member function</i> or as a <i>method</i> .
Signal	A message that causes a state transition in the state machine of an object. Known in some traditions as an <i>event</i> .
Supertype/subtype	Represents inheritance (also called <i>generalization</i>). If some entity B inherits from another entity A, then A is the supertype of B and B is a subtype of A. Other traditions may refer to this as <i>parent class</i> and <i>child class</i> , or as <i>base class</i> and <i>derived class</i> .

The next chapter describes how you create ObjectSwitch models in graphics or text.

This chapter describes how you model in ObjectSwitch. It introduces each ObjectSwitch modeling concept and shows how you use it to create ObjectSwitch models.

ObjectSwitch lets you choose between visual and textual modeling, at any stage. It provides you with modeling constructs, such as entities, attributes, operations, packages, that you can use in their visual or textual form to create your models.

The Unified Modeling Language is used to represent ObjectSwitch models visually. IDLos, an IDL-based modeling language, provides the textual equivalent. This chapter introduces these modeling concepts. For more detailed reference material, refer to the following:

Chapter 6 explains the lexical rules of ObjectSwitch models

Chapter 7 describes the ObjectSwitch type system

Chapter 8 provides a detailed IDLos language reference

After a general overview in “Models” this chapter introduces the following ObjectSwitch modeling constructs:

- Entities
- Local entities
- Attribute
- Key
- Operation
- Entity trigger
- Attribute trigger
- Role trigger
- Relationship
- Module
- Package
- Interface
- Local Interface

The section “State machine” shows you how to model finite state machines, and you will find a separate section covering each state machine construct:

- State
- Signal
- Transition

The final sections of this chapter cover:

- Inheritance
- Namespaces
- A big example

Models

ObjectSwitch applications are defined in terms of object models. Models can be expressed graphically in the Unified Modeling Language (UML) or textually in the IDLos modeling language; the two forms are fully equivalent.

You create UML models in the ObjectSwitch Visual Design Center. The Visual Design Center also lets you convert between the visual and textual representations by importing or exporting IDLos. You can also create IDLos models using an ordinary text editor.

Consider a UML model fragment in the Visual Design Center, as shown in Figure 2, and its corresponding IDLos representation.

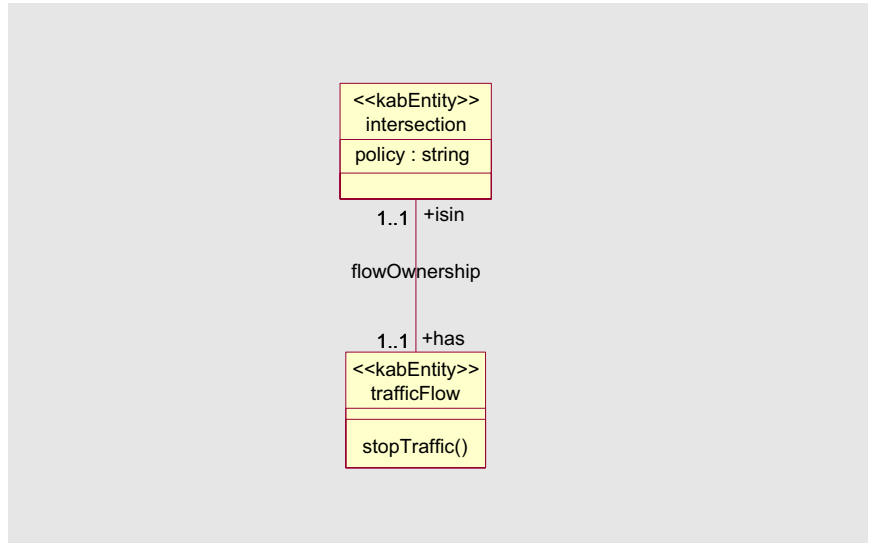


Figure 2: A model fragment
in the Visual Design Center

```

entity intersection
{
  attribute string policy;
};

entity trafficFlow
{
  void stopTraffic();
};

relationship flowOwnership
{
  role intersection has 1..1 trafficFlow;
  role trafficFlow isin 1..1 intersection;
};
  
```

This model fragment has two entities that are associated using a relationship. One entity has an attribute of type string, and the other has an operation that does not return a value. You would specify the implementation of this operation in Action Language using the action statement (not shown in this fragment).

You can model graphically in UML, textually in IDLos, or both; the choice is yours. In both cases your model is compiled into an application that you can deploy on an ObjectSwitch server.



The ObjectSwitch textual modeling language IDLos is based on IDL v 2.2, and existing IDL can be used directly in ObjectSwitch applications. IDLos extends the standard, providing relationships, state machines, and triggers; as well as entities and actions to specify implementations.

Object oriented

The ObjectSwitch modeling language is fully object-oriented (OO), supporting inheritance and polymorphism. It also supports some features often requested but not often found in OO languages: singletons, keys, triggers, and transactions.

In ObjectSwitch, inheritance includes relationships, state machines, and triggers. The inheritance of relationships is particularly powerful.

Component oriented

You organize your ObjectSwitch applications in *packages*. Objects in the same package have access to one another. To make objects visible outside a package, you define *interfaces*.

ObjectSwitch packages are more than an analysis principle. Packages represent the lowest level of granularity for building components. You can build one component for each package or group several packages into one component. And because of the ObjectSwitch runtime technology, you can add new components to your running application or update existing components without bringing down your application.

Richly typed

IDLos uses the IDL type system, which provides a rich system of basic and user-defined types. See Chapter 7 for more on the ObjectSwitch type system.

Adaptable

The Visual Design Center generates ObjectSwitch runtime components (processes executed by the ObjectSwitch runtime). You can use a range of ObjectSwitch adapter factories to expose your model to any number of technologies, for example, CORBA, Java, PHP, and SNMP.

When you model with ObjectSwitch, you ignore the specifics of the technology that you expose your model to. You make these implementation decisions later, at build time. While modeling, you concentrate solely on your business logic and the interfaces that expose your model.

Importing and exporting IDLos

You can easily convert models between IDLos and UML by importing or exporting them from the Visual Design Center.

Importing a model from an IDLos file From the Tools menu, select Kabira ObjectSwitch->Import IDLos and choose the IDLos source file from the dialog that appears.

Exporting a UML model to IDLos From the Tools menu, select Kabira ObjectSwitch->Export IDLos and name the IDLos destination file in the dialog that appears. Alternatively, you can right-click on a package in a class diagram and selecting Kabira ObjectSwitch->Export Package. You can export multiple packages by selecting the packages first (using shift-click or ctrl-click) and then exporting them using the right-click method.

Exporting UML from the command line The Visual Design Center installation includes a command you can use to export a UML model to IDLos. The command is located in the directory where the VDC was installed, and has the following syntax:

```
exportModel -model srcpath -exportPath destpath [-packages list]
```

Where:

srcpath is the fully-qualified path name of the Rose .mdl file representing your model.

dstpath is the fully-qualified path name of the subdirectory where the .soc and .act files will be created.

list is used with the optional -packages list to select particular packages to export. The argument *list* is a semicolon-separated list of package names: package1;package2;package3 and so on. By default (if you do not use the -packages switch) exportModel exports all packages.

The exportModel command creates separate static (.soc) and action language (.act) model files, with the same name as their corresponding packages. For example a package named MyPackage would be exported as MyPackage.soc.

Here's an example command line in the MKS Korn Shell to export all packages from the model test.mdl to the subdirectory c:\temp:

```
exportModel -model c:\\temp\\test.mdl -exportPath c:\\temp
```



You MUST use the double-backslash in the MKS Korn Shell. Forward slashes will not work because Rose is a Windows application that doesn't understand the forward slash.

Do not move the exportModel file itself. Either add the Add-In install directory to your path or invoke exportModel using the fully-qualified file name.

An example

This example shows two packages; one is a client and the other is a server. The client will ask the server to say hello, and the server will respond by printing “Hello World”.

Visual Design Center The server consists of the entity `TalkerImpl` and the interface `Talker` that exposes the operation `sayHi`. The `sayHi` operation is implemented in the entity `TalkerImpl`. Clients will invoke the `sayHi` operation through the `Talker` interface.

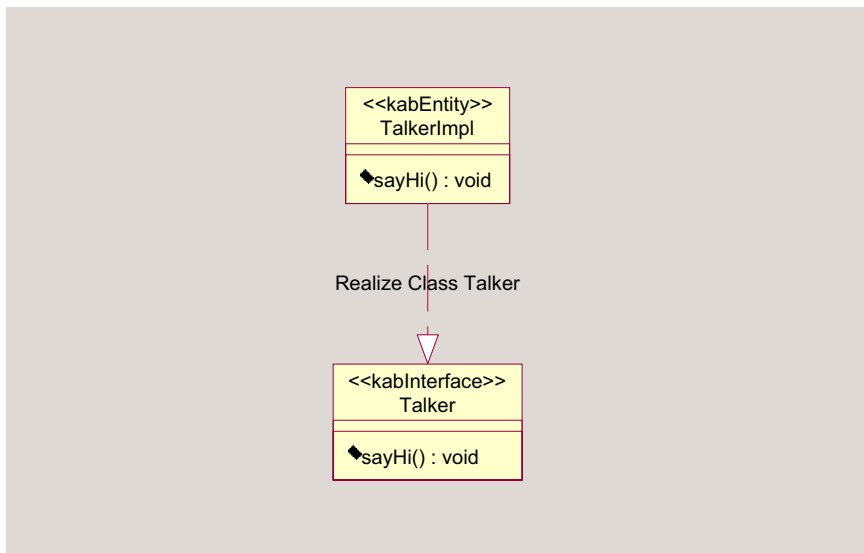


Figure 3: The server package

The operation `sayHi` is implemented with the following line of action language:

```
printf("Hello World\n");
```



The arrow connecting an interface with the entity that implements it may be drawn in either direction. The convention in this manual is to show the arrow pointing at the implementing entity, in the sense of the IDLs verb “expose”. You may wish to draw the arrow pointing to the interface, in the sense of the UML verb “realizes”.

The client package contains one local entity, `Startup`, with a lifecycle operation `init` that kicks off the application when the runtime initializes the client component.



Figure 4: The client package

The operation `init` contains the following lines of action language:

```
declare ::Server::Talker t;
declare ::swbuiltin::EngineServices es;
create t;
t.sayHi();
delete t;
es.stop(0);
```

IDLos Here is the same model in IDLos:

```
package Server
{
    entity TalkerImpl
    {
        oneway void sayHi();
    };
    interface Talker
    {
        oneway void sayHi();
    };
    expose entity TalkerImpl with interface Talker;
```



```

};
action ::Server::TalkerImpl::sayHi
{
    printf("Hello World\n");
};

package Client
{
    [local]
    entity Startup
    {
        [initialize]
        void init();
    };
};
action ::Client::Startup::init
{
    declare ::Server::Talker t;
    declare ::swbuiltin::EngineServices es;
    create t;
    t.sayHi();
    delete t;
    es.stop(0);
};

```

Entities

Modeling with ObjectSwitch means applying an object-oriented approach. Central questions of any object-oriented approach include

- what types of objects will my application be dealing with?
- what type of data will the objects provide?
- what behavior will the objects display?

For example, one type of object in your application could be specific customer records. The data of a customer record could include a customer's name, address, and phone number. The behavior of a customer record could involve changing the customer data, for example, when a customer moves or marries.

In ObjectSwitch to model a type (or class) of object, you use an *entity*. An object of that type is an *instance* of the entity. In an application there can be many instances of one entity.

An entity allows you to define the internal data structures (in the form of *attributes*) and behavior (as *operations*) for a type of object. For entities with many instances you can define *keys* to provide efficient selection of individual objects. *Triggers* let you invoke behavior when certain events happen, such as creating a new object or changing an attribute value. You can model the lifecycle, or states, of the entity's instances in a *state machine*.

Entities are the most powerful user-defined type. They are at the heart of every ObjectSwitch model.

The following table shows the modeling elements that you use to define an entity's data and behavior:

Attribute	Data member of an entity. Each instance has a data member of the type specified by an attribute.
Operation	Behavior that you invoke on an instance.
Key	Unique identifier. Keys enable you to efficiently query for individual instances.
Entity trigger	An event-operation pair. A trigger specifies an operation that is invoked on an instance at the onset of a certain event, for example, when you create, delete, refresh, or relate an instance.
State machine	Finite state machine. A state machine defines the life cycle of an instance, the states through which each instance can pass.

Nested types To execute their behavior, instances can also require types of data that are only valid within the scope of that instance. For example, this could be a constant that is valid for all instances, a nested entity, or simply an array of some type.

Within the scope of each entity you can define the following types, which are then only accessible for an instance of that entity:

typedef	an alias for a basic or composite type
const	a named constant
types	a basic or composite type

Namespace Each entity lives within some namespace (see “Namespaces” on page 88). You can define an entity only within the direct scope of either a package or a module.

Relationships You can define a *relationship* between entities, so that instances of one entity can be linked to instances of the other entity. See “Relationship” on page 38.

Inheritance An entity may inherit traits from another entity. See “Inheritance” on page 75.

Exposure There is no access to an entity from outside its package unless you explicitly enable it. You enable access to an entity’s attributes and operations from outside the package using an interface. See “Interface” on page 59.

Visual Design Center

The Visual Design Center represents an entity as a class with stereotype `Entity`. An entity has three sections: the header at the top (with stereotype and entity label), the attribute section in the middle, and the operation section at the bottom.

To add a new entity to a module or package in the Visual Design Center:

- 1 click on the entity icon in the toolbar
- 2 click somewhere in a package or module diagram

Class Specification dialog Once you add an entity to a module or package, you model its properties, attributes, operations, constants, keys, triggers and nested types in the Class Specification dialog.

To open the Class Specification dialog, right-click on the entity and select Open Specification

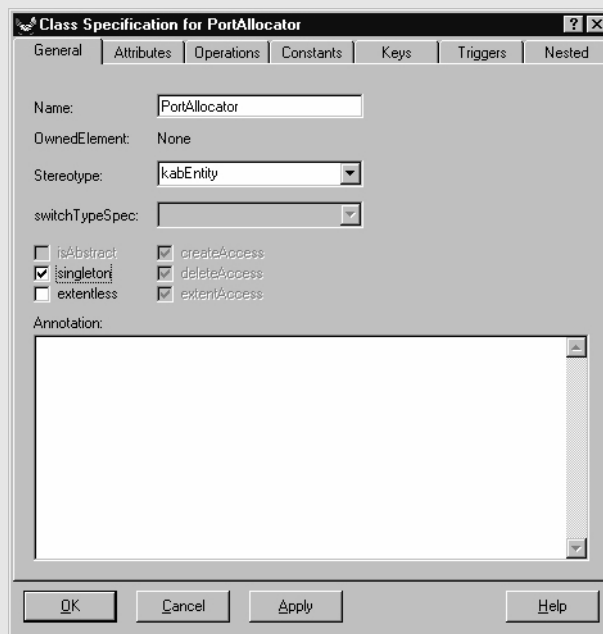


Figure 5: Class specification dialog

IDLos

Use the entity statement (see "entity" in Chapter 8 for syntax details).

Here is the IDLos for the example above:

```
entity TimerEventImpl
{
    // attributes
    attribute Road road;
    // operations
    oneway void generate ();
};
```

Entity properties

You can tag an entity with properties that affect the number of permitted instances as well as its representation in shared memory.

The following table shows the properties you can use in the definition of an entity.

singleton	Only one instance of a singleton entity is ever instantiated.
extentless	The extent (a list of all instances) is not generated.
dynamicLog	The runtime allocates the entity's log when you modify an instance and frees the log when the transaction commits or aborts. This reduces the shared memory footprint of instances; use this property for instances which you modify infrequently.

Setting properties In the Visual Design Center you set an entity's properties in the General tab of the Class Specification dialog (see "Class Specification dialog" on page 17).

In IDLoS entity properties appear at the beginning of the IDLoS statement, in square brackets.

```
[singleton] entity PortAllocator {};  
[extentless] entity InvoiceItem {};
```

singleton The singleton property denotes an entity that can have only one instance. At runtime, creating a singleton either:

- creates an instance if one does not already exist, otherwise it
- returns a reference to the existing instance

Singletons must not be local, and cannot be a supertype or subtype of another entity.

extentless You can use the extentless property to suppress the creation and management of extents in the application server.

ObjectSwitch normally keeps the *extent* of an entity in shared memory. The extent is a list of all instances of the entity. Even when an instance is flushed from shared memory, a reference to it stays in the extent.

Extentless entities, as the name implies, do not have an extent maintained for them. This places certain restrictions on what you can do, and it optimizes the behavior of the runtime under certain conditions.



When you iterate across all the instances of an extentless entity, you may miss some objects or encounter some twice, depending on other transactions' use of that type. Similarly, the result of the cardinality operator is not definitive for extentless entities.

So why would you use extentless objects? Because they may work better in your design for objects that are either:

- extremely numerous, and the extent itself consumes too much memory
- created and deleted frequently and concurrently, causing lock contention on the extent

This second point requires some explanation. When normal objects (as opposed to extentless ones) are created or deleted, their extent is locked until the completion of the transaction. This prevents other instances of the entity from being created or destroyed until the transaction completes.

When an extentless object is created or deleted, locking takes place at a much lower level and for a much shorter period. Once the object creation is complete, other instances of the type can be created right away. Using extentless objects can substantially improve performance in certain cases.

Local entities

Local entities like regular entities, but with some optimizations that make them suitable for the [initialize] and [recovery] engine operations. Normally you will only use them for these operations. Local entities have a number of restrictions:

- not distributable
- not recoverable
- cannot be used by clients in other components
- cannot be used as references in distributable types
- cannot be used as parameters in non-local operations

Local entities must not be singletons.

Like the local entity itself, a nested type defined in a local entity cannot be used as a parameter or return value for an operation, or as an attribute for a non-native entity.

Exposure There is no access to a local entity from outside its package unless you explicitly enable it. You enable access to a local entity's operations with the following element:

Local Interface	Defines operations that are accessible from outside a local entity's package.
-----------------	---

Lifecycle operations Use operations defined in local entities to perform lifecycle tasks. You specify that an operation is invoked when the runtime initializes, recovers, or terminates a component (see “initialization, recovery, termination” on page 33) or before any other lifecycle operation (see “packageinitialize” on page 33).

Visual Design Center

The Visual Design Center represents a local entity as a class with stereotype `kabLocalEntity`. Like an entity, a local entity has three sections: the header at the top (with stereotype and entity label, the attribute section in the middle (which is always empty), and the operation section at the bottom.

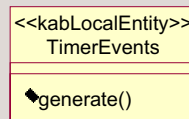


Figure 6: Local entity in the Visual Design Center

To add a new local entity to a module or package in the Visual Design Center:

- 1 click on the local entity icon in the toolbar
- 2 click somewhere in a package or module diagram

You can define operations and nested types for a local entity in the entity's Class Specification dialog (see "Class Specification dialog" on page 17).

IDLos

Use the entity statement. A local entity is an entity marked with the `local` property (see "local entity" in Chapter 8 for syntax details).

```
[local]
entity StartUp
{
    // operations
```



```
[initialize]
void initData ();
};

[local]
entity Utilities
{
    void who();
    void whatis();
};
```

Attribute

Attributes specify the data members of entities. You define an attribute in an entity or an interface. Defining an attribute in an interface enables cross-package access to the same attribute in the exposed entity.

An attribute has a type and an attribute label. Attributes can be of any valid type.

You can define a trigger that invokes an operation when an attribute is accessed (see “Attribute trigger” on page 46 for details).

Visual Design Center

The Visual Design Center represents attributes in the middle section of an entity. First the label appears and then the type, separated by a colon (see the figure below).

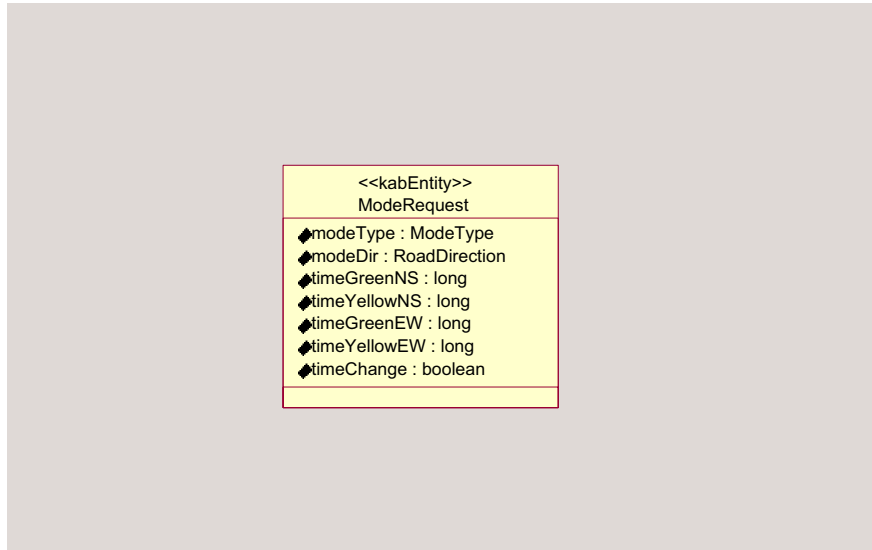


Figure 7: Attributes in Visual Design Center

To add an attribute to an entity or interface:

- 1 open the Class Specification dialog (see “Class Specification dialog” on page 17)
- 2 click on the Attributes tab
- 3 click on the Add button

This opens the Attribute Specification dialog (see the figure below), where you can specify the attribute's label in the Name field and select its type from the Type drop-down selection box and set the attribute to readOnly if desired.

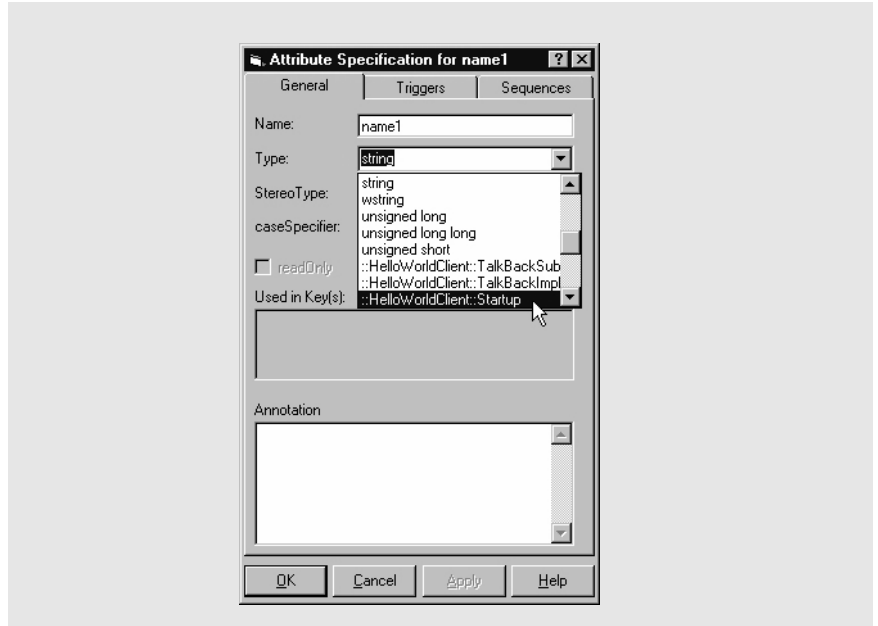


Figure 8: Attribute Specification dialog

IDLos

Use the attribute statement (see "attribute" in Chapter 8 for syntax details).

Here is the IDLos for the example in Figure 7:

```
entity ModeRequest
{
    // attributes
    attribute ModeType modeType;
    attribute RoadDirection modeDir;
    attribute long timeGreenNS;
    attribute long timeYellowNS;
    attribute long timeGreenEW;
    attribute long timeYellowEW;
    attribute boolean timeChange;
};
```

Read-only attributes

You can optionally use the `readonly` keyword to specify that an attribute may not be modified. Attributes are read-write by default unless you specify `readonly`.

Specifying a ready-only attribute In the Visual Design Center you specify this in the General tab of the Attribute Specification dialog (see Figure 8).

In IDLos you use the `readonly` keyword at the beginning of the attribute statement:

```
readonly attribute long employeeIndex;
```

Operation

Operations provide a way to invoke actions on an object. You define an operation in an entity or an interface. Defining an operation in an entity provides an entry point for invoking behavior; defining it in an interface exposes that operation to other packages.

Operations can be invoked and executed at any time. In contrast, use a state machine to specify behavior that takes place as objects pass through different states.

Each operation has a *signature*: name, parameters, and return type. An operation may also raise exceptions. An operation's name must be unique within the entity's namespace, and cannot be overloaded.

Parameters Operations can take parameters. Parameters have a type and a direction: `in`, `inout`, and `out`. Parameters with the `in` direction cannot be changed by the operation's implementation. Parameters with the `inout` and `out` direction may be changed.

You cannot call a non-const operation on a const reference. All `in` parameters are passed in as const references. Remember that in a const operation, `self` is a const reference.

Return type The return type can be any type valid in the context of the operation, or `void` if there is no type returned.

Exceptions You must specify the exceptions that an operation raises (for more on exceptions, see Chapter 2).

You implement operations using action language (for more on action language, see Chapter 3).

One-way and two-way operations By default, operations are two-way, or synchronous, operations. The work is dispatched to the destination and the caller blocks (suspends execution) until the operation returns. But you can explicitly define a one-way, or asynchronous, operation. When a caller invokes a one-way operation, the caller keeps executing even though the work may not yet be completed.

Visual Design Center

The Visual Design Center represents operations in the bottom section of an entity. The following figure shows an entity with the operation signatures displayed.

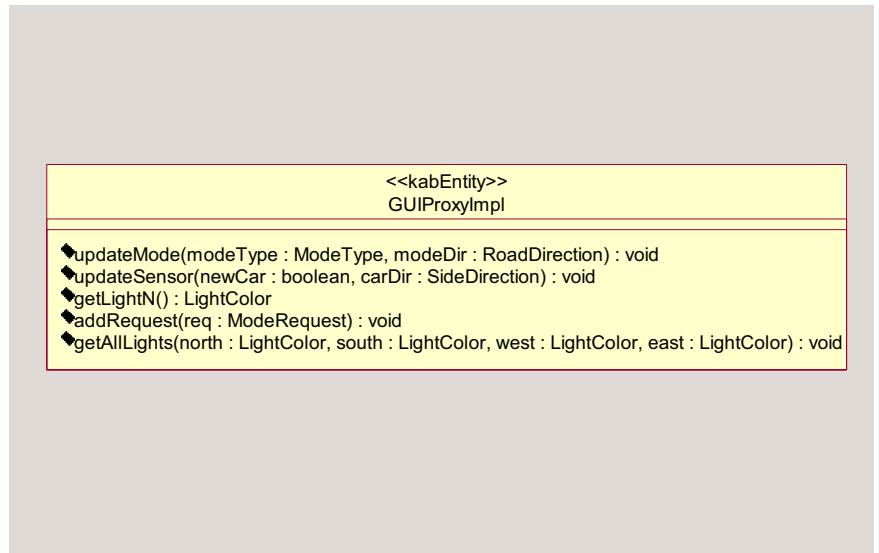


Figure 9: Operations in Visual Design Center

Operations are managed from the Operations tab of the Class Specification dialog. From this tab you can add and delete operations as well as specify a return type.

The operation label appears in the Name field and the operation return type in the Return type field (this field is a drop-down selection box), and parameters in the Parameters field (see the figure below).

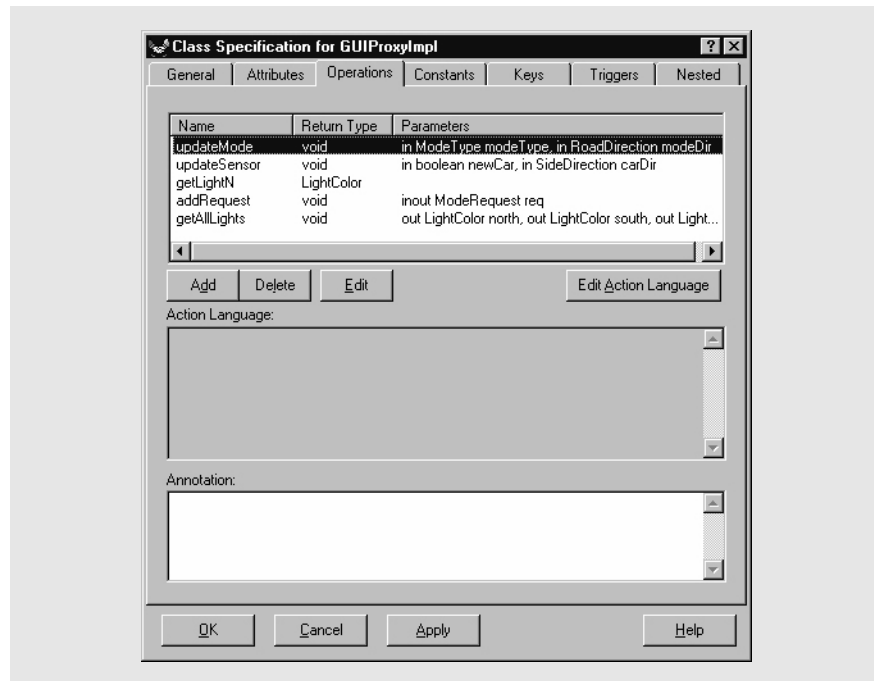


Figure 10: Adding an operation in the Visual Design Center

To add an operation to an entity, local entity, interface, or local interface:

- 1 open the Class Specification dialog (see “Class Specification dialog” on page 17)
- 2 select the Operations tab
- 3 click the Add button

This opens the Operation Specification dialog, where you can specify the operation’s signature, set properties, specify exceptions raised by the operation, and add the action language implementation. The Operation Specification dialog has four tabs: General, Parameters, Action Language, and Raises.

General tab In the General tab of the Operation Specification dialog (see the figure below) you can enter the operation’s identifier in the Name field, select a return type from the Return type selection box, and set properties (see “Operation properties” on page 31).

Parameters tab In the Parameters tab of the Operation Specification dialog (see the figure below) you can add a parameter with the Add button. Once you have added a parameter, you can edit the default values for direction, type, and parameter name. The Direction and Type fields are selection boxes that enable you to select a listed value. You can edit the Name field directly.

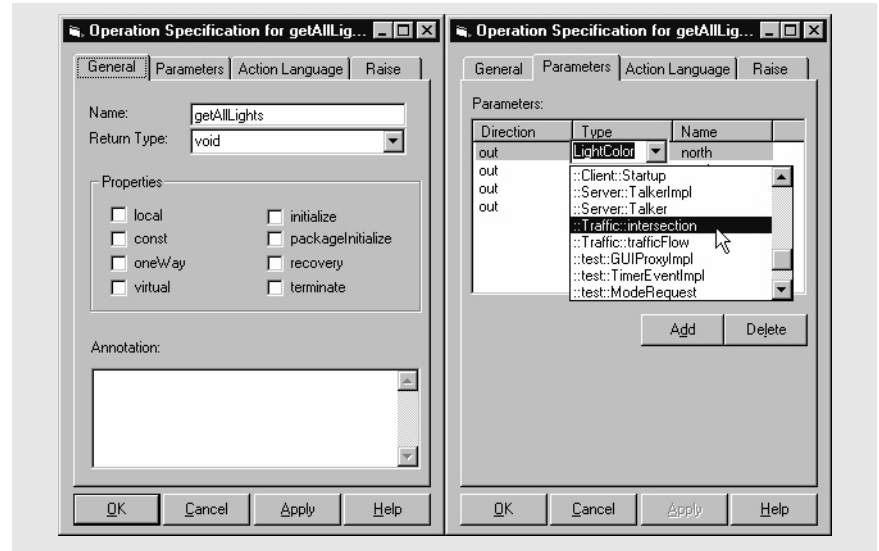




Figure 11: General and Parameters tab in the Operation Specification dialog

Action Language tab Click on the Add Action Language button to edit the operation’s action language implementation.

Modeless action language editor If you right-click on an operation, you can invoke the Action Language editor as modeless. This means that you can navigate the model to see your types, constants, etc. without having to first close the editor.

Raises tab You can specify exceptions that the operation can raise. The Exceptions pane shows exceptions that you can select for the operation. The Raises pane shows exceptions that are currently selected.

-  To move an exception from the Exceptions to the Raises pane, highlight the exception in the Exceptions pane and click on the arrow pointing to the right.
-  To move an exception from the Raises to the Exceptions and, highlight the exception in the Raises pane and click on the arrow pointing to the left.

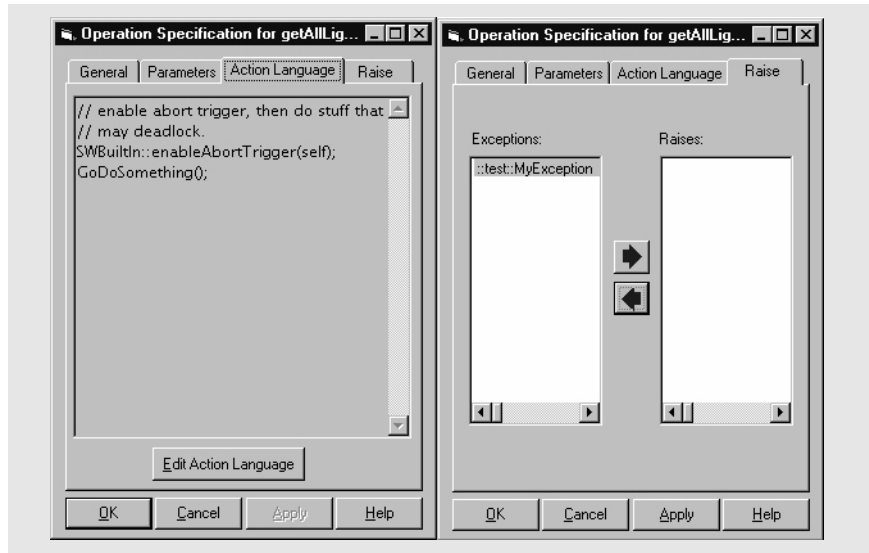


Figure 12: Action Language and Raises tabs in the Operations Specification dialog

IDLos

For a detailed description of the IDLos syntax for operations, see "operation" in Chapter 8.

Here is the IDLos for the example in Figure 9:

```
entity GUIProxyImpl
{
    // operations
    void updateMode (
        in ModeType modeType,
        in RoadDirection modeDir);
    void updateSensor (
        in boolean newCar,
        in SideDirection carDir);
    LightColor getLightN () raises (expLightNotFound);
    void addRequest (inout ModeRequest req);
    void getAIlLights (
```



```
    out LightColor north,  
    out LightColor south,  
    out LightColor west,  
    out LightColor east);  
};
```

Operation properties

The following table shows the properties you can use in the definition of an operation.

local	The runtime invokes a local operation directly, rather than dispatching it through the event bus (see “local” on page 32).
const	The operation cannot change the internal state of the object (see “const” on page 32).
oneway	The operation is asynchronous; all other operations are twoway, or synchronous (see “oneway” on page 33).
virtual	Defines a polymorphic operation (see “virtual” on page 33).
initialize	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation when the operation’s component initializes (see “initialization, recovery, termination”).
recovery	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation when the operation’s component recovers (see “initialization, recovery, termination”).

terminate	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation when the operation's component terminates (see "initialization, recovery, termination").
packageinitialize	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation before all other lifecycle operations when the operation's component initializes (see "packageinitialize" on page 33).

Setting operation properties In the Visual Design Center, you set an operation's properties in the General tab of the Operation Specification dialog (see Figure 11).

In IDLoS operation properties appear at the beginning of the IDLoS statement, in square brackets:

```
[oneway, virtual]  
void addRequest (inout ModeRequest req);
```

local When you build a component that invokes a local operation, you must link in the local operation's implementation.

Local entities may be used as operation parameters to local operations, or as parameters to operations defined in a local entity (see "Local entities" for more).

const The const property restricts what an operation can do. The restrictions that it imposes are:

- a const operation cannot change the internal state of the object
- the action language that implements the operation may not invoke any non-const operations

If an operation in an entity is marked const, then it must also be marked const in any interfaces that expose it. Similarly, if it is not marked const, then it must not be marked const in the interface. This can be stated as:



An exposing operation in an interface must match the “constness” of the operation in the underlying entity.

oneway Delivery of a oneway operation is guaranteed. A oneway operation must return void, can be neither inout nor out parameters, and cannot raise an exception.



Asynchronous operations are good for ObjectSwitch performance. They help ObjectSwitch take advantage of its multi-threaded architecture, and they alleviate object lock contentions by starting new transactions. Too many synchronous calls can get all the objects locked up in the same transaction. Use synchronous calls sparingly.

virtual This property forces an operation to be dispatched polymorphically. This means operations are invoked based on the actual instance type, rather than on the type of the handle. So even when you upcast a subtype object to the supertype, if an operation is marked virtual in the supertype, the subtype operation is invoked. See “Virtual operations and polymorphic dispatch” on page 82.

initialization, recovery, termination You can mark operations so that the ObjectSwitch runtime invokes them upon component initialization, recovery, or termination. You may define any number of such operations in any number of local entities.

You do not know the order that these three types of lifecycle operations are invoked. For example, if you define multiple initialize operations in a component, then these operations may execute in any order (although operations in singletons will execute before operations in other entities).



Although initialize and recovery operations are invoked when the engine starts, it is possible that external events (from other engines) may be processed before the lifecycle operation. Where this could cause a problem, use the packageinitialize property described below.

packageinitialize You can use the packageinitialize property, to designate an operation that executes before any other lifecycle operation. It also executes before any external events are processed; this lets you clean up process resources on initialization and recovery.

The `packageinitialize` operation must be a two-way operation with no parameters or return value. It can only access entities in the same package. Another way to say this is that it cannot access any interfaces from other packages. If this is attempted, the runtime will report a fatal error and exit the engine.



The `packageinitialize` property replaces the `package_initialize` property used in previous versions. The old spelling is supported for compatibility but is deprecated. Use `packageinitialize` instead.

Lifecycle example The example contains a local entity with several lifecycle operations. The runtime invokes `init` and `initializeServices` when the component is initialized, `initializeServices` and `checkNetConnections` when it recovers the component, and `cleanUp` when it terminates the component.

Consider the local entity `EngineEvents` in the following figure.

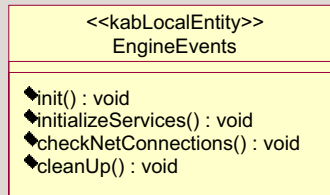


Figure 13: Lifecycle operations

The following figure shows the lifecycle settings for the operations `initializeServices` and `cleanUp`.

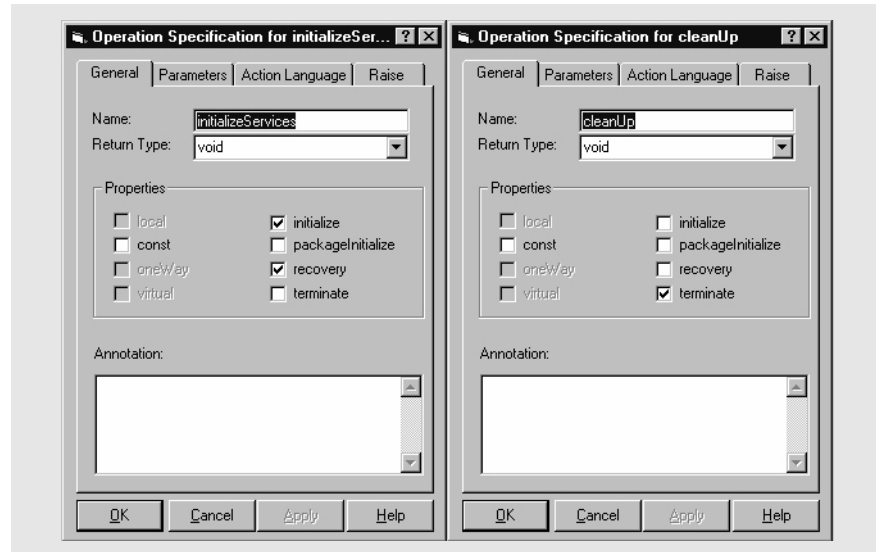


Figure 14: Setting lifecycle properties for an operation

IDLoS Here is IDLoS for the example above:

```
[local]
entity EngineEvents
{
    [initialize]
    void init();

    [
        initialize,
        recovery
    ]
    void initializeServices();
    [recovery]
    void checkNetConnections();

    [terminate]
    void cleanUp();
};
```

Automatic target instance All these lifecycle operations implicitly create a *new instance* of the entity and invoke the operation on that instance; the instance is destroyed when the operation completes. (If the entity is a singleton, the instance is created only if it does not already exist, and is not deleted when the operation completes.)



Although you can have lifecycle operations in any entity, it is better to use local entities only. The auditor will warn you if you use lifecycle operations on non-local entities.

Key

A key uniquely identifies an instance of an entity (see “Entities” on page 15). You define a key in an entity or an interface (see “Interface” on page 59). Defining a key in an interface enables cross-package access to the key in the exposed entity.

You define a key to use one or more attributes (see “Attribute” on page 23). Each key has a name. An entity may contain no more than three keys.

When you construct a query in action language, and you use all the attributes of the key in your where clause (see “select” in Chapter 9), ObjectSwitch optimizes the search. The key must evaluate to a unique value.

If a key appears in an interface, the identical key must exist in the entity that implements that interface. If an interface exposes all the attributes used in a key, but does not expose the key, the Design Center auditor will generate a warning.



You may not define a key in an entity that has any operations tagged with a lifecycle property (see “Lifecycle operations” on page 21).

There is currently no key support for attributes of these types: unbounded string, sequence, or array.

Visual Design Center

You manage an entity's keys in the Keys tab of the Class Specification dialog (see the figure below).

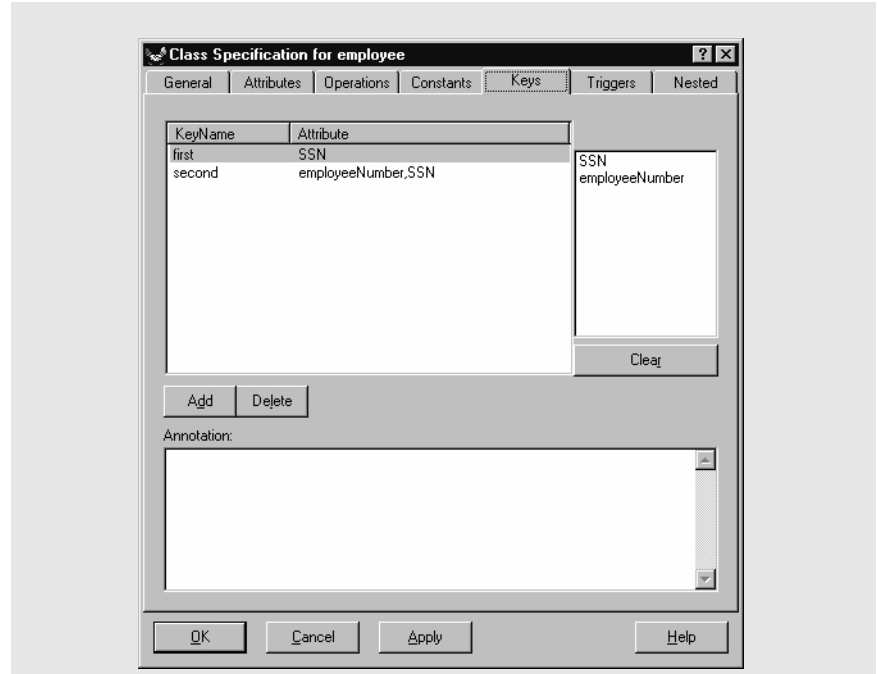


Figure 15: Keys in the Visual Design Center

The existing keys appear in the top-left pane. The key's label appears in the Keyname field, which you can edit directly. The attributes that make up the key appear in the Attribute field.

All of the entity's attributes that are available for constructing a key appear in the top-right pane.

To add a new key, click on the Add button.

To delete a key, select the key and click on the Delete button.

To add an attribute to a key, select the key and click on that attribute's name in the top-right pane.

To remove all attributes from a key's definition, select the key and click on the Clear button.

IDLos

Use the `key` statement (see "key" in Chapter 8 for syntax details). List the attributes comprising a key as a comma-separated list enclosed in braces.

```
entity employee
{
    attribute string firstname;
    attribute string middleinitial;
    attribute string lastName;
    attribute long SSN;
    attribute long employeeNumber;

    key first {SSN};
    key second {employeeNumber, SSN};
};
```

Relationship

Relationships define associations between entities. They let you link two instances, using `relate` in action language, or unlink them, using `unrelate`. When instances are linked, you can use action language to *navigate* across the links, to retrieve an associated instance, or retrieve a set. “Handling relationships” on page 115 has more information on `relate`, `unrelate` and navigation.

Relationships are a powerful abstraction, relieving you from writing numerous accessors and handling relationship integrity. The ObjectSwitch server optimizes link storage, set retrieval, and keyed searches. All the `relates` and `unrelates` are transactional. Relationship integrity is part of ObjectSwitch server transactions.

You can navigate a relationship from one object to the next, or back the other way. A relationship in ObjectSwitch has one or two *roles* defined to provide navigation in one or both directions.

Roles Roles have a name, a *from* entity, a *to* entity, and a multiplicity.

A relationship containing just one role is a *one-way* relationship, and cannot be navigated from the other side of the role. Only one-way relationships may cross package boundaries: the *to* entity may be in another package.

There is no guarantee that a relationship will be selected in the same order in which it was related. Ordering can be maintained with additional reflexive relations such as linked lists.

Visual Design Center

The Visual Design Center represents a relationship between two entities as a solid line. The name of the relationship appears next it. Role names and multiplicity appears next to the *to* entity.

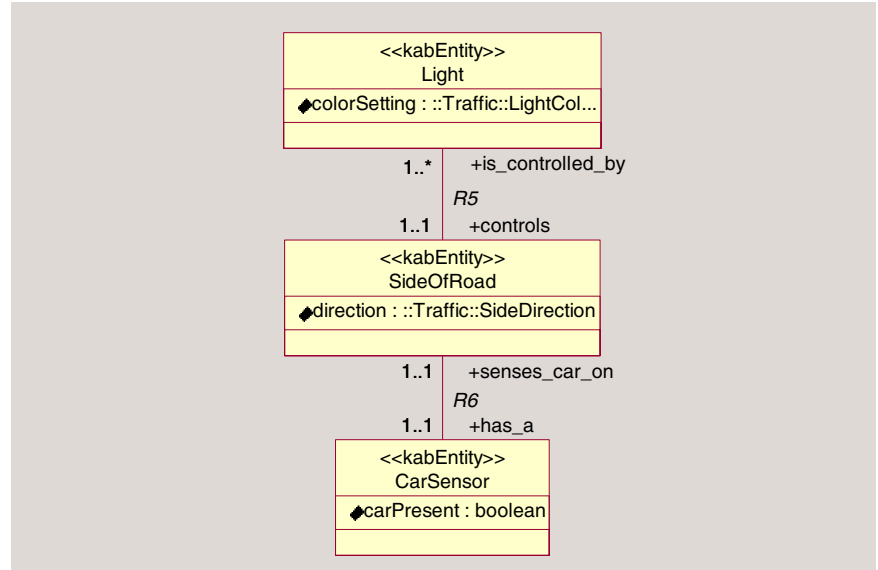


Figure 16: Relationship example

This example in the figure above defines two relationships. The textual descriptions are:

- Relationship R5: every SideOfRoad is controlled by one-or-more Lights, and each Light controls one-and-only-one SideOfRoad.
- Relationship R6: every CarSensor senses a car on one-and-only-one SideOfRoad, and every SideOfRoad has one-and-only-one CarSensor.

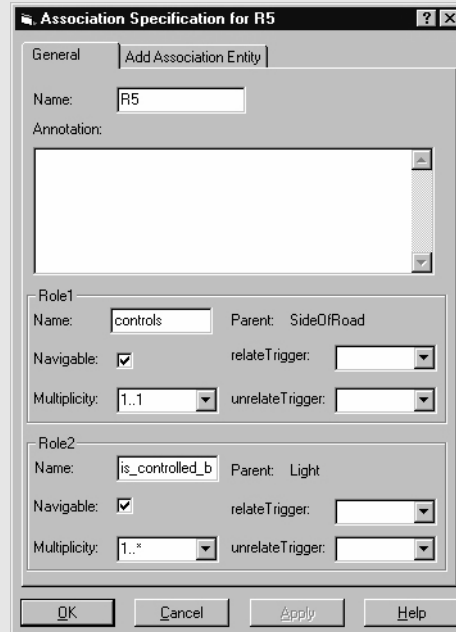
The example shows a role named controls. Light is the *from* entity; SideOfRoad is the *to* entity. And the multiplicity is 1..1.

To add a relationship between two entities:

- 1 select the Unidirectional Association icon in the toolbar

- 2 click and hold the mouse button pressed on one of the entities
- 3 drag the cursor to the second entity

To edit the relationship role names and multiplicity, open the Association Specification dialog (see the figure below). You can edit the Name field and select a value from the Multiplicity drop-down selection box for each role.



The image shows a dialog box titled "Association Specification for R5". It has two tabs: "General" and "Add Association Entity". The "General" tab is active. Inside the dialog, there is a "Name" field with the value "R5" and an "Annotation" text area. Below these are two sections for roles. "Role1" has a "Name" field with "controls", a "Parent" field with "SideOfRoad", a "Navigable" checkbox checked, a "Multiplicity" dropdown set to "1..1", and "relateTrigger" and "unrelateTrigger" dropdowns. "Role2" has a "Name" field with "is_controlled_b", a "Parent" field with "Light", a "Navigable" checkbox checked, a "Multiplicity" dropdown set to "1..*", and "relateTrigger" and "unrelateTrigger" dropdowns. At the bottom are "OK", "Cancel", "Apply", and "Help" buttons.

Figure 17: Association Specification dialog

You can also specify a relate and unrelate trigger for each role (see “Role trigger” on page 48 for more on role triggers).

IDLos

Use the relationship statement (for syntax details, see "Relationship" in Chapter 8.

Here is the IDLos for the example in Figure 16:

```
relationship R5
{
```

```

    role Light controls 1..1 SideOfRoad;
    role SideOfRoad is_controlled_by 1..* Light;
};
relationship R6
{
    role SideOfRoad has_a 1..1 CarSensor;
    role CarSensor senses_car_on 1..1 SideOfRoad;
};

```

Associative relationships

Sometimes, there is data associated with each link between two instances. For example, a separate marriage certificate belongs to every marriage, containing the wedding date, the location, and officiating official.

Visual Design Center The Visual Design Center displays an association between a relationship and an entity with a dashed line, as shown in the figure below.

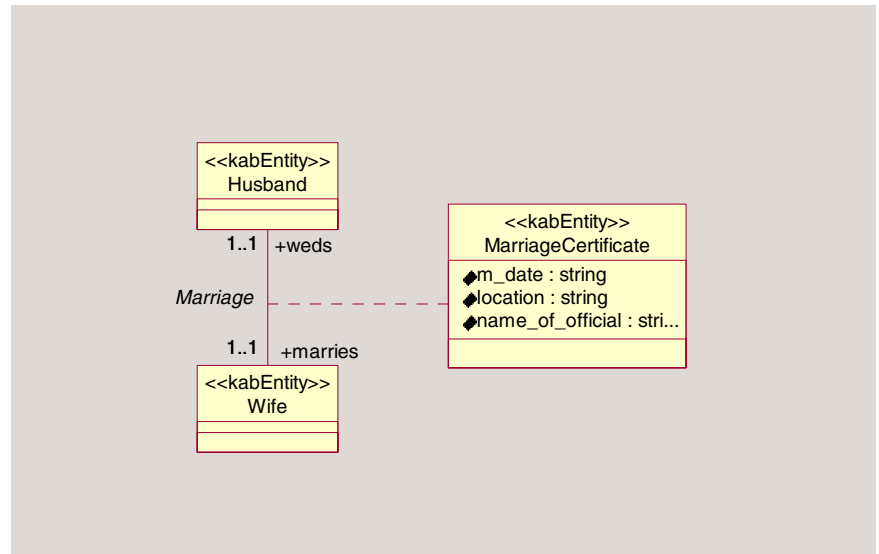


Figure 18: An associative relationship using an associative entity

To establish an association between a relationship and an entity:

- 1 right-click on the line representing the relationship
- 2 select Open Specification ... in the context menu

- 3 click on the Add Association Entity tab
- 4 select the entity from the Name drop-down list

IDLos Use the using statement to set up an association between a relationship and an entity.

```
entity Husband {};  
entity Wife {};  
entity MarriageCertificate  
{  
    string date;  
    string location;  
    string name_of_official;  
};  
relationship Marriage  
{  
    role Husband marries 1..1 Wife;  
    role Wife weds 1..1 Husband;  
    using MarriageCertificate;  
};
```

The using phrase in the relationship block indicates which entity to use for the association.

Entity trigger

Entity triggers allow you to specify operations that the runtime invokes at the onset of certain types of events, for example, when your application instantiates an entity or deletes an instance. You define an entity trigger within the context of an entity.

When you define an entity trigger, you can only use operations that you define directly in the entity or in a supertype.

Trigger type	Invokes the trigger operation when...
commit	a transaction in the entity commits (see “commit, abort” on page 43).
abort	a transaction in the entity aborts (see “commit, abort” on page 43).
create	you create an instance of entity type (see “create, refresh, state-conflict” on page 44).

Trigger type	Invokes the trigger operation when...
delete	you delete an instance of entity type (see “create, refresh, state-conflict” on page 44).
refresh	you refresh an instance of entity type, from a remote node or data store (see “create, refresh, state-conflict” on page 44).
state-conflict	a state conflict occurs.

commit, abort The trigger operation must be in the entity and satisfy the following conditions:

- it is two-way (cannot be a signal or oneway)
- its return type is void
- it has no parameters

Abort and commit triggers must be implemented by two-way operations, and need to be enabled in the action language in order for them to be called. e.g.:

```
action doit
{
    // enable abort trigger, then do stuff that
    // may deadlock.
    declare ::swbuiltin::ObjectServices os;
    os.enableAbortTrigger(self);
    GoDoSomething();
};
```



Abort and commit triggers are not invoked during recovery. When an engine recovers, any transactions that have begun to commit are committed, and all open transactions are aborted—but this does not invoke any triggers.



*Abort and commit triggers must not take any locks—because there is no valid transaction context for the lock.. Doing so causes a run-time error with unpredictable results. Be **very careful** when writing actions that implement abort and commit triggers. Refer to the section on locks in Advanced ObjectSwitch Modeling to understand what actions may take a lock.*

create, refresh, state-conflict The trigger operation must be in the entity and satisfy the following conditions:

- it is two-way, oneway, or signal
- its return type is void
- it has no parameters

If you specify initial values in a create statement (see "Manipulating data" in Chapter 3), then the create trigger fires *after* the values are set.

delete The trigger operation must be in the entity and satisfy the following conditions:

- it is two-way
- its return type is void
- it has no parameters



Create and delete triggers with inheritance: If you define create triggers for both the supertype (parent) and subtype (child), then the supertype trigger fires before the one in the subtype. Conversely, subtype delete triggers fire before those in a supertype.

Visual Design Center

You manage entity triggers in the Triggers tab of the entity's Class Specification dialog. There is a drop-down selection box for each type of entity trigger. The Visual Design Center only displays operations that satisfy the conditions for the respective trigger type (see the figure below).

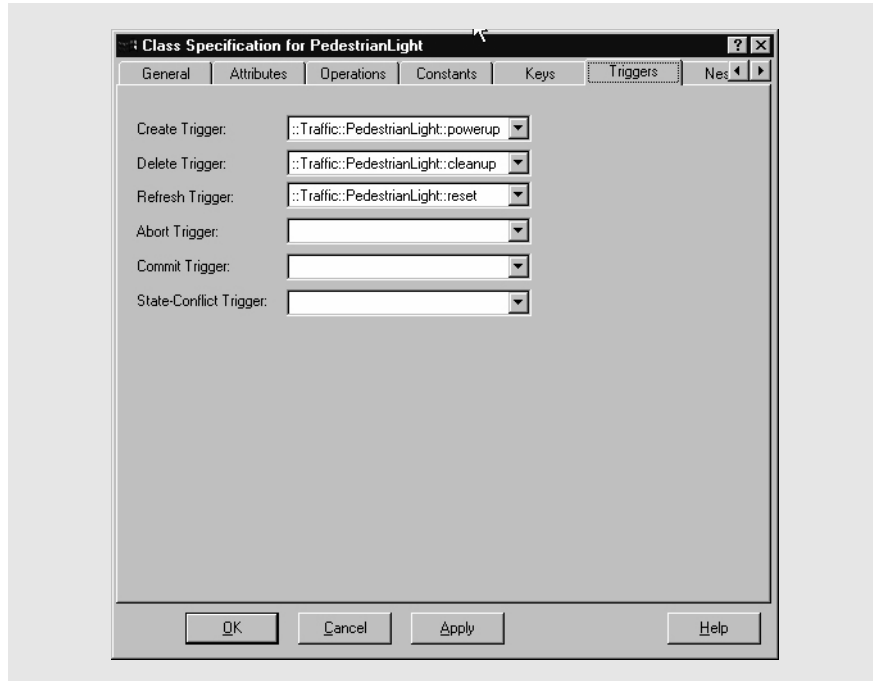


Figure 19: Action Language
and Raises tab

IDLos

Use the trigger statement (for syntax details, see "trigger" in Chapter 8).

The definition appears within the scope of the entity to which it applies.

The following IDLos shows several examples of entity triggers:

```
entity PedestrianLight
{
    // operations
    void powerup();
    void cleanup();
    void reset();
```

```
// triggers
trigger powerup upon create;
trigger cleanup upon delete;
trigger reset upon refresh;
};
```

Attribute trigger

Attribute triggers allow you to specify operations that the runtime invokes at the onset of certain types of events, for example, when your application sets or accesses an attribute value. You can define attribute triggers to occur both before or after an attribute is set, or before or after an attribute is retrieved.

Trigger type	Invokes the trigger operation when...
pre-get	before you access the value assigned to an attribute.
post-get	after you access the value assigned to an attribute.
pre-set	before you assign a value to an attribute.
post-set	after you assign a value to an attribute.

pre-get, post-get, pre-set, post-set The trigger must be defined in the same entity as the attribute to which it applies. The trigger operation can be inherited. The trigger operation must satisfy the following conditions:

- it is one-way, two-way, or a signal
- its return type is void
- it has no parameters

Visual Design Center

You manage attribute triggers in the Trigger tab of the Attribute Specification dialog. You can select an operation from the drop-down selection box for each attribute trigger type (see the figure below). The Visual Design Center only displays operations that satisfy the conditions for the respective trigger type.

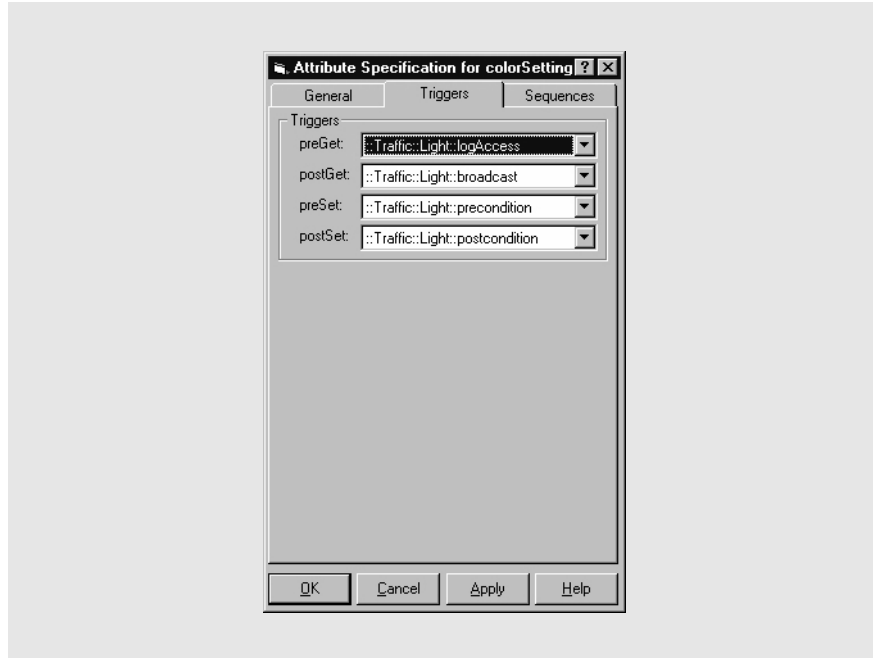


Figure 20: Action Language and Raises tab

IDLos

Use the trigger ... upon statement (for syntax details, see "trigger" in Chapter 8).

The following IDLos shows examples of each type of attribute trigger:

```
entity Light
{
    // attributes
    attribute LightColor colorSetting;

    // operations
    void precondition();
    oneway void postcondition();
}
```

```
oneway void broadcast();

// signals
signal logAccess();

// triggers
trigger precondition upon pre-set colorSetting;
trigger postcondition upon post-set colorSetting;
trigger logAccess upon pre-get colorSetting;
trigger broadcast upon post-get colorSetting;

};
```

Role trigger

Role triggers invoke operations when a role is related or unrelated. You specify role triggers for a specific role in a specific relationship; it must be defined for the same relationship as the role to which it applies.

The operation must have one parameter, which is the type of the *to* entity defined in the role. The operation must be defined in the *from* entity.

Trigger type	Invokes the trigger operation when ...
relate	you relate instances using the role.
unrelate	you unrelate instances related through the role, explicitly or when the <i>to</i> instance is deleted.

relate, unrelate The role trigger operation can be inherited. The role trigger operation must satisfy the following conditions:

- it is one-way, two-way, or a signal
- it is defined in the "from" entity
- its return type is void
- it has a single parameter of the "to" entity type, or the "associative" entity for associative relationships.



When two objects are related and one of them is deleted, the objects are implicitly unrelated. This invokes any unrelate trigger defined for the role “from” the remaining object. But if an unrelate trigger is defined “from” the deleted object, that trigger is not invoked by the implicit unrelate.

Visual Design Center

You manage role triggers in the General tab of the Association Specification dialog.

To define a relate role trigger for a relationship:

- 1 open the Association Specification dialog for the relation (right-click on the relationship and select Open Specification ...)
- 2 select the operation from the relateTrigger drop-down selection box

To define an unrelate role trigger for a relationship:

- 1 open the Association Specification dialog for the relation (right-click on the relationship and select Open Specification ...)
- 2 select the operation from the unrelateTrigger drop-down selection box

Consider the following model in the Traffic package.

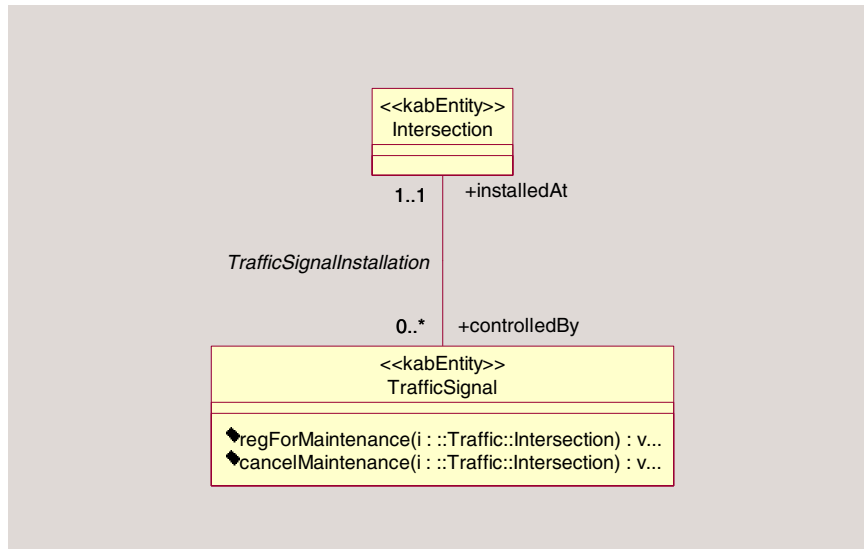


Figure 21: Relationship with operations for triggers

In the following figure, the regForMaintenance operation is specified as the relate trigger for the controlledBy role; the cancelMaintenance operation is specified as the unrelate trigger for the same role.

Figure 22: Relationship with operations for triggers

IDLos

Use the trigger ... upon statement (for syntax details, see "trigger" in Chapter 8.

Here is the IDLos for the example in Figure 21:

```
entity TrafficSignal
{
    oneway void regForMaintenance(in Intersection i);
    oneway void cancelMaintenance(in Intersection i);
};
entity Intersection
{
};
```

```
relationship TrafficSignalInstallation
{
    // roles
    role Intersection controlledBy 0..* TrafficSignal;
    role TrafficSignal installedAt 1..1 Intersection;

    // triggers
    trigger TrafficSignal::regForMaintenance
    upon relate controlledBy;
    trigger TrafficSignal::cancelMaintenance
    upon unrelate controlledBy;
};
```

Associative role triggers For associative relationships, you define the trigger operation in the “from” entity. This operation must take a single parameter whose type matches the associative entity.

Here is a modified version of the example for the marriage relationship presented in Figure 18.

In the Visual Design Center, to add a relate associative role trigger to the example:

- 1 add the trigger operation announceMarriage(in ::MarriageCertificate mc) to the Wife entity
- 2 open the Association Specification dialog for the Marriage relationship
- 3 select the trigger operation announceMarriage from the relateTrigger drop-down selection box

Here is the modified example in IDLs:

```
entity Wife
{
    // new operation as target for the trigger
    void announceMarriage(in MarriageCertificate mc);
};

relationship Marriage
{
    Husband marries 1..1 Wife;
    Wife weds 1..1 Husband;
    using MarriageCertificate;
    trigger Wife::announceMarriage upon relate weds;
};
```

```
action Pkg::Wife::announceMarriage
{
  // send out announcements
};
```

Module

Modules define a namespace. Inside a package, modules represent the main means of defining namespaces.

You can add a module to one of the following:

- Package
- Module

You can model anything in a module that you can in a package For a list of the elements see “Package”.

Visual Design Center

The Visual Design Center represents a module as a package with stereotype `kabModule`.



Figure 23: Relationship with operations for triggers

IDLos

Use the module statement (for syntax details, see "module" in Chapter 8).

Here is IDLos for the example above:

```
module myModule
{
  ...
};
```

Package

This section describes ObjectSwitch packages and how to use them. Everything in ObjectSwitch is organized into *packages*. Packages hide access to their entities, and each package is a namespace.

When you are implementing a package, you have open access to everything in your package. You also have open access to everything in other packages - *except* the entities. Entities are hidden inside their package. The only way to use an entity in another package is through an interface (see "Interface" on page 59).

ObjectSwitch packages may not be nested.

You can model all of the following within a package:

Entities	Class of objects.
Local entities	Non-distributable class of objects.
State machine	Alias for a basic or composite type.
Const	Named constant (on types, see Chapter 2).
ObjectSwitch types	Basic or composite type.
Relationship	Defines an association between the same or two different entities. You can link instances of related entities through a relationship.

Inheritance	Defines the inheritance relationship between two entities, the supertype and subtype; a subtype inherits the attributes and operations defined in the supertype.
Interface (and exposure)	Defines entity exposure across package borders.

Using Packages As a client of a package, you can:

- declare and create interfaces
- invoke exposed operations
- access exposed attributes
- navigate, relate, or unrelate exposed roles.
- implement abstract interfaces
- employ user-defined types

Visual Design Center

The Visual Design Center represents an ObjectSwitch package as a UML package with stereotype `kabPackage` (see the figure below).

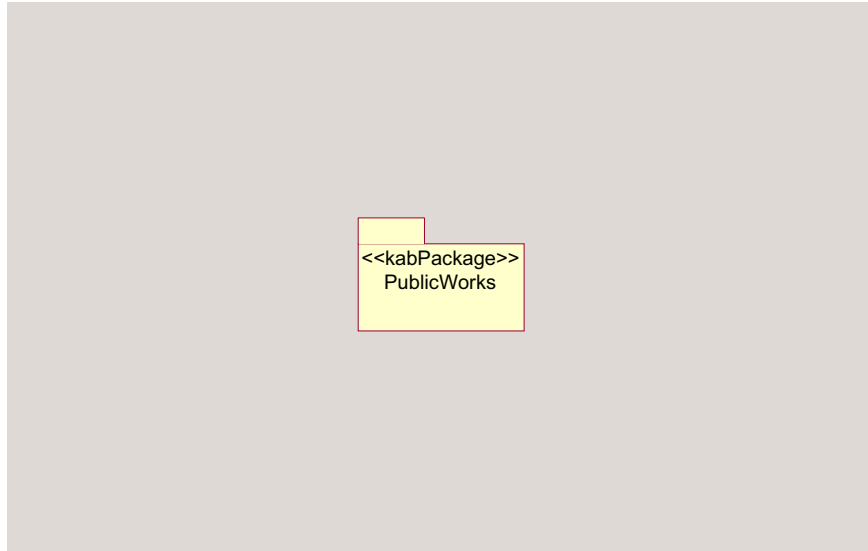


Figure 24: Package in the
Visual Design Center

To add a new package

- 1 click on the ObjectSwitch package icon in the toolbar
- 2 click anywhere in the Class Diagram: Logical View / Main window

IDLos

Use the package statement (for syntax details, see "package" in Chapter 8).

Here is the IDLos for the example above:

```
package PublicWorks
{
  ...
};
```

Example

Consider the PublicWorks package, which contains the following model, given first in its visual form, then in IDLos.

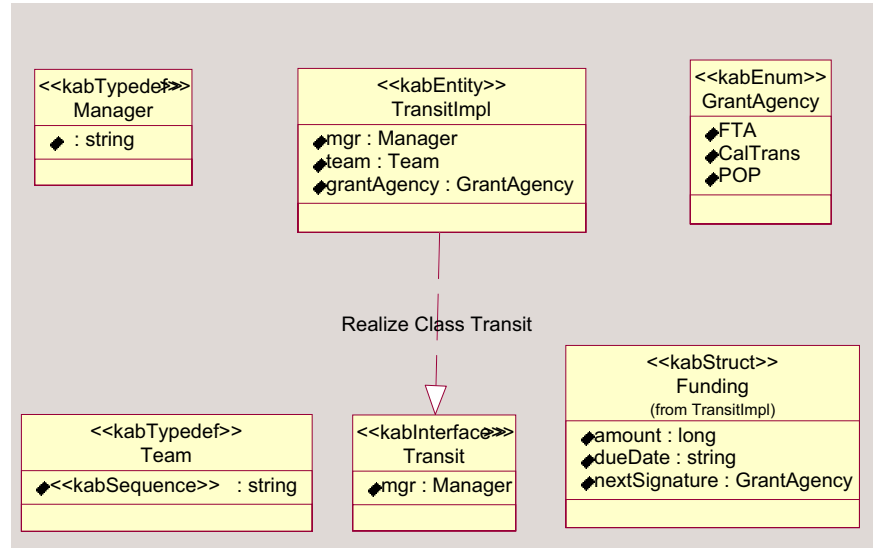


Figure 25: Package
PublicWorks

The same model in IDLos:

```

package PublicWorks
{
  typedef string Manager;
  typedef sequence<string> Team;
  enum GrantAgency { FTA, CalTrans, POP };
  entity TransitImpl
  {
    struct Funding
    {
      long amount;
      string dueDate;
      GrantAgency nextSignature;
    };

    attribute Manager mgr;
    attribute Team team;
    attribute GrantAgency grantAgency;
  };
  interface Transit
  {
    attribute Manager mgr;
  };
}

```

```
};  
    expose entity TransitImpl with interface Transit;  
};
```

A client of the PublicWorks package would have access to the Manager, Team, Transit, and GrantAgency types. But the client would *not* have access to TransitImpl type nor to the Funding structure nested in the entity.

The Transit interface does give the client access to the mgr attribute of TransitImpl through Transit.mgr. Interfaces are discussed in detail below.

Package properties

You can tag a package with properties that affect the IDLos exported for the package from the Visual Design Center. For example, you can specify IDLos statements that appear in the output *before*, *after*, and *inside* the IDLos package statement. You can also specify file names for exporting the IDLos.

The following table shows the properties you can use in the definition of an entity.

preInsert	IDLos statements to insert before the package
insert	IDLos statements to insert just inside the package
postInsert	IDLos statements to insert after the package
packageFilename	Name and path of file for exporting IDLos.
actionFilename	Name and path of file for exporting action language

Setting package properties In the Visual Design Center, you set package properties by opening a dialog directly from the package.

To open the Package Specification dialog, right-click on the package and select Kabira ObjectSwitch->ObjectSwitch Package Specification. In the dialog that appears, set the properties you want to apply to the package.

Interface

To give a client of your package access to an entity, you must provide an *interface*. The interface *exposes* the entity. The contents of the interface controls which attributes, operations, keys, and signals are exposed.

Each interface may expose only one entity. An entity may be exposed by more than one interface. Each operation or attribute in the interface must have a corresponding attribute, operation, or signal in the entity that it exposes.

An interface is an alias of the entity it exposes. You can use an interface type wherever you may use the entity type.

The following table shows the elements of an entity that an interface can expose:

Attribute	Attribute in the entity that the interface exposes.
Key	Key in the entity that the interface exposes.
Operation	Operation in the entity that the interface exposes.
Signal	Signal in the state machine of the entity that the interface exposes.

Nested types The nested types constraints that apply to entities also apply to interfaces (see “Nested types” on page 16).

Namespace The namespace constraints that apply to entities also apply to interfaces (see “Namespace” on page 17).

Association The association constraints that apply to entities also apply to interfaces (see “Relationships” on page 17).

Instantiation Instantiating an interface creates an object of the exposed entity type. You instantiate an interface with the same action language statements used to instantiate entities. (See “Creating objects” in Chapter 2 for details.)

Exposure The following table lists the elements that you can expose with an interface and shows how each element is exposed:

To expose ...	You must ...
an attribute	declare the same attribute in the interface (see “Attribute” on page 23). The attribute must match the entity’s attribute in name and type. Attribute access can also be marked readonly in an interface, exposing only the get accessor outside the package.
a key	declare the key just like it is declared in the underlying entity.
a signal	declare a oneway void operation of the same name and signature as the signal in the entity (see “Signal” on page 73).
an operation	declare an operation having the same name, signature (see “Operation” on page 26), and return type as the one in the entity. It must also match in its local properties. Operations marked oneway in an entity must also be marked oneway in an exposing interface. An operation in an interface must not be marked virtual.

To expose ...	You must ...
a relationship between entities	define it between interfaces (see “Relationship” on page 38). Do <i>not</i> define the same relationship between the corresponding entities; defining it between the interfaces also defines it for the entities.
user-defined types (such as structs, enumerations, and typedefs) defined in an entity	<p>move the type definition from the entity to one of its interfaces (on types, see Chapter 2).</p> <p>A type defined in an entity is hidden from other packages. A type defined in package scope, module scope, entity scope, or an interface scope is automatically exposed to other packages.</p> <p>Types defined in an interface are exposed, just as everything else defined in the interface is exposed.</p>

Interfaces and adapters ObjectSwitch adapter factories can generate different code for the same interfaces. This is why more than one interface per entity is allowed.

In the following model, three interface expose the same entity, one for each of the technologies SNMP, CORBA and other.

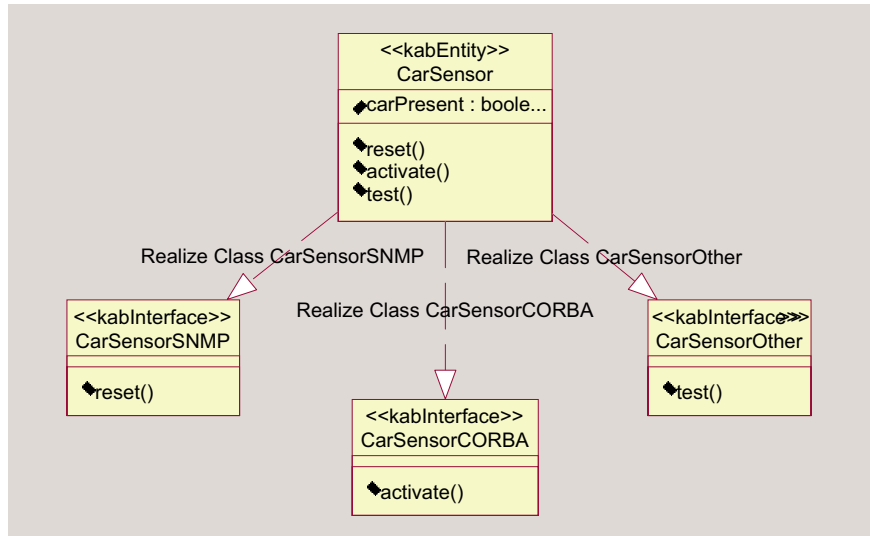


Figure 26: Interface in the Visual Design Center

Here is the IDLoS for the same model:

```

entity CarSensor
{
    void reset();
    void activate();
    void test();
};
interface CarSensorForSNMP
{
    void reset();
};
interface CarSensorCORBA
{
    void activate();
};
interface CarSensorOther
{
    void test();
};
expose entity CarSensor with interface CarSensorSNMP;
expose entity CarSensor with interface CarSensorCORBA;
expose entity CarSensor with interface CarSensorOther;
    
```


In the Design Center, the adapter could generate SNMP agent code for reset, CORBA server code for activate, and code for another technology for the test operation.

Visual Design Center

The Visual Design Center represents an interface as a class with stereotype `kabInterface`. An interface has three sections: the header at the top (with stereotype and interface label), the attribute section in the middle, and the operation section at the bottom (see the figure below).

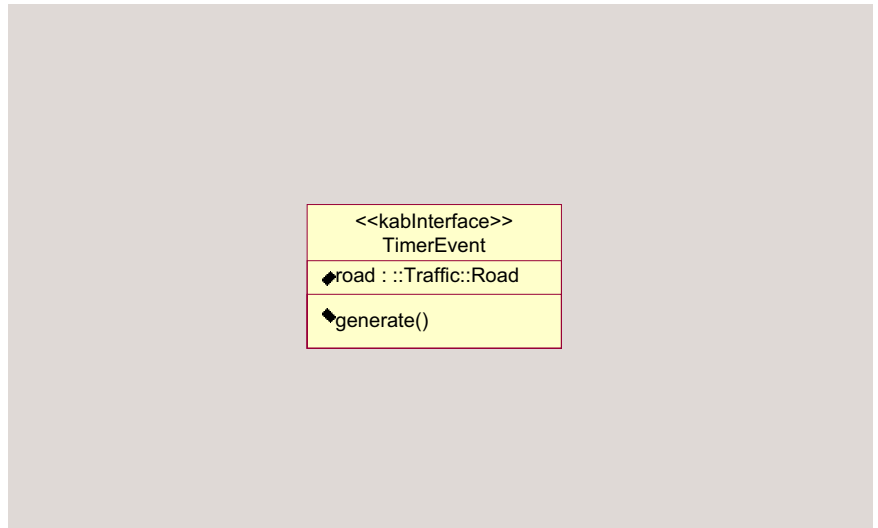


Figure 27: Interface in the Visual Design Center

To add a new interface to a module or package:

- 1 click on the interface icon
- 2 click somewhere in the package or module diagram

To expose an entity with an interface:

- 1 click on the realizes icon
- 2 click on the entity you want to expose and while holding the mouse button pressed, drag the cursor to the exposing interface

Once you have added an interface to a module or package, you model its properties, attributes, operations, constants, keys, triggers and nested types in its Class Specification dialog (see “Class Specification dialog” on page 17).

IDLos

Use the interface statement (for syntax details, see "interface" in Chapter 8).

Here is the IDLos for the example above:

```
interface TimerEvent
{
    attribute ::Traffic::Road road;
    void generate();
};
```

To expose an entity with an interface, use the expose statement (for syntax details, see "expose" in Chapter 8).

For example, to expose the entity TimerEventImpl with the TimerEvent interface, defined above, use the following IDLos:

```
expose entity TimerEventImpl with interface TimerEvent;
```

Interface properties

An interface can be tagged by properties that affect exposure. Also, an interface can expose a singleton entity only if it is a singleton and a local entity only if it is a local interface.

The following table shows the properties you can use in the definition of an interface.

isAbstract (abstract in IDLos)	Prohibits the interface from exposing an entity (see “abstract” on page 65).
createaccess	Permits and revokes permission to instantiate the exposed entity through the interface (see “Access control” on page 65).
deleteaccess	Permits and revokes permission to delete instances of the exposed entity through the interface (see “Access control” on page 65).

extentaccess	Permits and revokes permission to access the extent of the exposed entity through the interface (see “Access control” on page 65).
singleton	If an interface exposes a singleton, it must also have the singleton property (see “singleton” on page 19).
local (IDLos only)	If an interface exposes a local entity, it must also have the local property (see “Local entities” on page 21 and “Local Interface” on page 66)

Setting interface properties To set interface properties in the Visual Design Center, open the interface’s Class Specification dialog (see “Class Specification dialog” on page 17).

In IDLos interface properties appear before the interface statement in square bracket, for example,

```
[ abstract ]
interface TalkBack
{
    oneway void respond(in string response);
};
```

abstract An interface marked with the abstract property cannot be used in an *exposes* statement. Abstract interfaces define the operations which a client package should both inherit and implement. In other words, abstract interfaces define *callbacks* or *notifiers*.



Abstract interfaces may not contain attributes or typedefs.

“A big example” on page 93 contains an example of defining and using abstract interfaces.

Access control To permit more control over how an entity is exposed, three properties grant or revoke access to create, delete, or retrieve the extent of an interface. In this example, a client can retrieve handles to all the Crosswalks instantiated, but the client cannot create or delete any of them.

[

```

        createaccess=revoked,
        deleteaccess=revoked,
        extentaccess=granted
    ]
    interface Crosswalk {};

```

In this example, new BankCustomers can be created. But for security, clients cannot search through the existing customers, and they cannot delete any customers.

```

[
    createaccess=granted,
    deleteaccess=revoked,
    extentaccess=revoked
]
interface BankCustomer {};

```

Local Interface

To give a client of your package access to a local entity, you must provide an *interface*. The local interface *exposes* the local entity. The contents of the interface controls which operations are exposed.

To expose an operation, the local interface defines the same operation.

The following table shows the elements of a local entity that a local interface can expose:

Operation	Operation in the entity that the local interface exposes.
-----------	---

Visual Design Center

The Visual Design Center represents a local interface as a class with stereotype `kabLocalInterface`. A local interface has three sections: the header at the top (with stereotype and local interface label), the attribute section in the middle (which is always empty), and the operation section at the bottom (see the figure below).

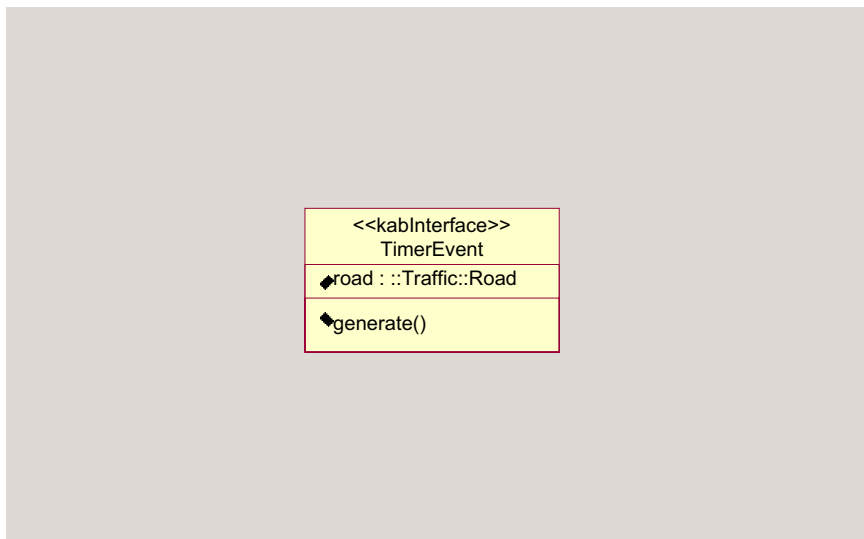


Figure 28: Local Interface in the Visual Design Center

To add a new local interface to a module or package:

- 1 click on the interface icon
- 2 click somewhere in the package or module diagram
- 3 open the local interface's Class Specification dialog (see "Class Specification dialog" on page 17)
- 4 select `kabLocalInterface` from the stereotype drop-down selection box

To expose a local entity with a local interface:

- 1 click on the realizes icon
- 2 click on the local entity you want to expose and while holding the mouse button pressed, drag the cursor to the exposing local interface

Once you have added a local interface to a module or package, you model its operations and nested types in its Class Specification dialog (see “Class Specification dialog” on page 17).

IDLos

Use the interface statement (A local interface is an interface tagged with the local property (for syntax details, see “interface” in Chapter 8).

```
[local]
entity UtilitiesImpl
{
    void who();
    void whatis();
};
[local]
interface Utilities
{
    void who();
    void whatis();
};
```

To expose a local entity with a local interface, use the expose statement (for syntax details, see “expose” in Chapter 8).

For example, to expose the entity UtilitiesImpl with the Utilities interface, defined above, use the following IDLos:

```
expose entity UtilitiesImpl with interface Utilities;
```

State machine

State machines define the *stages* of an instance’s life as well as the transitions between those stages. Use them when the instances of an entity must perform duties in a certain order, or when an instance must wait for certain signals before it can proceed. State machines are the best way to manage asynchronous communication with another instance or with a network protocol.

You can define a finite state machine for an entity. An action is executed as an instance enters each state. There are no actions associated with the signals or transitions. The actions are implemented in action statements using action language (on action language, see Chapter 3).

The following table shows the modeling elements that you use to model an entity's state machine:

State	A condition or situation during the life of an instance. An instance remains in a state until it receives a signal.
Signal	An event causing a state transition.
Transition	A progression from one state to another, or the same, state caused by a signal.

Visual Design Center

You model state machines in a Statechart diagram. You create a Statechart diagram for an entity by right-clicking on the entity and selecting Sub Diagrams > New Statechart Diagram. You open an exiting Statechart Diagram by right-clicking on the entity and selecting the diagram name.

In a Statechart diagram you model a state machine by adding new states (see “State”) and the transitions (see “Transition”) between them.



You can set the default state transition to either cannot happen or ignore using the State Diagram Specification dialog. Right-click on the State Chart Diagram and select ObjectSwitch State Diagram Specification.

A complete state machine and the UML representation of it is shown in the following figure.

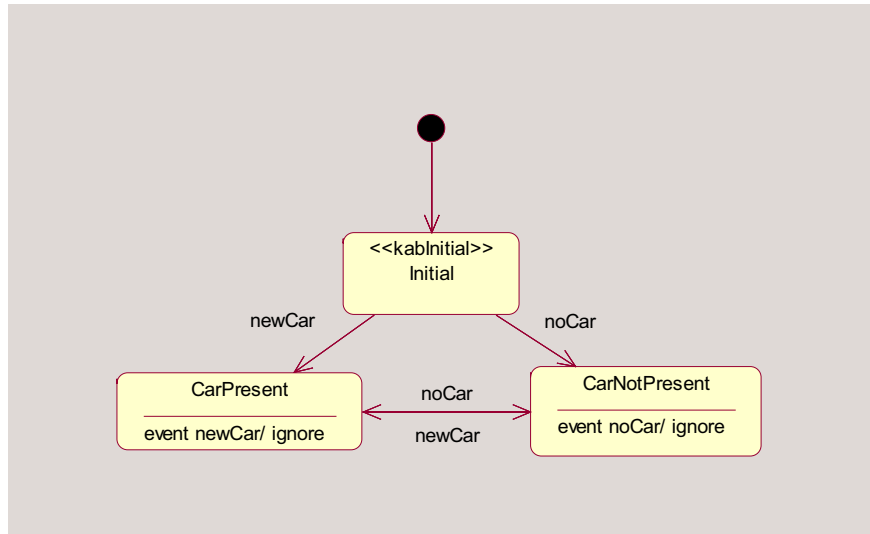


Figure 29: State model for car entity in sample model

IDLos

A state machine consists of a stateset statement and a group of transition statements, within the scope of the entity.

The stateset statement specifies all the states valid for the entity as well as the initial state and final states (for syntax details, see “stateset” in Chapter 2).

A transition statement specifies the *from* state and the *to* state as well as the signal initiating the transition.

```
entity CarSensor
{
    // attributes
    attribute boolean carPresent;

    // states
    stateset
    {
        Initial,
        CarPresent,
        CarNotPresent
    } = Initial;
```



```

// signals
signal newCar( );
signal noCar( );

// transitions
transition Initial to CarPresent upon newCar;
transition Initial to CarNotPresent upon noCar;
transition CarPresent to CarNotPresent upon noCar;
transition CarNotPresent to CarPresent upon newCar;
transition CarPresent to ignore upon newCar;
transition CarNotPresent to ignore upon noCar;
};

```

State

You use states to model state machines (see “Module”). A state represents a stage in the life of an instance. An instance stays in the same state until it receives a signal causing it to transition to a different state.

There are two special types of states:

- initial state
- final state

There can only be one initial state. This is the state that an instance first enters upon creation.



The state action for the initial state is executed following a transition back to the initial state, but not when the object is created. Actions on create can be expressed using create triggers—see “Entity trigger” on page 42.

There is no limit to the number of end states. When an instance enters an end state, it ceases to exist; the runtime deletes the instance.

Actions You implement operations and states in action statements.

```

action ::Traffic::CarSensor::CarNotPresent
// arguments: None
{
    self.carPresent = false;
};

```

Actions specify the name of an operation or a state. They can be defined inside an entity, outside the entity, or even outside the package, so use a scoped name when needed. The action language that implements the action appears between the backquotes (‘) in IDLos. See Chapter 3 for a description of the ObjectSwitch action language.

You must define an action statement for each operation and state. Although empty actions are legitimate, forcing you to write them explicitly reminds you to define an action where one is needed.

Everything between the backquotes is handled by a different parser from the rest of IDLos. This is why the IDLos namespace does not have reserved words from the action language, and vice-versa.

No parameter signature is required with the action statement..

Actions can be defined outside of packages so you can organize them into separate files, if you choose.

Visual Design Center

There are three different state icons in the Statechart diagram toolbar:

- State
- Initial State
- End State (final)

However, you only use the State and End State icons. An End State in the Visual Design Center corresponds to a state with the finished property of the enclosing stateset in IDLos. If you have multiple end states, their names will appear as comma-separated values for the finished property of the enclosing stateset.



UML start state semantics are different from the Kabira initial state's semantics. UML start states are "pseudo states"; they cannot have events on their outgoing transitions. Kabira initial states are like any other state (they can have actions, events on outgoing transitions, and transitions incoming and outgoing).

To add a state or final state:

- 1 click on the State or Final State icon in the toolbar
- 2 click somewhere in the Statechart diagram

To add an initial state, use a plain state with the `<<kabInitial>>` stereotype.

IDLos

The `stateset` statement defines all the states for an entity, as well as the initial and final states (for syntax details, see “`stateset`” in Chapter 2).

The initial state is the state that appears after the equals sign in the `stateset` statement.

finished This property designates the terminal state(s) in a `stateset`. It requires a list of states, for example:

```
[finished = {Retired, Lost}]
stateset {Made, Used, Retired, Lost} = Made;
```

In this example, ObjectSwitch automatically deletes the object after the `Retired` or `Lost` state finishes executing its action.

Signal

A signal causes a transition to a new state.

Only in parameters are allowed. A signal does not have a `raises` clause.

Visual Design Center [description not supplied in this edition — tracked as issue 020328-000009]

IDLos The `signal` statement defines the name of the signal and an optional list of parameters.

```
signal newCar( );
signal noCar( );
signal carLost(in string registration);
```

Transition

A transition specifies that an instance moves from one state to another when a certain signal is received. In the source state will perform certain specified actions and enter the destination state when a specified event occurs or when certain conditions are satisfied. A state transition is a relationship between two states, two activities, or between an activity and a state.

```
transition Initial to CarPresent upon newCar;  
transition Initial to CarNotPresent upon noCar;  
transition CarPresent to CarNotPresent upon noCar;  
transition CarNotPresent to CarPresent upon newCar;
```

The first transition above says that if an object is in the Initial state and the newCar signal is received, transition to the CarPresent state and execute its action.

If a signal is received, and there is no transition from the current state for that signal, an ObjectSwitch system exception is thrown by default. This causes the engine to exit with an error. You can explicitly express the default behavior by specifying a transition to cannot happen.

```
transition CarPresent to cannot happen upon lostCar;
```

The transition cannot happen (whether expressed explicitly or by default) indicates a condition that should never be possible in your model. Do *not* use this transition for expected error conditions.

To ignore signals when you are in a certain state, specify a transition to ignore.

```
transition CarPresent to ignore upon newCar;  
transition CarNotPresent to ignore upon noCar;
```



If two or more transitions to the same state occur upon different signals, those signals must all have the same parameter signature.

To relate

[description not supplied in this edition — tracked as issue 020328-000009]

```
entity TrafficSignal  
{  
    oneway void regForMaintenance(in Intersection i);  
    oneway void cancelMaintenance(in Intersection i);  
};  
entity Intersection
```

```

{
};
relationship TrafficSignalInstallation
{
    // roles
    role Intersection controlledBy 0..* TrafficSignal;
    role TrafficSignal installedAt 1..1 Intersection;

    // triggers
    trigger TrafficSignal::regForMaintenance
    upon relate installedAt;
    trigger TrafficSignal::cancelMaintenance
    upon unrelate installedAt;
};

```

Inheritance

Like most object-oriented languages, ObjectSwitch supports inheritance. This section describes how inheritance works in ObjectSwitch and discusses

- Entity inheritance
- Operations
- Interface inheritance

Entity inheritance

ObjectSwitch entities may inherit from other entities. This means that the subtype can share all the features of the supertype entity. The subtype *inherits* from the supertype.

The following constraints apply:

- a subtype may have only one supertype
- an entity may inherit from another entity if they are in the same package
- local entities may inherit only from other local entities

What do subtypes inherit? A subtype inherits these items:

- attributes
- operations; operations will be discussed in detail below
- roles; any role you can navigate, relate or unrelate on a supertype, you can navigate, relate or unrelate through a subtype
- state machines; subtypes inherit all signals, transitions; all states and their actions; a subtype cannot override the state machine of its supertype
- keys
- lifecycle (create/delete) and attribute triggers

What can subtypes add? A subtype may add:

- attributes, with triggers if desired
- keys
- operations, both virtual and non-virtual
- a state machine, if it did not inherit one; you cannot add to an existing state machine in a subtype; the state machine must be defined entirely within one entity.

Visual Design Center The Visual Design Center represents inheritance with a solid line from the subtype to the supertype. The tip of the line is a large hollow triangle pointing to the supertype.

For example, in the following model the subtype `RoadSide1` inherits from the supertype `SideOfRoad` (see the figure below). This means that all instances of `RoadSide1` will share the attribute `direction` and its accessors. It also means that anywhere you can use the type `SideOfRoad`, you can use the type `RoadSide1`.

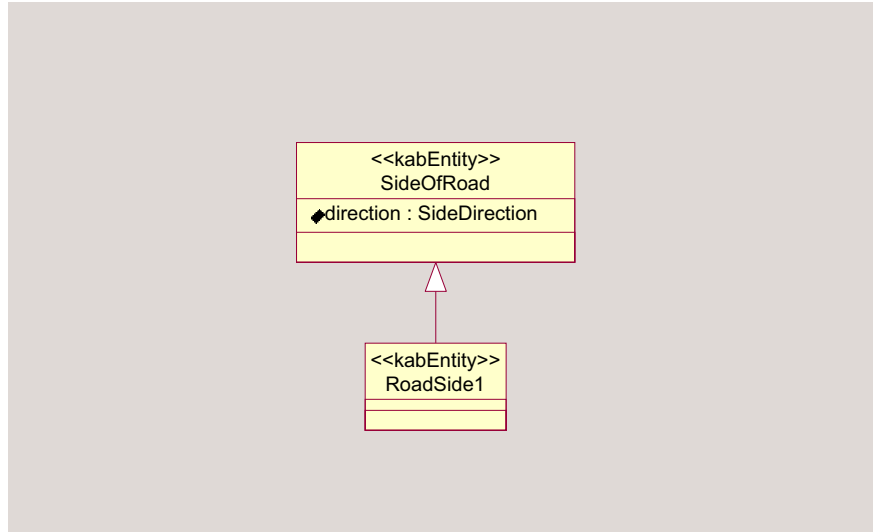


Figure 30: `RoadSide1` inherits from `SideOfRoad`

To add an inheritance relationship between two entities (or interfaces):

- 1 click on the Generalization icon in the toolbar
- 2 click and hold the mouse on the subtype
- 3 pull the mouse to the supertype and release

IDLos In IDLos use the colon to establish an inheritance relationship between two entities: `subtype : supertype`.

Here is the IDLos for the model defined in Figure 30:

```

entity SideOfRoad
{
    attribute SideDirection direction;
};
entity RoadSide1 : SideOfRoad
{
};
  
```

Shadow types Nested types defined in a supertype are accessible to a subtype. Consider the model where struct B is defined within the scope of entity A; entity C inherits from entity A; struct B is accessible inside entity C, for example, as a type in an attribute definition of b.

This situation is represented in the following model

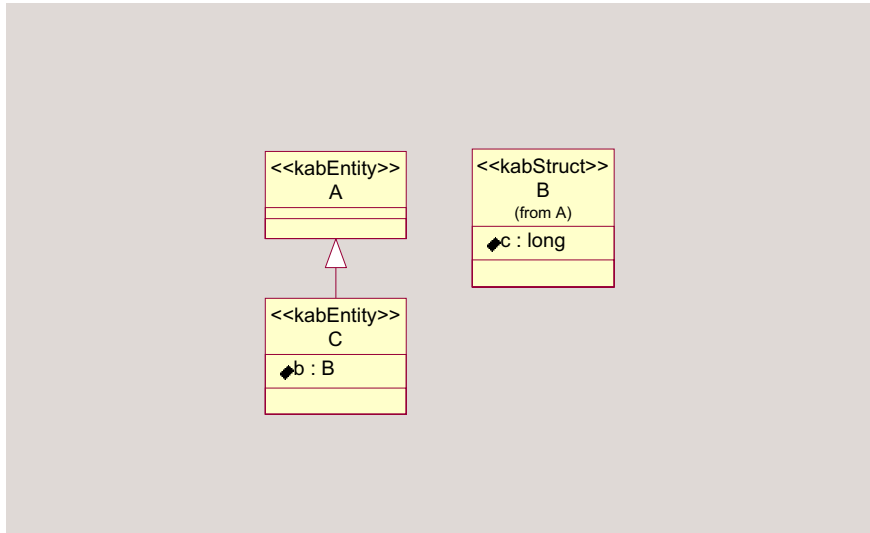


Figure 31: Inheritance and shadow types

Here is the same model in IDLs:

```
entity A
{
    struct B
    {
        long c;
    };
};
entity C : A
{
    attribute B b;
};
```

The example shows how entity C can use type B without any scoped name. These inherited, nested types are called *shadow types*.

Operations

All regular (non-virtual) operations are inherited: you can invoke the supertype's operation through the subtype. The supertype's implementation of the operation will be executed.

In the next model the entity Supertype has an operation named `greeting`. The implementation of `greeting` is to print "Hello" to stdout.

```
printf("Hello\n");
```

The entity Subtype inherits from Supertype, as shown in the figure below.



Figure 32: Supertype and Subtype

There is a third entity in the model called `Startup` that has an operation called `init`. `init` is a lifecycle operation that the runtime invokes when the component is initialized. The implementation of `init` is to create an instance of `Subtype` and invoke the operation `greeting` on the instance.

```
declare Subtype sub;
create sub;
sub.greeting();
```

Invoking `greeting` on the instance of `Subtype` sends "Hello" to stdout.

Here is the same model defined in IDLs:

```
// define the supertype
entity Supertype
{
    void greeting();
    action greeting
    {
        printf("Hello\n");
    };
};

// define the subtype
entity Subtype: Supertype
{};

// A local entity to start the engine
[local] entity Startup
{
    [initialize]
    void init();
};
action Startup::init
{
    // invoke through the subtype
    declare Subtype sub;
    create sub;
    sub.greeting();
};
```

Redefining operations When an inherited operation is redefined in a subtype, ObjectSwitch normally invokes either the supertype's or subtype's implementation, depending on the type of the object handle.

In the following model, the operation parting is defined in the supertype and its subtype, as shown in the figure below.

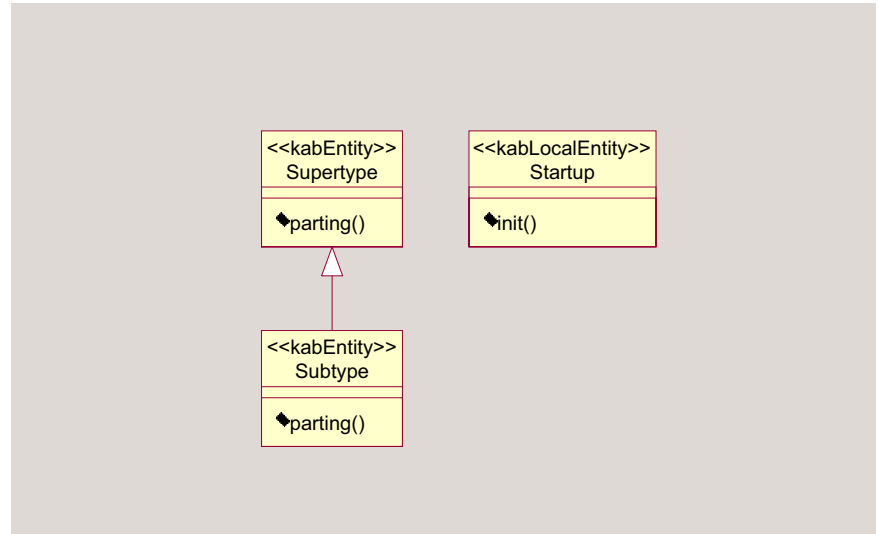


Figure 33: Supertype and Subtype

In the Supertype, parting sends “Bye ...” to stdout and in Subtype parting sends “Cheers ...”.

init declares an object handle of type Supertype and one of type Subtype. It creates an instance of Subtype and assigns it to the Supertype. Finally, it invokes parting first on the supertype instance and then on the subtype instance.

Here is the action language implementation of init:

```

// Create subtype
declare Subtype sub;
create sub;
// Make a supertype handle to the subtyped object
declare Supertype super;
super = sub; // these now refer to the same object

// Invoke via supertype and subtype
super.parting();
sub.parting();
  
```

The result of invoking init is “Bye...Cheers...”. Calling super.parting() prints “Bye...”, and then calling sub.parting() prints “Cheers...”.

Here is the same model in IDLs:

```
// define the supertype
entity Supertype
{
    void parting();
    action parting
    { ' printf("Bye..."); '};
};

// define the subtype
entity Subtype: Supertype
{
    void parting();
    action parting
    { ' printf("Cheers..."); '};
};
[local] entity Startup
{
    [initialize]
    void init();
};

action Startup::init
{ '
    // Create subtype
    declare Subtype sub;
    create sub;

    // Make a supertype handle to the subtyped object
    declare Supertype super;
    super = sub;    // these now refer to the same object

    // Invoke via supertype and subtype
    super.parting();
    sub.parting();
    '};
```

When executed, this model prints “Bye...Cheers...”. Calling `super.parting()` prints “Bye...”, and then calling `sub.parting()` prints “Cheers...”.

Virtual operations and polymorphic dispatch Sometimes you don’t want to execute the supertype’s implementation—you want to use an object as a supertype, but when you call an operation, you want to invoke the subtype implementation that corresponds to the actual object. This is called *polymorphism*.

IDLos's virtual property provides polymorphic dispatching of ObjectSwitch operations to the correct subtype implementation. The following example shows how virtual changes the way operations are dispatched. The preceding example printed "Bye...Cheers..." but this one prints "Cheers...Cheers" because it uses virtual.

The parting operation is declared virtual in the Supertype entity, as shown in the figure below.

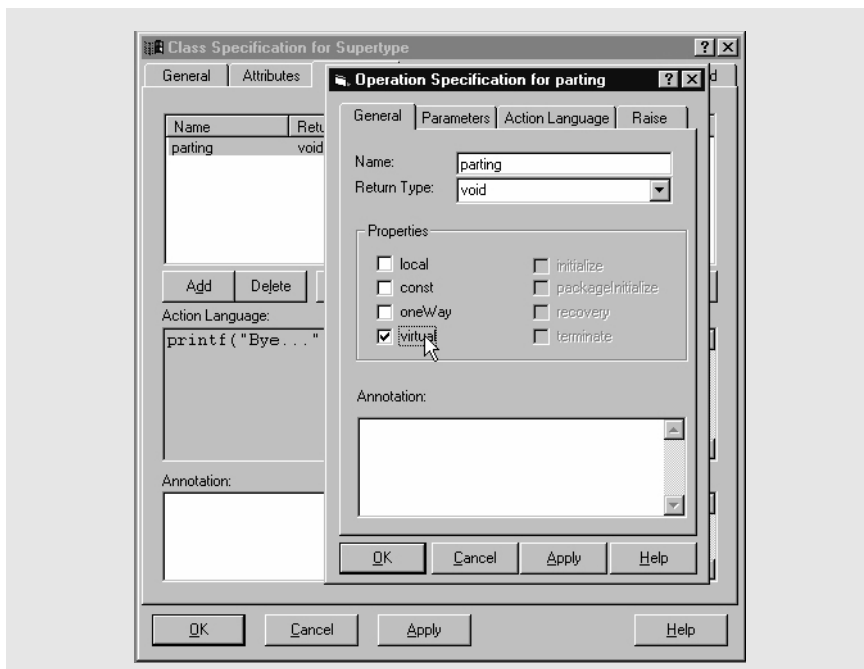


Figure 34: parting becomes virtual

Now when init executes it invokes the subtype's implementation twice because both object handles reference a subtype object.

In IDLos, you make the same change to the model by prepending the parting operation definition in the Supertype entity with [virtual], as shown below:

```
// define the supertype
entity Supertype
{
    [virtual] void parting();
    action parting
    { ' printf("Bye\n"); '};
};
```



There is no “pure virtual” in IDLos. When you mark an operation virtual, you still need to implement it in the supertype. You also must implement the operation for all subtypes.

Also, local entities cannot have virtual operations.

You can use the virtual property at any level of an inheritance hierarchy, but you can’t use it again further down the hierarchy.

In the following example (see the figure below), anOp is virtual in Middle and a polymorphic dispatch will occur on instance handles of Middle and Bottom, but not Top.

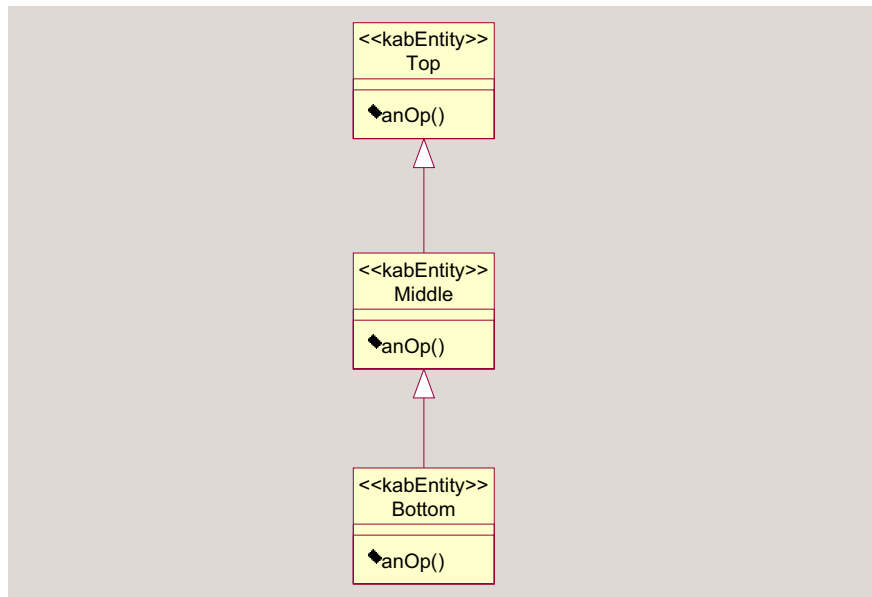


Figure 35:

Here is the same model in IDLs, which shows the virtual property:

```
entity Top
{
    void anOp();
};
entity Middle : Top
{
    [virtual]
    void anOp();
};
entity Bottom: Middle
{
    void anOp();
};
```

Adding an entity Floor that inherits from Bottom and then declaring anOp as virtual in Floor will create an error, since virtual has already been specified on the Middle level.



Don't try to make a subtype call the supertype implementation of a virtual operation. The supertype always invokes the subtype's implementation. This has important consequences: for the example above, the following implementation will cause an endless loop at runtime.

```
action Bottom::anOp
{
    declare Middle mid;
    mid = self; // upcast
    mid.anOp(); // will recurse forever
};
```

Instead of trying to invoke the virtual operation in the base entity directly, you can make it easy for subtypes to use the supertype's implementation using the following style:

```
entity Base
{
    string getDefaultString();
    [virtual] string getString();
};
entity Child : Base
{
    string getString();
    action getString
    {
        declare Base base;
        base = self;
        return base.getDefaultString();
    };
};
```

In this way, the base class provides both a default implementation and an operation that must be overridden by subtypes.

Interface inheritance

Interfaces can inherit from other interfaces in the same package or in other packages. The subtype inherits all interface definitions from the supertype.

Generally, the following rules apply to new definitions. The interface subtype can:

- introduce new types
- expose additional attributes
- expose additional operations
- restrict control access, but cannot grant more

Operations and attributes must expose an entity that is a subtype of the entity exposed by the supertype (where an entity is considered a subtype of itself)

Inheritance within a package When the supertype and subtype interfaces are in the same package, the subtype interface must expose an entity that is a subtype of the entity exposed by the supertype interface, where an entity is considered a subtype of itself.

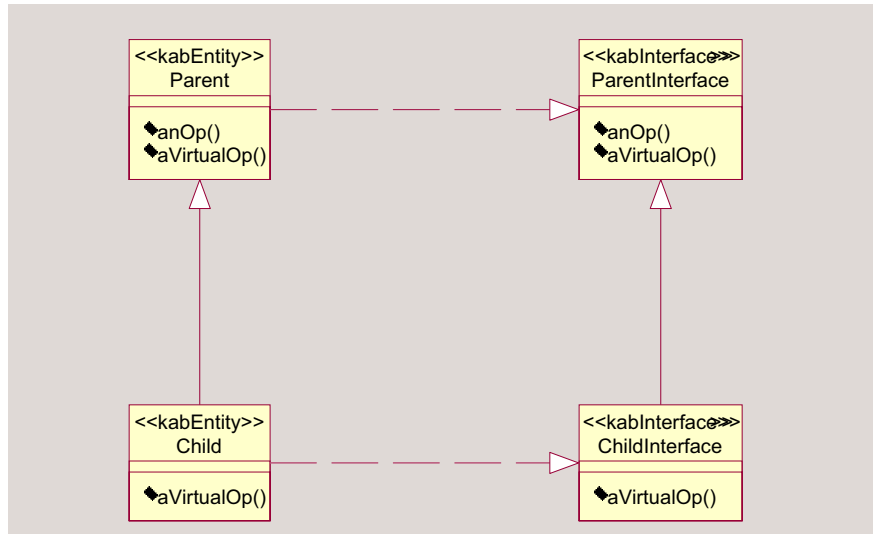


Figure 36: Interface inheritance

The model illustrated in Figure 36 is expressed in IDLos as follows:

```
package myPackage
{
    entity Parent
    {
        void anOp ();
        [ virtual ]
        void aVirtualOp ();
    };
    entity Child: Parent
    {
        void aVirtualOp ();
    };
    interface ParentInterface
    {
        void anOp ();
        [ virtual ]
        void aVirtualOp ();
    };

    interface ChildInterface: ParentInterface
    {
        void aVirtualOp ();
    };
    expose entity Parent with interface ParentInterface;
    expose entity Child with interface ChildInterface;
};
```

Cross-package inheritance When the supertype and subtype interfaces are in different packages, ObjectSwitch cannot validate operation redeclarations through the exposed entities; in ObjectSwitch cross-package inheritance relationships do not exist between entities. For this reason, it is illegal to redefine or redeclare an operation in the subtype.

Namespaces

ObjectSwitch namespaces are hierarchical, with packages forming the outermost namespace. Within a package, you can use modules to impose additional namespaces when needed. Within packages and modules, many ObjectSwitch elements define a namespace.

All identifiers in a namespace must be unique. For example, an entity defines a namespace, so it cannot contain an operation and an attribute with the same name.

Modules

Modules provide extra namespaces within a package if needed.

Modules can be nested. A module may contain entities, interfaces, enums, structs, exceptions, typedefs, nested modules, relationships, actions, and exposes statements.

When a module is declared more than once in IDLos, the module is re-opened and modified on each subsequent declaration. More information will be added to its definition each time that the module name appears in the model.

Model elements defining namespaces

Package and module are not the only containers that form a namespace. The following table includes all the IDLos language elements that form a namespace.

This namespace...	...may contain these namespaces
package	module, entity, interface, struct, exception, relationship
module	module, entity, interface, struct, exception, relationship
interface	struct, operation, exception
entity	struct, operation, exception, signal
struct	struct
exception	struct
operation	
signal	
relationship	role

Scoped names

In IDLos and action language, to identify an element in its namespace, use the scoped name. The scoped name is formed by listing each namespace of the hierarchy, separated by a pair of colons (::).

For example, consider the model in the following example:

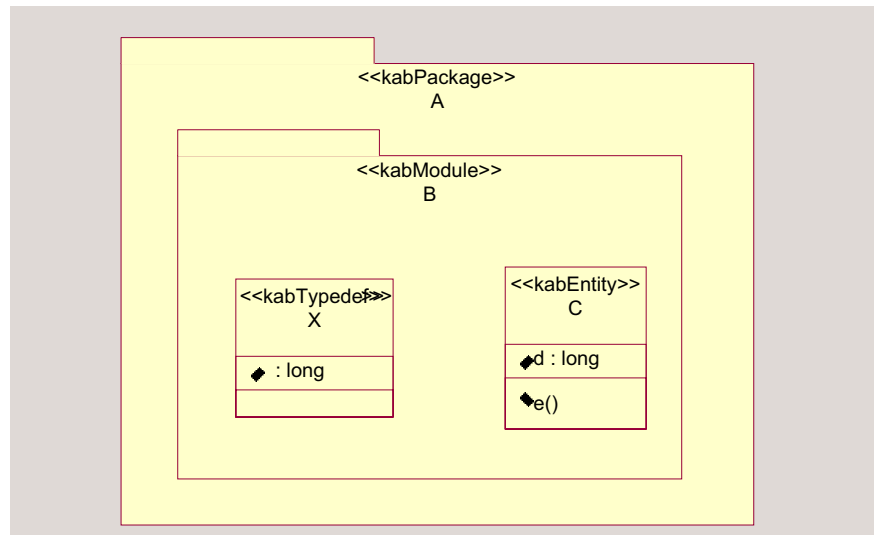


Figure 37: Namespace example

Here is the same model in IDLs:

```
package A
{
  module B
  {
    typedef long X;
    entity C
    {
      attribute long d;
      void e();
    };
  };
};
```

To refer to the attribute in the example, you would say

`::A::B::C::d`

The leading (`::`) mean the name is *globally scoped*. Globally scoped names are like absolute addresses. They tell IDLs to start resolving the name outside of any package scope (namespace).

A relative address is *partially scoped*. Partially scoped names are resolved starting from where they are used. For example, when you implement operation `::A::B::C::e`, you are in `e`'s namespace. Here are three ways to refer to the typedef `X` from within such an action:

```
action ::A::B::C::e
{
  declare ::A::B::X aa; // globally scoped
  declare A::B::X bb; // partially scoped
  declare B::X cc; // partially scoped
  declare X dd; // unscoped
};
```

A partially scoped name is searched for outwards from each enclosing scope. So in the example, the second declaration (`A::B::X`) starts a search in namespace `e`. Within `e`, it does not see a namespace or name `A`. So it goes outward to the next namespace, `C`. Within `C`, it does not see a namespace or name `A`, so it goes outward to `B`. Within `B`, it does not see an `A`. Out again to the global scope. In the global scope, it sees `A`. Within `A`, it sees `B`. Within `B` it sees `X`. The name is resolved.

Ordering and forward declarations in IDLos

IDLos is a one-pass parser. This means the parser must see the definition of something before it is used. This is true even within an entity. For instance, you must define the signals before they are used in transition statements.

Entities and interfaces can be forward declared (this is explained in “A big example”). Forward declarations satisfy the parser: having seen the name, the parser lets you use the entity or interface.

```
// forward declares
entity A;
entity B;
interface C;

relationship R
{
  role A owns 0..* B;
};
entity A
{
  attribute C c;
  attribute D d; // error! D not defined yet!
};
typedef long D;
```

The action language is not parsed until after the entire model is loaded into the Design Center and you request a build. So you don't have to worry if the types you refer to in your action are defined below or above the action in the IDLos file. Once they have all been loaded, the types will already be in the Design Center model sources and will be available to the entire model.

A big example

Here is a larger example model. It is an example of the notifier pattern. Use this pattern when you define “callbacks” or “notifiers” in your server component. It demonstrates the use of interfaces, inheritance and exposure:

- An abstract interface is defined in the server package.
- An interface in the client package inherits from the abstract interface.
- The subtype interface in the client package exposes an entity in the client.

At runtime the server can access a callback in the client through the abstract interface.

The example defines a server package and a client package. The server listens for a message from the client; if the server receives “Hello” then it replies “Bonjour”, and replies “I don’t understand” to any other message. When the client receives the server’s response, it prints “Ok” or “Not Ok”.

Visual Design Center

The following graphic shows the HelloWorldSever package in the Visual Design Center.

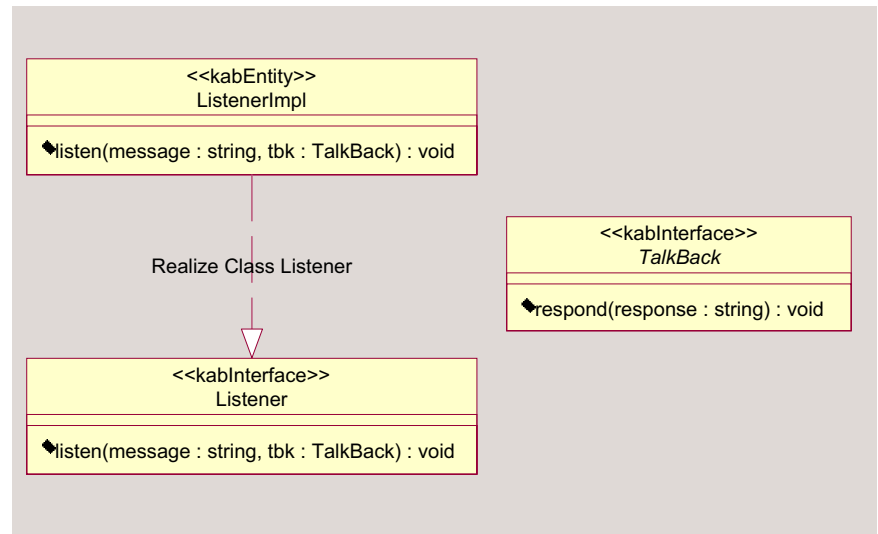


Figure 38: HelloWorldSever

The interface Listener exposes the operation listen in the entity ListenerImpl.

The following action language implements the operation listen:

```
if (message == "Hello")
{
    tbk.respond(response:"Bonjour");
}
else
{
    tbl.respond(response:"Bonjour");
}
```

Also, the abstract interface TalkBack exposes the operation respond.

The following graphic show the HelloWorldClient in the Visual Design Center.

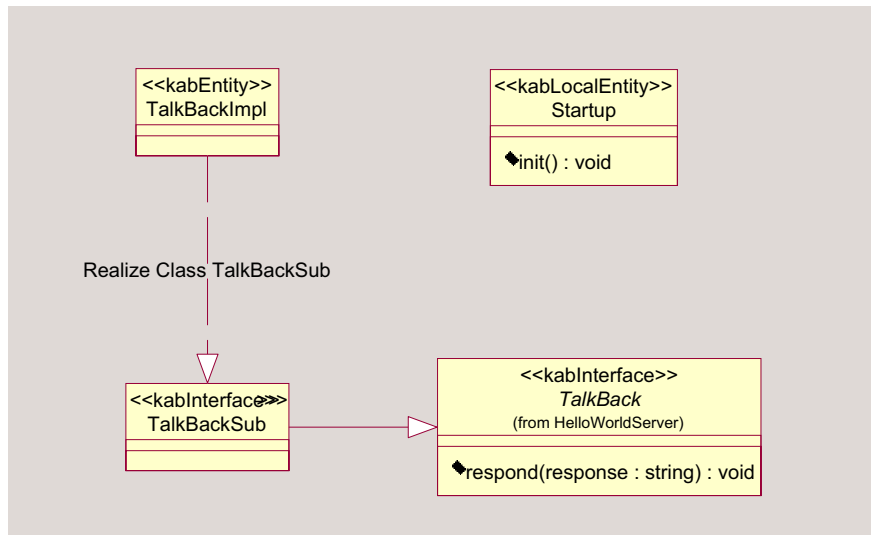


Figure 39:
HelloWorldClient

The interface TalkBackSub inherits from the abstract interface TalkBack defined in the HelloWorldServer package and exposes the entity TalkBackImpl.

The operation init in the local entity Startup is the lifecycle operation that kicks off the application. It creates an instance of TalkBackSub and Listener from the server package and invokes listen on the Listener instance passing the instance of TalkBackSub as the callback object.

The follow action language is the implementation of init:

```
declare ::HelloWorldServer::Listner lis;
declare TalkBackSub tks;
```



```
create lis;
create tks;
lis.listen(message:"Hello", tbk:tks);
```

Through the inherited abstract interface TalkBack, TalkBackSub exposes the signal respond defined in TalkBackImpl. The graphic below shows the state machine for the entity TalkBackImpl.

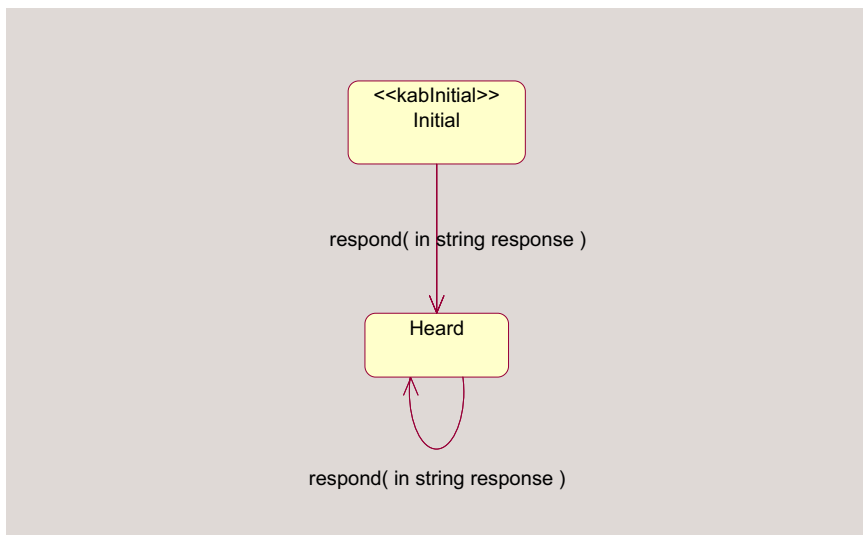


Figure 40: TalkBackImpl state machine

When an instance of TalkBackImpl receives the signal respond, it invokes the action language defined for the state Heard.

The following action language implements the state Heard:

```
if (response == "Bonjour")
{
    printf("Ok\n");
}
else
{
    printf("Not Ok\n");
}
```

IDLos

Here is the IDLos for the same model:

```
[annotation= "This server says 'Bonjour' when clients say 'Hello'"]
package HelloWorldServer
```

```

{
  // forward declares
  interface TalkBack;

  [annotation= "This entity does all the work."]
  entity ListenerImpl
  {
    [
      annotation= "The TalkBack reference must be passed in"
      " by the client. The server uses it to respond."
    ]
    oneway void listen(in string message, in TalkBack tbk);
  };

  [
    annotation= "This interface exposes the ListenerImpl"
    "entity. It allows create and delete access"
    "so that clients don't need a factory.",
    createaccess=granted,
    deleteaccess=granted
  ]
  interface Listener
  {
    oneway void listen(in string message, in TalkBack tbk);
  };
  expose entity ListenerImpl with interface Listener;

  [ annotation=
    "This is the notifier. Because clients inherit from "
    "this interface (and implement it), it must be "
    "abstract. We also grant access, because clients' "
    "subtype of this notifier will need it",
    createaccess=granted,
    deleteaccess=granted,
    abstract
  ]
  interface TalkBack
  {
    oneway void respond(in string response);
  };

  action ListenerImpl::listen
  {
    if (message == "Hello")
    {
      tbk.respond(response:"Bonjour");
    }
    else
  
```

```

        {
            tbk.respond(response:"I don't understand");
        }
    };
};

//
// Now the client.
//
package HelloWorldClient
{
    [
        annotation=
            "Let's implement the notifier. This is the key to"
            "the notifier pattern: inheriting the abstract"
            "interface, and implementing the derived interface"
            "in an entity. No operation declaration is needed in"
            "the TalkBackSub - it is inherited."
    ]
    interface TalkBackSub : ::HelloWorldServer::TalkBack
    {};

    [annotation="Process the response in a small state machine."]
    entity TalkBackImpl
    {
        signal respond(in string response);
        stateset {Initial, Heard} = Initial;
        transition Initial to Heard upon respond;
        transition Heard to Heard upon respond;


        [annotation="Implements the state Heard."]
        action Heard
        {
            if (response == "Bonjour")
            {
                printf("Ok\n");
            }
            else
            {
                printf("Not Ok\n");
            }
        }
    };
};
expose entity TalkBackImpl with interface TalkBackSub;

[
    annotation=
        "The operation in this native entity with the engine"
        "event property starts the client in motion.",

```

```
    local
  ]
entity StaRtup
{
  [initialize]
  void init();
  action init
  {
    declare ::HelloWorldServer::Listner lis;

    // It is important to instantiate a TalkBackSub,
    // rather than a TalkBack. In the type
    // hierarchy, TalkBackSub is both a TalkBack
    // and a TalkBackImpl.
    declare TalkBackSub tks;
    create lis;
    create tks;
    lis.listen(message:"Hello", tbk:tks);
  };
};
```



The next chapter explains how you describe state or operation behavior using action language.

3

Action language

This chapter describes the ObjectSwitch action language. The action language is used within action statements in IDLos. IDLos is the ObjectSwitch modeling language discussed in Chapter 2. Before reading this chapter, you should already be familiar with object-oriented software development, and you should have read the *Overview* and Chapter 2 of this book.

Overview

In ObjectSwitch, most of your application's behavior will be implemented by action language statements.

Action language and IDLos Action language is the dynamic counterpart to the IDLos structural modeling language. You use IDLos to design the entities, relationships, and states of your model. You use action language to define *what happens* during these states (the state actions) and what happens when the operations are invoked. You can also call C++ functions from your action language, or even include C++ code in-line.

Why is there an action language?

Suppose you have defined your entity model and your state model, using either IDLos or a third-party UML tool such as Rational Rose. This model describes a large part of your application. But you still need to describe what happens in each state, and what your operations do.

Action language lets you specify actions that implement the operations and state behavior of your model. But unlike regular programming languages, action language lets you specify this behavior at a very high level of abstraction. Instead of dealing with memory allocation and index tables, you do things like selecting objects or traversing relationships.

Action language

Modeling in ObjectSwitch is at a very high level of abstraction, higher than traditional object-oriented languages. For instance, you can define relationships between objects. You can define object state machines to handle asynchronous protocols. You can write queries to find objects, or to filter extents.

However, object models are not enough to implement an application, so ObjectSwitch has an action language. You implement your models at a high level of abstraction using action language. (See)

What is action language like?

The ObjectSwitch action language uses a similar syntax to that of C++, Java, and Visual Basic. If you know one of these languages, many features of the action language are already familiar to you. But action language is simpler, because of its higher level of abstraction. ObjectSwitch action language looks like this:

```
declare short currentCall;
for( currentCall=0; currentCall<maxCalls; currentCall++ )
{
    if( thisCustomer.callsToday == 0 )
    {
        break;
    }
    thisCustomer.enterServiceCall( currentCall );
}
```

Some basic features of the action language

This section describes a variety of basic features of action language. Object manipulation is described in the following section.

Keywords There is a section on each action language statement at the end of this chapter. Each of those statements is a keyword in action language. Additionally, all C++ keywords are also reserved in action language:

asm	for	static_cast
auto	friend	struct
bool	goto	switch
break	if	template
case	inline	this
catch	int	throw
char	long	true
class	mutable	try
const	namespace	typedef
const_cast	new	typeid
continue	operator	typename
default	private	union
delete	protected	unsigned
do	public	using
double	register	virtual
dynamic_cast	reinterpret_cast	void
else	return	volatile
enum	short	wchar_t
explicit	signed	while
extern	sizeof	
false	static	

Preprocessor directives Like IDLos, action language supports the `#include` preprocessor directive. But action language also supports the `#pragma include` directive, which includes a file after code generation and before C++ compilation. This allows you to include C++ header files for external libraries:

```
#pragma include <myCplusplusHeaders.h>
someCplusplusFunction();
```

Variables

Declaration Variable declarations define the name and type of a local variable. You must declare action language variables before using them:

```
declare type name;
```

You can also declare variables with initial values, or declare them as constants:

```
declare long x = 0;    // declaration with initial value
declare const long x = 0; // declaration of a constant
```

Strings can be declared as both bounded and unbounded.

```
//
//   Declare an unbounded string
//
declare string          unboundedString;

//
//   Declare a bounded string of 100 bytes
//
declare string < 100 >   boundedString;
```

In state actions, action language variables must not have the same name as parameters to the signal(s) that transition to that state. Similarly, where an action implements an operation, action language variables must not have the same name as parameters to the operation. For example:

```
signal cursed (in long howManyYears);
transition alive to zombie upon cursed;
action zombie
{
    declare long howManyYears; // invalid! Is parameter name
};
```

You can also include C++ variable declarations in your action language. This lets you use variable types not provided by the action language. However, these types may be incompatible with IDL types and the auditor cannot check them. For example:

```
action ::adventure::xyzyzy
{
    declare longlongVar1;
    long   longVar2;
    struct StatstatBuf;

    longVar1 = 17;
    // legal, since the types are compatible
    longVar2 = longVar1;

    // Type mismatch here, but the auditor can't see it.
    // Instead the C++ compiler will generate an error
    // and that is harder for the developer to deal with
    statBuf = longVar2;
}
```


Scope and lifetime When a variable is declared in an action; its scope is simply the scope of that action. When the action is completed, the variable ceases to exist:

```
action oneAction
{
    declare long myVariable;
    myVariable = 5;
};
action otherAction
{
    declare long anotherVariable;
    anotherVariable = myVariable;
    // that was illegal: "myVariable" is not in this scope
};
```

Code blocks form an inner scope. When a variable is declared within an action; its scope is the code block itself. When the code block is finished, the variable ceases to exist:

```
declare long myOuterVar;
while (x > 0)
{
    declare long myInnerVar;
    // do something
}
myOuterVar = myInnerVar; // illegal - out of scope
```

Also, it is illegal to redefine a variable within an inner code block:

```
declare long myVar;
while (x > 0)
{
    declare long myVar; // illegal - redeclared in scope
    ...
}
```

Accessing signal parameters from a state action In a state action, you can access parameters to the signal that caused the transition. For example:

```
signal cursed (in long howManyYears);
transition alive to zombie upon cursed;
action zombie
{
    declare long zTime;
    for( zTime=0; zTime < howManyYears; zTime++ )
    {
        // be a zombie for another year...
    }
}
```

```
'};
```

Object references You can declare a variable using the name of an entity in the package. This creates an empty *object reference* (also called a *handle*) that you can create or assign objects to:

```
declare
{
    Customer thisCustomer;
    Customer thatCustomer;
}
create thisCustomer;    // create a new object
thatCustomer=thisCustomer;
// thisCustomer and thatCustomer refer to the same object
// This assigned an object reference, is not a "deep" copy
```

Manipulating data

Assignment Assignment statements give values to a variety of expressions in action language. They can contain:

- declared variables
- literals
- attributes on declared objects
- operations on declared objects
- return values from operations on declared objects

For example:

```
myVariable = 3;
someObject.someAttribute = myVariable + 1;
```



The equals sign (=) in an assignment statement indicates assignment, not equality. Equality or equivalence in action language are indicated by a double equals sign (==).

Arithmetic The action language has the following simple arithmetic operators, shown here in descending order of precedence:

Operator	Meaning
*	Multiply
/	Divide
%	Modulus (remainder after division)
+	Add
-	Subtract
<< >>	Bitwise left or right shift, respectively.
&	Bitwise and
^	Bitwise complement
	Bitwise or

The << and >> operators can also be used for external C++ streams; they are passed through to the C++ compiler so that you can do things like:

```
extern ostream cout;
cout << "Hello World" << endl;
```

You can use parentheses in expressions to group items together.

```
1 + 2 * 4    // yields 9
(1 + 2) * 4  // yields 12
```

The following section describes all the action language operators.

Operators Unary and binary operators supported on fundamental IDLos types are shown in the following table.

operators	operand type	description
true false empty	boolean	Literals
!	boolean	Not
&& 	boolean	Boolean AND Boolean OR

operators	operand type	description
=	boolean, char, enum, long, short, unsigned long, unsigned long long, unsigned short, float, double, string, octet	Assignment
<> <= >=	char, float, double	Numeric comparison
<> <= >=	string	Lexographic comparison
==	boolean, char, enum, long, short, unsigned long, unsigned long long, unsigned short, float, double, string, octet	Equality/equivalence compare. Note: enum operands must be the same type.
+ - * / % << >>	float, double, long, short, unsigned long, unsigned long long, unsigned short	Add, subtract, multiply, divide, modulus, shift left, shift right
+	string	Concatenate
+= -= *= /= %= ^= = &=	These assignment operators perform both an arithmetic operation (or string concatenation, for +=) as in the descriptions above, together with an assignment. You use these operators anywhere you can use the ordinary assignment operator...	
<= =>	any	Copy to or from an any. These are the only operators supported for the any data type.

String operations In addition to the concatenation (+) and lexicographical comparison operators, ObjectSwitch strings have the following operations built in. String indexes begin at character zero. Note that `trim`, `remove`, `pad`, `insertChar`, and `insertString` (and assignment) alter the underlying string; all other string operations return a new string.

operators	description
<code>substring(in long start, in long end)</code>	Returns a substring of the string, bounded by start and end, inclusive. For example: <pre>s1 = "01234567890123456789"; s2 = s1.substring(3, 7);</pre> Now s2 contains "34567". See the following section "substring errors" for start and end boundary errors.
<code>trim()</code>	Returns the string with any leading and trailing whitespace removed.
<code>toupper()</code>	Returns the string as all uppercase.
<code>tolower()</code>	Returns the string as all lowercase.
<code>getCString()</code>	Returns the string as a C-style <code>char[]</code> . Useful for <code>printf</code> statements and other places where a C string is required.
<code>length</code>	An attribute (not an operation!) containing the number of characters in the string.
<code>remove(in long start, in long end)</code>	Removes characters from the string from start to end, inclusive. For example: <pre>s1 = "01234567890123456789"; s1.remove(3, 7);</pre> Now s1 contains "012890123456789".
<code>pad(in long len, in char padchar)</code>	Appends the character <i>padchar</i> to the string to make the string <i>len</i> characters long.
<code>insertChar(in long pos, in char data)</code>	Inserts the character <i>data</i> into the string at the position <i>pos</i> .

operators	description
<code>insertString(in long pos, in string data)</code>	Inserts the string <i>data</i> into the string at position <i>pos</i> .
<code>stringToLong(in NumericBase base)</code>	Converts the value in the string into a number. Values for NumericBase are: <i>SWString::Base10</i> , <i>SWString::Base16</i> and <i>SWString::Base8</i> . Note: you probably want to use Base10. This is <u>not</u> the default. (Base8 is the default.)



String operations may be performed on strings you declare in action language, but not on string attributes. You must copy the attribute and manipulate the copy. When you're done, you can assign the copy back to the string attribute.

You can construct a string from a `char[]` or concatenate a `char[]` to a string, but you cannot concatenate two `char[]`. Some examples:

```
declare string s;
declare string w = " world";
declare char x[100];
sprintf(x, "hello");

s = x;
s += x;      // All of these are ok.
s = x + w;

s = x + " world"; // This fails, since x is not a string
```

substring errors The function `substring` throws an exception in cases where `start > end`, `start < 0` or `end >= length`. Failure for any of these three tests will cause the exception, `ExceptionArrayBounds`, to be thrown.

Implicit conversion between string and numeric types

When you assign a string to a number or vice versa, `ObjectSwitch` automatically converts the value if possible, as described in the following paragraphs.

Assigning a number to a string When you assign a numeric type to a string, the value is implicitly converted to a string representation:

```
declare long x = 123;  
declare string myString;  
myString = x;           // myString is now "123"
```

However, you cannot concatenate a numeric type to a string.

```
declare string str = "the number " + 5; // str is now "umber "
```

This moves the pointer in the string "the number" five places to the right before performing the string assignment. To concatenate a converted numeric type to a string, assign the numeric type to a string variable first.

```
declare string x;  
x = 5;  
declare string str = "the number " + x; // str is "the number 5"
```

Assigning a string to a number When you assign a string to a numeric type, the string is interpreted as a numeric constant and implicitly converted to a number, if possible. The constant may be decimal, octal, or Hexadecimal, and may be preceded by a + or - sign.

```
declare string myString = "This is a string";  
declare long myLong = 0;  
myString = "4";  
myLong = myString; // myLong is now 4, not 0
```

type of constant	syntax
decimal constant	begins with a non-zero digit, and consists of a sequence of decimal digits
octal constant	begins with 0 followed by a sequence of the digits 0 to 7 only
hexadecimal constant	consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a..f (lower- or uppercase) corresponding to the values 10..15

errors If the string does not contain a valid constant, ObjectSwitch throws an `ExceptionDataError` exception, which your model can catch. If you don't catch this exception, the engine will exit.

Name spaces

Action language operates within the name spaces defined in the enclosing IDLos model. Identifiers in action language must be scoped according to the IDLos scoping rules, which precisely match those of IDL. See Chapter 2 for a discussion of IDLos scoping rules.

Data types

Action language uses the same fundamental types as IDLos does. You declare a variable using the action language declare statement.

Your model's action language can use any built-in type, or any user-defined type in the package where the action language appears. Types in other packages may also be used by qualifying them explicitly, as in:

packageName::typeName

Control structures

Loops

The action language provides three ways to iterate over a set of statements: for, for...in, and while.

for The for statement lets you loop using a range of values. It is identical to the C++ for statement:

```
for( x=0; x<myLimit; x++ )  
{  
    // do something  
}
```

for...in This statement iterates over a set of objects, using a different object each time through the loop:

```
for thisCustomer in setOfCustomers  
{  
    // do something  
}
```


The `for...in` statement may iterate over the instances of an extent or over the object in a relationship:

- extent—if the name of an entity appears after the `in` keyword, then the loop iterates over the extent (all the instances) of that entity
- relationship—if a relationship navigation appears after the `in` keyword, then the loop iterates over the related objects

`while` This statement loops as long as a given expression remains true. The expression is evaluated at the beginning of each loop:

```
while( money > 0 )
{
    // spend some money
}
```

Branches

The action language supports `if`, `else`, and `else if` constructs:

```
if( thisCustomer.balance < thisCustomer.creditLimit )
{
    // sell to the customer
}
else if( thisCustomer.alreadyWarned )
{
    // deactivate account
}
else
{
    // warn customer
}
```

Manipulating objects

This section introduces the principal ways that you can manipulate objects. It covers creating and deleting objects, handling object references, accessing the operations and attributes of objects, and using relationships.

Creating objects

To create a new object in action language, first declare a variable of the object's type (see "Object references" on page 104) and then use it with the create statement:

```
declare Customer someCustomer;  
create someCustomer;
```

This creates a new Customer object in the local shared memory, which the handle someCustomer refers to. When the current action finishes, someCustomer will no longer exist (see "Scope and lifetime" on page 103) but the new object will remain in shared memory. *The object remains even when the handle to it no longer exists.* You can locate the object again by selecting from the Customer extent or via any objects that you relate it to (see "Relating and unrelating objects" on page 115).

With initial values You can also create an object with initial values assigned to some or all of its attributes. For example, you can create the customer and provide an initial value:

```
declare Customer someCustomer;  
uniqueKey = self.allocateCustId(); // define this somewhere  
create someCustomer values (id:uniqueKey);
```

(The identifier self never needs to be declared; it always refers to the object in which the action is executing.)



When you create objects that must have unique keys, use the values clause to set these keys during creation to avoid creating objects with duplicate keys.

If an entity is marked in IDLos with the singleton property, you must use the create singleton statement (see "Singletons" on page 112) to create it.

Deleting objects

To remove an object from shared memory, use the delete statement:

```
delete someCustomer;
```

Singletons

If an entity is marked in IDLos with the singleton property, you must use the create singleton statement to create it:

```
// PortAllocator was defined as a singleton in IDLos
declare PortAllocator thePortAllocator;
create singleton thePortAllocator;
```

This creates a new PortAllocator object in shared memory unless one exists already, in which case it simply makes thePortAllocator refer to the existing object.

Object references

The preceding paragraphs have used object references in declarations, create statements, and other ways. When you first declare an object, it does not yet refer to anything—it is simply an empty reference:

```
declare Customer myNewestCustomer;
myNewestCustomer.name = "ObjectSwitch"; // invalid!
```

You need to create the object or assign the handle to an existing object before you can use it:

```
declare Customer myNewestCustomer;
create myNewestCustomer;
myNewestCustomer.name = "ObjectSwitch"; // this is OK
```

The “empty” keyword Sometimes you need to test whether an object reference is valid or not. For example, a select (see “Selecting objects” on page 116) may not return a valid object:

```
declare Customer aCustomer;
select aCustomer from Customer where (aCustomer.id == 1);
aCustomer.name = "ObjectSwitch"; // might be invalid!
```

In cases where you might have an invalid object reference, you can check it using the empty keyword:


```
declare Customer aCustomer;
select aCustomer from Customer where (aCustomer.id = 1);
if( empty aCustomer )
{
    // whatever you do when there's no customer 1
}
else
{
    aCustomer.name = "ObjectSwitch"; // OK
}
```

Operation and signal parameters

When you define an operation (or signal) you can specify *formal parameters* that the operation takes. The following paragraphs describe the calling conventions and notation for the *actual parameters* that you supply when you call the operation from action language.

Calling conventions Actual parameters that you pass in when you call an operation or signal are passed either by reference or by value:

- objects are always passed by reference
- other types (fundamental, complex, etc.) are always passed by value

 *These calling conventions may have performance or functional impact on your application, so be clear about the difference. For example, using a large struct as a parameter to a remote operation may cause much more network traffic than using an object.*

Positional and named parameters The action language lets you call operations and signals using either positional or named parameters. If you use any named parameters, you must name all parameters — you cannot use named and positional parameters in the same call.

For example, given the IDLoS:

```
entity CallingOpsAndSignals
{
    signal aSignal( in long inL, in boolean inB);
};
```

you can invoke the signal in action language using either named or positional parameters:

```
declare long    aLong = 1;
declare boolean aBoolean = false;

//
// Call aSignal using positional parameters
//
self.aSignal(aLong, aBoolean);

//
// Call aSignal using named parameters. Notice that
// the parameter order was reversed.
//
```

```
self.aSignal(inB:aBoolean, inL:aLong);
```

Accessing operations and attributes

As demonstrated in previous examples, you access the attributes of an object using the “dot” operator. This is also how you access its operations:

```
if( thisCustomer.creditLimit > requestedAmount )
{
    thisCustomer.sendApprovalLetter();
}
```

You can’t access nested members by chaining together multiple “dot” operators:

```
declare short    since;
since = thisCustomer.profile.customerSince; // illegal
```

Instead, you need to use an intermediate variable:

```
declare CustProfile thisProfile;
declare short    since;
thisProfile = thisCustomer.profile;
since = thisProfile.customerSince;
```

Handling relationships

A relationship between entities in an IDLos model specifies the objects that may be related to each other. Relating specific objects to other objects is something that you do at run time using the role name:

Relating and unrelating objects A relationship between entities in an IDLos model specifies the objects that may be related to each other. Relating specific objects to other objects is something that you do at run time using the role name:

```
declare Customer thisCust;
declare Invoice thisInvoice;

//
// Select customer, create invoice, and relate the two
//
select thisCust from Customer where (thisCust.id=123);
create thisInvoice;
relate thisCust isBilledBy thisInvoice;
```

This leaves the two objects related in the shared memory, so that you can always find one given the other. For example, you can process all the invoices for a customer:

```
for oneInvoice in Invoice->Customer[bills]
{
    // process the invoice
};
```

When you no longer want two objects to be related, use the `unrelate` statement:

```
unrelate thisCust belongsTo thisSalesRep;
```



When you delete an object that is related to another object, the Application Server does an automatic `unrelate` for you. You are not left with a dangling relationship from the surviving object. If you are using `unrelate` triggers, note the specific behavior of the triggers for this implicit `unrelate`, described in “Role trigger” on page 48.

Navigation You can “navigate” (traverse) across multiple relationships to find one object that is related to another via one or more intermediate objects. For example, suppose for each overdue invoice we want to notify the manager of the customer’s sales representative:

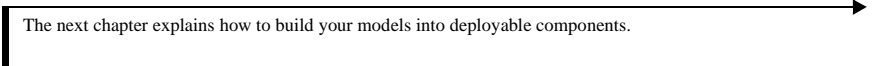
```
declare
{
    Invoice theInv;
    SalesManager theMgr;
}
for theInv in Invoice
{
    if( theInv.isLate )
    {
        theMgr = Invoice
            ->Customer[bills]
            ->SalesRep[belongsTo]
            ->SalesManager[worksFor];
        theMgr.notify();
    }
}
```

Selecting objects You use the `select` statement to select a single object from a relationship, extent, or singleton. You always include a `where` clause:

```
select aCustomer from Customers where (aCustomer.id=123);
```

except when you're selecting a singleton or from a “one” end of a relationship:

```
select aCustomer from thisInvoice->Customer[bills];
```



The next chapter explains how to build your models into deployable components.

4

Building ObjectSwitch components

When you build your model into a component, everything needed to deploy the model on an ObjectSwitch node is wrapped into a single file.

This chapter describes the ObjectSwitch tools and commands you use to generate an ObjectSwitch component. To build a component from your model you:

- 1 Design an implementation
- 2 Build the model into a deployable component

A component specification is used to build an implementation of your model and is introduced in this first section of the chapter:

- The ObjectSwitch component
- What is a component specification?
- Defining a component specification

Chapter 4: Building ObjectSwitch components

The ObjectSwitch component

This chapter is a guide to developing a component specification. It is presented as follows:

- Creating a project
- Working with model sources
- Defining a component
- Putting packages in a component
- Importing another component
- Adding adapters
- Adding model elements to adapters
- Saving a component specification

The final section of the chapter describes how you build a deployable component from your component specification:

- Building the component

The ObjectSwitch component

An ObjectSwitch component is an executable model that can be deployed and reused. You deploy a component in the ObjectSwitch runtime using the Engine Control Center (see the *Deploying and Managing ObjectSwitch Applications*). To reuse a component, you import the component providing the service into a new component.

You turn your model into a component using a component specification.

What is a component specification?

A model is an abstract definition of your application logic, with little or no consideration for implementation. Your component specification, by contrast, specifies how you want the model implemented. A component specification defines:

- model source files to include
- model elements to implement in the component
- service adapters to implement for model elements
- dependencies with other components
- compile time options

You use a component specification to build applications from models. It describes how to turn your model into a deployable Objectswitch component.

Defining a component specification

Whether you use the graphical user interface or text-based commands to compose your specification you can view this as a three step process.

- 1 Create the project. Define the structure of your implementation (for example: two components, one with a database adapter).
- 2 Populate the specification. Select the elements from your model for each part of the implementation (for example: package A for component A, interface B for adapter C).
- 3 Add properties. Select and add the properties to the model elements you are implementing.

Figure 41 shows a conceptual view of this process.

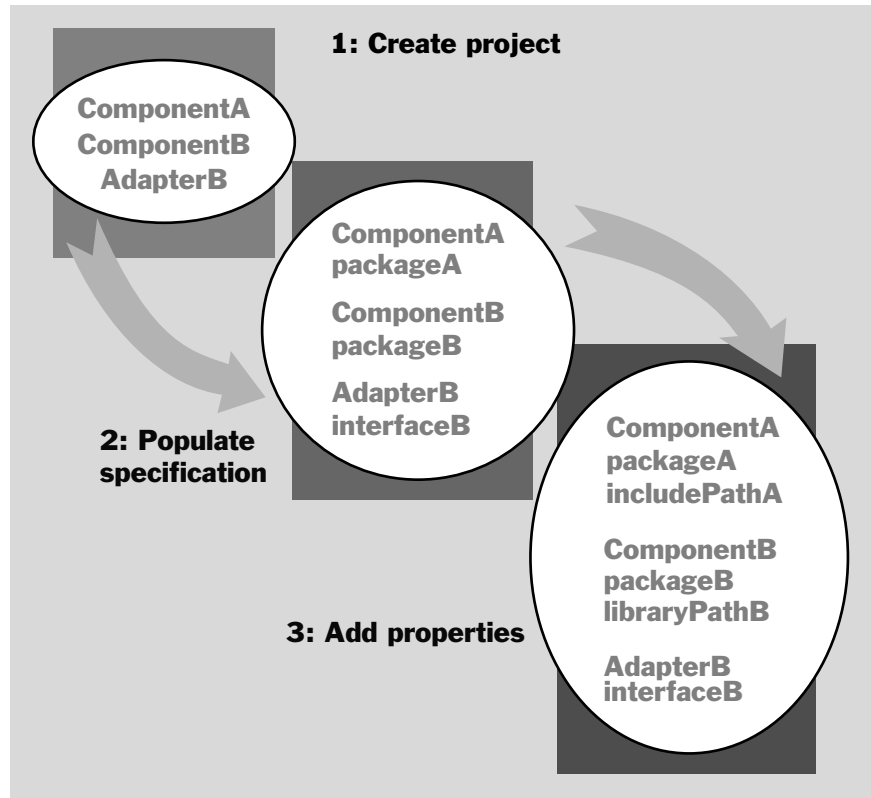


Figure 41: The three general stages in defining a project

Although Figure 41 shows this as three discrete stages you can mix the individual steps in any way that is convenient. The order is not important.

The individual steps by which you define a project are described in greater detail in the following sections.

Graphics vs Text to build a component specification

The Visual Design Center window (or “VDC”) lets you design your component specification graphically. Or you can write your component specification in a text editor and run the build directly from the command line using any text editor that saves a straight flat file, such as `vi` on the Solaris platform.

The following sections show, in parallel, the process of developing the component specification with the Visual Design Center and in a text file.

Creating a project

The component specification begins with a project. A project is the container in which you will develop your component specification. You can specify multiple component definitions in a single project. The components you add to your project define the runtime partitioning of your application model.

Figure 42 presents a hierarchical view of project elements.

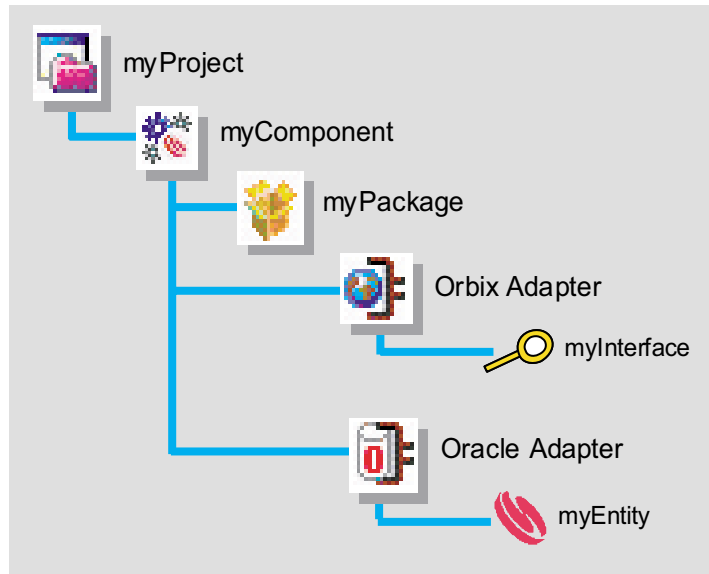


Figure 42: a project tree

myInterface and myEntity are model elements from myPackage.

Visual Design Center In the Visual Design Center you need to start a new project before you can define a component specification.

To start a new project:

- 1 You select the project icon from the palette and drag it into the project window.
- 2 To rename the project, right click on the project name and select Rename from the menu. Type in the new project name.

Figure 43 shows a new project icon before it has been renamed.

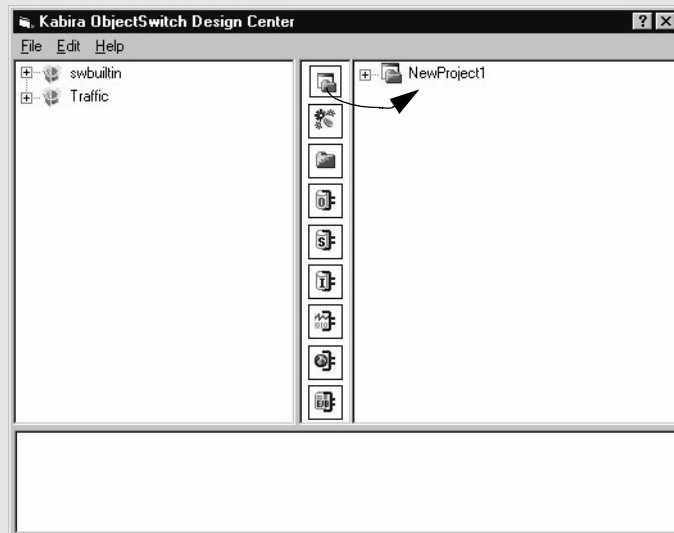


Figure 43: Adding a new project

Text The component specification text file is equivalent to a project in the Visual Design Center. The component specification file is one of the parameters passed to the swbuild utility when you build the deployable component. Refer to "Starting a build in text" on page 140.

Project properties

You can define properties, such as `buildPath` and `classPath`, at the project level. If you set properties at the project level they will be defined globally for the entire project. For a complete list of properties available at the project level, see “Properties” on page 363.

Visual Design Center In the VDC, to set a project property:

- 1 Click on the “+” sign on the left of the project icon to expose the Properties tag immediately below.
- 2 Right click to open the pop-up menu and select Add Properties.

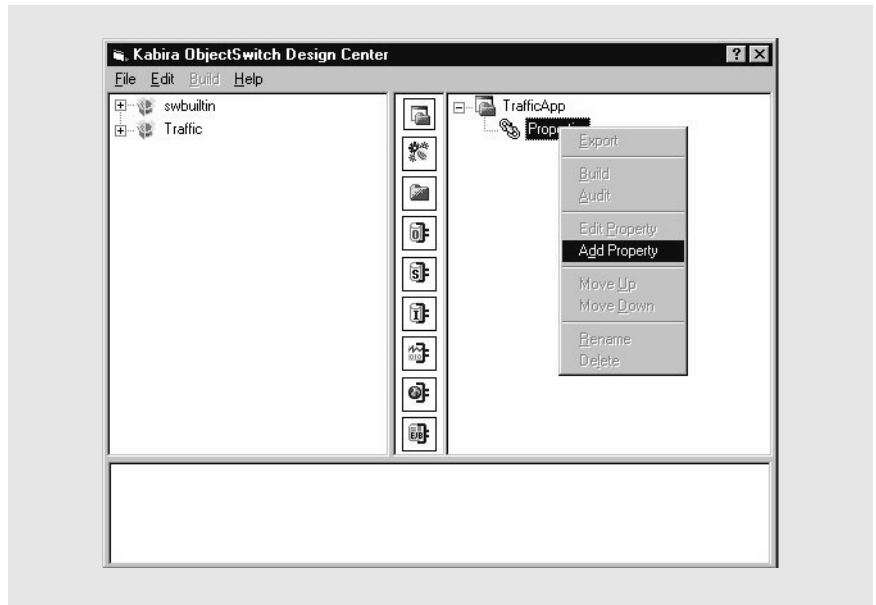


Figure 44: add properties option

- 3 Highlight the property you wish to add from the property list and click OK. This adds the new property below the Properties tag.

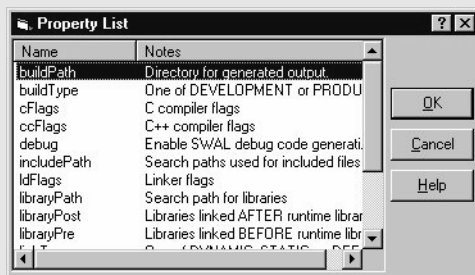


Figure 45: properties list window

- 4 Right click the new property and select Edit Properties.

- 5 Enter the property value, then highlight the selection. Click on OK.

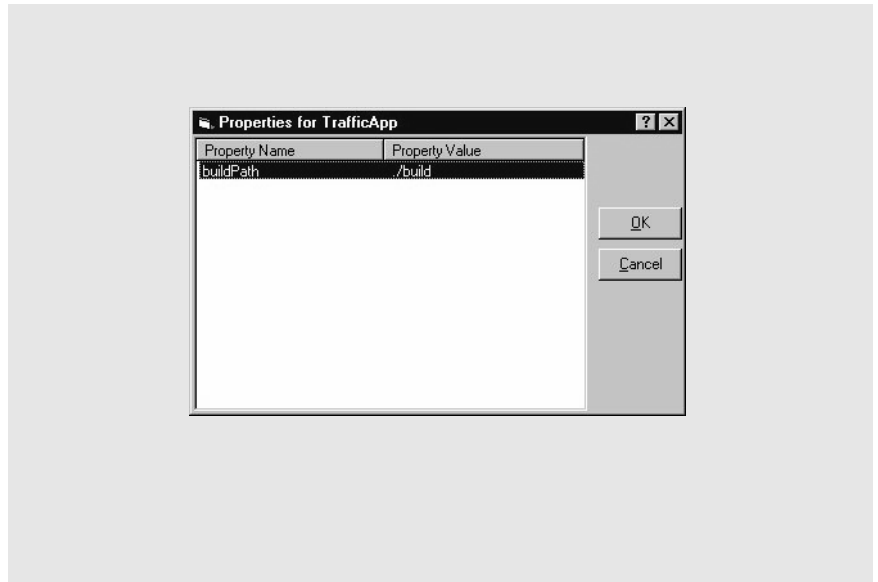


Figure 46: Editing a new property

Editing a project property To change the property that you've added to a project, right click on the property and edit it as described above.

Text To set a project property, define the property outside the scope of all the components in the component specification file. For example:

```
// project property
buildPath = "some/build/path";
component Component1
{
};
component Component2
{
};
```

Working with model sources








The core of the project is your model sources. The component specification is used to build a deployable or reusable implementation of this model.

Visual Design Center The Visual Design Center window opens with all the model packages you currently have open in Rational Rose available in the model browser.

In the case of legacy IDLos models, if your model includes other files, such as the IDLos `#include` directive, you need to specify the include path for those files. See “Setting up Design Center access”.

Whenever you build a component, the model is automatically refreshed from the source(s) that you have selected.

You can view your model source in the Visual Design Center model browser by clicking on the “+” to the left of the model name. The table below describes which model element each icon in the browser represents.

	Package
	Entity, attribute, operation (shown from left to right)
	Interface
	Key
	Relationship
	Role
	Signal

Text You identify the sources for a component with `source` keyword. These files will be loaded into the Design Center server prior to a build. Source files must be listed in the correct order of dependency. If you

use a source statement in the component specification rather than `#include` directives in IDL files, the Design Center server can optimise reloading.

In the example below, one source statement has an associated `includePath` property, which is valid only for that source statement. Source statements may have other properties, as well.

```
component MyComponent
{
    source /path/to/myfile.soc
    source anotherfile.soc
    {
        includePath = some/file/path;
    };
};
```

Defining a component

A component specification will contain all the information necessary to build a deployable and reusable component from your model.

Visual Design Center Select and drag the component icon into the project. Figure 47 shows a component icon added to the project and renamed TrafficComponent.

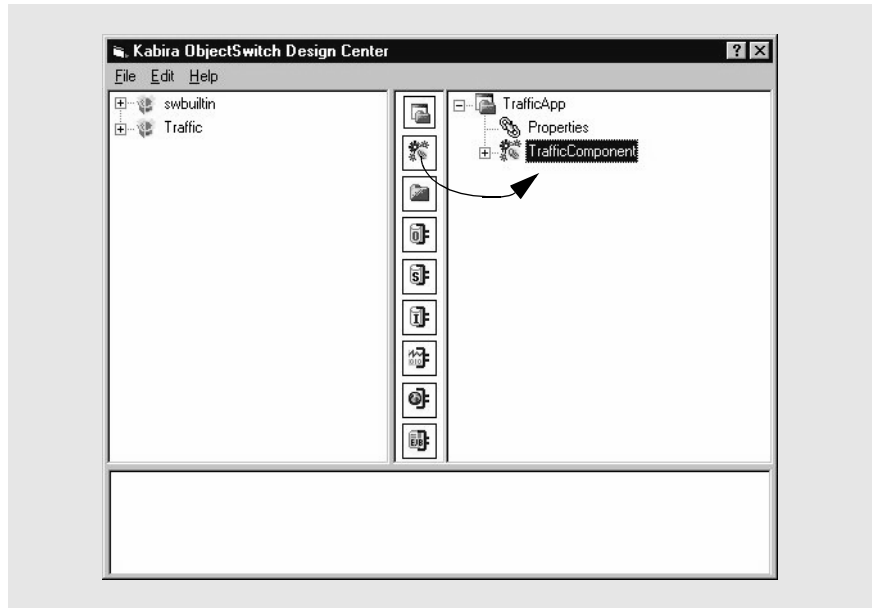


Figure 47: Adding a new component

Text You start the textual component specification with the component keyword. This starts the definition of a component block. For example:

```
component TrafficComponent
{
    // a component specification for the Traffic application
};
```

A component implements model packages. It may contain adapters, or model elements; these may be grouped.

Component properties

There are a number of properties that you can set for a component that control how the compilers generate code. More than one property may be set for a component. For a complete list of properties available at the component level, see “Properties” on page 363.

Visual Design Center You add, and edit, component properties like you add and edit project properties. Select the `Properties` tag below the component and refer to the VDC section of “Project properties” on page 125 for step by step instructions.

Text To set a component property, define the property inside the component block in the component specification file.

```
component Component1
{
    includePath = "some/include/path";
}
```

Many of these properties can also be set on a project so that they apply to all the components contained in the project. For more information refer to “Project properties” on page 125.

Putting packages in a component

You add complete packages to a component. It is not possible to add some entities in a package to one component, and other entities in that package to another component.

Chapter 4: Building ObjectSwitch components

Putting packages in a component

Visual Design Center You specify the packages that you want in a component by dragging the package from the model browser into the component, as shown in Figure 48.

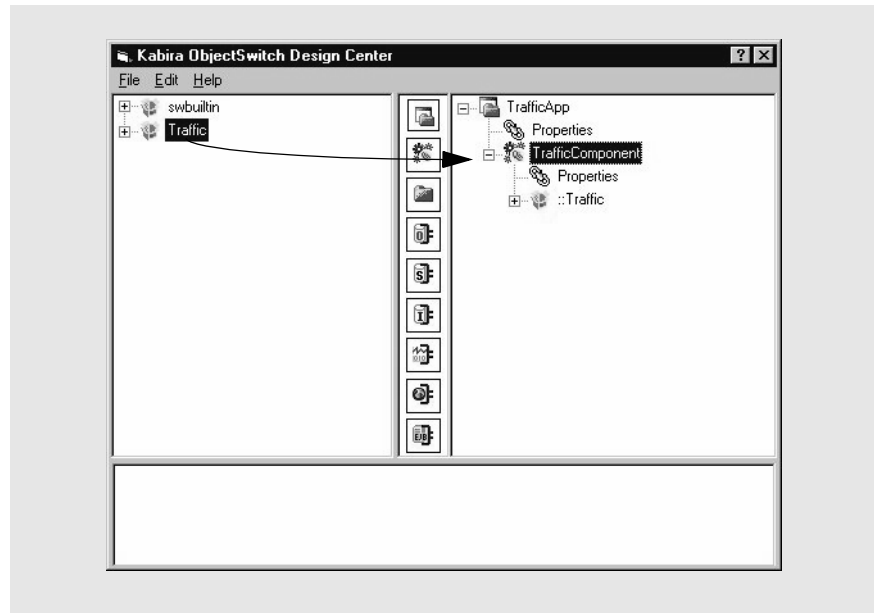


Figure 48: Adding a package to a component

Text The keyword `package` signifies an element type. A package can belong to a component, an adapter, or a group.

```
component MyComponent
{
    package MyPackage;
}
```



You cannot partition a package across components, but you can implement a package in more than one component. This constraint should be taken into consideration at the modeling stage.

Importing another component

You can use a component as a service in another component. To do this, you import one component—the one providing the service—into the other. You will need to create a client model to use that new service. For example, the client model may instantiate and use interfaces in the service component. Figure 49 illustrates this concept.

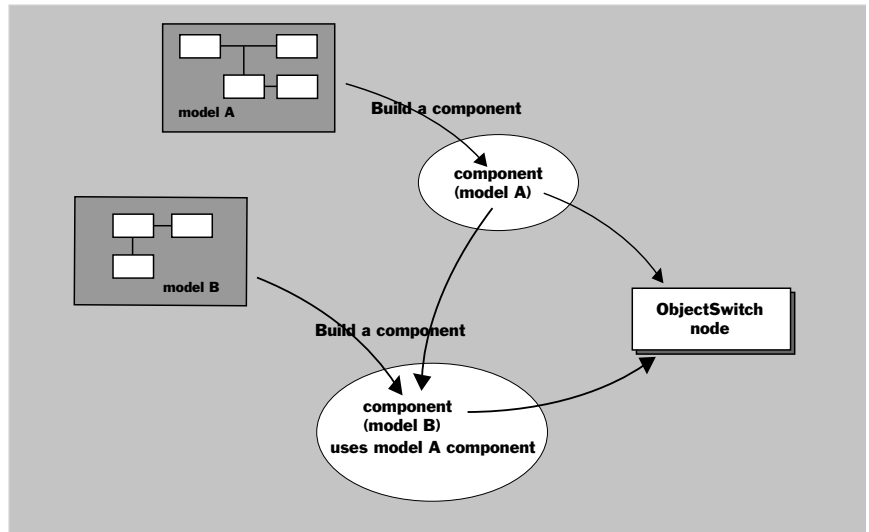


Figure 49: Two models used as client and server components

Visual Design Center To import another component into the Visual Design Center window:

- 1 Open File -> Import from the Design Center menu.

- 2 Select the component from the component list and click OK.

Figure 50 shows the Import window.

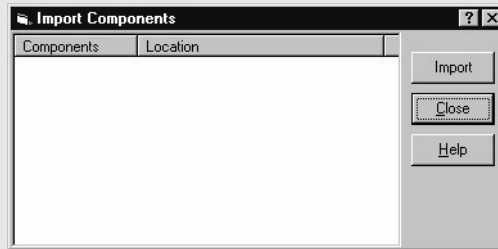


Figure 50: Importing a component



The component you wish to import should reside in the component include path defined in Rose under Project Properties. By default, "." and \$SW_HOME/distrib/\$SW_PLATFORM/component are searched.

Text When a component A depends upon another component B, you declare A's dependency upon B with the *import* statement. In the example below, the default include path is augmented with the `includePath` property for all components referenced in component A.

```
component A
{
    importPath = /some/path;
    import B;
};
```


Adding adapters

You add adapters to a component to implement selected entities or interfaces in a database or service. For example, if you want selected entities in a component to be implemented in Oracle, you would add an oracle adapter to the component. When you build your application, the component will include code to maintain and synchronize the database and shared memory copies of its objects.

Visual Design Center You specify an adapter that you want in a component by dragging its icon into the component.

Figure 51 shows an adapter added to a component.

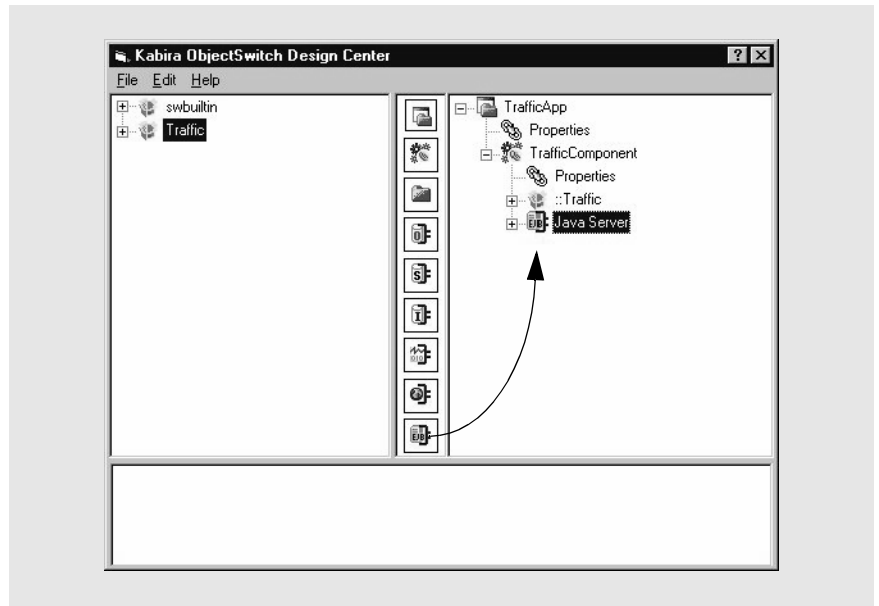


Figure 51: Adding an adapter to a component

Chapter 4: Building ObjectSwitch components

Adding model elements to adapters

Text You add an adapter to a component with the `adapter` keyword followed by the name of the adapter. The example below shows how component A uses the CORBA adapter `swi ona` to implement the interface `::MyPackage::MyInterface`:

```
component A
{
    adapter swi ona
    {
        interface ::MyPackage::MyInterface;
    }
};
```

Adapter properties

There are a few properties that you can set for adapters. You add and edit adapter properties like you add and edit project and component properties. Each type of adapter has different properties; for these properties and the values they can take, refer to the documentation for each individual adapter.

Adding model elements to adapters

When you add an adapter to a component you also need to designate which entities or interfaces, depending on the adapter type, are used with that adapter.

For instance, you connect a modeled entity to an external implementation, such as a database, using a database adapter. You connect a model interface to an external protocol such as CORBA or SNMP using a protocol adapter. Refer to the specific adapter in each service adapter for more information on what can be implemented in that adapter.

Visual Design Center You specify the adapter that you want in a component by dragging its icon to the component. Figure 52 shows an interface added to an EJB adapter.

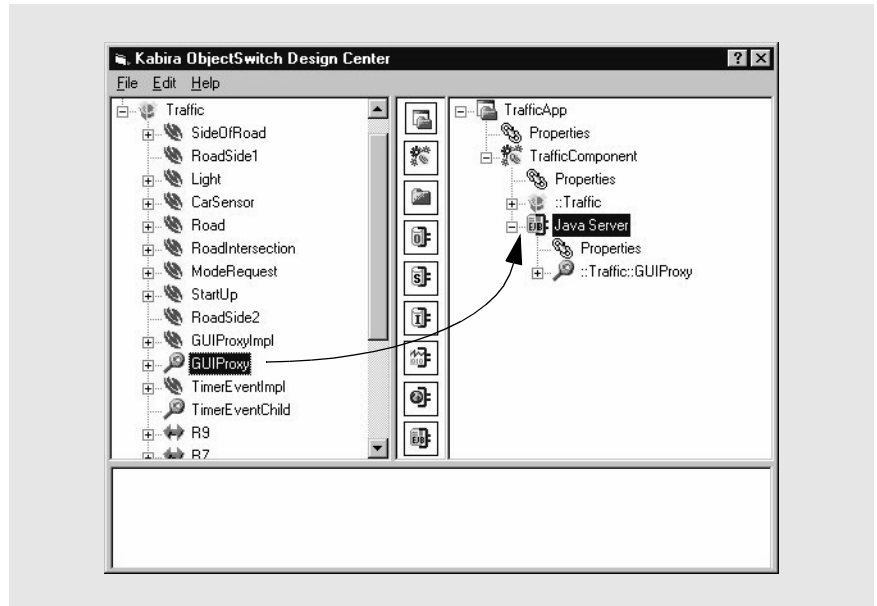


Figure 52: Adding an interface to an adapter

Text The model element types are: package, module, interface, entity, relationship, operation, signal, attribute, role and key.

```
component MyComponent
{
  adapter CORBA
  {
    i nterface :: MyPackage :: MyI nterface;
  };
  adapter Oracle
  {
    e nti ty :: Mypackage :: MyEnti ty
    {
      readStri ng = "Some SQL String";
    };
  };
};
```

Model element properties

Implementation adapters let you set properties on an entity to control how those objects are cached, and to support legacy databases. Refer to the documentation for Database Adapters to learn how to use these properties.

Some protocol adapters allow you to set properties on the interface. Refer to the documentation for Database Adapters to learn how to use these properties.

You add and edit model element properties within the element block like you add and edit project and component properties. Refer to “Project properties” on page 125 for step by step instructions in both the VDC and text.

Putting relationships and roles into adapters

Refer to the documentation on individual adapters for more about relationships and roles in adapters.

Putting attributes into adapters

You add attributes to an adapter to override the settings for that attribute in the interface or entity. For example, a read-write attribute in an interface can be made read-only for use with an adapter.

Setting attribute properties Some adapters let you customize attributes to override the default definition for the attribute in its enclosing entity or interface.

You add and edit attribute properties within the attribute block like you add and edit project and properties.

Saving a component specification

Visual Design Center The component specification is automatically saved with the model file, .mdl. However, at times you might want to save a text version. For example you might want to automate a build.

You can save a text version of the component specification for an entire project or a single component by right clicking on the selected item and choosing Export from the pop-up menu. Select the directory you wish to save the file in and click on Export. This will save a file using the name of the item you have selected with the extension . osc.

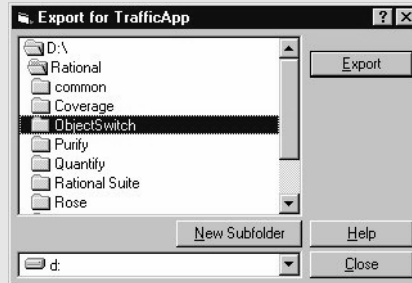


Figure 53: export window

It also exports the . soc and . act files.

Building the component

This section describes how you build a deployable component. Before you can build the component, you must have selected model sources and completely defined your component specification.

What can you build?

You can build either an entire project or a single component. Building a project is the same as building each component in the project separately.

Starting a build in Visual Design Center To build a component or project, right-click on it and choose Build from the pop-up menu..

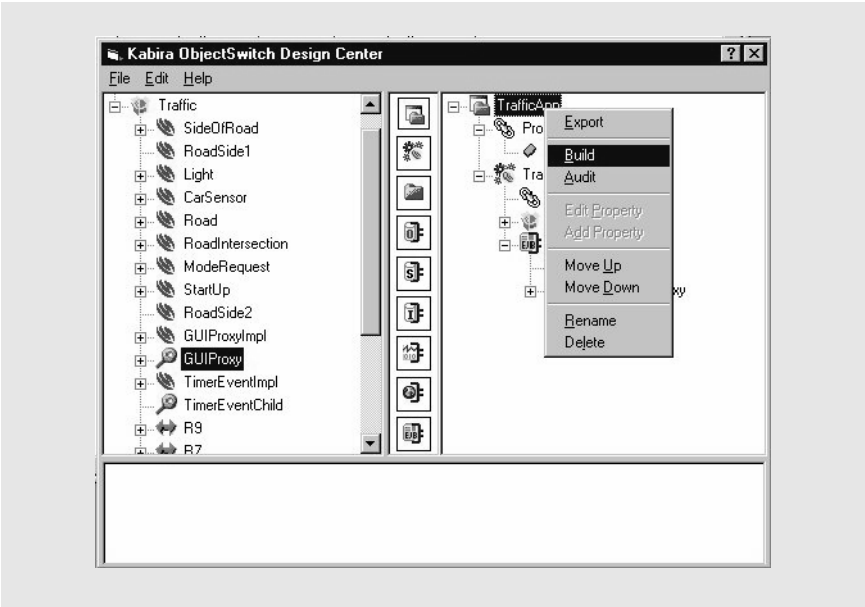


Figure 54: Starting the build process

Starting a build in text The utility `swbuild` allows you to build ObjectSwitch applications from the command line. It requires the model and the component specification files and is as follows:

```
swbuild [options] mySpecFileName
```

By default, `swbuild` performs both an audit and a build. The `[]` define options to perform an audit only, use the “-a” option. The optional “-o <macro>=<value>” assigns a value to a macro used in the build (see “Macros” on page 361.) The “spec file” is the Component specification file. If none is specified, standard input is used.

option	description
a	performs an audit
o	assigns a value to a macro used in the build

What is auditing?

Before the build begins the Design Center server will run an audit. This is automatically part of the build process. You may also run a separate audit before you build.

Auditing your model checks for many different kinds of errors that can be expressed in valid modeling syntax. The audit that takes place before a build includes not only model syntax, but action language too; this also performs adapter-specific checks for any adapters in the build. Some of the audit checks are:

- Are all element names unique within their scopes?
- Is the model free of inheritance “loops”?
- Are all entity and interface definitions complete?
- Are all modules contained in packages?
- Are all relationships contained entirely within single packages?
- Do relationships have at most one role per direction?
- Do state transitions use signals and states that are defined for the entity?
- Do interfaces expose only what exists?
- Do operations, attributes, parameters, and data types used in action language agree with their IDLos definitions?
- Are relationships traversals in action language possible in the IDLos model?
- Are properties such as key and singleton observed in action language?

What you get after you build

After a successful build, you get a set of output files. By default, these files, and some subdirectories, are created in the directory running the Design Center server. You can alter this location by specifying a directory using the `buildPath` property. See “Project properties” on page 125 or “Component properties” on page 130.

Chapter 4: Building ObjectSwitch components

Building the component

Although the build generates many files, there is only one that you need to be concerned with. That file is the component archive, named *componentname.kab* (where *componentname* is the name you gave your component in the Visual Design Center or the component block in the text file). This is the archive file that you deploy on an ObjectSwitch node using the Engine Control Center. See the *Deploying and Managing ObjectSwitch Applications* for more about deploying components using the Engine Control Center.

5

Accessing ObjectSwitch through PHP

PHP is a widely-used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. The functionality of PHP can be extended by building dynamically loaded *extensions*.

The ObjectSwitch PHP4 extension provides access to ObjectSwitch objects from within PHP scripts. This lets you invoke requests on a web server that interact directly with ObjectSwitch applications.

ObjectSwitch also has a built-in PHP interpreter, so you can execute PHP scripts from within ObjectSwitch actions.

This chapter begins with several sections that describe how to use the PHP extensions generally:

- Overview
- Data types
- PHP script language
- PHP4 Extension

The chapter continues with a detailed description of each function in the ObjectSwitch PHP extension:

- `os_connect`
- `os_create`
- `os_delete`
- `os_disconnect`
- `os_extent`
- `os_get_attr`
- `os_invoke`
- `os_relate`
- `os_role`
- `os_set_attr`
- `os_unrelate`

The chapter finishes with the following sections that describe specific aspects of the PHP extension and provide an example:

- Web Server—Apache
- Command Line Utility
- Execute PHP in action language
- Transactions
- A PHP example

Overview

The ObjectSwitch PHP4 extension provides a set of PHP functions to access ObjectSwitch applications using PHP scripts. The scripts can be executed in any of several ways:

- from a web browser
- from a command line using the CGI version of PHP
- from within an ObjectSwitch action

The following table presents a brief description of each function call in the PHP extensions.

PHP extension function	Description
os_connect	create a new connection to the ObjectSwitch osw engine
os_create	creates an ObjectSwitch entity and returns its reference
os_delete	delete an ObjectSwitch entity
os_disconnect	disconnect from the ObjectSwitch engine
os_extent	retrieve the extent of object handles for a type
os_get_attr	retrieves the value of all attributes in an ObjectSwitch entity
os_invoke	invoke an operation on an ObjectSwitch entity
os_relate	relate two interfaces
os_role	retrieves an array of handles by navigating across a relationship
os_set_attr	set the value of an ObjectSwitch attribute
os_unrelate	un-relate two interfaces

For more detailed information on each of the function calls see the reference in Chapter 11.

Data types

By nature, PHP is a loosely typed system and ObjectSwitch is a strongly typed system. Therefore, there is no exact type mapping between PHP and ObjectSwitch. The ObjectSwitch PHP extension supports basic types, enums and object references. It also supports arrays and sequences of basic types, enums and object references. All variables default to type string. At runtime, PHP decides the variable's type by its context.

Basic types

The table below lists the basic types that are supported. The following sections provide details of how each of the types is supported.

ObjectSwitch types	PHP types
short	string
long	
unsigned short	
unsigned long	
long long	
unsigned long long	
float	
double	
char	
octet	
string	
wchar	
wstring	

Boolean

In PHP the boolean value for false is 0 or "" (empty string) and all other values are true. The tables below show the mapping between ObjectSwitch and PHP boolean values.

These ObjectSwitch booleans...	map to these PHP booleans
SW_FALSE	0
SW_TRUE	1

These PHP booleans...	map to these ObjectSwitch booleans
0 (zero) or "" (empty string)	SW_FALSE
All other values	SW_TRUE

Enum

The symbolic name of enums are used in PHP.

Object references

References to ObjectSwitch objects in PHP are represented by strings. The internal format of the string is not documented. Do not manipulate object references directly. One special case is when the object reference represents an empty object. It is "" (empty string).

Arrays and sequences

ObjectSwitch arrays, sequences and bounded sequences are mapped to PHP arrays. PHP arrays are truly dynamic. It does not have any size limitation nor do the elements have to have the same type.

When arrays are used with the ObjectSwitch PHP extension the elements must be either:

- basic type
- enum
- object reference

You must keep in mind that ObjectSwitch has bound checking for arrays and bounded sequences.

Note: See `os_set_attr()` and `os_invoke()` for sequence type limitations.

Unsupported types

The following ObjectSwitch types are not supported.

- struct
- union
- any
- native

PHP script language

The ObjectSwitch PHP4 extension must be executed from within a PHP script. This section briefly discusses how to write PHP scripts.

PHP Syntax

The following is a PHP script example

```
<html><head><title>PHP Test</title></head>
<body>
<?php echo "Hello World<p>"; ?>
</body></html>
```

The code between `<?php` and `?>` is PHP script. A PHP script need not be embedded inside of HTML file. It could simply be

```
<?php echo "Hello World<p>"; ?>
```

PHP script is free syntax and loosely typed language. All variables used in the script must have a prefix of dollar sign `$`. All statements must end with `;`.

```
<?php
    $name = "Jone Doe";
    $num = 5;
    $flag = true;
    $list = array (1, 2, 3, 4);
    $attrList = array("m_name" => "John", "ssn" =>55555555);
?>
```

Using the extension

You must call the function `os_connect()` before using any other ObjectSwitch extension functions. This gets a connection to the ObjectSwitch Web Engine (osw) and subsequent requests are processed by osw.

```
<?php
    $conn_id = os_connect("local host", 7654);
    $obj_r = os_create($conn_id, "myPkg::mylfc");
    $msg = os_invoke($conn_id, $obj_r, "getMessage");
    print $msg;
    os_delete($conn_id, $obj_r);
    os_disconnect($conn_id);
?>
```

Error handling

The error handling behavior in PHP is controlled by the settings in `php.ini` file.

<code>error_reporting</code>	<i>bit mask controls what to report</i>
<code>display_errors</code>	On or Off
<code>track_errors</code>	On or Off

If `display_errors` is on, error message will show up on the web page. If `track_errors` is on, when error happens the message will also be set in a PHP variable called `$php_errormsg`. The following script segment shows how to use `$php_errormsg`

```
<?php
    $php_errormsg = "";
    $conn_id = os_connect("local host", 7654);
    if ($php_errormsg != "")
    {
        print "Connect trouble: $php_errormsg\n";
        exit(-1);
    }
    ...
?>
```



More information on standard PHP and HTML scripting languages can be found in any PHP or HTML manual. Or visit the PHP web site at <http://www.php.net>

PHP4 Extension

This section provides a detailed description of each function in the ObjectSwitch PHP extension. This information can also be found in Chapter 11.

os_connect

Creates a new connection to ObjectSwitch osw engine.

Syntax

```
$conn_id = os_connect ($host, $port);  
$conn_id = os_connect ($host);  
$conn_id = os_connect ( );
```

Description

This function creates a connection between the PHP process and the *osw* engine.

conn_id is a PHP variable.

host and *port* are optional parameters that locate the *osw* engine.

If *host* and *port* are not set they will default to the *host* and *port* specified in the `php.ini` file.

Warnings

None

Error Conditions

- Not able to connect to ObjectSwitch server

os_create

Creates an ObjectSwitch object and returns its reference.

Syntax

```
$objr = os_create ($conn_id, $scopedName, $attrList);  
$objr = os_create ($conn_id, $scopedName);
```

Description

This function creates an object and returns its reference in the variable *objr*. If the object has attributes, an optional *attrList* could be passed in to set the attribute initial values.

When the *scopedName* type is a singleton, this call acts like the *create singleton* in action language. It creates the singleton if it does not exist. Otherwise, it returns the object reference.

conn_id is the value returned by `os_connect()`.

objr is the return value.

scopedName is a fully scoped ObjectSwitch interface name.

attrList is an optional associative array of attribute names and values.

Example

```
//  
// create a sales person with "name" initialized  
//  
$salesType = "myPackage::Salesman";  
$salesAttrArray = array('name' => 'Slick Willy');  
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);  
  
//  
// create a customer - without attribute array  
//  
$custType = "myPackage::Customer";  
$custHandle = os_create($conn_id, $custType);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid scoped name
- Invalid attribute name
- Unsupported type
- No create access
- Duplicate Key

os_delete

Deletes an ObjectSwitch object.

Syntax

```
os_delete($conn_id, $objrList);  
os_delete($conn_id, $objr);
```

Description

The second parameter may be either a single object reference or an array containing a list of object references.

conn_id is the value returned by `os_connect()`.

objrList is an array of valid object instances to delete.

objr is a valid object instances to delete.

Warnings

None.

Example

```
//  
// Delete all Orders  
//  
$sn = "mypackage::Order";  
$orderList = os_extent($conn_id, $sn);  
os_delete($conn_id, $orderList);  
  
//  
// delete a single object  
//  
$objr = os_create($conn_id, $sn);  
os_delete($conn_id, $objr);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- No delete access

os_disconnect

Disconnect from the ObjectSwitch engine.

Syntax

```
os_disconnect ($conn_id);
```

Description

conn_id is the return value from a call to `os_connect()`.

Warnings

None.

Example

```
/* close a connection to the osw engine */  
os_disconnect($xconn);
```

os_extent

Retrieve the extent of object handles of a given type.

Syntax

```
SobjrList = os_extent($conn_id, $scopedName, $attrList);  
SobjrList = os_extent($conn_id, $scopedName);
```

Description

This function will select objects of a given type. If *attrList* is provided, it contains a list of name-value pairs that are *AND*ed together as a where clause to filter the number of object handles being returned.

If the objects are keyed and *attrList* has key coverage, a keyed lookup will be performed.

conn_id is the value returned by `os_connect()`.

objrList is the return value and is a PHP array of scalar values.

scopedName is a string containing the fully scoped name of a type.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//  
// get all customers  
//  
$sn = "myPackage::Customer";  
$custList = os_extent($conn_id, $sn);  
  
//  
// return all customers in California  
//  
$sn = "myPackage::Customer";  
$whereClause = array('state' => 'CA');  
$custList = os_extent($conn_id, $sn, $whereClause);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid scoped name
- Invalid attribute name
- Unsupported type or bad value
- No extent access

os_get_attr

Retrieves the values of the attributes in an ObjectSwitch object.

Syntax

```
$attrList = os_get_attr($conn_id, $objr, $filter);
$attrList = os_get_attr($conn_id, $objr);
```

Description

This function retrieves attribute values from an object. If *filter* exists, only the values of those attributes named in the filter array will be returned. Otherwise, all attributes values will be returned.

conn_id is the value returned by os_connect().

attrList is an optional associative array of attribute names and values.

objr is an ObjectSwitch object handle.

filter is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//
// get all attributes of a cusotmer
//
$attrs = os_get_attr($conn_id, $custHandle);
for ( reset($attrs); $name = key($attrs); next($attrs) )
{
    $avalue = $attrs[$name];
    print "$name = $value\n";
}

//
// get the state attribute only
//
$filter = array ( "state" => "" );
$attrs = os_get_attr($conn_id, $custHandle, $filter);
$state = $attrs["state"];
print "This customer is in state of $state\n";
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid attribute name

os_invoke

Invoke an operation on an ObjectSwitch object.

Syntax

```
$returnValue = os_invoke($conn_id, $objr, $opName, $param, $ex);  
$returnValue = os_invoke($conn_id, $objr, $opName, $param);  
$returnValue = os_invoke($conn_id, $objr, $opName);
```

Description

This function is used to invoke an operation on an object. The *returnValue* is not required on *void* operations. If a parameter is an *inout* or *out* parameter the value of that array element in *param* will be modified with the result parameter value after a call. If the operation raises user defined exception, *\$ex* will contain the name of the exception type as a string upon return.

conn_id is the value returned by `os_connect()`.

returnValue is the return value of the operation upon completion.

objr is a valid object instance handle.

opName is the name of an operation.

param is a nested associative array of parameter name and value pairs.

ex is the name of user exception thrown by the operation.

os_invoke does not work with *in* or *inout* parameters of sequence type.



Example

```
//
// call a void operation that does not have any params
//
os_invoke($conn_id, $objr, "runtest");

//
// call an operation with parameters that returns a boolean
//
$params = array ("name" => "Smith", "number" => 5);
$ret = os_invoke($conn_id, $objr, "register", $params);
if ($ret)
{
    print "OK\n";
}
else
{
    print "register failed\n";
}
```

When an operation raises user defined exceptions, `os_invoke()` can get the exception type, but not the exception data if it has member fields.

```
//
// operation with user defined exceptions
//
$userex = "";
$params = array();
os_invoke($conn_id, $objr, "myOp", $params, $userex);
if ($userex != "")
{
    print "Caught user exception $userex\n";
}
```

Array types can be used as *in*, *out*, *inout* parameters and return values. Sequence types can be used as *out* parameter or return values.

```
//
// array or sequence type as out param
//
$params = array ( "myList" => array() );
$ret = os_invoke($conn_id, $objr, "getList", $params);
$list = $params["myList"];
for ($i = 0; $i < count($list); $i++)
{
    $value = $list[$i];
    print "list[ $i ] = $value\n";
}
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid operation name
- Invalid parameter name
- Unsupported type or bad value
- Application exception

os_relate

Relates two interfaces.

Syntax

```
os_relate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function relates two objects using the relationship role *roleName*.

conn_id is the value returned by `os_connect()`.

fromObjr is a valid object reference handle.

roleName is the name of a role in a relationship between the “from” and “to” objects.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$salesType = "myPackage::Salesman";
$salesAttrArray = array('name' => 'Slick Willy');
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);

$custType = "myPackage::Customer";
$custAttrArray = array('name' => 'Lucent', 'state' => 'NJ');
$custHandle = os_create($conn_id, $custType, $custAttrArray);
os_relate($conn_id, $salesHandle, "hasCust", $custHandle);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid role name

os_role

Retrieves an array of handles by navigating across a relationship.

Syntax

```
$objrList = os_role($conn_id, $objr, $roleNam, $attrList);  
$objrList = os_role($conn_id, $objr, $roleNam);
```

Description

This function returns an array of object references as it navigates across the appropriate relationship. The name-value pairs in *attrList* are *AND*ed together to act as a *where* clause to filter the number of object handles being returned.

conn_id is the value returned by os_connect().

objrList is a list of object references.

objr is an ObjectSwitch object handle.

roleName is a relationship role name.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
/*  
** assume Salesperson is related 1:M with customer  
** and that $salesHandle is already populated with a  
** valid Salesperson.  
*/  
$cList = os_role($conn_id, $salesHandle, "sellsto");  
for ( reset($cList); $cust = current($cList); next($cList))  
{  
    /* do something with $cust */  
}
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid role name
- Invalid attribute name
- Unsupported type or bad value

os_set_attr

Sets a value in an attribute of an ObjectSwitch object.

Syntax

```
os_set_attr($conn_id, $objr, $attrList);
os_set_attr($conn_id, $objr, $name, $value);
```

Description

This function sets a value of an object attribute. More than one attribute value can be set in an object.

conn_id is the value returned by `os_connect()`.

objr is an ObjectSwitch object handle.

attrList is an optional associative array of attribute names and values.



Setting uninitialized attributes of sequence type does not work. However, there is a work around using pre-set triggers.

Example

```
//
// set the name of a customer
//
$attrArray = array('name' => 'John');
os_set_attr($conn_id, $objr, $attrArray);

//
// or using
//
os_set_attr($conn_id, $objr, "name", "John");

//
// set array attribute
//
$value = array (1, 2, 3, 4, 5);
$attrArray = array ("longList" => $value);
os_set_attr($conn_id, $objr, $attrArray);
```

Using pre-set trigger to set sequence attributes This section describes the workaround that lets you set sequence attributes from PHP.

```
//
// set sequence attribute with pre-set trigger
// this is what needs to be done in the model.
//
package Example
{
    typedef sequence<string>StringList;

    interface Complex
    {
        attribute StringList t_stringlist;
    };

    entity ComplexImpl
    {
        attribute StringList_stringlist;

        void t_stringlist_init();
        trigger t_stringlist_init upon pre-set t_stringlist;
    };

    expose entity ComplexImpl with interface Complex;
};

action :: complexTest :: !ComplexImpl :: t_stringlist_init
{
    declare StringList l;

    disableTriggers();
    self.t_stringlist = l;
    enableTriggers();
};
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid attribute name
- Attribute is readonly
- Unsupported type or bad value
- Duplicate Key

os_unrelate

Unrelates two objects.

Syntax

```
os_unrelate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function unrelates two objects that are currently related using the relationship role *roleName*.

conn_id is the value returned by `os_connect()`.

fromObjr is a valid object reference handle.

roleName is a relationship role name.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$custList = os_role($conn_id, $salesHandle, ":sellsto");  
for (reset($custList); $cust = current($custList); next($custList))  
{  
    os_unrelate($conn_id, $salesHandle, 'hasCust', $cust);  
}
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid role name

Web Server—Apache

When a web browser executes a PHP script that uses the ObjectSwitch PHP extension, it needs a web server that is PHP enabled and has ObjectSwitch extension loaded. Various web servers can be used with PHP, but Kabira has only certified the PHP extensions using the Apache server.

Figure 55 illustrates the flow of information between the web browser and the ObjectSwitch application.

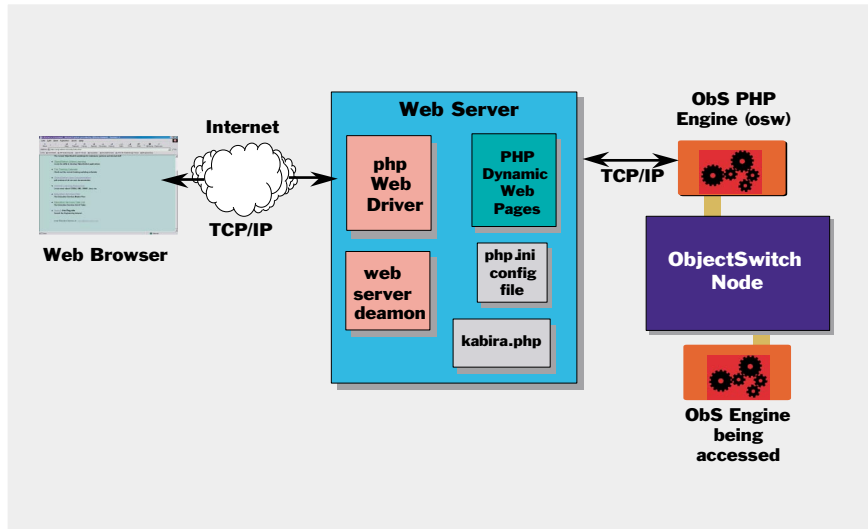


Figure 55: PHP extension architecture

Apache and PHP software are freely available. In order to build Apache and PHP, a number of GNU tools are required.

Please refer to \$SW_HOME/OS.README.<version> file for details of how to build Apache and PHP.

Refer to \$SW_HOME/OS.3RDPARTY.<version> for the supported version of Apache and PHP software.

Command Line Utility

A shell script wrapper for the CGI version of PHP is shipped with the release. The script name is “swphp”.

Usage

Usage: swphp [-p path] [-g var=value] script_file

-p Set runtime path (default .)

-g Set init value for \$var in script.

Note: -g option must appear after all other options.

Like PHP scripts executed in a web browser, swphp also connects to the *osw* engine to process ObjectSwitch extension functions. The *swstart* command creates a *php.ini* file in the current directory. It has the settings for swphp to start up the ObjectSwitch extension within PHP.

The -p option tells swphp the directory path of where swstart was run. If it is executed in the same directory as swstart, -p option can be omitted.

The -g option sets variable initial values in the PHP script.

Example

```
swphp -g sn=myPkg:mylfc -g opName=dolt run.php
```

```
<? php
// File: run.php
//
// no need to give host and port, swphp will find the
// right host and port from the registry.
//
$x_conn = os_connect();

//
// the value of $sn and $opName are set by -g option of swphp
//
$obj_r = os_create($x_conn, $sn);
$msg = os_invoke($x_conn, $obj_r, $opName);
os_delete($x_conn, $obj_r);
print "calling $sn $opName returns $msg\n";
?>
```

Execute PHP in action language

This capability is also known as “PHP callout”. The PHP interpreter is modeled in the *swmetadii* package, an interface called PHP. The implementation is built into *osw* engine.

```
package swmetadii
{
    interface PHP
    {
        void executeScript(
            in string scriptFile,
            in NameValueList inParamList,
            inout NameValueList outParamList)
            raises (Failed);

        void evalString(
            in string scriptString,
            in NameValueList inParamList,
            inout NameValueList outParamList)
            raises (Failed);
    };
};
```

To build an an ObjectSwitch application that uses PHP callouts, you need to import the *osw* component in your build specification as shown below.

```
component MYCOMP
{
    import osw;
    ...
};
```

If you are using the Visual Design Center you also need to import the *osw* component and use the *swmetadii* package. See the online documentation for the *swmetadii* package.

The following action language segment shows how to use this component to perform callouts to PHP scripts.

```
declare swmetadii::PHPcaller;
create caller;

declare swmetadii::NameValueListparamlist1;
declare swmetadii::NameValueListparamlist2;
declare swmetadii::NameValueparam;

param.name = "VarName";
param.value = "VarValue";
paramlist1[0] = param;

param.name = "outmsg";
param.value = "";
paramlist2[0] = param;

declare string scriptFile = "test.php";
try
{
    caller.executeScript(scriptFile, paramlist1, paramlist2);
    param = paramlist2[0];
    if (param.value != "passed")
    {
        // something went wrong.
    }
}
catch (swmetadii::Failed e)
{
    msg = e.emsg;
    fprintf(stderr, "php call failed [%s]\n", msg.getCString());
}
```

Chapter 5: Accessing ObjectSwitch through PHP

Execute PHP in action language

The content of the PHP script “test.php” might look like the example below:

```
<?php
    if ($VarName == "VarValue")
    {
        print "in param has correct value\n";
        $outmsg = "passed";
    }
    else
    {
        print "in param does not have correct value\n";
        $outmsg = "failed";
    }
?>
```


Transactions

This section describes how transactions are managed when using the PHP extensions.



The transaction boundary varies, depending on whether PHP scripts are executed in a web browser, from the command line, or from action language.

Web browser or command line transactionality

In a web browser or from a command line, each PHP extension function call creates its own transaction. For instance,

```
$obj r = os_create($conn_id, "myPkg::myIfc");
```

When an *os_create()* call returns to the PHP script, two things have already happened:

- an object has been created
- the transaction of creating the object has committed

The next call that uses *\$obj r* to invoke an operation will be in a separate transaction.

The only way to ensure transactionality over a series of calls is to wrap the calls in a single function in the ObjectSwitch model, then invoke that one function from PHP. Suppose you want to write a PHP script such as:

```
os_create(...);
os_set_attr(...);
os_relate(...);
...
```

If you want to make all of these calls execute in the same transaction, you need to model these functions as a single operation in your ObjectSwitch application, for example:

```
action create_and_relate()
{
    create obj;
    obj.name = "John";
    relate obj ... ;
    ...
};
```

These functions will now be performed as a single transaction by invoking the operation with *os_invoke()*:

```
os_invoke( . . , "create_n_relate()", . . . );
```

Action language callout transactionality

When a PHP script is executed from within action language, it inherits the transaction environment from the caller object. The extension functions *do not* start their own transactions. Therefore the whole script is executed in the same transaction.

A PHP example

This example demonstrates how to create object references, set and get attributes, create and traverse relationship roles, and invoke operations on an object using a PHP script.

There is a model with customers and orders, and PHP scripts to create new customers, create new orders, and get customer balances.

The model

This section shows the UML and IDLos versions of the model used in this example.

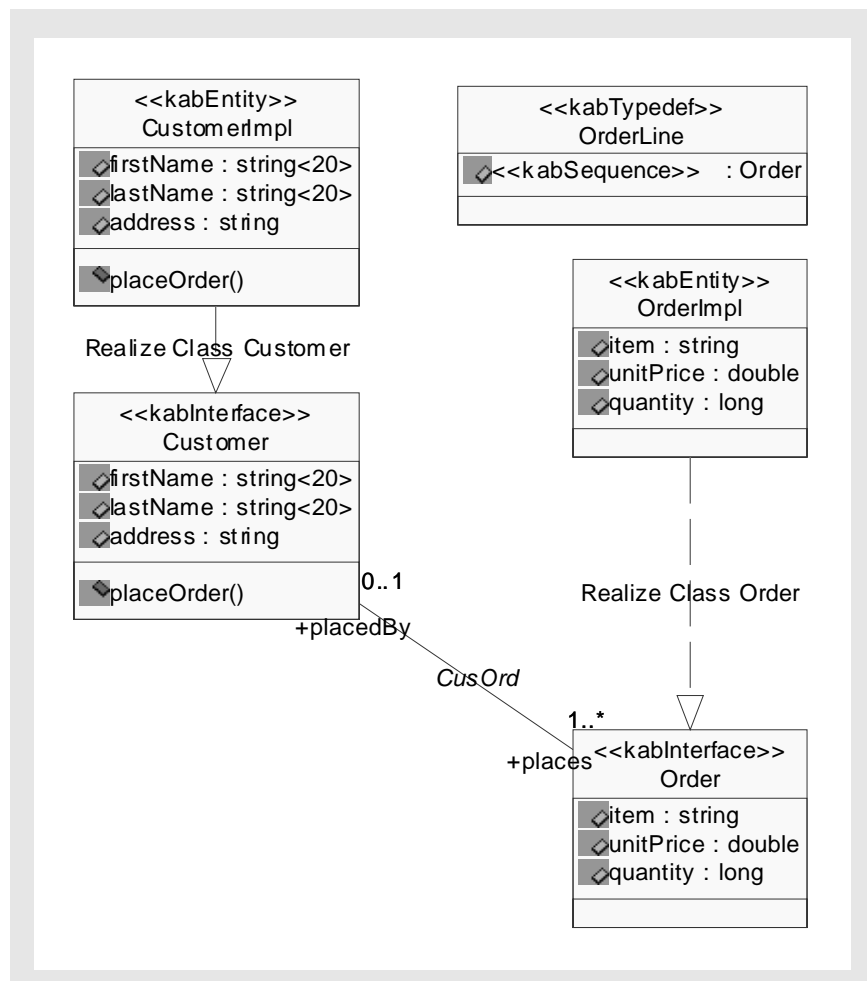


Figure 56:

```
package orderMngt
{
    interface Order {
        attribute string item;
        attribute double unitPrice;
        attribute long quantity;
    };
    entity OrderImpl {
        attribute string item;
        attribute double unitPrice;
        attribute long quantity;
    };
    expose entity OrderImpl with interface Order;

    typedef sequence<Order>OrderLine;

    interface Customer {
        attribute string<20> firstName;
        attribute string<20> lastName;
        attribute string address;
        key PKey { firstName, lastName };

        boolean placeOrder(inout Order ord);
    };
    entity CustomerImpl {
        attribute string<20> firstName;
        attribute string<20> lastName;
        attribute string address;
        key PKey { firstName, lastName };

        boolean placeOrder(inout Order ord);
        action placeOrder {`
            if (empty ord) {
                return false;
            }
            relate self places ord;
            return true;
        `};
    };
    expose entity CustomerImpl with interface Customer;

    relationship CusOrd {
        role Customer places 1..* Order;
        role Order placedBy 0..1 Customer;
    };
};
```

Example scripts

This section shows the PHP scripts for the example.

add_customer.php This PHP script adds a new customer.

```
<?php
    $php_errormsg = "";
    $conn_id = os_connect();
    if ( $php_errormsg != "" )
    {
        print "connect to OSW failed: $php_errormsg\n";
        exit (-1);
    }

    $attrList = array ( "firstName" => "John",
                        "lastName" => "Smith" );

    $typeName = "orderMngt::Customer";

    $cus = os_create($conn_id, $typeName, $attrList);
    if ( $php_errormsg != "" )
    {
        print "add customer John Smith failed: $php_errormsg\n";
        os_disconnect($conn_id);
        exit (-1);
    }

    print "Added new customer John Smith";

    os_disconnect($conn_id);
?>
```

add_order.php This PHP script adds an order for an existing customer.

```
<?php
    $php_errormsg = "";
    $conn_id = os_connect();
    if ( $php_errormsg != "" )
    {
        print "connect to OSW failed: $php_errormsg\n";
        exit (-1);
    }
```

```

$attrList = array ("firstName" => "John",
                  "lastName" => "Smith");

$typeName = "orderMngt::Customer";

$cusList = os_extent($conn_id, $typeName, $attrList);
if ( $php_errormsg != "" )
{
    print "find customer John Smith failed: $php_errormsg\n";
    os_disconnect($conn_id);
    exit(-1);
}
if (count($cusList) == 0){
    print "did not find John Smith";
    os_disconnect($conn_id);
    exit(-1);
}

$attrList = array ("item" => "Apple",
                  "unitPrice" => 0.99,
                  "quantity" => 50);

$typeName = "orderMngt::Order";

$ord = os_create($conn_id, $typeName, $attrList);
if ( $php_errormsg != "" )
{
    print "create order failed: $php_errormsg\n";
    os_disconnect($conn_id);
    exit(-1);
}

$params = array ("ord" => $ord);
$cus = $cusList[0];
$status = os_invoke($conn_id, $cus, "placeOrder", $params);
if (! $status)
{
    print "place order for John Smith failed\n";
    os_disconnect($conn_id);
    exit(-1);
}

print "added order for John Smith\n";

os_disconnect($conn_id);

```

?>

Chapter 5: Accessing ObjectSwitch through PHP

A PHP example

get_balance.php This script gets the current balance for an existing customer's account.

```
<?php
    $php_errormsg = "";
    $conn_id = os_connect();
    if ( $php_errormsg != "" )
    {
        print "connect to OSW failed: $php_errormsg\n";
        exit (-1);
    }

    $attrList = array ( "firstName" => "John",
                        "lastName" => "Smith" );

    $typeName = "orderMngt::Customer";

    $cusList = os_extent($conn_id, $typeName, $attrList);
    if ( $php_errormsg != "" )
    {
        print "find customer John Smith failed: $php_errormsg\n";
        os_disconnect($conn_id);
        exit(-1);
    }
    if (count($cusList) == 0){
        print "did not find John Smith";
        os_disconnect($conn_id);
        exit(-1);
    }

    $orderList = os_role($conn_id, $cusList[0], "places");

    $total = 0.0;

    for ($i = 0; $i < count($orderList); $i++)
    {
        $attrList = os_get_addr($conn_id, $orderList[$i]);
        $unitPrice = $attrList["unitPrice"];
        $quantity = $attrList["quantity"];

        $total += $unitPrice * $quantity;
    }

    print "send a bill to John Smith for $total\n";

    os_disconnect($conn_id);

?>
```


Part Two: ObjectSwitch Reference

6

Lexical and syntactic fundamentals

This chapter describes the basic lexical rules of IDLos. These rules follow the IDL (version 2.2). This also applies to action language and build specifications, so you'll need to understand the rules whether you're modeling textually or graphically.

The chapter covers the character set, tokens, white space, comments, pre-processing directives, and the syntax for properties.

Character set

IDLos supports the ISO Latin-1 (8859.1) character set. This consists of:

Decimal digits These are: 0 1 2 3 4 5 6 7 8 9

Graphic characters The graphic characters are: ! " # \$ % & ' () * + = . / : ; < = > ? @ [\] ^ _ ' { | } ~ , plus the extended set.

Formatting characters The formatting characters are: BEL, BS, HT, LF, NL, VT, FF, CR.

Alphabetic characters The alphabetic characters are a-z and A-Z. plus these: àÁ áÂ âÃ äÅ æÆ çÇ èÈ éÉ êË ëË ìÍ íÎ ïÏ ñÑ òÒ óÓ ôÔ õÕ öÖ øØ ùÛ úÛ ûÛ üÛ ÿ and the Icelandic thorn and eth characters, both capital and lowercase.

Tokens

There are five kinds of tokens: keywords, identifiers, literals, operators and other separators.

Keywords The following table lists all the keywords reserved by IDLoS; keywords that do not occur in IDL are shown in **bold**.

Although this is a large table, there are only about 20 statements that need to be defined in the IDLoS reference pages. IDLoS really is a small language; many of the statements use two or three keywords, and there are 14 basic types of keywords. The trigger statement alone uses 11 keywords.

abort	fixed	relationship
abstract	float	role
action	from	sequence
annotation	ignore	singleton
any	in	short
attribute	initialize	signal
boolean	inout	stateset
cannothappen	interface	string
case	key	struct
char	local	switch
commit	long	terminate
const	module	to
context	native	transition
create	Object	trigger
createaccess	octet	true
default	oneway	TRUE
delete	out	typedef
deleteaccess	package	unsigned
double	post-get	union
during	post-set	unrelate
enum	pre-get	upon
entity	pre-set	using
exception	raises	void
expose	readonly	wchar
extentaccess	refresh	with
false	recovery	wstring
FALSE	relate	virtual
finished		

The keywords context, fixed, native, wchar and wstring are parsed, but are not used by ObjectSwitch Design Center at this time.

Identifiers An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore (“_”) characters. The first character must be an alphabetic character. Only one of the letters A-Z or a-z may be used as the first character. Identifiers are case sensitive.

Unlike IDL, identifiers in IDLos are case sensitive.

i

If you need to use a keyword as an IDLos identifier, you can delimit it with quotation marks, for example:

```
entity purchase
{
    attribute long amount;
    attribute boolean "commit";
}
...
declare purchase p;
create p;
p."commit"=FALSE;
```

You can use any IDLos or action language keyword as a quoted IDLos identifier, unless it is also a C++ or IDL reserved word. Be aware that in some cases this can lead to ambiguity, for example:

```
void myAction(in long "commit"); // quoted keyword as parameter
...
// if the action contains this expression
if ("commit" == 1) // then "commit" is parsed as a string literal
// and not as the input parameter
```

Literals Literals may be integer, character, floating point, string, or boolean:

integer	A sequence of numerals beginning with 0 (e.g. 077) is interpreted as an octal number. A sequence of numerals beginning with any other number (e.g. 63) is interpreted as a decimal number. A sequence of numerals prefixed with 0x or 0X (e.g. 0x3F) is interpreted as a hexadecimal number. Hexadecimals may use a-f or A-F. A sequence of numerals appended with LL is interpreted as a 64-bit integer (long long).
---------	---

character	A character literal is enclosed in single quotes (e.g. 'Z'). Standard IDL escapes are allowed, such as '\n' for newline and '\x67' for hexadecimal rendition. There is no support for wide character literals at this time.
-----------	---

floating point	A floating point literal consists of an integer part, a decimal point, a fraction part, and e or E, and an optionally signed integer exponent (e.g. 1. 024+e13). The decimal point <i>or</i> the exponent may be missing, but not both. The integer part <i>or</i> the fraction part may be missing, but not both.
string	String values are placed in double quotes (e.g. "Thi s i s a stri ng"). Adjoining strings are concatenated to form long strings. Newlines and other characters must be embedded with escape sequences.
boolean	A boolean literal: TRUE, true, FALSE, or false.

Escape sequences The following table defines the escape sequences

Description	Escape sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
value of character in octal	\ooo
value of character in hexadecimal	\xhh

Operators The operator tokens in IDLos are: - + ~ = ! || && / * () ?.

Other separators Other separators in IDLs are:

- preprocessor token #
- action language delimiter ‘
- statement delimiter ;
- list delimiter ,
- inheritance delimiter :
- scoping delimiter ::
- block grouping { }
- list grouping ()
- array definition, property grouping []
- sequence declaration < >
- comment tokens // /* */
- role multiplicity delimiter ..

White space

White space is comprised of blanks, horizontal and vertical tabs, newlines, formfeeds and comments. White space is ignored except as it serves to separate tokens. For example, when an inherited interface must be scoped

```
interface A : ::SomePackage::B {};      // correct
interface B ::SomePackage::B {};      // wrong! :: causes error
```

Comments

Single line comments start with // and continue to a newline. C-style comments start with a /* and continue until a */, and may include newlines.

Comments do not nest. Once a comment has started, all comment delimiters are ignored except the end delimiter for that style of comment.

Preprocessing directives

The following preprocessing directives are supported. Any other line where the first non-white space character is '#' is ignored.

`#include`

When `#include` is the first non-white space on a line, the text after the `#include` is interpreted as a file name and will be included at that point in the text. The file is processed as IDLs.

`#pragma include`

Include the specified file in the generated output. The file won't be parsed by the IDLs loader.

ObjectSwitch IDLs supports `#include "headerfile.h"`. The IDLs loader adds the local directory "." to the include path so there is no difference between `#include <headerfile.h>` and `#include "headerfile.h"`. This differs from normal cpp rules for `#include`.

The next chapter describes the ObjectSwitch type system.

7

ObjectSwitch types

This chapter describes the ObjectSwitch type system.

Types are labels that specify what category of information a symbol indicates. Usually one category is not comparable with another (for example, a string is not comparable with a float). ObjectSwitch uses the IDL (version 2.2) type system, which provides a rich system of basic and user-defined types.

For each type, there is a brief description of the semantics of the type, and, where appropriate, how to specify the type in the Visual Design Center, IDLos and/or Action Language. That will be followed by one or more examples, and any general information.

The following types are supported by ObjectSwitch. Simple types are marked with an asterix(*).

- any*
- array
- boolean*
- bounded sequence
- bounded string
- bounded wstring
- char*
- const
- context
- double*
- entity
- enum
- exception
- extern
- fixed*
- float*
- interface
- long*
- long double*
- long long*
- native
- Object
- octet*
- pipe
- sequence
- short*
- string
- struct
- typedef

- union
- unsigned long*
- unsigned long long*
- unsigned short*
- void
- wchar
- wstring

any

Semantics

The type any is an IDL simple type. It can hold any simple IDLos type. An any logically contains a TypeCode, and a value that is described by the TypeCode.

Visual Design Center syntax

Select the kabEnti ty class and open the specification. Click on the Attri butes tab and then click on the Add button to create a new attribute. Click on the Type pulldown selection and select any.

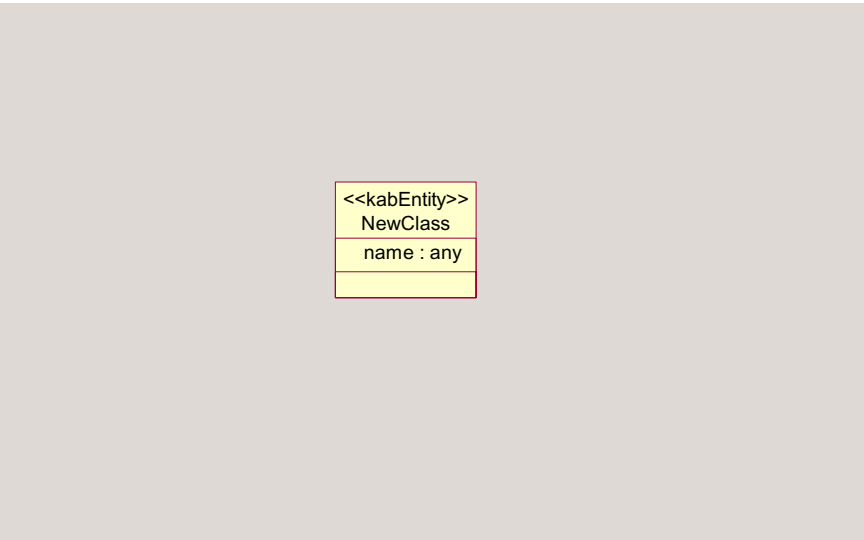


Figure 57: Create a class attribute of type "any"

To create a sequence or array of type any, open a kabTypedef class specification. Click on the Attributes tab and select an attribute of type any. Click on the Edit button and set the Stereotype to kabArray, kabSequence or kabSequenceOfArray. Click the Sequences tab and set the dimensions of the sequence or array.

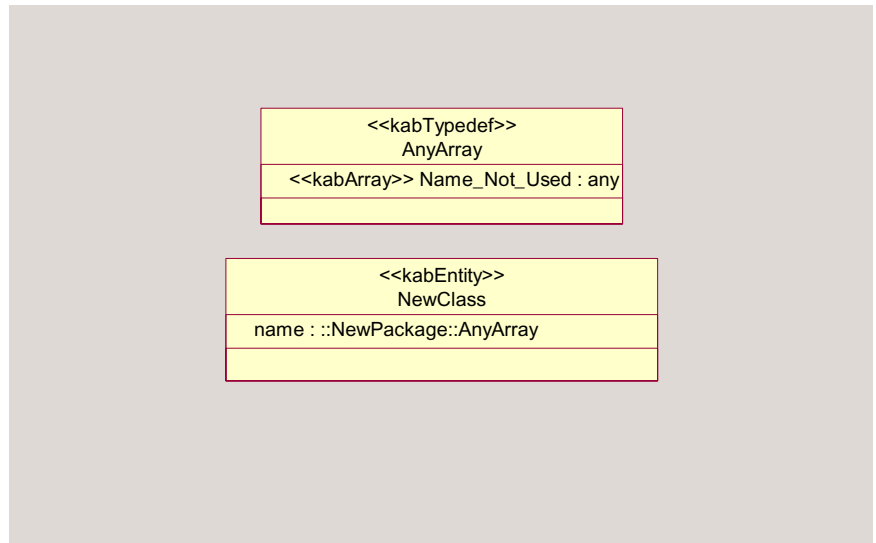


Figure 58: Create an array of type “any”

IDLos syntax

Inside of an entity block, define an attribute “name” of type any with:

```
attribute any name;
```

Inside of an entity block, define an array attribute “name” of type any with:

```
attribute any name[5];
```

Inside of an entity block, define a sequence “name” of type any with:

```
typedef sequence<any> name;
```

Inside of an entity block, define a bounded sequence “name” of type any with:

```
typedef sequence<any, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type any with:

```
declare any name;
```

Inside of an action language function, allocate an array “name” of type any with:

```
declare any name[5];
```

Example

```
declare any myAny;  
declare any myAny2;  
declare string str;  
declare string str2;  
str = "A string lives here.";   
myAny <<= str;  
myAny2 = myAny;  
myAny2 >>= str2;
```

General Information

You must unpack an any into a C-style array in order to pass it to a C++ function call.

Operator Descriptions:

=	Assigns one any to another any
<<=	Inserts a value into an any.
>>=	Extracts a value from an any.

The any is always the first operand of the insertion and extraction operators.

If there is a type mismatch between the value contained in an any and the target of an extraction operator, the following exception occurs:

```
swbutil::ExceptionAnyTypeMismatch
```

You should wrap an extraction operation in a try-catch block that catches the above exception.

```
action myPackage::myEntity::myOp
{
    declare any myAny;
    declare string str;
    str = "A string lives here.";
    myAny <<= str;
    ...
    try
    {
        myAny >>= str;
        printf("%s\n", str.getCString());
    }
    catch (swbuiltin::ExceptionAnyTypeMismatch)
    {
        printf("A string doesn't live here any more.\n");
    }
};
```

array

Semantics

The type array is an IDL composite type. It represents a known-length series of a single type. An array must be named by a typedef declaration in order to be used as a parameter, an attribute, or a return value. You can omit a typedef declaration only for an array that is declared within a structure definition.

Visual Design Center syntax

See Visual Design Center syntax for a simple type, such as any.

IDLos syntax

See IDLos syntax for a simple type, such as any.

Action language syntax

See Action language syntax for a simple type, such as any.

Example

This struct defines 256x256 array of 8-bit octets to hold an image:

```
struct SmallImage
{
    octet pixel [256][256];
};
```

General Information

You must unpack an array into a C-style array in order to pass it to C++ function call.

IDLos and Action Language support any number of dimensions. All dimensions in an array must be bounded. To define an array with unbounded dimensions, use a sequence.

boolean

Semantics

The type `boolean` is an IDL simple type. It represents a data item that can only be assigned the values of `true` or `false`.

Visual Design Center syntax

Select an appropriate class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `boolean`.

To create a sequence or array of type `boolean`, open a `callback` class specification. Click on the **Attributes** tab and select an attribute of type `boolean`. Click on the **Edit** button and set the **Stereotype** to `callbackArray`, `callbackSequence` or `callbackSequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `boolean` with:

```
attribute boolean name;
```

Inside of an entity block, define an array “name” of type `boolean` with:

```
attribute boolean name[5];
```

Inside of an entity block, define a sequence “name” of type `boolean` with:

```
typedef sequence<boolean> name;
```

Inside of an entity block, define a bounded sequence “name” of type `boolean` with:

```
typedef sequence<boolean, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type boolean with:

```
declare boolean name;
```

Inside of an action language function, allocate an array “name” of type boolean with:

```
declare boolean name[5];
```

Example

```
declare boolean result;  
result = true;
```

bounded sequence

Semantics

The type `bounded sequence` is an IDL template type. Bounded sequences can hold any number of elements, up to the limit specified by the bound. A `bounded sequence` must be named by a `typedef` declaration in order to be used as a parameter, an attribute, or a return value. You can omit a `typedef` declaration only for a sequence that is declared within a structure definition.

Visual Design Center syntax

See Visual Design Center syntax for a simple type, such as any.

IDLos syntax

```
typedef sequence<any, 5> name;
```

Action language syntax

Sequences cannot be defined other than as a `typedef` or as a member of a `struct` or `exception`.

Example

```
const long constBoundsValue = 50;
typedef string stringArray[constBoundsValue];
typedef sequence<stringArray, constBoundsValue>
boundedSeqOfStringArrays;
struct LimitedAccounts {
    string bankSortCode<10>;
    sequence<Account, 50> accounts; // max sequence length is 50
};
```

General Information

You must unpack into a C-style array in order to pass to C++ function call. An attribute `length` represents the current length of the sequence at runtime.

bounded string

Semantics

The type `bounded string` is an IDL template type. It is a series of any number of possible 8-bit quantities except 0x00, up to the limit specified by the bound.

Visual Design Center syntax

Select an appropriate class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `string`. Click on the **Edit** button and change the type to `string<value>`, where `value` is the bound.

IDLos syntax

You may define a `bounded string` anywhere you can define a simple type.

```
attribute string<1024> buffer;
```

Action language syntax

```
declare string<10> myString;
```

Example

```
struct ShortMessagePacket
{
    string destination;
    string<1024> shortMessage;
};
```

bounded wstring

Semantics

The type `bounded wstring` is an IDL template type. It is not used by ObjectSwitch at this time.

Visual Design Center syntax

Select an appropriate class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `wstring`. Click on the **Edit** button and change the type to `wstring<value>`, where `value` is the bounded value.

IDL syntax

You may define a `bounded wstring` anywhere you can use a simple type.

```
struct ShortMessagePacket
{
    wstring destination;
    wstring<1024> shortMessage;
};
```

Action language syntax

```
declare wstring<10> wi deLabel ;
```

Example

```
struct ShortMessagePacket;
{
    wstring destination;
    wstring<1024> shortMessage;
};
declare wstring<1024> wi deBuffer;
```

char

Semantics

The type `char` is an IDL simple type. It is an 8-bit quantity that encodes a single-byte character from any byte-oriented code set. The type `char` can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

Visual Design Center syntax

Select the `Entity` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `char`.

To create a sequence or array of type `char`, open a `typedef` class specification. Click on the **Attributes** tab and select an attribute of type `char`. Click on the **Edit** button and set the **Stereotype** to `Array`, `Sequence` or `SequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `char` with:

```
attribute char name;
```

Inside of an entity block, define an array “name” of type `char` with:

```
attribute char name[5];
```

Inside of an entity block, define a sequence “name” of type `char` with:

```
typedef sequence<char> name;
```

Inside of an entity block, define a bounded sequence “name” of type `char` with:

```
typedef sequence<char, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `char` with:

```
declare char name;
```

Inside of an action language function, allocate an array “name” of type `char` with:

```
declare char name[5];
```

Example

```
declare char name;
```

const

Semantics

Used in conjunction with other IDL types, `const` assigns a symbol to an unchanging value.

These types can be made `const`: `boolean`, `char`, `double`, `float`, `long`, `long long`, `short`, `string`, `unsigned long`, `unsigned long long`, `unsigned short`, `wchar`, `wstring`.

Visual Design Center syntax

Select the `Entity` class and open the specification. Click on the **Constants** tab and then click on the **Add** button to create a new constant attribute. Click on the **Type** pulldown selection and select one of the available types.

IDL syntax

```
entity name
{
    const short parm = 5;
};
```


Example

```

const long numOfEnglishLetters = 26;
typedef string EnglishIndexURLs[numOfEnglishLetters];
entity htmlIndex
{
    attribute EnglishIndexURLs URLs;
    boolean getIndexURL(in long i, out string url);
    action getIndexURL
    {
        if (i >= 0 && i < numOfEnglishLetters)
        {
            url = self.URLs[i];
            return true;
        }
        else
        {
            url = "";
            return false;
        }
    }
};

```

General Information

There is no such thing as an any constant value.

User-defined types, even those which define primitive types, cannot be used to define constant values.

context

Semantics

An IDL operation's `context` expression specifies which elements of the client's context might affect the performance of a request by the object. The runtime system makes the context values in the client's context available to the object implementation when the request is delivered. Context is not used by `ObjectSwitch` at this time.

double

Semantics

The type `double` is an IDL simple type. It represents IEEE double-precision floating point numbers. Types `float` and `double` follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

Visual Design Center syntax

Select the `Entity` class and open the specification. Click on the `Attributes` tab and then click on the `Add` button to create a new attribute. Click on the `Type` pulldown selection and select `double`.

To create a sequence or array of type `double`, open a `typedef` class specification. Click on the `Attributes` tab and select an attribute of type `double`. Click on the `Edit` button and set the `Stereotype` to `Array`, `Sequence` or `SequenceOfArray`. Click the `Sequences` tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `double` with:

```
attribute double name;
```

Inside of an entity block, define an array “name” of type `double` with:

```
attribute double name[5];
```

Inside of an entity block, define a sequence “name” of type `double` with:

```
typedef sequence<double> name;
```

Inside of an entity block, define a bounded sequence “name” of type `double` with:

```
typedef sequence<double, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type double with:

```
declare double name;
```

Inside of an action language function, allocate an array “name” of type double with:

```
declare double name[5];
```

Example

```
declare double name;
```

entity

Semantics

The type `entity` is an IDL composite type.

With an entity, you can:

- define attributes and operations
- inherit from other entities
- define any number of relationships with other entities
- make it accessible to other packages using an interface
- trigger operations when it is created, deleted, refreshed, or related to another entity
- define keys to use in queries
- manage its states with a finite state machine
- define types nested in the entity

Visual Design Center syntax

Open the specification for the class and select `entity` as the class stereotype.

IDL syntax

```
entity name
{
    ...
}
```

Example

```
entity TimerEventImpl
{
    // attributes
    attribute Road road;
    // operations
    oneway void generate ();
};
```

General Information

There is no “pure virtual” in IDLos. When you mark an operation virtual, you still need to implement it in the supertype. You also must implement the operation for all subtypes.

enum

Semantics

The type `enum` is an IDL composite type.

An `enum` (enumerated) type lets you assign identifiers to the members of a set of values.

Visual Design Center syntax

Open the specification for the class and select `kaEnum` as the class stereotype.

IDL syntax

```
enum name { <list of values> };
```

Example

```
enum SideDirection {North, East, South, West};
```

This defines `SideDirection` as a type which takes on the values `North`, `East`, `South` or `West`.

Action Language example:

```
enum SideDirection LocalDirection;  
LocalDirection = East;
```

General Information

The actual ordinal values of a `enum` type vary according to the language implementation. The CORBA specification only guarantees that the ordinal values of enumerated types monotonically increase from left to right. All enumerators are mapped to a 32-bit type.

exception

Semantics

The type `exception` is an IDL composite type.

The IDL `exception` type itself is similar to a `struct` type; it can contain various data members (but no methods).

An `exception` is used to alter the flow of control of a section of code. When a method raises an `exception`, the method “returns” at that point in its code. When the calling code receives the `exception`, control passes to any matching catch blocks, or the `exception` is rethrown.

Uncaught user and system exceptions will cause a runtime engine failure.

Visual Design Center syntax

Open the specification for the class and select `exception` as the class stereotype.

IDL syntax

```
exception name { <attribute list> };
```

Example

```
exception IsDead { string causeOfDeath; };  
void wakeup() raises (IsDead);
```

This defines the user `exception` `IsDead` as a type with attribute `causeOfDeath`. The function `wakeup()` can instantiate a variable of type `IsDead`, set the value of `causeOfDeath` and throw the variable as a user exception. To catch the above exception:

```
try { self.wakeup(); } catch (IsDead id) { ... }
```

General Information

User exceptions are defined in IDLs. User exceptions defined in IDLs cannot be inherited.

System exceptions are defined in `swbuiltin.sdl`. They can be raised by the ObjectSwitch runtime and adapters. The user should not throw System exceptions, but should catch them. To catch system exceptions in action language, include `swbuiltin.sdl` in the model.

Handling the system exceptions `ExceptionDeadlock` and `ExceptionObjectDestroyed` in application code not executed as part of spawn will produce unpredictable but bad results.

Operations that throw an exception must have a `raises` clause in the operation signature.

You must unpack an exception into a C-style array in order to pass it to a C++ function call.

An exception definition cannot include itself as a member.

extern

Semantics

Designates global variables and/or code segments.

Visual Design Center syntax

Open the specification for the class and select kabExtern as the class stereotype.

Action language syntax

This is a reserved word for C++.

Example

```
extern C { ... }
```

fixed

Semantics

The type `fixed` is an IDL template type. It provides fixed-point arithmetic values with up to 31 significant digits. You specify a `fixed` type with the following format:

```
typedef fixed< digit-size, scale > name;
```

Unlike IEEE floating-point values, type `fixed` is not subject to representational errors. IEEE floating point values are liable to inaccurately represent decimal fractions unless the value is a fractional power of 2. For example, the decimal value 0.1 cannot be represented exactly in IEEE format. Over a series of computations with floating-point values, the cumulative effect of this imprecision can eventually yield inaccurate results. Type `fixed` is especially useful in calculations that cannot tolerate any imprecision, such as computations of monetary values. Type `fixed` is not used by ObjectSwitch Design Center at this time.

float

Semantics

The type `float` is an IDL simple type.

It represents IEEE single-precision floating point numbers. Types `float` and `double` follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

Visual Design Center syntax

Select the `kaBEntity` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `float`.

To create a sequence or array of type `float`, open a `kaBTypedef` class specification. Click on the **Attributes** tab and select an attribute of type `float`. Click on the **Edit** button and set the **Stereotype** to `kaBArray`, `kaBSequence` or `kaBSequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDLos syntax

Inside of an entity block, define an attribute “name” of type `float` with:

```
attribute float name;
```

Inside of an entity block, define an array “name” of type `float` with:

```
attribute float name[<size>];
```

Inside of an entity block, define a sequence “name” of type `float` with:

```
typedef sequence<float> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `float` with:

```
declare float name;
```

Inside of an action language function, allocate an array “name” of type `float` with:

```
declare float name[<size>;
```

Example

```
declare float name;
```

interface

Semantics

Interfaces are the fundamental abstraction mechanism of CORBA. An interface defines a type of object, including the operations that the object supports in a distributed enterprise application.

An IDL interface generally describes an object's behavior through operations and attributes:

Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.

An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

Visual Design Center syntax

Open the specification for the class and select `ka-bl interface` as the class stereotype.

IDL syntax

```
interface Employee
{
    // only these parts of EmployeeImpl will be
    // accessible outside the package.
    void promote();
    oneway void makewaves();
    attribute string name;
};

entity EmployeeImpl
{
    void promote();
    attribute string name;
    attribute string m_nickname;
    signal makewaves();
};
```

Example

In this example, the `EmployeeImpl` entity has two attributes, an operation, and a signal. The interface exposes all but the `m_nickname` attribute. Notice that the signal is exposed as an operation.

```
interface Employee
{
    // only these parts of EmployeeImpl will be
    // accessible outside the package.
    void promote();
    oneway void makewaves();
    attribute string name;
};
entity EmployeeImpl
{
    void promote();
    attribute string name;
    attribute string m_nickname;
    signal makewaves();
};
// This statement is self-explanatory.
expose entity EmployeeImpl with interface Employee;
```

General Information

An interface may also contain definitions for typedefs, attributes, operations, enumerations, and structures.

Each interface may expose only one entity. An entity may be exposed by more than one interface. Each operation or attribute in the interface must have a corresponding attribute, operation or signal in the entity.

Abstract interfaces may not contain attributes or typedefs.

To permit more control over how an entity is exposed, three properties grant or revoke access to create, delete, or retrieve the extent of an interface.

long

Semantics

The type `long` is an IDL simple type. It represents 32-bit signed integer values. IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types.

Visual Design Center syntax

Select the `long` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `long`.

To create a sequence or array of type `long`, open a `long` class specification. Click on the **Attributes** tab and select an attribute of type `long`. Click on the **Edit** button and set the **Stereotype** to `array`, `sequence` or `sequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `long` with:

```
attribute long name;
```

Inside of an entity block, define an array “name” of type `long` with:

```
attribute long name[size];
```

Inside of an entity block, define a sequence “name” of type `long` with:

```
typedef sequence<long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `long` with:

```
declare long name;
```

Inside of an action language function, allocate an array “name” of type `long` with:

```
declare long name[<size>];
```


Example

```
declare long name;
```

long double

Semantics

The type `long double` is an IDL extended simple type.

Like 64-bit integer types, platform support varies for `long double`, so usage can yield IDL compiler errors.

The type `long double` is *not currently supported* by ObjectSwitch.

long long

Semantics

The type `long long` is an IDL simple type. IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types.

The `long long` type represents 64-bit signed integer values.

The 64-bit integer types `long long` and `unsigned long long` support numbers that are too large for 32-bit integers. Platform support varies. If you compile IDL that contains one of these types on a platform that does not support it, the compiler issues an error.

Visual Design Center syntax

Select the `kaBEntity` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `long long`.

To create a sequence or array of type `long long`, open a `kaBTypedef` class specification. Click on the **Attributes** tab and select an attribute of type `long long`. Click on the **Edit** button and set the **Stereotype** to `kaBArray`, `kaBSequence` or `kaBSequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDLos syntax

Inside of an entity block, define an attribute “name” of type `long long` with:

```
attribute long long name;
```

Inside of an entity block, allocate an array “name” of type `long long` with:

```
attribute long long name[<size>];
```

Inside of an entity block, define a sequence “name” of type `long long` with:

```
typedef sequence<long long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type long long with:

```
declare long long name;
```

Inside of an action language function, allocate an array “name” of type long long with:

```
declare long long name[<size>];
```

Example

```
declare long long name;
```

native

Semantics

Indicates a platform specific class. It is not used by ObjectSwitch Design Center at this time

Visual Design Center syntax

Open the specification for the class and select `kabNative` as the class stereotype.

Object

Semantics

The type `Object` is an IDL simple type. It represents a reference to a user-defined interface or entity.

Visual Design Center syntax

Select the `Object` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `Object`.

To create a sequence or array of type `Object`, open a `typedef` class specification. Click on the **Attributes** tab and select an attribute of type `Object`. Click on the **Edit** button and set the **Stereotype** to `Array`, `Sequence` or `SequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `Object` with:

```
attribute Object name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `Object` with:

```
declare Object name;
```

Inside of an action language function, allocate an array “name” of type `Object` with:

```
declare Object name[<size>];
```

Example

```
declare Object name;
```

General Information

You must unpack the type `Object` into a C-style array in order to pass it to a C++ function call.

octet

Semantics

The type `octet` is an IDL simple type.

It represents an 8-bit quantity guaranteed not to undergo any conversion when transmitted by the communication system. This lets you safely transmit binary data between different address spaces. Avoid using type `char` for binary data, inasmuch as characters might be subject to translation during transmission. For example, if client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

Visual Design Center syntax

Select the `Entity` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `octet`.

To create a sequence or array of type `octet`, open a `typedef` class specification. Click on the **Attributes** tab and select an attribute of type `octet`. Click on the **Edit** button and set the **Stereotype** to `array`, `sequence` or `sequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `octet` with:

```
attribute octet name;
```

Inside of an entity block, define a sequence “name” of type `octet` with:

```
typedef sequence<octet> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `octet` with:

```
declare octet name;
```


Inside of an action language function, allocate an array “name” of type octet with:

```
declare octet name[<size>;
```

Example

```
declare octet name;
```

pipe

Semantics

The type `pipe` is an IDL composite type. It is not supported by ObjectSwitch.

sequence

Semantics

The type `sequence` is an IDL template type. It represents an array with unbounded dimensions. Unbounded sequences are automatically grown at runtime as required. If the index being accessed is larger than the current sequence size, the sequence is resized. When a sequence is resized, the values of all new sequence members are undefined. They must be initialized to a known value by the application.

A sequence must be named by a `typedef` declaration in order to be used as a parameter, an attribute, or a return value. You can omit a `typedef` declaration only for a sequence that is declared within a structure definition.

Visual Design Center syntax

See Visual Design Center syntax for a simple type, such as any.

IDLos syntax

```
typedef sequence<string> NameList;
```

Example

```
const long constBoundsValue = 50;
typedef string stringArray[constBoundsValue];
typedef sequence<stringArray> SeqOfStringArrays;
struct UnlimitedAccounts
{
    string bankSortCode<10>;
    sequence<Account> accounts; // no max sequence length
};
```

General Information

You must unpack a sequence into a C-style array in order to pass it to a C++ function call. Sequences cannot be defined other than as a `typedef` or as a member of a struct or exception. An attribute `length` represents the current length of the sequence at runtime.

short

Semantics

The type `short` is an IDL simple type. It represents 16-bit signed integer values. IDL supports `short` and long integer types, both signed and unsigned. IDL guarantees the range of these types. For example, an unsigned `short` can hold values between 0-65535. Thus, an unsigned `short` value always maps to a native type that has at least 16 bits. If the platform does not provide a native 16-bit type, the next larger integer type is used.

Visual Design Center syntax

In Visual Design Center, select the `Entity` class and open the specification. Click on the `Attributes` tab and then click on the `Add` button to create a new attribute. Click on the `Type` pulldown selection and select `short`.

To create a sequence or array of type `short`, open a `typedef` class specification. Click on the `Attributes` tab and select an attribute of type `short`. Click on the `Edit` button and set the `Stereotype` to `Array`, `Sequence` or `SequenceOfArray`. Click the `Sequences` tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `short` with:

```
attribute short name;
```

Inside of an entity block, allocate an array “name” of type `short` with:

```
attribute short name[<size>];
```

Inside of an entity block, define a sequence “name” of type `short` with:

```
typedef sequence<short> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type short with:

```
declare short name;
```

Inside of an action language function, allocate an array “name” of type short with:

```
declare short name[<size>];
```

Example

```
declare short name;
```

string

Semantics

The type `string` is an IDL template type. It is a series of all possible 8-bit quantities except 0x00. IDL prohibits embedded NUL characters in strings. Unbounded `string` lengths are generally constrained only by memory limitations.

A bounded `string`, such as `string<10>`, can hold only the number of characters specified by the bounds, excluding the terminating NUL character. Thus, a `string<6>` can contain the six-character string “cheese”.

Visual Design Center syntax

In Visual Design Center, select an appropriate class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `string`.

IDLos syntax

```
attribute string name;
```

Action language syntax

```
declare string name;
```

Example

```
typedef string myString;
attribute myString shortName;
```

General Information

An attribute `length` represents the current length of the `string` at runtime.

struct

Semantics

The type `struct` is an IDL composite type. A `struct` data type lets you encapsulate a set of named members of various types.

Visual Design Center syntax

Open the specification for the class and select `kabStruct` as the class stereotype.

IDLos syntax

```
struct AStruct
{
    long    aLongMember;
    short   aShortMember;
};
```

Action language syntax

```
declare AStruct    aStruct;
aStruct.aLongMember = 76543;
```

Example

```
struct Packet
{
    long destination;
    octet data[256];
    boolean retry;
};
```

This defines `Packet` as a type with members `destination`, `data` and `retry`.

For structs, you can define the types of their members in-line.

```
struct A
{
    boolean firstMember;
    struct B
    {
        unsigned long a;
        long b;
        string c;
    } secondMember;
    long thirdMember;
};
```

General Information

You must unpack a struct into a C-style array in order to pass to C++ function call. A struct must include at least one member. Because a struct provides a naming scope, member names must be unique only within the enclosing structure.

Recursive definitions of a struct are not allowed. A C-like workaround is available, e.g.,

```
struct nodeList
{
    string name;
    typedef sequence<nodeList> NextNode;
    NextNode nextNode;
};
```


typedef

Semantics

A typedef lets you define a new legal type; The typedef keyword allows you define a meaningful or simpler name for an IDL type.

Visual Design Center syntax

Open the specification for the class and select kabTypedef as the class stereotype.

IDLos syntax

```
typedef sequence<unsigned long> name;
```

Example

```
typedef enum status {started, finished} racerStatus;
```

General Information

You may rename any legal type. In a typedef, you can define a sequence, struct, or enum in-line.

union

Semantics

The type `union` is an IDL composite type. A `union` type lets you define a structure that can contain only one of several alternative members at any given time. All IDL unions are discriminated.

Visual Design Center syntax

Open the specification for the class and select `Union` as the class stereotype.

For multiple labels, enter a list of labels in the `caseSpecifier` field of the attribute specification dialog. For example, for the equivalent of the IDLs:

```
case 'S' :
case 's' :
    string aString;
```

you would enter `'S','s'` in the `caseSpecifier` field.

IDLs syntax

You declare a `union` type with the following IDL syntax:

```
union name switch (discriminator)
{
    case label1 : element-spec;
    case label2 : element-spec;
    [...]
    case labeln : element-spec;
    [default : element-spec; ]
};
```

Action language syntax

```
union MyUnion switch (char)
```

Example

You can access the value of the discriminator in action language using the special `_d` attribute:

```

union MyUnion switch (char)
{
    case 'S':
    case 's':
        string aString;
    case 'L':
    case 'l':
        long aLong;
    default:
        long justANumber;
};

enum Direction
{
    North;
    South;
    East;
    West;
};

union DirectionUnion switch (Direction)
{
    case South:
    case North:
        long latitude;
    case East:
    case West:
        long longitude;
};

struct ComplexStructure
{
    string name;
    MyUnion myUnion;
};

...
declare MyUnion mu;
mu.aString = "some data";
...
if (mu._d == 'S' || mu._d == 's')
{
    // it's a string
}

```

You can also set the discriminator...

```
mu._d = 'L';
```

...but ObjectSwitch does not check that you are setting the discriminator to a legitimate value.

You can also set the union to null...

```
declare DirectionUnion geoLine;
geoLine = null;
```

To handle a union in a complex structure, you must declare a local union to set and then move it as a whole into the structure...

```
declare ComplexStruct myStruct;
declare MyUnion mu;
myStruct.name = "Joe";
mu.aLong;
myStruct.myUnion = mu;
```

General Information

The union is stored at runtime as an array of slots for each union member. This storage is not optimized; the runtime size of a union is simply the sum of the union members.



The above means that unions are not the best choice for systems where small size, or high performance are important. Avoid using unions as a placeholder for incomplete analysis; try to use them only when they are the best solution to a problem.

The ObjectSwitch Monitor displays the discriminator and the current values of all the union members.

If you access an “incorrect” union member (that is, the discriminator indicates another union member is active), the accessor throws an `ExceptionDataError` system exception.

A struct, union, or exception definition cannot include itself as a member.

A union's discriminator can be integer, char, boolean or enum, or an alias of one of these types; all case label expressions must be compatible with this type. Because a union provides a naming scope,

member names must be unique only within the enclosing union. Unions allow multiple case labels for a single member. Unions can optionally contain a default case label. The corresponding member is active if the discriminator value does not correspond to any other label.

unsigned long

Semantics

The type `unsigned long` is an IDL simple type. It represents 32-bit unsigned integer values.

Visual Design Center syntax

In Visual Design Center, select the `Entity` class and open the specification. Click on the `Attributes` tab and then click on the `Add` button to create a new attribute. Click on the `Type` pulldown selection and select `unsigned long`.

To create a sequence or array of type `unsigned long`, open a `typedef` class specification. Click on the `Attributes` tab and select an attribute of type `unsigned long`. Click on the `Edit` button and set the Stereotype to `Array`, `Sequence` or `SequenceOfArray`. Click the `Sequences` tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `unsigned long` with:

```
attribute unsigned long name;
```

Inside of an entity block, allocate an array “name” of type `unsigned long` with:

```
attribute unsigned long name[<size>];
```

Inside of an entity block, define a sequence “name” of type `unsigned long` with:

```
typedef sequence<unsigned long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `unsigned long` with:

```
declare unsigned long name;
```

Inside of an action language function, allocate an array “name” of type unsigned long with:

```
declare unsigned long name[<size>];
```

Example

```
declare unsigned long name;
```

unsigned long long

Semantics

The type `unsigned long long` is an IDL simple type. It represents 64-bit unsigned integer values. IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types.

Visual Design Center syntax

In Visual Design Center, select the `Entity` class and open the specification. Click on the **Attributes** tab and then click on the **Add** button to create a new attribute. Click on the **Type** pulldown selection and select `unsigned long long`.

To create a sequence or array of type `unsigned long long`, open a `typedef` class specification. Click on the **Attributes** tab and select an attribute of type `unsigned long long`. Click on the **Edit** button and set the **Stereotype** to `Array`, `Sequence` or `SequenceOfArray`. Click the **Sequences** tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `unsigned long long` with:

```
attribute unsigned long long name;
```

Inside of an entity block, define an array attribute “name” of type `unsigned long long` with:

```
attribute unsigned long long name[<size>];
```

Inside of an entity block, define a sequence “name” of type `unsigned long long` with:

```
typedef sequence<unsigned long long> name;
```


Action language syntax

Inside of an action language function, allocate a handle “name” of type `unsigned long long` with:

```
declare unsigned long long name;
```

Inside of an action language function, allocate an array “name” of type `unsigned long long` with:

```
declare unsigned long long name[<size>];
```

Example

```
declare unsigned long long name;
```

unsigned short

Semantics

The type `unsigned short` is an IDL simple type. It represents 16-bit unsigned integer values. IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types.

Visual Design Center syntax

In Visual Design Center, select the `Entity` class and open the specification. Click on the `Attributes` tab and then click on the `Add` button to create a new attribute. Click on the `Type` pulldown selection and select `unsigned short`.

To create a sequence or array of type `unsigned short`, open a `typedef` class specification. Click on the `Attributes` tab and select an attribute of type `unsigned short`. Click on the `Edit` button and set the Stereotype to `Array`, `Sequence` or `SequenceOfArray`. Click the `Sequences` tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `unsigned short` with:

```
attribute unsigned short name;
```

Inside of an entity block, define an array attribute “name” of type `unsigned short` with:

```
attribute unsigned short name[<size>];
```

Inside of an entity block, define a sequence “name” of type `unsigned short` with:

```
typedef sequence<unsigned short> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned short with:

```
declare unsigned short name;
```

Inside of an action language function, allocate an array “name” of type unsigned short with:

```
declare unsigned short name[<size>;
```

Example

```
declare unsigned short name;
```

void

Semantics

The type `void` is an IDL simple type.

The `void` keyword is valid only in an operation. In an operation declaration, it may be used to indicate an operation that does not return a function result value.

Visual Design Center syntax

In Visual Design Center, select the `kaBEnti ty` class and open the specification. Click on the `Operations` tab and then click on the `Add` button to create a new operation. Click on the `Return Type` pulldown selection and select `void`.

IDLos syntax

```
void myOperation(in myParm);
```

Example

```
void myOperation(in myParm);
```

wchar

Semantics

The type `wchar` is an IDL simple type. It encodes wide characters from any character set. The size of a `wchar` is platform-dependent.

Visual Design Center syntax

In Visual Design Center, select the `kaBEntity` class and open the specification. Click on the `Attributes` tab and then click on the `Add` button to create a new attribute. Click on the `Type` pulldown selection and select `wchar`.

To create a sequence or array of type `wchar`, open a `kaBTypedef` class specification. Click on the `Attributes` tab and select an attribute of type `wchar`. Click on the `Edit` button and set the `Stereotype` to `kaBArray`, `kaBSequence` or `kaBSequenceOfArray`. Click the `Sequences` tab and set the dimensions of the sequence or array.

IDLos syntax

Inside of an entity block, define an attribute “name” of type `wchar` with:

```
attribute wchar name;
```

Inside of an entity block, define an array “name” of type `wchar` with:

```
attribute wchar name[<size>];
```

Inside of an entity block, define a sequence “name” of type `wchar` with:

```
typedef sequence<wchar> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `wchar` with:

```
declare wchar name;
```

Inside of an action language function, allocate an array “name” of type `wchar` with:

```
declare wchar name[<size>];
```

Example

```
decl are wchar name;
```

wstring

Semantics

The type `wstring` is an IDL template type. It is the wide-character equivalent of type `string`. Like `string`-types, `wstring` types can be unbounded or bounded. Wide strings can contain any character except NUL.

Since the ObjectSwitch implementation of the `string` type also allows for multi-byte strings, there is little difference between `string` and `wstring`. The only current difference is that `wstrings` cannot be compared as greater-than or less-than other `wstrings`.

Visual Design Center syntax

In Visual Design Center, select the `kaBEntity` class and open the specification. Click on the `Attributes` tab and then click on the `Add` button to create a new attribute. Click on the `Type` pulldown selection and select `wstring`.

To create a sequence or array of type `wstring`, open a `kaBTypedef` class specification. Click on the `Attributes` tab and select an attribute of type `wstring`. Click on the `Edit` button and set the `Stereotype` to `kaBArray`, `kaBSequence` or `kaBSequenceOfArray`. Click the `Sequences` tab and set the dimensions of the sequence or array.

IDL syntax

Inside of an entity block, define an attribute “name” of type `wstring` with:

```
attribute wstring name;
```

Inside of an entity block, define an array “name” of type `wstring` with:

```
attribute wstring name[<size>];
```

where `<size>` is an integer and is the number of bytes allocated for the string, (which may not be the number of characters).

Inside of an entity block, define a sequence “name” of type `wstring` with:

```
typedef sequence<wstring> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type `wstring` with:

```
declare wstring name;
```

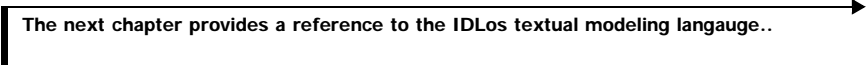
Inside of an action language function, allocate an array “name” of type `wstring` with:

```
declare wstring name[<size>];
```

Example

```
declare wstring name;  
name = "a single-byte character string";
```

There is no built in support for wide string literals.



The next chapter provides a reference to the IDL's textual modeling language..

8

IDLos Reference

This section contains reference pages for each IDLos statement, in alphabetical order. Each page describes the syntax of the statement, a discussion of usage, and an example.

The page for properties contains a table of all the IDLos properties.

Understanding the notation

The syntax description for each statement or function uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

plain text indicates a feature of the notation, as follows:

(*item*) parentheses group items together

(*item*) * the item in parentheses appears zero or more times

(*item*) + the item in parentheses appears one or more times

{ *item* } items in braces are optional

(*item1* | *item2*) the vertical bar indicates either *item1* or *item2* appears

Common elements of the grammar

The following element of the IDLos grammar appear frequently throughout the reference pages.

Grammar element	Meaning
<i>identifier</i>	any supported characters, digit, or underscore. Must start with a-z or A-Z.
<i>scoped_name</i>	{::} <i>simple_declarator</i> (:: <i>simple_declarator</i>)*
<i>simple_declarator</i>	<i>identifier</i>

Files and engines in IDLos

This section discusses how IDLos can be arranged into files and mapped to ObjectSwitch engines.

Files

Usually you will put one package in a file. But there is no restriction on how IDLos packages are arranged in files except that each package must be syntactically complete.

If a package name appears again in a model, IDLos reopens the package and adds information to it. In this way, a package might be divided up among many files. For example, you might have each module from that package defined in its own file.

Actions might be defined in separate files. This is similar to the arrangement of C++ implementations kept in different files than the class definitions.

The Design Center does not care how your IDLos files are arranged. Whether you have one file that `#includes` all the necessary files, or you load each file by hand - the Design Center can build your model as long as you have all the necessary packages loaded when you build your engine.

When a package is declared more than once in IDLos, the package is re-opened and modified on each subsequent declaration. More information will be added to its definition each time that the package name appears in the model.

Engines

You will usually put one package in a file and generate one engine from that package.

Packages are mapped to engines using the ObjectSwitch Design Center. There is no restriction in the Design Center on how many packages can be built into one engine. You may compile as many packages into an engine as you want.

When you partition ObjectSwitch applications into different components, the package represents the smallest level of granularity you can use.

With the ObjectSwitch server, there is usually very little advantage to putting packages in the same engine. An event bus dispatch to an object on the same node is just as fast whether that object resides in the same engine, or another. Of course, if engines are running on different nodes, i.e. distributed, then there is a performance cost.

If your application is very message intensive, you can get a performance gain by using as many `local` operations and `local` entities as practical, since these are invoked directly, not through the ObjectSwitch event bus.

action

Defines the behavior of a state or an operation.

Syntax

```
acti on scoped_name { ' actionLanguage ' };
```

Description

The implementation of a state or action is defined in action language, between the backquotes '. An action can be defined in an entity, a module, a package, or outside of a package.

Warnings

None

Example

```
acti on :: Traffic :: GUI ProxyImpl :: updateMode
// arguments:
//in ModeType      modeType
//in RoadDirection modeDir
//
{`
    // some action language
`};
```

See also

operation, stateset.

attribute

An object data member, and its accessors.

Syntax

```
[readonly] attribute param_type_spec simple_declarator;
```

Description

In an entity, the attribute defines a data member, a set accessor and a get accessor. In an interface, the attribute exposes those same accessors. `readonly` may only be specified in an interface. If the attribute is `readonly` in the interface, only the get accessor is exposed.

Attributes are inherited. To expose the attribute in the interface, it must match the attribute in the entity in both name and type.

Attributes cannot be structs or arrays. However, user-defined types defined as structs or arrays can be attributes.

When an object is created, the values of its attributes are undefined.

Warnings

None

Example

```
attribute long timeGreen;  
attribute LightColor colorTarget;
```

See also

[trigger](#)

const

Defines a constant value.

Syntax

```
const const_type identifier = literal;
```

Description

literal must be a valid literal type: integer, character, floating point, string, or boolean (see “Literals” on page 187 for more details).

Assigning *literal* to an *identifier* of type *const_type* must represent a valid operation. *const_type* can be a scoped name. The following table shows valid IDLos types for each literal type:

literal	const_type
integer	long long long
character	char wchar
floating point	float double long double
string	string wstring
boolean	boolean

ObjectSwitch also performs implicit conversions between string and numeric types in a `const` statement (see “Implicit conversion between string and numeric types” for more details).

Warnings

None.

Example

```
const long constBoundsValue = 50;
const char littleN = 'n';
const float smallFloat = 73.033e-32;
const string msg1233 = "Don't do that!";
const boolean SureThing = TRUE;

// used as bound in string, array and sequence
typedef string<constBoundsValue> boundedString;
typedef string stringArray[constBoundsValue];
typedef sequence<string, constBoundsValue> stringSequence;
```


enum

Defines a type whose values are a fixed set of tokens

Syntax

```
enum identifier { identifier (, identifier)* };
```

Description

Enums contain a comma-separated list of one or more enumerators, which are just identifiers.

Warnings

None

Example

```
enum ModeType  
{  
    tyTi med,  
    tySensed,  
    tyHol dGreen,  
    tyBl i nkYel l ow,  
    tyBl i nkRed,  
    tyNoChange  
};
```

entity

Defines the implementations of objects.

Syntax

```
entity simple_declarator { : scoped_name } { (entityContent) * };
```

```
entity simple_declarator;
```

Description

The first syntax is for entity definition, the second for forward declaration.

Entities can contain actions, attributes, enums, exceptions, keys, operations, signals, transitions, statesets, structs, triggers, and typedefs.

An entity statement may have `annotation`, `local`, and `singleton` properties.

To make an entity accessible outside its package, expose the entity with an interface. An entity may be exposed by more than one interface.

The `: scoped_name` following the entity declarator designates a supertype for inheritance. Entities may inherit from other entities if they are in the same package, and have the same value of their `local` property.

Entities may have relationships to other entities in the same package.

Forward declares allow you to use an entity type before it is defined.

Warnings

Local entities have such a different description, they are covered on their own page.

Multiple inheritance is not supported.

Example

```
// forward declares
entity TimerEventImpl;

// definitions
entity TimerEventImpl
{
    attribute Road road;
    oneway void generate ();
};
```

See also

Local entity, trigger

exception

A user-defined type that can be thrown and caught upon error.

Syntax

```
exception identifier { (type_spec declarator)* };
```

Description

Define exceptions when you need to report errors in your applications.

Warnings

None

Example

```
exception ExpFileNotFound  
{  
    string path;  
    string filename;  
};
```

See also

operation

action language throw, catch, and exceptions

expose

Specifies which entity an interface exposes.

Syntax

```
expose entity scoped_name with interface scoped_name;
```

Description

An `expose` statement may have the `annotation` property. An abstract interface may not be used in an `expose` statement. You may define `expose` statements in a package or module.

Warnings

None

Example

```
expose entity GuiProxyImpl with interface GuiProxy;
```

See also

interface, entity

interface

Interfaces provide access to entities for clients in other packages.

Syntax

```
interface simple_declarator { : scoped_name } { (interface_body) * };
```

```
interface simple_declarator;
```

Description

The first syntax is for interface definition. The second syntax is for forward declaration.

Interfaces can contain attributes, enums, exceptions, operations, structs, and typedefs.

Interfaces define what part of an entity gets exposed outside of the package boundary. Everything has open access within a package.

Interfaces contain the attributes and operations of the entity that you intend to expose to outside clients.

Signals are exposed as oneway void operations. They must be implemented in the entity.

The : *scoped_name* following the interface declarator designates a supertype for inheritance. Interfaces may inherit from other interfaces, either within the same package or across package boundaries.

An interface statement may have abstract, annotation, createaccess, deleteaccess, extentaccess, and singleton properties.

You may also use interfaces for access to the entity from within the package; but it is not necessary.

Warning

Relationships defined between interfaces are implemented between the entities they expose. Interfaces may not participate in associative relationships.

All attributes and operations listed in an interface must be defined in the exposed entity.

Example

```
interface Employee
{
    // only these parts of EmployeeImpl will be
    // accessible outside the package.
    void promote();
    oneway void makewaves();
    attribute string name;
};

entity EmployeeImpl
{
    void promote();
    attribute string name;
    attribute string m_nickname;
    signal makewaves();
};

expose entity EmployeeImpl with interface Employee;
```

See also

entity, expose

key

A list of attributes whose value uniquely identifies an entity.

Syntax

```
key simple_declarator {simple_declarator (, simple_declarator) *};
```

Description

Each key is a named list of attributes.

A key can contain as many attributes as you want. Multiple keys may be defined for a single entity.

A key statement may have the annotation property.

Warnings

Arrays, sequences, and unbounded strings are not valid in keys.

When a key is inherited (that is, if an entity containing a key has subtypes), the index for the key is maintained in the type where the key is defined. This means that keys must be unique across instances of that type and all its subtypes.

Avoid modifying key attributes. If it is necessary to modify a attribute which is part of a key, enclose the modifications in a try..catch() block. This allows ObjectSwitch to update the keys immediately after they are modified, and it lets you catch the ExceptionObjectNotUnique exception that can be thrown when you change a key. If you don't catch this exception, the engine will crash.

Here is an example of how you might use a temporary object when you need to change an attribute which is part of a key:

```
entity KeyedEntity
{
    attribute Long longAttr;
    key Key { longAttr };
};

. . . . . IDLs above, action language below . . . . .

declare KeyedEntity      keyedEntity;
declare Long             saveLongAttr;
// . . .
saveLongAttr = keyedEntity.longAttr;
try
{
    declare KeyedEntity tempKeyedEntity = keyedEntity;
    tempKeyedEntity.longAttr = newValue;
}
catch (swbuiltins::ExceptionObjectNotUnique)
{
    keyedEntity.longAttr = saveLongAttr;
}
```

Multipart keys should be modified as a group within the try block.

Example

```
entity Employee
{
    attribute string firstName;
    attribute string lastName;
    attribute Long SSN;
    attribute Long employeeNumber;
    key primary {employeeNumber};
    key secondary {SSN};
};
```

See also

[attribute](#)

[action language select, for...in](#)

local entity

An entity that cannot be distributed. A local entity is essentially a C++ class defined in IDLos and implemented in action language. Its operations are invoked directly without being dispatched through the ObjectSwitch event bus, so they execute somewhat faster.

Though their operations are executed in a transaction, local entities are not recoverable. This means that the invocation of operations in a local entity is not logged in the transaction log, and changes to the local entity are not logged in the transaction log, but everything that the local entity does to other entities *is* logged.

You specify a local entity by placing the `local` property before the entity declaration.

Syntax

```
[local] entity simple_declarator { : scoped_name }
{(entityContent) *};

entity simple_declarator;
```

The first syntax is for entity definition, the second for forward declaration.

Description

Local entities can contain actions, enums, exceptions, operations, structs, and typedefs. To make a local entity accessible outside its package, expose the entity with an interface. A local entity may:

- be exposed by more than one interface
- inherit from other local entities in the same package
- use the `annotation` property
- have operations that use the `initialize`, `recovery`, and `terminate` properties

Local entities may *not* have relationships to other entities. Forward declares allow you to use a local entity type before it is defined.

Warnings

A local entity cannot be used as an actual parameter when calling an operation on a non-local entity.

Local entities are not recoverable.

Local entities have no state, so they cannot contain statesets.

Example

```
// forward declare
entity StartUp;

[local]
entity StartUp
{
    [initialize]
    void initData ();
};
```

See also

[entity](#)

module

Used to provide extra namespaces within a package as needed.

Syntax

```
modul e simple_declarator {(definition)+};
```

Descriptions

Modules define a namespace. Modules may be nested.

A module may contain entities, interfaces, enums, structs, exceptions, typedefs, nested modules, relationships, actions, and exposes statements. Modules are declared inside of packages.

A `modul e` statement may have the `annotati on` property.

When a module name appears more than once in a model, information is added to the module definition each subsequent time that its name appears.

Warnings

None

Example

```
package Traffic
{
    modul e Pedestri ans
    {
        enti ty Joggers {};
    };
    modul e Cars
    {
        enti ty Trucks {};
    };
    relati onshi p
    {
        rol e Pedestri ans::Joggers avoid 0..* Cars::Trucks;
        rol e Cars::Trucks stop 0..* Pedestri ans::Joggers;
    };
};
```

operation

Declares an operation on an entity or an interface.

Syntax

```
{oneway}(param_type_spec | void) simple_declarator
    ( { param_dcl (, param_dcl)* } )
    { raises (scoped_name (, scoped_name)* )};
```

param_dcl:

```
(i n | out | i nout) param_type_spec simple_declarator
```

Description

Use **oneway** for asynchronous operations.

In an entity, an operation may have annotation, const, local, and virtual properties.

In an interface, an operation may have annotation, const, and local properties.

In a local entity, an operation may have annotation, const, initialize, recovery, and terminate properties.

Operations are implemented in action statements.

Warnings

Operations have constraints on whether they can be oneway, twoway, or have parameters when they:

- are used in triggers
- are used as engine events
- implement exposed signals

See the appropriate sections of the manual.

Example

```
void updateMode (
    in ModeType modeType,
    in RoadDirection modeDir);
void updateLightTimes (
    in long timeGNS,
    in long timeGEW,
    in long timeYNS,
    in long timeYEW);
void updateSensor (
    in boolean newCar,
    in SideDirection carDir);
LightColor getLightN ();
LightColor getLightS ();
LightColor getLightE ();
LightColor getLightW ();
void getAllLights (
    out LightColor north,
    out LightColor south,
    out LightColor west,
    out LightColor east);
```

See also

action, signal, trigger

package

Packages define components.

Syntax

```
package simple_declarator { (definition) * };
```

Description

IDLos applications require packages. Packages define components. They form an access boundary: you cannot access an entity in another package unless it is exposed by an interface.

A package may contain entities, interfaces, enums, structs, exceptions, typedefs, modules, relationships, actions, and exposes statements.

A package statement may have the annotation property.

When a package name appears more than once in a model, information is added to the package definition each subsequent time that its name appears.

An IDLos file may contain more than one package, and a package may be contained in more than one file.

Warnings

There is no implicit package scope for a file. All IDLos statements except action definitions must be encapsulated in an explicit package.

Example

This fragment of the sample model has been edited for clarity's sake.

```
package Traffic
{
    enum LightColor
    {
        tyRed,
        tyYellow,
        tyGreen,
        tyBlackRed,
        tyBlackYellow
    };
    interface GUIProxy
    {
        // operations
        LightColor GetLightN ();
        LightColor GetLightS ();
        LightColor GetLightE ();
        LightColor GetLightW ();
    };
};
```


relationship / role

Defines a relationship between entities or between interfaces.

Syntax

```

relati onshi p simple_declarator
{
    rol e scoped_name identifier(0..1 | 0..* | 1..1 |
                                1..*) scoped_name;
    {rol e scoped_name identifier(0..1 | 0..* | 1..1 |
                                1..*) scoped_name; }
    {usi ng scoped_name; }
};

```

Description

You specify roles in one or both directions between entities in the relationship statement. Each role has a name, a multiplicity, and specifies the two related entities. You can also specify an associative entity with the optional `usi ng` clause.

You can specify a relationship between interfaces instead of between entities. This is how you expose a relationship. Do not mix entities and interfaces in a relationship. Interfaces may not be used as associative objects; relationships between interfaces may not contain a `usi ng` clause.

A `relati onshi p` statement may have the `annotati on` property.

Warnings

The two roles in a relationship must have different names.

Example

```

relati onshi p fl owOwnershi p
{
    rol e i ntersecti on has 1..1 traffi cFl ow;
    rol e traffi cFl ow i s i n 1..1 i ntersecti on;
};

```

See also

action language, relate, trigger, unrelate

signal

Defines an event for a state machine.

Syntax

```
signal simple_declarator (param_dcl (, param_dcl)*);
```

param_dcl:

```
in param_type_spec simple_declarator
```

Description

A `signal` statement may have the `annotation` property.

Warnings

Signals may have only `in` parameters.

Example

```
signal blinkRed( );  
signal goRed( );  
signal blinkYellow( );  
signal greenTimed( );  
signal greenSensor( );  
signal greenHeld( );  
signal timeout( );  
signal carArrived( );  
signal goYellow( );
```

See also

`stateset`, `transition`, `operation`

stateset

Defines the allowed states for an entity.

Syntax

stateset

{simple_declarator (, simple_declarator)} = simple_declarator;*

Where the last *simple_declarator* designates the initial state.

Description

Defines the states in an entity's state machine, and its initial state. When the entity is created, it is placed into its initial state.

Each state (including initial and final states) must be associated with an action. In ObjectSwitch, the action of an initial state is *not* executed upon object creation. You must always transition into a state with a signal in order for an action to be executed.

A **stateset** statement may have annotation and finished properties.

Warnings

State names are in the entity's namespace.

Example

```
stateset
{
    Initial,
    BlinkRed,
    Red,
    TimedGreen,
    TimedYellow,
    SensorGreenPreferred,
    WaitForCar,
    WaitForGreenTimeout
} = Initial;
```

See also

action, signal, transition

struct

A type to hold data.

Syntax

```
struct identifier { (type_spec declarator);*};
```

Description

Use structs as a container for data.

Warnings

None

Example

```
struct Result  
{  
    typedef sequence<string> Messages;  
    Messages messages;  
    boolean status;  
};
```

transition

Defines the transition from one state to another when a specific signal is received.

Syntax

`transition scoped_name to (scoped_name | cannot happen | ignore)
upon simple_declarator;`

Description

Transition statements specify how a state machine should respond to a certain signal when the entity is in a particular state. You can transition to another state, ignore the signal, or cause an exception to be thrown by specifying `cannot happen`. If a transition is not specified, the default is `cannot happen`.

Warnings

There are no automatic transitions in IDLos. All transitions must be explicit.

Example

```
transition Initial to BlinkRed upon blinkRed;
transition BlinkRed to Red upon goRed;
transition Red to BlinkRed upon blinkRed;
transition Red to BlinkYellow upon blinkYellow;
transition Red to TimedGreen upon greenTimed;
transition Red to SensorGreenPreferred upon greenSensor;
transition TimedYellow to Red upon timeout;
transition CarWaiting to ignore upon carArrived;
transition CarWaiting to ignore upon goYellow;
transition HoldGreen to ignore upon carArrived;
transition WaitForGreenTimeout to ignore upon carArrived;
```

See also

signals, stateset

trigger

Triggers invoke operations upon the occurrence of some event.

Syntax

```

trigger simple_declarator
upon (create | delete | refresh | commit | abort);

trigger simple_declarator
upon (pre-get | post-get | pre-set | post-set) simple_declarator;

trigger scoped_name upon relate|unrelate simple_declarator;

```

Description

The first syntax is for entity triggers: The *simple_declarator* specifies an operation name. You define entity triggers in entities.

The second syntax is for attribute triggers. The first *simple_declarator* specifies an operation or signal. The next *simple_declarator* specifies the attribute upon which to trigger. These are also defined in entities.

The third syntax is for role triggers. The *scoped_name* identifies an operation or signal in the *from* entity of the role (the first one mentioned in the role statement). The *simple_declarator* specifies the role name. Role triggers are defined within relationship statements.

A trigger statement may have the annotation `property`.

Create and delete triggers with inheritance If you define create triggers for both the supertype (parent) and subtype (child), then the supertype trigger fires before the one in the subtype. Conversely, subtype delete triggers fire before those in a supertype. Some readers will find this familiar, as it resembles class constructor/destructor behavior in other languages.

Warnings

Operations used in triggers must return void and must have no parameters, and delete triggers may not be one-way operations. The operations must be defined in the entity for which the triggers are defined or in a supertype.

Example

```
trigger initializeAttributes upon create;  
trigger calculatePayment upon pre-get mortgagePayment;  
trigger deleteCustomer upon unrelate patronizes;
```

See also

attribute, operation, signal, entity, relationship

typedef

Give a new name to a type. Define a new sequence or array type.

Syntax

typedef *type_spec declarator*;

Description

The *declarator* is the name of the new legal type. A *declarator* can include array dimensions.

The *type_spec* can be any legal type already defined, either a basic type, or a user-defined type. It could also be a **sequence** or a bounded string.

Warnings

None

Example

```
typedef octet byte;  
typedef sequence<Employee> Division;  
typedef boolean low_pass_filter[3][3];
```


Complete IDLos grammar

```
accessValues :  
    granted | revoked  
  
actionStatement :  
    action scoped_name swalBlock ;  
  
annotation :  
    annotation = string_literal ( string_literal ) *  
  
annotations :  
    [ annotation ]  
  
any_type :  
    any  
  
array_declarator :  
    identifier ( fixed_array_size ) +  
  
assignedProperty :  
    annotation | finished = finishedStateList | createaccess = accessValues |  
    deleteaccess = accessValues | extentaccess = accessValues  
  
assocSpec :  
    using scoped_name  
  
attr_dcl :  
    { readonly } attribute param_type_spec simple_declarator { = string_literal }  
    ( , simple_declarator { = string_literal } ) *  
  
attributeList :  
    { simple_declarator ( , simple_declarator ) * }  
  
attributeTrigger :  
    ( pre-set | post-set | pre-get | post-get ) simple_declarator  
  
base_type_spec :  
    floating_pt_type | integer_type | char_type | wide_char_type | boolean_type | octet_type  
    | any_type | object_type  
  
boolean_literal :  
    TRUE|true | FALSE|false  
  
boolean_type :  
    boolean  
  
booleanProperty :  
    abstract | const | initialize | local | extentless | recovery |  
    singleton | terminate | virtual  
  
case_clause :  
    ( case_label ) + element_spec ;
```

Chapter :

Complete IDLos grammar

```
case_label :
    case_literal : | default :

char_type :
    char

character_literal :
    character-literal

complex_declarator :
    array_declarator

const_dcl :
    const const_type simple_declarator = const_exp

const_exp :
    literal

const_type :
    integer_type | char_type | wide_char_type | boolean_type | floating_pt_type | string_type
    | wide_string_type | fixed_pt_const_type | scoped_name

constr_type_spec :
    struct_type | union_type | enum_type

context_expr :
    context ( string_literal ( , string_literal ) * )

declarator :
    simple_declarator | complex_declarator

declarators :
    declarator ( , declarator ) *

definition :
    type_dcl ; | const_dcl ; | except_dcl ; | interface ; | module ; | entity ; | exposeState-
    ment ; | relationshipStatement ; | actionStatement | includeStatement | pragmaState-
    ment

element_spec :
    type_spec declarator

entity :
    entity ( entityStatement | forward_entity_dcl )

entityBlock :
    { ( { properties } entityContent ) * }

entityContent :
    signal_dcl ; | outerStateSetStatement ; | transitionStatement ; | entityTriggerStatement
    ; | keyStatement ; | type_dcl ; | const_dcl ; | except_dcl ; | attr_dcl ; | op_dcl ; |
    actionStatement
```

```
entityStatement :  
    simple_declarator { inheritance_spec } entityBlock  
  
entityTrigger :  
    ( create | delete | refresh | commit | abort )  
  
entityTriggerStatement :  
    trigger simple_declarator upon ( attributeTrigger | entityTrigger )  
  
enum_type :  
    enum simple_declarator { enumerator ( , enumerator ) * }  
  
enumerator :  
    simple_declarator  
  
except_dcl :  
    exception identifier { ( except_member ) * }  
  
except_member :  
    type_spec declarators ;  
  
export :  
    type_dcl ; | const_dcl ; | except_dcl ; | attr_dcl ; | op_dcl ;  
  
exposeStatement :  
    expose ( entity scoped_name with interface scoped_name )  
  
finishedStateList :  
    { stateName ( , stateName ) * }  
  
fixed_array_size :  
    [ positive_int_const ]  
  
fixed_pt_const_type :  
    fixed  
  
fixed_pt_literal :  
  
  
fixed_pt_type :  
    fixed < positive_int_const , integer_literal >  
  
floating_point_literal :  
    floating-point-literal  
  
floating_pt_type :  
    float | double | long double  
  
forward_dcl :  
    simple_declarator  
  
forward_entity_dcl :  
    simple_declarator
```

Chapter :

Complete IDLos grammar

```
includeStatement :
    < filename

inheritance_spec :
    : scoped_name ( , scoped_name ) *

initState :
    stateName

integer_literal :
    decimal-integer-literal | hexadecimal-integer-literal | octal-integer-literal

integer_type :
    signed_int | unsigned_int

interface :
    interface ( interface_dcl | forward_dcl )

interface_body :
    ( { properties } export ) *

interface_dcl :
    simple_declarator { inheritance_spec } { interface_body }

keyStatement :
    key simple_declarator attributeList

literal :
    integer_literal | fixed_pt_literal | floating_point_literal | boolean_literal | string_literal
    | character_literal

member :
    type_spec declarators ;

member_list :
    ( member ) +

module :
    module simple_declarator { ( { properties } definition ) + }

object_type :
    Object

octet_type :
    octet

op_attribute :
    oneway

op_dcl :
    { op_attribute } op_type_spec simple_declarator parameter_dcls { raises_expr }
    { context_expr }
```

```
op_type_spec :  
    param_type_spec | void  
  
outerStateSetStatement :  
    stateSetSpec = initState  
  
packageBlock :  
    { ( { properties } definition ) * }  
  
packageScope :  
    simple_declarator ::  
  
  
packageStatement :  
    package simple_declarator packageBlock ;  
  
param_attribute :  
    in | out | inout  
  
param_dcl :  
    param_attribute param_type_spec simple_declarator  
  
param_type_spec :  
    base_type_spec | string_type | wide_string_type | fixed_pt_type | scoped_name  
  
parameter_dcls :  
    ( { param_dcl ( , param_dcl ) * } )  
  
positive_int_const :  
    integer_literal  
  
pragmaStatement :  
    include filename  
  
properties :  
    [ property ( , property ) * ]  
  
property :  
    assignedProperty | booleanProperty  
  
qualifiedName :  
    simple_declarator ( . simple_declarator ) *  
  
raises_expr :  
    raises ( scoped_name ( , scoped_name ) * )  
  
relationshipStatement :  
    relationship simple_declarator { roleStatement ; ( roleTriggerStatement ; ) * {  
    roleStatement ; ( roleTriggerStatement ; ) * { assocSpec ; ( roleTriggerStatement ; ) *  
    } } }  
  
roleMult :  
    1..1 | 0..1 | 1..* | 0..*
```

Chapter :

Complete IDLos grammar

```
roleStatement :  
    { annotations } role scoped_name simple_declarator roleMult scoped_name  
  
roleTriggerStatement :  
    trigger scoped_name upon ( relate | unrelate ) simple_declarator  
  
scoped_name :  
    { :: } qualifiedName ( :: qualifiedName ) *  
  
sequence_type :  
    sequence < simple_type_spec { , positive_int_const } >  
  
signal_dcl :  
    signal simple_declarator parameter_dcls  
  
signed_int :  
    signed_long_int | signed_short_int | signed_longlong_int  
  
signed_long_int :  
    long  
  
signed_longlong_int :  
    long long  
  
signed_short_int :  
    short  
  
simple_declarator :  
    identifier  
  
simple_type_spec :  
    base_type_spec | template_type_spec | scoped_name  
  
socleContent :  
    packageStatement | actionStatement | includeStatement | pragmaStatement  
  
socleStatement :  
    ( { properties } socleContent ) +  
  
stateName :  
    simple_declarator  
  
stateSetSpec :  
    stateset { stateName ( , stateName ) * }  
  
string_literal :  
    string-literal  
  
string_type :  
    string { < positive_int_const > }  
  
struct_type :  
    struct simple_declarator { member_list }
```



```
swalBlock :  
    { delimited-action-language }  
  
switch_body :  
    ( case_clause )+  
  
switch_type_spec :  
    integer_type | char_type | boolean_type | enum_type | scoped_name  
  
template_type_spec :  
    sequence_type | string_type | wide_string_type | fixed_pt_type  
  
transitionStatement :  
    transition qualifiedName to transitionTarget upon simple_declarator  
  
transitionTarget :  
    qualifiedName | cannothappen | ignore  
  
type_dcl :  
    typedef type_declarator | struct_type | union_type | enum_type | native  
    simple_declarator  
  
type_declarator :  
    type_spec declarators  
  
type_spec :  
    simple_type_spec | constr_type_spec  
  
union_type :  
    union identifier switch ( switch_type_spec ) { switch_body }  
  
unsigned_int :  
    unsigned ( unsigned_short_int | unsigned_long_int | unsigned_longlong_int )  
  
unsigned_long_int :  
    long  
  
unsigned_longlong_int :  
    long long  
  
unsigned_short_int :  
    short  
  
wide_char_type :  
    wchar  
  
wide_character_literal :  
  
wide_string_literal :
```

Chapter :
Complete IDLos grammar

```
wide_string_type :  
    wstring { < positive_int_const > }
```



The next chapter provides a complete reference to the ObjectSwitch action language.

9

Action language reference

This section provides a reference to the ObjectSwitch action language, organized by keyword. Each statement or function in the action language appears on its own page, with a top-level syntax definition, a description of the statement or function, and an example. See “Complete action language grammar” on page 349 for a complete syntactic definition of the action language.

About the notation

The syntax description for each statement or function uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

plain text indicates a feature of the notation, as follows:

(*item*) parentheses group items together

(*item*) * the item in parentheses appears zero or more times

(*item*) + the item in parentheses appears one or more times

{ *item* } items in braces are optional

(*item1* | *item2*) the vertical bar indicates either *item1* or *item2* appears

For example:

```
keyword { optionalKeyword }  
  { optionalThing } ( zeroOrMoreRepeatingThing ) *  
  ( oneAlternative | otherAlternative | thirdAlternative )
```

Some common elements

The following paragraphs describe some common elements found in the action language grammar. These are not statements or functions, so they do not appear under their own headings later in this chapter. Nor does the action language grammar (see “Complete action language grammar” on page 349) explain their meaning.

handle A handle is a reference to an object. When initially declared, a handle is empty and may be used only as the target of an assignment, select, or create statement.

chainSpec A chainSpec represents a relationship traversal. When one entity or interface has a relationship to another entity or interface, two objects may be related using a `relate` statement. The relationship can then be traversed from one object to the other; this involves a chain spec of the form:

scopedType [roleName]

For example, the `for...in` statement:

`for handle in setSpec { where whereSpec } statementBlock`

accepts either a chainSpec or an entity name for *setSpec*. The chainSpec might be used as in either of the following examples:

```
// verify this subscriber's phone services
for s in thisSubscriber->Service[subscribesTo]
{
    service.verify();
}

// reorder all the parts used in this assembly
for p in thisAssembly->Component[contains]->Part[includes]
{
    part.reorder();
}
```

A chainSpec may also represent an associative relationship traversal. An associative relationship is where one entity has a relationship with another. This relationship can only be traversed using the third entity. For example:

```
select thisSeller from Seller where thisSeller.name = "fred";  
for aBuyer in thisSeller->Sale[sell_to]->Buyer[sell_to]  
{  
    aBuyer.verifyCredit();  
}
```

break

Exit from a while or for loop.

Syntax

```
break;
```

Description

This loop control statement lets you exit immediately from a `while`, `for...in`, or `for` loop. If while or for loops are nested, the `break` exits from the containing loop only.

IDLos Constraints

None.

Warnings

None.

Example

```
declare long x;  
for (x = 0; x < 10; x++)  
{  
    //  
    // Check for termination condition. If  
    // found get out of loop immediately  
    //  
    if (someCondition)  
    {  
        break;  
    }  
}
```

See Also

`continue`, `for`, `for...in`, `return`, `while`

cardinality

Return the number of objects in an extent or relationship.

Syntax

```
cardinality ( ( className | (handle | self) -> chainSpec ) );
```

Description

`cardinality` returns the number of objects in an extent or relationship. Cardinality can be used on extents and a single relationship chain. Extended relationship chains are not allowed.

`cardinality` should be used instead of iterating over an extent or relationship to determine the number of objects. This has a large performance advantage over iteration since extents and relationships maintain the number of objects directly, eliminating the requirement to iterate over the entire extent or relationship.

IDLos Constraints

None

Warnings

When you use `cardinality` on extentless entities, the result is approximate.

Example

```
entity E { };
entity I { };

relationship EToI
{
    role E toI 1..1 I;
    role I toE 1..* E;
};

// --- Get the number of instances of E (the extent) ---
declare long numE = cardinality (E);

// --- Get the number of instances of E related to I ---
```

Chapter 9: Action language reference

cardinality

```
declare long numItoE = 0;
declare I    anI;

// Iterate over the extent adding up the number of
// related instances of E
for anI in I
{
    numItoE = numItoE + cardinality (anI->E[toE]);
}
```

See Also

```
for ... in
in
```


continue

Skip immediately to the next iteration of a `for`, `for...in`, or `while` statement.

Syntax

```
continue;
```

Description

This loop control statement skips immediately to the next iteration of a `for`, `for...in`, or `while` statement.

IDLos Constraints

None.

Warnings

None.

Example

```
declare Customers aCustomer;  
for aCustomer in Customers  
{  
    // Check to see if we have already processed  
    // this customer. If we have, skip this customer  
    // and proceed to the next  
  
    if (aCustomer.seen)  
    {  
        continue;  
    }  
  
    // First time we have seen this customer. Welcome  
    // them.  
}
```

See Also

`break`, `for`, `for ... in`, `return`, `while`

create

Create an object at a specific location with initial values.

Syntax

```
create handle
{ on locationSpec }
{ values ( valueSpec (, valueSpec)* ) } ;
```

Description

This statement creates an object using a previously declared handle of entity or interface type. A handle is set to empty upon declaration, and is invalid (that is, it does not refer to any object) until it is assigned to a valid object or used in a create statement.

locationSpec is the name of the ObjectSwitch node where the object will be created. If the on clause is omitted, the object is created on the local node.

valueSpec is an initial value assignment for the attributes or relationships, in the form *attributename: value* (for relationships, *chainSpec: value*) If attributes are not assigned initial values, the value of the attribute is undefined. The only exception to this is object references. Object references are initialized to a value of empty.

IDLos Constraints

The entity being created cannot have been declared as a singleton.

Warnings

It is an error to use `create` on entities that have been declared as `singleton` in your model. You must use `create singleton` instead.



Attributes that are part of a key must be initialized to a unique key value to avoid duplicate key exceptions during object creation.

Relationships that are used for referential integrity in a backing database should be initialized during creation to ensure that referential integrity is maintained.



A race condition during object creation allows two extentless objects to be created with the same object ID. Always enclose extentless creates in a try..catch block to catch the `ExceptionObjectNotUnique` exception.

Examples

```
entity Customer
{
    attribute string  name;
    attribute long    id;
    key CustId { id };
};

declare Customer lclCustomer;
declare Customer rmtCustomer;
declare long     uniqueKey;

//
// allocateCustId allocates a unique customer id
//
uniqueKey = self.allocateCustId();

//
// Create a customer on the local node
//
create lclCustomer values (name:"Joe", id:uniqueKey);

//
// Allocate another customer id
//
uniqueKey = self.allocateCustId();

//
// Create a customer in Paris
//
create rmtCustomer on "Paris" values (name:"Jean-Louis",
                                     id:uniqueKey);
```

See Also

create singleton, delete

create singleton

Create a singleton object at a specific location with initial values.

Syntax

```
create singleton handle  
{ on locationSpec }  
{ values ( valueSpec ( , valueSpec ) * ) };
```

Description

This statement creates a singleton object using a previously declared handle of entity or interface type. There is only one instance of a singleton entity.

`create singleton` can be called multiple times. If the single instance has not been created it is created and returned to the caller. If the instance already exists, a handle to the single instance is returned to the caller.

IDLos Constraints

Entity must be declared as a singleton.

Warnings

It is an error to use `create singleton` on entities and interfaces that have not been declared as a singleton in IDLos.

Example

```
[ singleton ]  
entity PortAllocator  
{  
    void doInitialization();  
};  
  
//  
// Create a singleton  
//  
declare PortAllocator myAllocator;  
  
create singleton myAllocator;
```

```
//  
//   Invoke a two-way operation to ensure that  
//   it is initialized before using it  
//  
myAllocator.doInitialization();
```

See Also

create, del ete

declare

Declare a variable.

Syntax

```

declare type identifier { [ integerLiteral ] | assignmentExpression };
declare { ( type identifier { [ integerLiteral ] | assignmentExpression }; )+ }
declare const type identifier { [ integerLiteral ] | assignmentExpression };

```

Description

All variables must be declared before they are used. The ObjectSwitch action language is strongly typed. Variables are scoped to an operation, state, or code block defined within an operation or state.

The value of a variable is undefined until it is assigned a value. The only exception is object handle which are initialized to empty when they are declared. Variables can be assigned initial values in the declare statement.

Variables can be declared as arrays. There is no support for initializing the values of an array in the declare statement.

Strings can also be declared as either bounded or unbounded.

IDLos Constraints

None.

Warnings

A variable should always be initialized to a known value. The value of an uninitialized variable is undefined and may change in the future. The only exception is for object handles, which are empty after being declared.

Example

```

//
// Declare some variables
//
declare long    aLong = 0;

```

```
declare AnEntity anEntity;

//
// Declare an array of shorts
//
declare short    anArrayOfShorts[100];

//
// Declare a variable inside of a for loop
//
declare long i;
for (i = 0; i < 10; i++)
{
    declare long tmpVar;
}
// tmpVar is out of scope here

//
// Bounded strings
//
declare string<10> firstName;

//
// Unbounded string
//
declare string    unboundedString;
```

See Also

Types

delete

Destroy an object.

Syntax

`delete handle;`

Description

Destroy an object. All memory associated with the object is discarded. After invoking `delete` on an object handle, using that handle will produce a runtime error.

An implicit `unrelate` is performed on all relationships in which this object participates.

IDL's Constraints

None.

Warnings

None.

Example

```
declare Customer aCustomer;  
create aCustomer;  
  
// Do something with customer  
  
// Destroy the customer  
delete aCustomer;  
  
// aCustomer handle is now invalid
```

See Also

`create`
`create singleton`

empty

Test and set handles to empty.

Syntax

```

declare type_identifier handle = empty;
empty handle
handle = empty;

```

Description

`empty` returns a boolean value if used in an expression. `true` is returned if the handle is currently unassigned to a valid object reference. `false` is returned if the handle holds a valid object reference.

`empty` can also be used to initialize a handle to known value that indicates that the handle does not contain a valid object reference.

IDLos Constraints

None

Warnings

None

Example

```

entity Customer { };

...

declare Customer aCustomer = empty;

//
// If the customer handle is empty create a new customer
//
if (empty aCustomer)
{
    create aCustomer;
}

```

Chapter 9: Action language reference

empty

```
//  
// Set the customer handle to empty  
//  
aCustomer = empty;
```

See Also

Exceptions

Exception handling.

Syntax

```
try statementBlock ( catchStatement )+  
catch( scopedType ) statementBlock  
throw handle;
```

Description

These statements provide support for handling user and system exceptions.

User exceptions are defined in IDLos using the IDL `excepti on` and `raises` keywords. System exceptions are a predefined set of exceptions that can be raised by the ObjectSwitch runtime and adapters. System exceptions cannot be thrown by the user, but they can be caught.

The supported system exceptions are defined in the `swbuiltin` component. You must import this component to catch system exceptions in action language. For details on the individual system exceptions, refer to the on-line documentation for the *swbuiltin* component.

IDLos Constraints

User exceptions are defined using IDLos. Operations that throw an exception must have a `raises` clause in the operation signature.

Warnings

Uncaught user and system exceptions will cause a runtime engine failure.

System exceptions should not be thrown by users.



Handling the system exceptions `Excepti onDeadLock` and `Excepti onObj ectDestroyed` *in application code not executed as part of spawn will produce unpredictable but bad results.*

Example

```
package Life
{
  entity A
  {
    exception IsDead
    {
      string causeOfDeath;
    };
    exception IsTired
    {
      long nextTime;
    };

    void wakeUp(in string reason) raises (IsDead, IsTired);
    void checkIt(void);
  };

  //
  // wakeUp operation. Throws exception on how
  // the wakeup was processed
  //
  action Life::A::wakeUp
  {`
    //
    // Figure out my current state
    //
    if (imDead)
    {
      declare IsDead id;
      id.causeOfDeath = "boredom";
      throw id;
    }
    if (imTired)
    {
      declare IsTired it;
      it.nextTime = 60;
      throw it;
    }
  `};

  //
  // checkIt handles exceptions thrown by wakeUp
  //
  action Life::A::checkIt
  {`
```

```

try
{
    self.wakeUp("it's noon!");
}
catch (IsDead id)
{
    declare string cd = id.causeOfDeath;
    printf("id.causeOfDeath = %s\n", cd.getCString());
}
catch (IsTired it)
{
    printf("it.nextTime = %s\n", it.nextTime);
}
`;

//
// Example of string overflow exception
//
entity StringOverflow
{
    attribute string<5> stringFive;
};

//
// Assign a string larger than 5 characters to
// stringFive. This will cause an exception
//
try
{
    self.stringFive = "123456";
}
catch (ExceptionStringOverflow)
{
    printf("caught ExceptionStringOverflow\n");
}

//
// Example invalid array exception
//
typedef long ArrayFive[5];

//
// Use an index > 4. This will cause an exception.
// Remember that arrays are zero based.
//
declare ArrayFive arrayFive;
try
{
    arrayFive[5] = 1;

```

Chapter 9: Action language reference

Exceptions

```
    }  
    catch (ExceptionArrayBounds)  
    {  
        printf("caught ExceptionArrayBounds\n");  
    }
```

See Also

spawn, Transactions

extern

Specifies an external symbol.

Syntax

extern (*scopedType* | *cType*) *quotedIdentifier*

Specifies an external symbol that will be referenced in your action language. The model compiler will resolve this symbol at build time. This removes the need to create an SDL wrapper for simple symbols that your model references.

IDLos Constraints

None.

Warnings

None.

Example

```
extern int myGlobalVar;  
myGlobalVar = 123;
```

for

Iterate over statements with iterator expression and termination condition.

Syntax

for (*lval* { *assignmentExpr* } ; *conditionExpr* ; *iteratorExpr*) *statementBlock*

Description

An iteration statement; enables you to loop through and execute a block of code until a condition becomes false.

assignmentExpr is performed once before iteration begins. It is typically used to set the initial value of a variable used in the next two expressions.

conditionExpr is evaluated before each iteration. If the expression is nonzero, then *statementBlock* is executed.

iteratorExpr is performed each iteration after *statementBlock* is executed.

IDLos Constraints

None.

Warnings

None.

Example

```
for ( x = 0; x < maxTimes; x++ )
{
    //
    //   Repeat this work maxTimes
    //
}
```

See Also

for in

while

for ... in

Iterate over an extent or relationship.

Syntax

for handle in setSpec { where whereSpec } statementBlock

Description

Iterates over a set of objects. If the set is empty, the statement block is never executed. The handle must be of the same type as the set it iterates over.

IDLos Constraints

None.

Warnings

Do not confuse this statement with the `in` function.

Example

Consider the following IDLos definitions:

```
entity Customer { attribute string firstName; };
entity Queues { };
```

```
relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};
```

Queues and Customers that have been created and related can be iterated across using the `for ... in` statement in various ways:

```
//
// --- welcome all the customers held in some queue q ---
//
for cust in q->Customer[holds]
{
    // Welcome this customer
}
```

```
//  
// --- Send a bill to every customer that exists ---  
//  
for cust in Customer  
{  
    // Send a bill to this customer  
}  
  
//  
// --- Welcome customers called Dan in some queue q ---  
//  
for cust in q->Customer[holds] where (cust.firstName == "Dan")  
{  
    // Welcome this Dan  
}
```

See Also

cardinality
select
in

if else else if

Conditionally execute a statement block.

Syntax

```
if booleanExpression statementBlock  
  { ( else statementBlock | elseIfStatement ) }
```

```
elseIfStatement :  
  else if booleanExpression statementBlock  
  { ( else statementBlock | elseIfStatement ) }
```

Description

A conditional control flow operator. The `else` clause is optional. If statements may be nested.

IDLos Constraints

None.

Warnings

None.

Example

```
if (aCustomer.balanceDue < aCustomer.creditLimit)  
{  
  //  
  // Allow withdrawal  
  //  
}  
else if (aCustomer.isPaymentOverdue)  
{  
  //  
  // Refuse order with rude message  
  //  
}  
else
```

```
{  
  //  
  // Credit limit is expired, but payments  
  // current. Politely refuse withdrawal  
  //  
}
```

See Also

for
while

in

Test whether a given object reference refers to a member of some set of objects.

Syntax

handle in handleOrSelf -> chainSpec

Description

Returns true if the object referenced by the handle is a member of the set specified. Otherwise it returns false.

IDLos Constraints

None.

Warnings

Do not confuse this function with the for..in statement.

Example

```
//  
// If is one of my cars, drive it  
//  
if( thisCar in self->Car[owns] )  
{  
    thisCar.drive();  
}
```

See Also

cardinality
for ... in

isnull

Test and set an attribute's null indicator.

Syntax

```
i snull attributeName
attributeName = i snull ;
```

Description

`i snull` returns a boolean value if used in an expression. `true` is returned if the object is created without an assigned initial value. `false` is returned if the object is created with an explicit value.

`i snull` can also be used to set the null indicator of an attribute.

IDLos Constraints

None.

Warnings

If `i snull` is used to set the null indicator of an attribute the data for the attribute does not change though subsequent tests for `i snull` will be `true`.

Example

```
entity Invoice
{
    ...
    attribute long total;
}
declare Invoice anInvoice;
create anInvoice;
...
if (i snull anInvoice.total)
{
    // the attribute total is null.
}
```

relate

Relates one object to another across a relationship.

Syntax

```
relate (handle | self) relationshipSpec (handle | self)
{ using (handle | self) } ;
```

Description

Establishes a relationship between two objects, or an associative relationship between three objects if a `using` clause is provided.

Relationships can be used as ordered lists by defining a 1..1 relationship from an entity to itself. The order of the handles in the `relate` statement defines the order of the objects in the relationship.

The order of the two handles is significant only if they refer to the same entity.

If the cardinality of any side of a role is 1, performing a `relate` on that role will cause an implicit `unrelate` to occur for any objects previously in that relationship.

IDLos Constraints

None.

Warnings

None.

Example

```
entity Customer { };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};
```



```
relationship CustomerOrder
{
    role Customer inFrontOf 1..1 Customer;
    role Customer inBackOf 1..1 Customer;
};

//
// Put a customer in a queue
//
relate aCustomer waitsIn aQueue;

//
// This is also valid for putting a customer
// in a queue
//
relate aQueue holds aCustomer;

//
// Put impatient customer in front of nice customer
//
relate impatientCustomer inFrontOf niceCustomer;
```

See Also

unrelate

return

Return from an operation or state.

Syntax

```
return { expression } ;
```

Description

A return statement signals completion of an operation or a state and returns control to the caller. Operations with return values must have a return statement and provide a return value expression.

There may be any number of return statements in an operation or state.

IDLos Constraints

Operations must be declared with the correct return type.

Warnings

It is illegal to return a value from a state.

Example

```
entity ATM
{
    long getCash();
    void pressOk();
};

//
//  Implementation of getCash operation
//
action getCash
{
    declare long    cashAmount = 12345;
    return cashAmount;
*};

//
//  Implementation of pressOk operation
//
```

```
action pressOk  
{  
    return;  
};
```

See Also

select

Select an object using a relationship or an extent.

Syntax

```
select handle from (handle | self) ( ( -> ) chainSpec )+ { where whereSpec };
```

```
select handle from className where whereSpec;
```

```
select handle from singleton className;
```

Description

select always returns a single object. It can be used against an extent or a many relationship only with a where clause to narrow the select to return a single object.

The three forms of select above are:

- navigate a relationship chain with a where clause. The where clause is not required if this navigation is towards the one side of a relationship role.
- select a single object from an extent using a where clause.
- select a singleton.

IDLos Constraints

None.

Warnings

It is a runtime error if a select statement returns more than a single object.

Example

```
[ singleton ] entity TheBoss { };
entity Customer
{
    attribute long custId;
    key CustomerId { custId };
};
entity Contact { };
```

```
relationship CustomerContacts
{
    role Contact isFrom 1..1 Customer;
    role Customer has 0..* Contact;
};

///// ---- above was IDLos, below is action language ---- /////

//
//  Select the next customer in line
//
select aCustomer from aContact->Customer[isFrom];

//
//  Select customer 123 from all customers
//
select aCustomer from Customer where (aCustomer.custId == 123);

//
//  Find the boss
//
select boss from singleton TheBoss;
```

See Also

for...in

select...using

Syntax

```
select handle from className using keyName where whereSpec
{ read lock | write lock | no lock }
{ on empty create { values valuesSpec } }
```

Description

select...using looks up an entity using the specified key. By default, if the object is found, it is locked for reading. If you specify `no lock`, the object will not be locked. Specifying `write lock` locks the object for writing.

If you specify `on empty create` and the object is not found, `select...using` creates a new object with the key values you supplied in the `where` clause. You can set other attribute values using the optional `values` clause.

Warnings

If no object is found, and `on empty create` was not specified, it is possible that an entity matching the specification may be created in another transaction. This could cause a subsequent create of the object to throw `ExceptionObjectNotUnique`.

Examples

The following examples are based on this IDL's entity definition:

```
entity Obj
{
    attribute long x;
    attribute long y;

    key key1 { x };
};
```

To select the instance of entity `Obj` where `x` is zero:

```
declare Obj obj;
select obj from Obj using key1 where (obj.x == 0);
```

If no instance exists where *x* is zero, the handle "obj" will be empty. To create an object with the given key if it is not found, use `on empty create`:

```
select obj from Obj using key1 where (obj.x == 0)
on empty create;
```

This instantiates an object with the attribute *x* initialized to zero. The attribute *y* will be unset. To initialize additional attributes when the entity is created, a `values` clause may be used:

```
select obj from Obj using key1 where (obj.x == 0)
on empty create values (y: 46);
```

Note that the key attributes' values are taken from the `where` clause, and may not be specified again in the `values` clause.

Locking examples

By default, when an entity is found, it is locked for reading. This ensures that the entity will not be modified by another transaction after it is selected. If the entity is to be modified, a write lock may be taken via the `writelock` option:

```
select obj from Obj using key1 where (obj.x == 0)
writelock;
```

This prevents a lock promotion, which can increase the likelihood of a deadlock.

The `nolock` option may be used to specify that the object should not be locked. This is especially useful when sending oneway events to objects which may be locked by another transaction:

```
select obj from Obj using key1 where (obj.x == 0)
nolock;
```



Using the `nolock` option means that another transaction can modify the object you have just selected, while you are handling it.

self

Reference the current object.

Syntax

`self`

Description

The `self` keyword is used to reference the current object in which an operation or state is executing.



In spawned operations, `self` refers to the thread of execution, not to an actual object. See “spawn” on page 335 and “Terminating spawned threads” on page 227 for more information.

IDLos Constraints

None.

Warnings

Assigning `empty` or any handle to `self` is illegal.

Example

```
entity Customer
{
    attribute long customerId;
    void anOperation();
};

action anOperation
{
    declare long x;
    x = self.customerId; // Access customerId in the current object
};
```

See Also

`empty`

spawn

Spawn a user thread of control.

Syntax

```
spawn scopedType ( { paramSpec ( , paramSpec ) * } );
```

Description

Spawn is used to create a thread of control that is managed by the user. This provides a mechanism for users to call blocking external functions and manage transactions as work is injected into ObjectSwitch.

For example, `spawn` can be used to create a listener thread for a protocol stack. As messages are received on the protocol stack a transaction is started using the transaction support in action language and the work is injected into ObjectSwitch.



The `spawn` statement can only be used on operations in local entities. Thus the spawned operation is not associated with any transaction or shared memory.

Object references used as parameters to the `spawn` operation and ones declared in the `spawn` operation are not in a transaction until a `begin transaction` statement is executed. Access to these objects remains valid until a `commit` or `abort transaction` statement is executed. Accessing these objects outside of a transaction will cause a runtime exception.

For example, the following action language will cause a runtime exception:

```
ALocal Entity : aSpawnMethod(in Long arg1)
{
    declare AnEntity    anEntity;

    // Causes a runtime exception since no transaction
    //      was started
    create anEntity;

    begin transaction;
    ...
}
```

IDLos Constraints

You can only spawn an operation that:

- is defined in a local entity
- is a two-way void operation
- uses only in parameters

Warnings

Spawned operations must use `begin transaction` before accessing any objects. Similarly, they must use `commit transaction` once they have finished.

Spawned operations must explicitly manage all exception handling, including transaction deadlocks. (A transaction deadlock can occur when accessing an object reference, accessing an attribute, or invoking a two-way operation.) The spawned operation must catch the `ExceptionDeadLock` system exception and abort the current transaction. Any uncaught system exceptions will cause a fatal engine failure.

Spawned operations cannot use `return while` in an active transaction. Doing so causes a fatal engine error.



When an engine is shut down, the thread manager kills spawned threads that are not in an active transaction. During transactions in spawned threads, you should periodically check whether the thread needs to terminate. Refer to the online documentation for the `swbuiltin` component to learn more about using the `shouldTerminate()` operation.

A spawned thread that blocks in an external function with a transaction active can prevent clean shutdown. Do not block within transactions in spawned threads.

Example

This is a simple example of how the `spawn` and `transaction` statements can be used to implement a simple server.

```
package AServer
{
    [ local ]
```

```

entity Initialize
{
  [ initialize ]
  void startServer(void);
};

[ local ]
entity ServiceThread
{
  void runServer(in long msgSize);
  void doWork(inout RuntimeServer rs);
};

//
// The following entity method implementations are
// left as an exercise to the reader.
//
entity WorkerObject
{
  oneway void doit(in string data);
  ...
}

[ local ]
entity RuntimeServer
{
  exception NetworkFailure { };
  void init(in long msgSize) raises (NetworkFailure);
  void recv(out string data) raises (NetworkFailure);
};

//
// Implementation of startServer
//
action AServer::Initialize::startServer
{
  //
  // Assumes a builtin interface to registry.
  //
  declare swbuiltin::Registry registry;
  declare long msgSize;
  msgSize = registry.getInt32("SocketServer", "msgSize", 0);
  declare long numThreads;
  numThreads = registry.getInt32("SocketServer", "numThreads", 5);

  //
  // Spawn numThreads threads of control
  //

```

Chapter 9: Action language reference

spawn

```
    declare long    i;
    for (i = 0; i < numThreads; i++)
    {
        spawn ServiceThread::runServer(msgSize:msgSize);
    }
};
//
//  Implementation of runServer
//
action AServer::ServiceThread::runServer
{
    //
    // Call native method to do server initialization
    //
    declare RuntimeServer    rs;
    declare boolean    runForever = true;
    while (runForever)
    {
        declare boolean    initComplete = false;
        try
        {
            rs.init(msgSize:msgSize);
            initComplete = true;
        }
        catch (RuntimeServer::NetworkFailure)
        {
            printf("Could not init server\n");

            //
            // Sleep and retry the server
            // initialization.
            //
            swbuiltin::nanoSleep(60, 0);

            //
            // Or we could kill the engine here.
            //
            // swbuiltin::stop(1);
            //
        }

        //
        // Call dowork to process work until the
        // connection is lost. This shows that we can
        // call other IDLos local entity operations
        // in a spawned thread of control.
        //
        if (initComplete == true)
```

```

    {
        self.doWork(rs);
    }
};

```

action AServer::ServiceThread::doWork

```

{
    //
    // Create a worker object. Probably actually want
    // to create a pool of them, and manage a free list
    // (via a relation) for injecting work.
    //
    declare WorkerObject wo;

    //
    // Need to manage lifecycle of this object.
    //
    begin transaction;
    create wo;
    commit transaction;

    declare boolean ok = true;
    while (ok)
    {
        try
        {
            declare string data;

            //
            // Assume this call reads in data
            // from the network via the ::recv()
            // socket call. Since it blocks, we don't
            // want to be in a transaction.
            //
            rs.recv(data);

            //
            // Inject work into objectswitch (see
            // note above about worker pool).
            //
            begin transaction;
            wo.doit(data);
            commit transaction;
        }
        catch (swbuiltin::ExceptionDeadLock)
        {
            //
            // Here we throw away work and try

```

Chapter 9: Action language reference

spawn

```
// another recv(). Could be updated to
// continually try and post the last work
// item.
//
printf("got deadlock\n");

//
// Abort any pending work.
//
abort transaction;
}
catch (RuntimeServer::NetworkFailure)
{
    //
    // Example of catching a user defined
    // exception. Note we don't call
    // abort, since this was thrown from the
    // recv call.
    //
    printf("Caught a network failure\n");
    ok = false;
}
//
// Should catch and abort on all system
// exceptions
//
}

//
// returning here causes the network
// code to restart.
//
return;
`};
```

See Also

Exceptions, Transactions

Transactions

Explicitly manage ObjectSwitch transactions.

Syntax

(begin | commit | abort) transaction ;

Description

The three types of transaction statements let you explicitly manage transactions. This is only required and only allowed in an operation that was invoked using the `spawn` statement. All other operations and states are implicitly in a transaction when they are executed.

Objects that are created or selected in a spawned operation must be done in the context of a valid transaction that was started using `begin transaction`.

Once an object has been created or selected in a valid transaction it is implicitly reattached to any new transactions

See also “Transaction processing” on page 244 and related topics for a discussion of how transactions, locks, and deadlocks are handled at run time.

IDLos Constraints

None.

Warnings

Accessing objects passed as parameters are declared in a spawn operation that are not in a valid transaction will cause a runtime exception.

Example

This is a simple example of creating and selecting objects in a spawned local operation. A more detailed example of using transaction control is also provided under “spawn” on page 335.

```
ALocalEntity::aSpawnOperation(in long arg1)
```

Chapter 9: Action language reference

Transactions

```
{
  declare MyObject  mobj;
  declare SelectObject sobj;

  //
  //  Create and select in a valid transaction
  //
  begin transaction;
  create mobj;
  select sobj from singleton SelectObject;
  commit transaction;

  for (;;)
  {
    begin transaction;

    //
    //  No need to select these objects again.
    //  They are implicitly reattached to the new
    //  transaction.
    //
    mobj.numThings++;
    sobj.doIt(mobj.numThings);

    commit transaction;
  }
}
```

See Also

Exceptions

spawn

Types

Action language data types.

Syntax

See IDLos chapter.

Description

The action language uses the IDL type system, just like the rest of IDLos does. The `decl` are statement is used to define a type in an operation or state.

Structure members are accessed using the “.” notation.

Arrays and sequences are both accessed using “[n]” indexing to access a specific element in an array or sequence. Arrays and sequences use zero based indexing.

Unbounded sequences are automatically grown at runtime as required. If the index being accessed is larger than the current sequence size, the sequence is resized. When a sequence is resized, the values of all new sequence members is undefined. They must be initialized to a known value by the application.

IDLos Constraints

None.

Warnings

None.

Example

```
struct AStruct
{
    long   aLongMember;
    short  aShortMember;
};

typedef sequence<char> UnboundedCharSequence;
typedef long ALongArray[10];
```

Chapter 9: Action language reference

Types

..... Above was IDLoS, below is action language

```
// --- Accessing a structure member ---
declare AStruct  aStruct;
declare long     aLong;
aLong = aStruct.aLongMember;

// --- Accessing index 23 in unbounded sequence. ---
// --- NOTE - Sequences and arrays are zero based. ---
declare UnboundedCharSequence seq;
declare char                aChar;
aChar = seq[22];

// --- Accessing the first index in an array ---
declare ALongArray array;
declare long        aLong;
aLong = array[0];

// --- Using an any ---
declare any  anAny;
declare long aLong;
declare string aString;

// --- Assign a long to an any ---
aLong = 1;
anAny <=<= aLong;
// anAny now contains 1

// --- Now assign a string to the same any ---
aString = "hello world";
anAny <=<= aString;
// anAny now contains "hello world"
```

See Also

declare
Exceptions

unrelate

Unrelate objects.

Syntax

```
unrelate (handle | self) relationshipSpec (handle | self)
{ using (handle | self) } ;
```

Description

The `unrelate` statement terminates the relationship between the specified objects. If the relationship is an associative relationship the `using` clause must be specified to define the third-party in the associative relationship. Deleting a related object performs an implicit `unrelate`.

IDLos Constraints

None.

Warnings

None.

Example

```
entity Customer { };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};

..... Above was IDLos, below is action language .....

// --- Remove a customer from a queue ---
unrelate aCustomer waitsIn aQueue;

// --- Remove a customer from a queue the other way works too ---
unrelate aQueue holds aCustomer;
```

See Also

delete
relate

while

Loop over a statement block while a condition remains true.

Syntax

while booleanExpression statementBlock

Description

The while statement loops and executes a set of statements repeatedly as long as a condition is true. You may use `break` to exit a while loop at any time, or use `continue` to skip immediately to the next iteration.

IDLs Constraints

None.

Warnings

None.

Example

```
while (aClerk.atWork)
{
    // --- Can this clerk help this customer? ---
    if (!aCustomer.canHelp)
    {
        continue;
    }

    // --- Time to quit? Just leave ---
    if (aClerk.doneForTheDay)
    {
        break;
    }

    //
    //   Service customers
    //
}
```

See Also

break
continue
for

Complete action language grammar

This section contains the grammar that is used by the action language parser. Grammatical elements in this list occasionally differ from their equivalents in the SWAL statement and functions section, because in that section a few changes were made for the sake of clarity:

- minor name changes added missing semantic information
- unimplemented parts of the grammar were omitted
- some elements were expanded or condensed according to the grammar

The elements defined in the grammar are listed in alphabetical order. A few elements—such as *string-literal* or *identifier*—are defined by the IDL grammar. See the IDLos grammar in Chapter 2 for a definition of these items.

action:

type *scopedMethodName* *inputParameterList* { **const** } *statementBlock* |
includeStatement | *idStatement* | **package** *identifier* ;

actionFile:

(*action*) +

addingExpression:

multiplyingExpression ((+ | -) *multiplyingExpression*) *

arrayExpression:

{ :: } (*identifier* ::) * *identifier* | *integerLiteral*

arrayList:

([*expression*]) *

assignEqualOp:

* = | /= | %= | += | -= | <=< | >>= | &= | ^= | |=

assignmentExpression:

(= | *assignEqualOp*) (*expression* | **empty**) | (++ | --)

attributeName:

identifier

binaryOperator:

+ | - | * | /

booleanExpression:

(*expression*)

booleanLiteral:

true | **false**

Chapter 9: Action language reference

Complete action language grammar

booleanOperator:
 && ||

cardinalityFunction:
 cardinality ((*scopedType* | *handleOrSelf* -> *chainSpec*))

catchExpression:
 scopedType { *identifier* }

catchStatement:
 catch (*catchExpression*) *statementBlock*

chainSpec:
 scopedType [*relationshipSpec*]

charLiteral:
 character-literal

className:
 scopedType

comparisonOperator:
 < | > | <= | >= | == | !=

constDeclare:
 { **const** }

createStatement:
 create { **singleton** } *handle* { **on** *locationSpec* }
 { **implementation** *implementationSpec* }
 { **values** (*valueSpec* (, *valueSpec*) *) }

cType:
 unsigned short | **short** | **unsigned int** | **int** | **long** |
 long long | **unsigned (long | long long)** | **float** | **double** | **unsigned char** |
 char | **void**

declareSpecStatement:
 type identifier { [*arrayExpression*] | *assignmentExpression* } ;

declareStatement:
 declare *declareSpecStatement* | **declare** { (*declareSpecStatement*) + }

deleteStatement:
 delete *handle* | **delete** [] *handle*

elseIfStatement:
 else if *booleanExpression* *statementBlock* { (*elseStatement* | *elseIfStatement*) }

elseStatement:
 else *statementBlock*

expression:
 relationalExpression (*booleanOperator* *relationalExpression*) *

extendedChain:

((->) *chainSpec*) +

floatLiteral:

floating-point-literal

forStatement:

for **handle in** *setSpec* { **where** *whereSpec* } { **order by** *attributeName* } *statement-Block* | **for** (*lval* { *assignmentExpression* } ; *expression* ; *expression*) *statement-Block*

function:

cardinalityFunction | *inFunction*

genAssignDeclareStatement:

{ **const** } (*cType* | *scopedType* | **self**) (*paramList* | *userDereference identifier* { ([*arrayExpression*] | *assignmentExpression* | *paramList*) } | . *identifier arrayList* (*paramList* | = (*expression* | **empty**) | *assignEqualOp expression*))

handle:

identifier

handleOrSelf:

handle | **self**

idStatement:

identifier (*expression*) ;

ifStatement:

if *booleanExpression* *statementBlock* { (*elseStatement* | *elsifStatement*) }

implementationSpec:

identifier | *integerLiteral*

includeStatement:

include < *identifier* ((. | /) *identifier*) * >

inFunction:

handle in handleOrSelf -> *chainSpec*

inputParameter:

type & *identifier*

inputParameterList:

((*inputParameter* (, *inputParameter*) * | **void** |))

integerLiteral:

decimal-integer-literal | *hexadecimal-integer-literal* | *octal-integer-literal* | *decimal-integer-literal_LL* | *hexadecimal-integer-literal_LL* | *octal-integer-literal_LL*

literal:

stringLiteral | *charLiteral* | *integerLiteral* | *floatLiteral* | *booleanLiteral*

locationSpec:

Chapter 9: Action language reference

Complete action language grammar

identifier | *stringLiteral*

loopControlStatement:
break | **continue**

lval:
{ (++ | --) } *identifier* *arrayList*

lvalStatement:
lval { *assignmentExpression* }

methodName:
identifier

multiplyingExpression:
signExpression ((* | / | %) *signExpression*) *

nativeType:
unsigned short | **short** | **unsigned (long | long long)** | **long** | **long long** |
float | **double** | **boolean** | **char** | **wchar** | **octet** | **Object** | **any** | **string** | **wstring**
| **void**

operationName:
identifier

parameterName:
identifier

paramList:
({ *paramSpec* (, *paramSpec*) * })

paramSpec:
parameterName : *expression* | *expression*

primitiveExpression:
{ (& | *) } *identifier* *arrayList* { (++ | --) } | *literal* | *function* |
handleOrSelf . *attributeName* (. *attributeName*) * (*paramList* | *arrayList*) | *user-*
MethodOrVar | *staticMethodOrEnum* | (*expression*) |
sizeof ((*identifier* | *cType*)) | **empty** *handleOrSelf* |
new ({ :: } (*identifier* ::) * *identifier* | *cType*) { [*arrayExpression*] } | **self**

relateStatement:
relate *handleOrSelf* *relationshipSpec* *handleOrSelf* { **using** *handleOrSelf* }

relationalExpression:
addingExpression (*comparisonOperator* *addingExpression*) *

relationshipName:
identifier

relationshipSpec:
relationshipName

returnStatement:

```

    return { expression }

scopedMethodName:
    ( identifier :: )+ identifier

scopedType:
    { :: } ( identifier :: )* identifier

selectStatement:
    select { distinct } handle from ( handleOrSelf extendedChain { where whereSpec } | singleton className | className where whereSpec )

setSpec:
    className | handleOrSelf extendedChain

signExpression:
    { + - } unaryExpression

spawnStatement:
    spawn scopedType ( { paramSpec ( , paramSpec )* } )

statement:
    declareStatement | createStatement ; | deleteStatement ; | relateStatement ; | unrelateStatement ; | selectStatement ; | forStatement | ifStatement |
    whileStatement | tryStatement | throwStatement ; | loopControlStatement ; |
    transactionStatement ; | spawnStatement ; | genAssignDeclareStatement ; |
    userMethodStatement ; | lvalStatement ; | returnStatement ; | statementBlock

statementBlock:
    { ( statement )* }

staticMethodOrEnum:
    { :: } ( identifier :: )+ operationName { paramList }

stringLiteral:
    ( string-literal )+

throwStatement:
    throw handle

transactionStatement:
    ( begin | commit | abort ) transaction

tryStatement:
    try statementBlock ( catchStatement )+

type:
    constDeclare ( nativeType | userDefinedType )

unaryExpression:
    { unaryOperator } primitiveExpression

unaryOperator:
    ! | -

unrelateStatement:

```

Chapter 9: Action language reference

Complete action language grammar

```
unrelate handleOrSelf relationshipSpec handleOrSelf { using handleOrSelf }  
userDefinedType:  
  scopedType  
userDereference:  
  { ( * | & ) }  
userMethodOrVar:  
  handle -> methodName { paramList } | operationName paramList  
userMethodStatement:  
  handle ( -> methodName )+ { paramList } { assignmentExpression }  
valueSpec:  
  attributeName : expression | chainSpec : expression  
whereExpression:  
  ( whereSpec | wherePrimitive )  
wherePrimitive:  
  ( handleOrSelf . attributeName arrayList | handleOrSelf ) comparisonOperator  
  primitiveExpression  
whereSpec:  
  ( whereExpression ( booleanOperator whereExpression )* )  
whileStatement:  
  while booleanExpression statementBlock
```



The next chapter provides a complete reference to the build specification language.

10

Build specification reference

This chapter provides a reference for the keywords and commands you use to compose and execute a build specification. These appear in alphabetical order. Each page includes the syntax of the keyword or command, a discussion of usage, and an example.

A simple example

The example below shows a very simple build specification. For a full build specification that uses all the standard features, see “Complete build specification example” on page 370.

```
component Component1
{
    source MyIDLosFile.soc

        import ImportedComponent
        {
            includePath=/this/is/where/the/component/is;
            libraryPost=someLibraryThisComponentNeeds;
        };

        package Package1;
        package :: Package2
        adapter Sybase
        {
            entity Package1::PersistentEntity
        };
    };
};
```

Understanding the syntax notation

The syntax description for each keyword or command uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

plain text indicates a feature of the notation, as follows:

(*item*) parentheses group items together

(*item*)+ the item in parentheses appears one or more times

{ *item* } items in braces are optional

adapter

Defines an adapter block.

Syntax

```
adapter adapterName { buildLanguage };
```

Description

adapterName is the name of a valid, installed service adapter.

An adapter block defines a service adapter for a specific model element. Each adapter must contain an interface or entity, depending on its type.

Adapters also have properties that affect the way they operate. Refer to the documentation for a specific service adapter for more information on the properties used by that adapter.

Example

```
adapter CORBA
{
    interface :: MyPackage:: anInterfaceA;
    interface :: MyPackage:: anInterfaceB;
};
```

component

Defines a component block.

Syntax

```
component identifier { implementation=service-name }  
{ buildLanguage };
```

Description

A component block defines a component being built and will contain all the information needed to build the component. A component may implement one or more packages, or may simply contain interfaces to an external implementation. All packages in a given component must be listed in the order of dependency.

A component can act as a *wrapper* around a service implemented in another technology. The generated wrapper will behave just like any other component except that it uses a foreign implementation.

serviceName is the name of the foreign service being implemented.

Refer to the documentation for the individual adapter generator for more information on wrapping a foreign service.

Warnings

None

Example

```
component MyComponent  
{  
    source MyIDLos.soc;  
    package MyPackage;  
};  
component AnExternalComponent implementation=corba  
{  
    idlPath=MyIDL.idl;  
};
```


group

Defines a logical collection of a given type.

Syntax

```
group identifier { buildLanguage };
```

Description

Allows for grouping model references and properties.

Warnings

None

Example

```
component MyComponent
{
    adapter SNMP
    {
        group O I D
        {
            snmpPackageO I D = 1.3.6.1.4.1.3004.2.1.4;
            i n t e r f a c e MyPackage: : a n I n t e r f a c e A;
            i n t e r f a c e MyPackage: : a n I n t e r f a c e B;
        };
    };
};
```

import

Specifies a component dependency, or aggregates components.

Syntax

```
import identifier { { importImplementation = ( TRUE | FALSE ) } };
```

Description

A component may depend on the services provided by another component. You import a component so that its interface definitions and link libraries can be loaded into the Design Center server at build time.

The Design Center always knows where to locate built-in ObjectSwitch components. If you want to import your own components from an arbitrary location, you can specify this location using the `importPath` property. Note that this property follows normal scoping rules, so you can use different import paths for different components.

You can also aggregate components using the `importImplementation` property in the build specification. This property defaults to `FALSE`. Setting it to `TRUE` links the imported component's implementation into the one being built, with the performance benefit that all two-way operations in the imported component are now local calls (not dispatched through the event bus). The configuration for a component with imported implementations is automatically generated from the imported components' configurations, and library dependency issues are handled automatically.

Example

```
component MyComponent
{
    // set the include path to the component location
    importPath = /some/include/path;
    import DependentComponentA
    {
        importImplementation=TRUE;
    };
};
```

Macros

Allows you to substitute parameters from the command line.

Syntax

Use in the build spec: `$(macro-name)`

Definition on the command line: `swbuild -o macro-name=value`

Description

Before the build specification is parsed, macros are evaluated and substituted with values from the command line. Forms that do not correspond to the syntax are not macros, and no substitution takes place. To explicitly escape the `$(` sequence, you can use `$$`. Thus `$(NOTAMACRO)` becomes `$(NOTAMACRO)` after substitution. Because macros are evaluated before parsing, they can be used anywhere:

```
component $(COMPONENT_NAME)
{
    name=$(VALUE);
    $(NAME)=$(VALUE);
    name="Macros can even be in $(STRINGS)";
};
```

An undefined macro evaluates to the name of the macro. For example, if `$(VALUE)` is not defined, it evaluates to `VALUE`, and the Design Center issues a warning.

Example

A typical use of macros is to pass property values from the command line.

```
component dcapi plugin
{
    source=dcapi.soc;
    package dcapi;

    includePath=../include;
    buildType=$(BUILD);
    numParallelCompiles=$(MAXPROCESS);
};
```

The command line associated with the example above would be something like:

```
swbuild -o BUILD=PRODUCTION -o MAXPROCESS=4
```

Warnings

Macros are expanded anywhere, even in comments. This means you can't simply comment out lines containing undefined macros—you must remove the offending macro call.

See also

```
swbuild
```

Properties

Specify options in a build specification.

Syntax

propertyName = *value*;

Description

Each property is a name-value pair that sets compiler build options for the containing component or adapter.

propertyName is the name of a defined build property from the properties table below.

value is a value as defined in the description column of the properties table below.

The following table is a complete list of project and component properties. For adapter properties refer to the appropriate section in the documentation for the relevant adapter.

Property	Type	Default	Description	Applies to
importPath	directory	.	Search paths used for imported components	project, component, component (import, group)
buildPath	directory	.	Directory for generated output.	project, component
buildType	choice	DEVELOPMENT	One of DEVELOPMENT or PRODUCTION build	project, component
cFlags	string		C compiler flags	project, component
ccFlags	string		C++ compiler flags	project, component
debug	choice	FALSE	Enable SWAL debug code generation.	project, component

Chapter 10: Build specification reference

Properties

Property	Type	Default	Description	Applies to
includePath	directory	.	Search paths used for included files	project, component
importImplementation	choice	FALSE	Whether to aggregate the imported component into the one being built.	imported component
ldFlags	string		Linker flags	project, component
libraryPath	directory		Search path for libraries	project, component
libraryPost	string		Libraries linked AFTER runtime libraries	project, component
libraryPre	string		Libraries linked BEFORE runtime libraries	project, component
methodsPerFile	numeric	50	Number of methods generated per file. Min: 10, Max: 100	project, component
name	string		Name of the generated engine	component
numParallelCompiles	numeric	1	Number of parallel compilations, min. 1	project, component
description	string		Description string for this engine	component
engineGroup	string	Applications	ECC engine group for this component	component

Property	Type	Default	Description	Applies to
engineService	string		Add an engine service to this engine.	component
extraArchive-File	file		Extra file to place in engine archive	component

When you use an adapter factory, additional properties specific to that adapter may also be available. See the relevant adapter factory documentation for these properties and the values they take.

Example

```
// a global property will apply to both components
buildPath = some/build/path;
component MyComponentA
{
    // property local to MyComponentA
    source = MyIDLosA.soc
    {
        // property local to MyIDLosA
        includePath = some/include/path;
    };
};
component MyComponentB
{
    // properties local to MyComponentB
    source = MyIDLosB.soc
    name=myWonderful Component
};
```

source

Specifies a source file for a component.

Syntax

`source identifier;`

Description

identifier specifies the filename.

Source files can be added to a component specification at a global or component level. A source file can be an IDLos, action language or header file.

Source files must be listed in the correct order of dependency. It is best to list all source files, including .act files, in the specification instead of as a `#include` in the IDLos file. This will optimize Design Center server reloading.

Properties may be associated with a source file, as follows:

- `includePath`
- `libraryPath`

Warnings

None.

Example

```
component MyComponent
{
    source /some/source/path/MyIDLos.soc;
    source /some/source/path/MyAL.act;
};
```


swbuild

Build a component from the command line.

Syntax

```
swbuild { -a } { -o macro-name=value } {specification-file}
```

Description

The `-a` option suppresses building; the Design Center will perform an audit only.

The `-o` option allows you to set options for macros defined in the build specification.

The *specification-file* indicates the source file for the build specification. If you omit this argument, the Design Center reads the build specification from standard input.

Example

```
swbuild mybuildspec.osc
```

See also

Macros

Complete build grammar

```
componentSpecification :  
    ( componentStatement | propertyStatement )* end-of-file  
  
componentStatement :  
    component identifier ( implementation identifier )*  
    { componentBlock } ;  
  
componentBlock :  
    { ( sourceStatement | importStatement | propertyStatement |  
      adapterStatement | modelrefStatement | includeStatement )* }  
  
importStatement :  
    import identifier { propertyBlock } ;  
  
sourceStatement :  
    source valueToken { propertyBlock } ;  
  
propertyBlock :  
    { ( propertyStatement )* }  
  
propertyStatement :  
    nameToken = { valueToken } ;  
  
nameToken :  
    identifier  
  
valueToken :  
    compoundQuotedLiteral | unquotedLiteral  
  
adapterStatement :  
    adapter identifier { adapterBlock } ;  
  
adapterBlock :  
    { ( propertyStatement | groupStatement | modelrefStatement )* }  
  
groupStatement :  
    group identifier { groupBlock } ;
```

```
groupBlock :  
    { ( propertyStatement | modelrefStatement )* }  
  
modelrefStatement :  
    model-reference scoped_name { propertyBlock } ;  
  
scoped_name :  
    { :: } identifier ( :: identifier )*  
  
unquotedLiteral :  
    literal-content  
  
compoundQuotedLiteral :  
    quotedLiteral ( quotedLiteral )*  
  
identifier :  
    [a-zA-Z][a-zA-Z0-9_]*  
  
includeStatement :  
    include ( < | " ) filename ( > | " )
```

Complete build specification example

```
/*
 * Global properties. These appear outside of any component
 * block, and apply to all components.
 *
 * The following tests global properties and tries to trick
 * our parser with macros and comments.
 */

global Name1=global Value1;
global $(NAME)2=global $(VALUE)2;
$(GLOBALNAME3)=$(GLOBALVALUE3);

/*comment*/global $(NAME)4/*$(comment)*/=global $(VALUE)4;

// Empty values are allowed. All three of the following are empty
values:
emptyValue="";
emptyValue=;
emptyValue=$(EMPTYVALUE);

// A component statement describes a component to be built.
// In the plugin service project tree, this maps to ElementEngine

component Component1
{

local Name1=local Value1;
local $(NAME)2=local $(VALUE)2;
$(LOCALNAME3)=$(LOCALVALUE3);

quotedValue = "
```

What you see is what you get... almost.

Escape sequences in this string follow the same rules as IDLs.

So, you can \"escape a string\"
insert a tab \t, etc.

Check out unquoted literals if you are specifying file

```
paths.

";

escapedString="Escaped quote:\" Newline:\n Escaped backslash:\\";

quotedCompoundValue =
    "Notice how strings are allowed to "
    "continue, just like in C++;";

quotedMacroValue = "quoted$(VALUE)1";

// Note how unquoted literals don't have any escape sequences.
// This makes file names on NT easy.

pathValue1=/some/file;
pathValue2=C:\some\file;

escapedMacro1 = $$ (NOTAMACRO);
escapedMacro2 = "$$(NOTAMACRO)";
escapedMacro3 = "This is $$ (NOTAMACRO), OK?";

// The source statement indicates a source file to be
// loaded into the DC server. Normally source files
// do not need properties. However, if you are using
// #include in your source file, you may wish to define
// an includePath which only applies to your file. Be
// aware that using #include is a bad idea. You should
// list all your files here (both IDLs and Action Language)
// because it allows the DC to optimize reloads.

source MyIDLSourceFile.soc
{
    includePath = /some/path/name;
};

source MyActionLanguageFile.act;

// The import statement establishes a dependency to
// another component. The DCAPI plugin will search for
// this component using includePath. You can specify
// additional includePath, libraryPost, etc, parameters
// for this component only by using a property block.

import ImportedComponent
{
    includePath=/this/is/where/the/component/is;
    libraryPost=someLibraryThisComponentNeeds;
};
```

Chapter 10: Build specification reference

Complete build specification example

```
// You must list the packages that will be
// implemented in your component. If leading
// "::-" is left off, it is implicit. You can also
// assign properties to model references.

package Package1;
package ::Package2
{
    someproperty=somevalue;
};

// You may use the services of an adapter to
// bind parts of your model to a particular
// implementation. It is up to each adapter as
// to what sort of model references and properties
// they allow

adapter Sybase
{
    entity Package1::PersistentEntity
    {
        readString="Some SQL string";
        writeString="Some SQL string";
    };
};

// You can have more than one adapter, of course.

adapter CORBA
{
    interface /*comment*/
```

```
MySecondPackage : MyPublicInterface;
};

// SNMP used plugin service "folders" to group
// configuration information. We need a way to
// preserve this. Essentially, this adds hierarchical
// properties to the DC. We're still using ElemFolder
// in the plugin service for this. We just call them
// "groups" here.

adapter SNMPAgent
{
    group OID
    {
        snmpPackageOID=1.2.3.4.5.6;
        interface Package1::Interface1;
        interface Package2::Interface2;
        module Package1::Module1;
    };
};

// You can build multiple components in the same
// spec. Componets can also have different implementations

component Component2 implementation java
{

    // blah, blah

};

// It should be possible to completely parameterize
// a build

component $(Component3)
{
    package $(Package3);
};
```



Each PHP extension function call appears in this section on its own page with a top-level syntax definition, a description of the function and an example.

About the notation

The syntax description for each statement or function uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

i>

plain text indicates a feature of the notation, as follows:

(*item*) parentheses group items together

(*item*) * the item in parentheses appears zero or more times

(*item*) + the item in parentheses appears one or more times

{ *item* } items in braces are optional

(*item1* | *item2*) the vertical bar indicates either *item1* or *item2* appears

For example:

```
keyword { optionalKeyword }
    { optionalThing } ( zeroOrMoreRepeatingThing ) *
    ( oneAlternative | otherAlternative | thirdAlternative )
```

os_connect

Creates a new connection to ObjectSwitch osw engine.

Syntax

```
$conn_id = os_connect ($host, $port);  
$conn_id = os_connect ($host);  
$conn_id = os_connect ( );
```

Description

This function creates a connection between the PHP process and the *osw* engine.

conn_id is a PHP variable.

host and *port* are optional parameters that locate the *osw* engine.

If *host* and *port* are not set they will default to the *host* and *port* specified in the `php.ini` file.

Warnings

None

Error Conditions

- Not able to connect to ObjectSwitch server

os_create

Creates an ObjectSwitch object and returns its reference.

Syntax

```
$objr = os_create ($conn_id, $scopedName, $attrList);
$objr = os_create ($conn_id, $scopedName);
```

Description

This function creates an object and returns its reference in the variable *objr*. If the object has attributes, an optional *attrList* could be passed in to set the attribute initial values.

When the *scopedName* type is a singleton, this call acts like the *create singleton* in action language. It creates the singleton if it does not exist. Otherwise, it returns the object reference.

conn_id is the value returned by `os_connect()`.

objr is the return value.

scopedName is a fully scoped ObjectSwitch interface name.

attrList is an optional associative array of attribute names and values.

Example

```
//
// create a sales person with "name" initialized
//
$salesType = "myPackage::Salesman";
$salesAttrArray = array('name' => 'Slick Willy');
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);

//
// create a customer - without attribute array
//
$custType = "myPackage::Customer";
$custHandle = os_create($conn_id, $custType);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid scoped name
- Invalid attribute name
- Unsupported type
- No create access
- Duplicate Key

os_delete

Deletes an ObjectSwitch object.

Syntax

```
os_delete($conn_id, $objrList);  
os_delete($conn_id, $objr);
```

Description

The second parameter may be either a single object reference or an array containing a list of object references.

conn_id is the value returned by `os_connect()`.

objrList is an array of valid object instances to delete.

objr is a valid object instances to delete.

Warnings

None.

Example

```
//  
// Delete all Orders  
//  
$sn = "mypackage::Order";  
$orderList = os_extent($conn_id, $sn);  
os_delete($conn_id, $orderList);  
  
//  
// delete a single object  
//  
$objr = os_create($conn_id, $sn);  
os_delete($conn_id, $objr);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- No delete access

os_disconnect

Disconnect from the ObjectSwitch engine.

Syntax

```
os_disconnect ($conn_id);
```

Description

conn_id is the return value from a call to `os_connect()`.

Warnings

None.

Example

```
/* close a connection to the osw engine */  
os_disconnect($xconn);
```

os_extent

Retrieve the extent of object handles of a given type.

Syntax

```
$objrList = os_extent($conn_id, $scopedName, $attrList);  
$objrList = os_extent($conn_id, $scopedName);
```

Description

This function will select objects of a given type. If *attrList* is provided, it contains a list of name-value pairs that are *AND*ed together as a where clause to filter the number of object handles being returned.

If the objects are keyed and *attrList* has key coverage, a keyed lookup will be performed.

conn_id is the value returned by `os_connect()`.

objrList is the return value and is a PHP array of scalar values.

scopedName is a string containing the fully scoped name of a type.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//  
// get all customers  
//  
$sn = "myPackage::Customer";  
$custList = os_extent($conn_id, $sn);  
  
//  
// return all customers in California  
//  
$sn = "myPackage::Customer";  
$whereClause = array('state' => 'CA');  
$custList = os_extent($conn_id, $sn, $whereClause);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid scoped name
- Invalid attribute name
- Unsupported type or bad value
- No extent access

os_get_attr

Retrieves the values of the attributes in an ObjectSwitch object.

Syntax

```
$attrList = os_get_attr($conn_id, $objr, $filter);  
$attrList = os_get_attr($conn_id, $objr);
```

Description

This function retrieves attribute values from an object. If *filter* exists, only the values of those attributes named in the filter array will be returned. Otherwise, all attributes values will be returned.

conn_id is the value returned by `os_connect()`.

attrList is an optional associative array of attribute names and values.

objr is an ObjectSwitch object handle.

filter is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//  
// get all attributes of a cusotmer  
//  
$attrs = os_get_attr($conn_id, $custHandle);  
for ( reset($attrs); $name = key($attrs); next($attrs) )  
{  
    $avalue = $attrs[$name];  
    print "$name = $value\n";  
}  
  
//  
// get the state attribute only  
//  
$filter = array ( "state" => "" );  
$attrs = os_get_attr($conn_id, $custHandle, $filter);  
$state = $attrs["state"];  
print "This customer is in state of $state\n";
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid attribute name

os_invoke

Invoke an operation on an ObjectSwitch object.

Syntax

```
$returnValue = os_invoke($conn_id, $objr, $opName, $param, $ex);  
$returnValue = os_invoke($conn_id, $objr, $opName, $param);  
$returnValue = os_invoke($conn_id, $objr, $opName);
```

Description

This function is used to invoke an operation on an object. The *returnValue* is not required on *void* operations. If a parameter is an *inout* or *out* parameter the value of that array element in *param* will be modified with the result parameter value after a call. If the operation raises user defined exception, *\$ex* will contain the name of the exception type as a string upon return.

conn_id is the value returned by `os_connect()`.

returnValue is the return value of the operation upon completion.

objr is a valid object instance handle.

opName is the name of an operation.

param is a nested associative array of parameter name and value pairs.

ex is the name of user exception thrown by the operation.

os_invoke does not work with *in* or *inout* parameters of sequence type.



Example

```
//
// call a void operation that does not have any params
//
os_invoke($conn_id, $objr, "runtest");

//
// call an operation with parameters that returns a boolean
//
$params = array ("name" => "Smith", "number" => 5);
$ret = os_invoke($conn_id, $objr, "register", $params);
if ($ret)
{
    print "OK\n";
}
else
{
    print "register failed\n";
}
```

When an operation raises user defined exceptions, `os_invoke()` can get the exception type, but not the exception data if it has member fields.

```
//
// operation with user defined exceptions
//
$userex = "";
$params = array();
os_invoke($conn_id, $objr, "myOp", $params, $userex);
if ($userex != "")
{
    print "Caught user exception $userex\n";
}
```

Array types can be used as *in*, *out*, *inout* parameters and return values. Sequence types can be used as *out* parameter or return values.

```
//
// array or sequence type as out param
//
$params = array ( "myList" => array() );
$ret = os_invoke($conn_id, $objr, "getList", $params);
$list = $params["myList"];
for ($i = 0; $i < count($list); $i++)
{
    $value = $list[$i];
    print "list[ $i ] = $value\n";
}
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid operation name
- Invalid parameter name
- Unsupported type or bad value
- Application exception

os_relate

Relates two interfaces.

Syntax

```
os_relate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function relates two objects using the relationship role *roleName*.

conn_id is the value returned by `os_connect()`.

fromObjr is a valid object reference handle.

roleName is the name of a role in a relationship between the “from” and “to” objects.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$salesType = "myPackage::Salesman";
$salesAttrArray = array('name' => 'Slick Willy');
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);

$custType = "myPackage::Customer";
$custAttrArray = array('name' => 'Lucent', 'state' => 'NJ');
$custHandle = os_create($conn_id, $custType, $custAttrArray);
os_relate($conn_id, $salesHandle, "hasCust", $custHandle);
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid role name

os_role

Retrieves an array of handles by navigating across a relationship.

Syntax

```
$objrList = os_role($conn_id, $objr, $roleNam, $attrList);  
$objrList = os_role($conn_id, $objr, $roleNam);
```

Description

This function returns an array of object references as it navigates across the appropriate relationship. The name-value pairs in *attrList* are *AND*ed together to act as a *where* clause to filter the number of object handles being returned.

conn_id is the value returned by `os_connect()`.

objrList is a list of object references.

objr is an ObjectSwitch object handle.

roleName is a relationship role name.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
/*  
** assume Salesperson is related 1:M with customer  
** and that $salesHandle is already populated with a  
** valid Salesperson.  
*/  
$cList = os_role($conn_id, $salesHandle, "sellsto");  
for ( reset($cList); $cust = current($cList); next($cList))  
{  
    /* do something with $cust */  
}
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid role name
- Invalid attribute name
- Unsupported type or bad value

os_set_attr

Sets a value in an attribute of an ObjectSwitch object.

Syntax

```
os_set_attr($conn_id, $objr, $attrList);  
os_set_attr($conn_id, $objr, $name, $value);
```

Description

This function sets a value of an object attribute. More than one attribute value can be set in an object.

conn_id is the value returned by `os_connect()`.

objr is an ObjectSwitch object handle.

attrList is an optional associative array of attribute names and values.



setting uninitialized attributes of sequence type does not work. However, there is a work around using pre-set triggers.

Example

```
//  
// set the name of a customer  
//  
$attrArray = array('name' => 'John');  
os_set_attr($conn_id, $objr, $attrArray);  
  
//  
// or using  
//  
os_set_attr($conn_id, $objr, "name", "John");  
  
//  
// set array attribute  
//  
$value = array (1, 2, 3, 4, 5);  
$attrArray = array ("longList" => $value);  
os_set_attr($conn_id, $objr, $attrArray);
```

Using pre-set trigger to set sequence attributes This section describes the workaround that lets you set sequence attributes from PHP.

```
//
// set sequence attribute with pre-set trigger
// this is what needs to be done in the model.
//
package Example
{
    typedef sequence<string>StringList;

    interface Complex
    {
        attribute StringList t_stringlist;
    };

    entity ComplexImpl
    {
        attribute StringList_stringlist;

        void t_stringlist_init();
        trigger t_stringlist_init upon pre-set t_stringlist;
    };

    expose entity ComplexImpl with interface Complex;
};

action :: complexTest :: IComplexImpl :: t_stringlist_init
{
    declare StringList I;

    disableTriggers();
    self.t_stringlist = I;
    enableTriggers();
};
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid attribute name
- Attribute is readonly
- Unsupported type or bad value
- Duplicate Key

os_unrelate

Unrelates two objects.

Syntax

```
os_unrelate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function unrelates two objects that are currently related using the relationship role *roleName*.

conn_id is the value returned by `os_connect()`.

fromObjr is a valid object reference handle.

roleName is a relationship role name.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$custList = os_role($conn_id, $salesHandle, ":sellsto");  
for (reset($custList); $cust = current($custList); next($custList))  
{  
    os_unrelate($conn_id, $salesHandle, 'hasCust', $cust);  
}
```

Error Conditions

- Not able to connect to ObjectSwitch server
- Invalid handle
- Invalid role name

Index

Page numbers in this index are prefixed with letters, for example AG:21-24, to show which volume of the user documentation contains that entry.

DG = Creating ObjectSwitch Applications
SY = Advanced ObjectSwitch Modeling

AG = Managing ObjectSwitch Applications

A

- abort transaction action language statement 223, 229–230
- abstract IDLos property and interfaces 89
- action IDLos statement 127–127
 - implements operation 51
- action language 165–??, 185–242
 - arithmetic operators 171
 - declarations 167
 - enclosed in IDLos action statement 127
 - instant example 166
 - keywords 167
 - parameters 180
 - variable names must not match parameter names 168
 - variables 167
- actions 127–127
 - execution of 19–21
 - implementing states and operations 96
 - scope of 169
- adapters
 - adding attributes to 26
 - and interfaces 85
- Application Server
 - configuring 63–69
 - See also* nodes
- applications
 - partitioning 13
- arithmetic
 - operators 171
- attribute IDLos statement 47, 128–128
- attributes 128–128
 - defined 5, 1
 - Design Center icon for 16
 - in IDLos 47
 - initial values 192
 - as keys 138–139

- read-only 84
 - setting properties for 26
 - triggers on 153
- audit ??–29

B

- begin transaction action language statement 223, 229–230
- binary operators 171
- boolean operators 171
- branching *See if*
- break action language statement 188
- building
 - projects 27–30
- built-in functions 7–??

C

- C++
 - including header files 167
 - interfacing to 12–13
- callbacks
 - using abstract interfaces for 89
- cardinality action language operator 189
- catch action language statement 201–204
- chainSpec 186
- commands
 - regedit 51
 - swadmi ntool 11
 - swcoord 7
 - swdc -c cl obber 60
 - swdc -c forcestop 60
 - swmon 27
 - swregi stry 53
- commit transaction action language statement 223, 229–230
- components *See* packages
- configuring
 - engines 61

Index

- hash table sizes 69
- ObjectSwitch nodes 63–69
- registry variables 61–69
- System Coordinator 66–69
- continue action language statement 191
- control flow *See if; loops*
- create action language statement 192–195
 - initial state of object 95
- creating
 - objects 192–195
 - objects, initial state of 95
 - singletons 194–195
- D**
- data members *See* attributes
- data types
 - action language 231–232
 - in action language 176
 - C++ and equivalent ObjectSwitch types 13
 - IDLos 34
 - inheritance of "shadow" types 102
 - scoping 176
 - user-defined 155–155
 - user-defined types, exposing 85
- database adapters
 - defining transient attributes in persistent objects 26
- date functions 11
- deadlocks
 - automatic detection of 23
 - explicitly managing deadlocks in spawned threads 224
- debugging
 - See also* trace files
- declarations 196–197
 - forward 115, 132
 - order of 115
- declare action language statement 196–197
- delete action language statement 198
 - See also* finished
- deleting
 - events sent to deleted objects 18
 - objects 198
- deployment directory
 - of engines 21
- Design Center ??–30
 - administration 57–60
 - erasing repository 60

- icons 16
- internal details 57–??
- setting up new users 57–59
- destroying
 - objects 198
- destroying objects
 - C++ code, memory deallocation in 13
- distribution
 - creating objects on remote nodes 192
 - location code used in object reference 33
 - strategies 13

E

- else clause in action language *See if*
- empty action language operator 199
- Engine Control Center 3, 11–25
 - starting 11
- engines
 - adding to nodes 20
 - configuring 16, 61
 - defined 5, 2
 - deployment directory of 21
 - removing from nodes 21
 - started by coordinator 6
 - starting and stopping 15
 - viewing status of 12
- entities 132–133
 - associative 65, 147
 - defined 5, 2
 - Design Center icon for 16
 - exposing 135–135
 - exposing with interfaces 82–87
 - in IDLos 39–96
 - local 45–60, 140–141
 - as namespaces 113
 - states 77–96
 - triggers on 66, 153
- entity IDLos statement 39–96, 132–133, 140–141
- enum IDLos statement 131–131
- enumerations 131–131
- environment variables
 - PATH 58
 - SW_HOME 57
- event bus
 - local operations not dispatched via 55
 - operations in local entities not dispatched via 45
- events
 - asynchronous 20

- defined 5, 2
- processing of 17–18
- synchronous 20
- to deleted objects 18
- See also* signals
- excepti on IDLos statement 134–134
- exceptions 134–134, 201–204
 - example 202
 - rai ses IDLos keyword 51
 - system exceptions 201–??
 - See also* throw; catch; try
- expose IDLos statement 135–135
- extentl ess IDLos property 43
- extents
 - defining extentless entities 43
 - determining number of objects in 189
 - selecting from 218

F

- filters
 - trace messages 4
- fi ni shed 97
- for action language statement 206
 - using break to exit 188
 - using conti nue to skip iterations 191
- for. . . in action language statement 208
- functions
 - built-in 7–??
 - member *See* operations 7

G

- generalization *See* inheritance

H

- handle *See* object reference
- hash tables 69

I

- icons
 - Design Center 16
- IDL
 - and IDLos syntax 34
- IDLos 31–164
 - attributes 47
 - correspondence to UML 32
 - entities 39–96
 - keys in 60
 - long example 117–122

- signals 97
- strings, built-in operations on 173
- triggers 153–154
- types 34
- i f action language statement 210–212
- importing
 - models ??–16
- i n
 - parameter 50
 - See also* for. . . in
- include path
 - for IDLos includes 16
- inheritance 92–110
 - of interfaces 110
 - and operations 103
 - restrictions 99
- initialization
 - automatically invoking operations for 57
 - coordinator starts engines 6
 - coordinator's role in 5
- i ni ti a l i z e IDLos property
 - order of invocation 16
- i ni ti a l i z e IDLos property 57
- i nout parameter 50
- i nterface IDLos statement 82–87, 136–137
- interfaces 136–137
 - abstract 89
 - and adapters 85
 - cross-package inheritance 112
 - defined 5, 2
 - Design Center icon for 16
 - exposing entities 135–135
 - IDL syntax identical to IDLos 34
 - inheritance 110
 - instantiating 83
 - as namespaces 113
 - and packages 78, 82–87
 - and relationships 85
 - relationships between 136
- iteration *See* loops

K

- key IDLos statement 60, 138–139
- keys 138–139
 - Design Center icon for 16
- keywords
 - action language 167

Index

L

- libraries
 - using 12–13
- local entities 45–60, 140–141
 - and interfaces 89
- local operations 55
- location codes
 - used in object references 33
- locks
 - deadlocks 23
 - transactions 21
- loops
 - for 206
 - for . . . i n 208
 - whi l e 235
- l oops
 - using break to exit 188
 - using conti nue to skip iterations 191

M

- member functions *See* operations
- members *See* attributes; operations
- messages *See* signals
- models
 - importing ??–16
 - See also* IDLos
- modul e IDLos statement 142–142
- modules 113, 142–142

N

- names
 - namespaces 112–116
 - resolution of partially scoped names 115
 - scope of 112–116
 - scoped 113
- namespaces 112–116
 - different model elements and their namespaces 113
 - packages as 112, 113
 - scoping names in 113
- navigating
 - relationships 186, 218
- nodes
 - configuring 63–69
 - Design Center node 57
- notifiers
 - example 117–122
 - using abstract interfaces for 89

O

- object references 170
 - empty 199
 - OID 33
 - sel f 222
- objects
 - creating 192–195
 - defined 6, 2
 - deleted, events sent to 18
 - destroying 198
 - destroying, memory deallocation in C++ code 13
 - empty reference to 199
 - reference to self 222
 - See also* object references; entities
- ObjectSwitch
 - languages *See* IDLos, action language
- ObjectSwitch modeling
 - overview 32
- ObjectSwitch Monitor 3
 - starting 27
 - user interface 29
 - viewing objects in shared memory 34
- OID
 - object references 33
- operati on IDLos statement 143
- operations 143
 - defined 6, 2
 - Design Center icon for 16
 - inherited 103
 - i ni ti a l i ze 57
 - local 55
 - recovery 57
 - returning values from 216
 - termi nate 57
 - virtual 106–110
- operators 171
 - boolean 171
 - precedence 171
 - unary and binary 171
- os 265, 265, 297, 297
- os_connect php extensions 252, 252, 252, 284, 284, 284
- os_createphp extensions 253, 253, 285, 285
- os_del etephp extensions 255, 255, 287, 287
- os_di sconnect php extensions 256, 288
- os_exten t php extensions 257, 257, 289, 289
- os_get_attr php extensions 259, 259, 291, 291
- os_i nvoke php extensions 261, 261, 261, 293, 293,

293
 os_relate php extensions 264, 296
 os_role php extensions 265, 265, 297, 297
 os_set_attr php extensions 267, 299
 os_unrelate php extensions 269, 302
 out parameter 50

P

package IDLos statement 145–146
 packages 145–146
 allocating to components 19
 allocating to engines 20
 as namespaces 112, 113
 as organizing components 34
 Design Center icon for 16
 and interfaces 82–87
 user-defined types in 176
 parameters
 accessing in action language 169
 in action language 180
 in, out, inout 50
 named vs. positional 180
 names must not match variable names 168
 partitioning
 strategies 13
 PATH *See* environment variables
 performance
 and transaction size 23
 polymorphism 106–110
 projects
 building 27–30

R

raises IDLos keyword 51
 read-only attributes 84
 recovery 6
 automatically invoking operations for 57
 engine restart timing, configuring 67
 local entities not recoverable 140
 recovery IDLos property
 order of invocation 16
 recovery IDLos property 57
 regedit *See* commands; registry, editor
 registry
 contents of 61–69
 default 5
 editor ??–53
 engine configuration 61

graphical vs. command-line editing 3
 modifying from command line 51–55
 used by system coordinator 5
 variables 61–69
 viewing 52
 relate action language statement 214–215
 relationship IDLos statement 147–148
 relationships 62–73, 147–148
 associative 65
 between interfaces 136
 Design Center icon for 16
 determining number of related objects 189
 exposing with interfaces 85
 initialization on object creation 192
 navigating 186, 218
 relating objects with 214–215, 233
 roles 62
 selecting across 218
 See also relate
 repository
 erasing 60
 return action language statement 216
 role IDLos statement 147–148
 roles 62, 147–148
 Design Center icon for 16
 triggers on 153

S

scope 112–116
 of an action 169
 SDL wrapping 33
 select action language statement 218
 selecting
 objects, extents, and singletons 218
 self action language keyword 222
 in const operations 50
 shared memory
 configuring size of 65
 object references & type IDs 33
 specifying access permissions for 65
 signal IDLos statement 97, 149–149
 signals 149–149
 accessing parameters in action language 169
 defined 6, 2
 Design Center icon for 16
 singleton IDLos property 43
 singletons
 creating 194–195
 designating 43

Index

- and interfaces 89
- selecting 218
- spawned thread, example of 224
- state machines 77–96
 - stateset 150
 - subtype cannot override 100
 - transition events using signal 149
 - transitions 152–152
- state transitions 152–152
- states 77–96
 - defining state actions 96
 - transitions 98
- stateset IDLos statement 150
- strings
 - built-in operations 173
- struct 151–151
- structures 151–151
- subtypes
 - defined 6, 6, 2, 2
- supertype/subtype 92–110
- SW_HOME *See* environment variables
- swadmintool *See* commands; Engine Control Center
- swcoord *See* commands; System Coordinator
- swdc *See* commands
- swmon *See* commands; System Monitor
- swregistry *See* commands
- System Coordinator 5–9
 - adding engines 20
 - attaching to 12
 - configuring 66–69
 - configuring engines for 16
 - removing engines 21
 - starting 7
 - using Engine Control Center with 11–25
- system exceptions 201–??

T

- terminate IDLos property
 - order of invocation 16
- terminate IDLos property 57
- termination
 - automatically invoking operations for 57
- threads
 - spawned, example 224
 - spawning new 223–228
- throw action language statement 201–204
- time functions 11

- timers 7–9
- trace files 3
 - configuring tracing 67–68
- transaction action language statements 223, 229–230
- transactions
 - abort transaction 223, 229–230
 - begin transaction 223, 229–230
 - commit transaction 223, 229–230
 - handling deadlocks in spawned threads 224
 - local entities not recoverable 140
 - locks 21
 - managing transactions in spawned threads 229
 - processing 21
 - in spawned threads 223, 223, 223, 223
- transition IDLos statement 98, 152–152
- trigger IDLos statement 70–??, 153–154
- triggers 153–154
 - entity 66
- try action language statement 201–204
- typedef 155–155
- types
 - supertype/subtype, defined 6, 2

U

UML

- instant overview 4, 2
- unary operators 171
- unrelate action language statement 233
- users
 - setting up Design Centers for 57–59
- using IDLos keyword 147

V

- variables 167
 - declaring 196–197
 - names must not match parameter names 168
- virtual IDLos property 57, 106–110

W

- while loops in action language 235
 - using break to exit 188
 - using continue to skip iterations 191