Software through Pictures®

Millennium Edition

Using ACD Templates

UD/UG/ST0000-10149/001

Software through Pictures
ACD Templates
Millennium Edition
January, 2002

Aonix® reserves the right to make changes in the specifications and other information contained in this publication without prior notice. In case of doubt, the reader should consult Aonix to determine whether any such changes have been made. The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

World Headquarters
5040 Shoreham Place, Suite 100
San Diego, CA 92122
Phone: (800) 97-AONIX
Fax: (858) 824-0212
E-mail: info@aonix.com
http://www.aonix.com

Table of Contents

1 Introduction

Generating UML Solutions with Architecture Component Development

Architecture Component Development (ACD) is a feature of StP that provides a more powerful and flexible approach for automatically generating solutions from your StP/UML models. Using ACD, a majority of the implementation code for your application can be generated automatically from your model. ACD implements a template-based approach that allows you to specify the architectural aspects of the system under construction.

Corresponding to OMG's Model Driven Architecture (MDA), ACD is designed to let you model your system from a design perspective, rather than focusing on implementation details that can be handled by the templates.

There are two documents that address ACD in detail:

- *Using ACD Templates* (this document) explains how to use standard ACD templates.
- *ACD Programming Guide* is intended as a reference document which provides information on the implementation and more detailed information on templates.

Intended Audience

The audience for this manual includes analysts and developers who wish to generate Java, Ada 95, or C++ code from StP/UML diagrams.

The information in this chapter assumes you are familiar with:

- The fundamentals of StP, including using diagram editors, the Object Annotation Editor, and annotation templates
- Object-oriented methodology, as documented in *Unified Modeling Language User Guide* (Booch et al., 1997) or *Unified Modeling Language Reference Manual* (Rumbaugh et al., 1997)
- Your operating system directory and file structures
- Conventions of Windows systems
- Target language for generating code

ACD Templates

StP provides a library of standard templates:

- Java (see Chapter 2, “Generating Java Code”)
- Ada 95 (see Chapter 3, “Generating Ada 95 Code”)
- C++ (see Chapter 4, “Generating C++ Code”)
- EJB

In addition to the standard templates, templates for other languages, including C, COM, CORBA, IDL, and Visual Basic are available and can be activated by toggling the appropriate variables in the StP desktop's ToolInfo editor advanced section. These templates typically are associated with a companion white paper, which can be located in *templates/uml/qrl/code_gen/<language>*. Please contact Aonix for more information about these additional templates.

If you need to generate code in a language that is not currently supported by an ACD template, you can create a template so that it supports the programming language of your choice.

How ACD Differs from Traditional Applications

Traditional Code Generators

Traditional code generators give you a one-to-one mapping from the design model to code. For each class that is represented in the model there is one class specified in the code. Hence, more implementation code is needed to complete your system.

So, to get this extra implementation code, you must either represent extra details in your model or input the code manually. This leads to models which are constructed to achieve maximum code generation, rather than accurately representing the business or user requirements in a maintainable way.

The ACD Approach

ACD allows you to design your models according to OMG's MDA approach.

Instead of employing a one-to-one mapping from model to code, the ACD approach allows you to raise the level of abstraction in UML models. These domain-specific models can then be mapped to the target system and ACD templates generate the technical infrastructure. These types of models are much easier to understand and to maintain than traditional UML Models.

Using the ACD approach, you:

- Create a domain-specific model in UML notation
- Apply an ACD template to transform your domain model to source code
- Add additional code to complete the business-logic functionality

Reverse Engineering

Reverse engineering is used to maintain the one-to-one mapping between model and code. With traditional code generators, a majority of the implementation code must be input manually. Often, this code is *reverse engineered* into the model, so that the details in the model accurately reflect the output code.

Reverse engineering is not necessary with ACD because a one-to-one mapping does not exist between model to code. It would only put unneeded details back into the design model; details for which ACD's templates *know* to automatically generate code, even without representation in the design model. For example, in COM, ACD generates IDL files, interfaces, classes, class factories, and proxy/stub code.

For legacy systems with no corresponding UML model, you may use reverse engineering to develop an initial model of your application. This UML model will, however, be a one-to-one mapping of your implementation.

In order to leverage the power of ACD templates, this model then needs to be abstracted. Or in other words, implementation detail needs to be removed from the reverse-engineered model.

Depending on the target architecture, the templates then need to be tailored to generate these code parts. This is an important step towards automation and avoiding implementation-specific code constructs appearing redundantly in your model, which in turn allows much more flexible change management. (That is, a change only needs to be implemented in one place in your templates, rather than many places in your model.)

Note: OMG's MDA does not encourage round-trip engineering.

How ACD Works

Source code can be generated in different levels of granularity:

1. For a complete model:

   In the StP desktop's **Code** menu, call **Generate <language> by ACD template**.

2. For individual diagrams:

   In the StP desktop's right hand pane for class diagrams, select the desired diagram, right-click, and call **Generate <language> by ACD template**.

3. For individual packages or classes:

   In the StP desktop's right hand pane for the Repository View, select the desired package or class, right-click, and call **Generate <language> by ACD template**.

In the property sheet which pops up for these commands, you can change the output directory for the generated code and switch off generation of the IMF file (the latter should always be activated unless you develop templates and know that the UML model has not changed since the last time the IMF file was generated).

Note: Typically, all other parameters which influence the way the code is generated are set from within the templates because they generally do not need to be modified every time code is generated.

2 Generating Java Code

Introduction

This chapter discusses using ACD to generate Java code from models created with StP / UML 8.x. The code produced by ACD from StP / UML models is standards-compliant, compiles out of the box, and features good coding style.

Besides generating standard elements such as code skeletons for classes, interfaces and exceptions with attributes, constructors, and operations, ACD also generates the following:

- Implementation methods for interfaces and abstract classes
- Accessor methods (*setter* and *getter* methods)
- Various constructor types
- Implementation of state machines from state charts
- Standard operations overriding those in *java.lang.Object*

All these additional features are optional and created only when the feature is not in the model. Furthermore, you can generate code selectively for single or multiple:

- Class diagrams
- Class tables
- Packages
- Individual classes

Additional reference information can be found in Appendix A, "Java Notes", which contains notes and discussions on:

- Source code examples
- Structure of the code generator
- Notes and items used in code generation
- Stereotypes and tagged values
- Global variables for code generation

Recommended Environments and Tools

In order to take full advantage of Java code generation, we suggest you install the Java Development Kit (JDK), available from Sun Microsystems Inc.

When building an application, the generator creates *makefiles*. Currently, the Java-based maketool ANT, from the Apache Software Foundation, is supported and can be downloaded from their website.

Compatibility

With Language Specifications

The generated code conforms to the Java Language Specification 2.0, which is downloadable from the Sun Microsystems website.

With Previous Code Generators

In general, the generated code is not compatible with code created by previous versions of the Java code generator. Some of the additional user-defined sections have changed their signature, making the merge process no longer able to merge source code produced by newer versions with code originating from older versions.

General Mechanisms

Before discussing classes and interfaces in greater detail, we will review:

- Model elements that determine code generation
- Invocation of the code generator
- Adding information to enrich the model

Model Elements That Determine Code Generation

While ACD can access other model elements such as Use Cases, Actors, and Requirements, only the elements listed below are relevant in Java code generation:

- Packages
- Classes
- Attributes
- Operations
- Associations
- Association classes
- State machines
- Annotations

Invocation of the Code Generator

The Java code generator is either invoked for the whole model or for individual model elements or files.

For a whole model, choose **Code > Java > Generate Java for Whole Model by ACD Template** from the StP desktop.

For a file or element, choose one or several model elements from either the class diagram view, class table view, package object view, or class objects view. Invoke **Generate Java for <element> by ACD Template** where *<element>* is a class diagram, class table, package, or class.

See Figure 1, which shows how to generate Java code for a single object, in this case a class diagram.

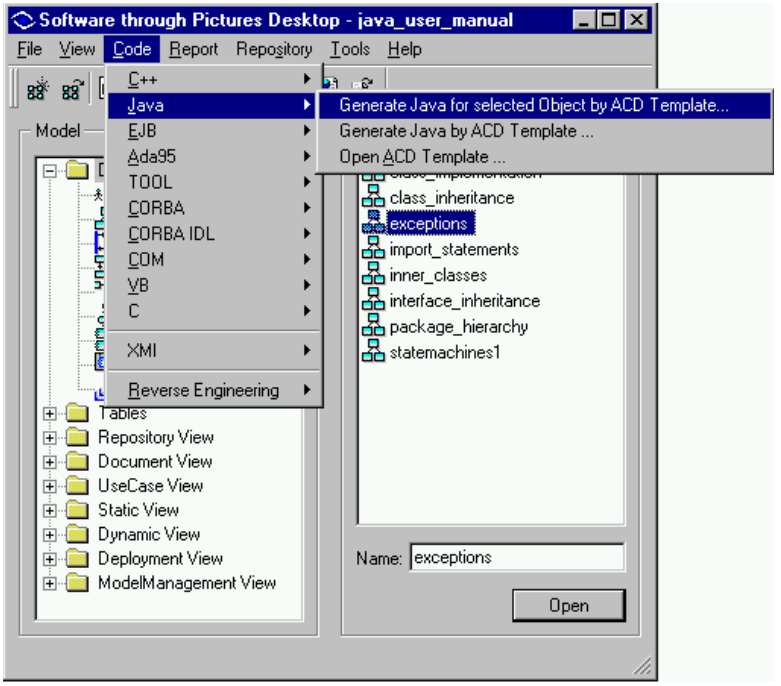


Figure 1: Starting Java Code Generation

Adding Information to Enrich the Model

In StP / UML there are four mechanisms to add information to a model:

- Diagrams (e.g., class and state diagrams)
- Tables (e.g., class and state tables)
- Various property sheets
- Object Annotation Editor (OAE)

Usually you begin with a diagram and enrich your model elements with specific notes and items. For a class, you normally use the class table editor, with its general section (e.g., Analysis Items) and language-specific sections (e.g., Java Items).

For some code constructs, the property sheet for an object is used to enter the appropriate stereotypes and tagged values. Stereotypes and tagged values play an important role in Java code generation. (See "Stereotypes and Tagged Values" on page A-4 for an overview of all used stereotypes and tagged values.)

In rare cases, when there is no table section or property sheet, you may use the OAE to provide the missing information.

Refer to the appropriate chapters in *Creating UML Models* and *Generating and Reengineering Code* for more in-depth discussions.

Basic Code Generation

This section discusses the minimum amount of Java source code generated by ACD to be compliant with the Java language specification. This is also the *factory* default if you have a freshly installed StP/UML.

Exclusion from Generation

Most of the time, you want the generated code to represent exactly what you have modeled. There are situations, however, when you might wish to exclude certain elements from generation; e.g., library packages and classes such as those from the JDK, or packages and classes imported from other models.

For code exclusion, you can mark classes and packages to be excluded in the property sheet for these elements. For packages, this mechanism is recursive. An *excluded* package excludes all subpackages and all contained classes and subpackages from generation. For example, if you imported the whole *java.lang.** and *java.util.** package hierarchy from the JDK, you need only set the package *java* to *excluded*. None of the contained packages and classes are generated.

Visibility

Visibility or access control to classes, attributes, and operations is an important feature for the object-oriented paradigm as a whole. This principle is known as information hiding and ensures that code *from outside* is not dependent on (private) implementations of (public) services of a class.

Standard UML has three possible values for visibility:

- Public
- Protected
- Private

In Java there is an implicit fourth value called *package private*, meaning *no visibility modifier at all*. Only classes in the same package can access identifiers with this visibility.

Common Code Fragments

Certain code artifacts are common to every Java source file, whether they be a class or an interface. Common features are:

- Directory structure into which files are generated
- Basic and important features applicable to both classes and interfaces:
  - File headers
  - Source code comments
  - Package statements
  - Import statements

Directory Structure

For every modeled class named <CLASS>, a Java source file named *<CLASS>.java* is created.

The location of this class file depends on the global TDL variable [srcdir], which you set when starting code generation (See Figure 1 on page 2-4) and the package containment hierarchy of the class.

The variable [srcdir] defaults to *<current_project>/<current_system>/src_files*.

The class Account in will therefore created in directory *<current_project>/<current_system>/src_files/com/aonix/banking*. (See Figure 2 on page 2-8.)

File Headers

A source file may have a preamble or file header, which consists of copyright information and various metainformation such as author and creation date.

For every class / interface, a preamble is generated whose exact content you can influence. Set the global TDL variable `[AUTHOR]` to the generator of the sources; set `[COMPANY]` to your company name.

You can customize the template gen_java_header(MClass) in file *code/java_Common.tdl* for your particular application. For the class Account in Figure 2 on page 2-8, the file header is similar to the following example:

```
/************************************************
* Account.java
* -------------
*
* Author:  YOUR_NAME
* Company: YOUR_COMPANY
* Date:    01/27/02
*
* Copyright (c), 2002, YOUR_COMPANY
************************************************/
```

Comments

A very important feature is the transformation of model element descriptions into Java source code comments. For every class / interface, attribute, or operation in the model, you can extract the description (accessible in StP / UML with <CTRL><T>) and put the description into a source code comment.

This feature can be activated or deactivated by setting the global TDL variables [COMMENT_(CLASSES | ATTRIBUTES | METHODS)]. These variables default to True.

Package and Import Statements

Java source files usually contain exactly one class or interface that normally has the following structure:

- A package statement
- One or more import statements
- The class / interface body

Package Statement

Classes and interfaces are generally modeled inside packages. As long as a class is not modeled as *toplevel* (not contained in any package), a package statement is generated according to the containment hierarchy. The structure shown in Figure 2 generates the package statement:

```
package com.aonix.banking.Account;
```

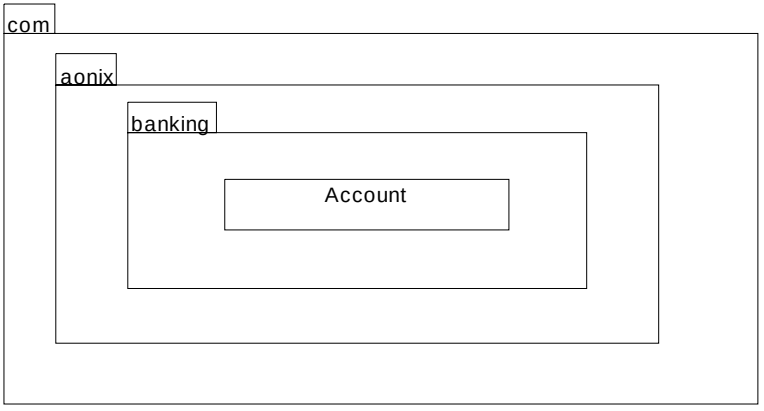


Figure 2: Package Hierarchy

You do not have to model package hierarchies this way. Instead, you can nest package *aonix* inside *com*, and in a second diagram nest package *banking* inside *aonix*.

Import Statement

Every usage of a reference type (class, interface, exception), which is neither the current class / interface nor inside the same package as the current class / interface, results in an import statement for these reference types. Figure 3 on page 2-10 shows some possible import statements:

```
package source.SClass1;
import target.TClass2;
import target2.*;
import attributes.AttribType2;
import operations.OpType2;
public class SClass {...}
```

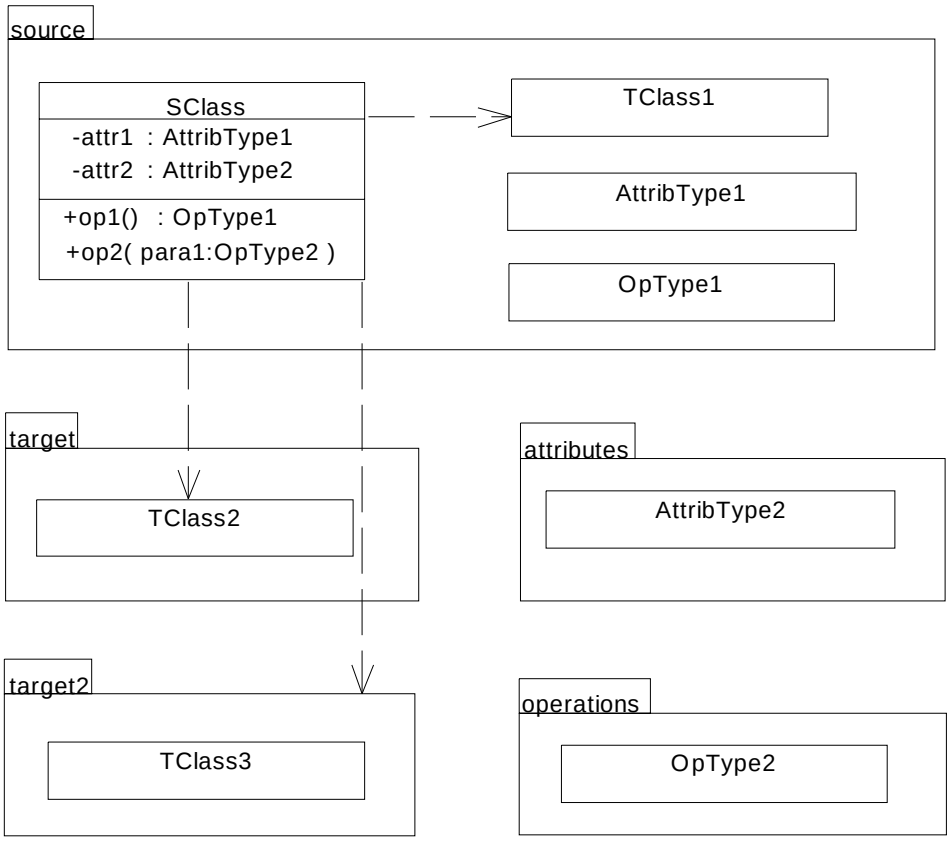


Figure 3: Importing Reference Types

What is shown in Figure 3 is the import of attribute types, parameter types, and types resulting from dependencies.

Other possible import statements may result from generalization and implementation (discussed below). If you draw a dependency to a package instead of a class/interface, you can generate import statements for whole packages. A package wildcard statement (<PACKAGE>.*) is generated in this case.

The classes TClass1, AttribType1, and OpType1 are not imported, because they are located in the same package as SClass. Also not imported are reference types inside operations and constructors, which result from manually inserted source code.

Interfaces

Interfaces describe services that classes implementing these interfaces must expose to the outside world. In Java, interfaces are a core language feature of Java (in contrast to C++, where there is not an extra keyword and where interfaces are simulated with the help of abstract classes and *pure virtual* functions). An interface and all its contained methods are *abstract* by default.

In StP, interfaces are modeled by giving a class the stereotype <<interface>>. Once an interface is defined, an additional presentation option, for interface, the lollipop symbol, is made available. You can use this symbol anywhere the class symbol can be used.

Since nearly all modifiers for an interface are determined by the language specification, there is no need to provide additional information. Interfaces do not need a class table when they contain only methods with a return type of *void*.

General Structure for Interfaces

Following is a description of the general structure of a source file for a Java interface generated by ACD. For details, please refer to "Common Code Fragments" on page 2-6, "Attributes" on page 2-21, and "Operations" on page 2-29.

- General header (copyright and other metainformation)
- Package statement, if the interface is contained in a package
- Import statements, if any reference type outside the current interfaces package is used
- Interface declaration statement, eventually with extends clauses
- Static attributes (constants also), if present
- Modeled operations, if present

- Nested interfaces (which in turn can repeat the structure above recursively)

After the sections for attributes and methods, there is a user-defined section for inserting your own attributes, constructors, and methods.

See Appendix A, "Java Notes," for an example of the structure above.

Inheritance

A Java interface can have generalization relationships because an interface can inherit from more than one superinterface. The superinterface type is imported when it is in another package. The superinterface generates an *extends* statement in the interface declaration. Figure 4 below shows an example for which the following code is generated:

```
package transform3d;
import transform.Shearable;
import transform.Translateable;
import transform.Rotateable;
public interface Rotate3D extends
   Shearable, Translateable, Rotateable {
   ...
   public void rotate3d(float alpha, float beta, float
gamma);
   ...
}
```

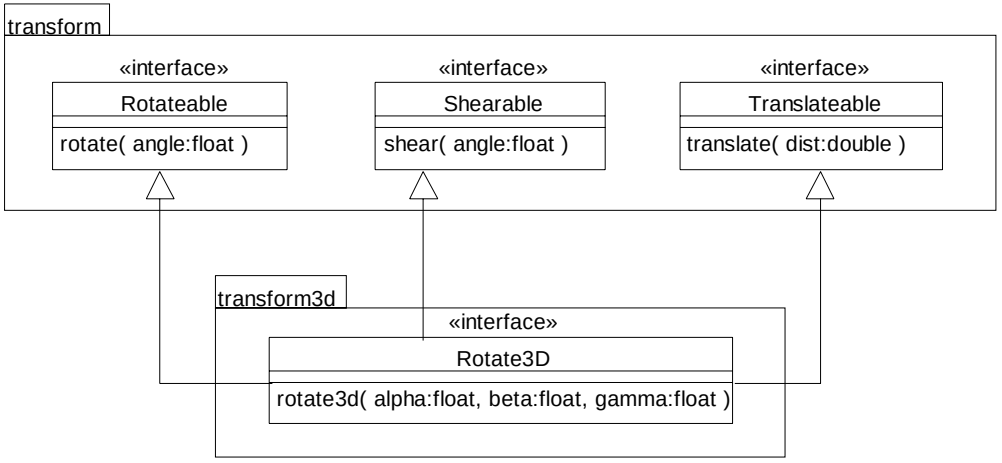


Figure 4: Interface Inheritance

Note: You cannot draw interface generalizations to interface lollipops, only to class symbols.

Classes

General Structure for Classes

Following is a description of the general structure of a source file generated by ACD for a Java class. For details, see "Common Code Fragments" on page 2-6, "Attributes" on page 2-21, and "Operations" on page 2-29.

- General header containing copyright and other metainformation
- Package statement, if the class in contained in a package
- Import statements, if any reference type outside the current class's package is used
- Class declaration statement (eventually with extends and implements clauses)
- Static attributes and constants, if present

- Instance attributes, both modeled and from reference, if present
- Constructors, if modeled or if the creation of artificial ones is enabled
- Modeled operations, if present
- Methods from implemented interfaces:
  - Accessor methods arising from instance attributes, if present and enabled
  - Eventually, finalize(), if the global TDL variable CREATE_FINALIZER is set to True
  - Eventually, standard operations such as toString(), equals(), or clone(), if the global TDL variable CREATE_STD_OPS is set to True
- Nested classes or interfaces, which in turn can recursively repeat the structure
- If the class has a state machine, code for a finite state machine is created, either by nested switch statements or by a pattern of inner classes.

After the sections for attributes, constructors, and methods, there is a user-defined section for inserting your own attributes, constructors, and methods.

If not explicitly modeled, the visibility of a class is determined by the global variable [DEF_CLASS_VISIB], which defaults to *public*. You can assign a visibility different from the class property sheet.

See "Source Code Example" in Appendix A, "Java Notes," for an example of the structure above.

Modifiers for Classes

Java classes can be *abstract*. This means that they cannot be instantiated directly, but rather through non-abstract subclasses. You set abstraction via the property sheet.

The opposite modifier is *final*. Final classes cannot be subclassed and, therefore, they must be non-abstract. You can set this modifier only as an annotation from the OAE. Add the Class Java Definition note and the Final item and set the latter to True. Using the OAE, you can override the visibility set from the property sheet by adding the Visibility item in the Class Java Definition note and setting it to a different value (e.g., *package*

private). See Table 2 on page A-3 in Appendix A, “Java Notes” for a summary of the notes and items for classes, which can be set only via the OAE.

Inheritance

A Java class can have a generalization relationship to at most one superclass. The superclass type is imported, if the type is defined in another package. The superclass generates an *extends* statement in the class declaration. An example of class inheritance is shown in Figure 5. The example generates the following statements:

```
package sub;
import super.SuperClass;
public class SubClass extends SuperClass {...}
```

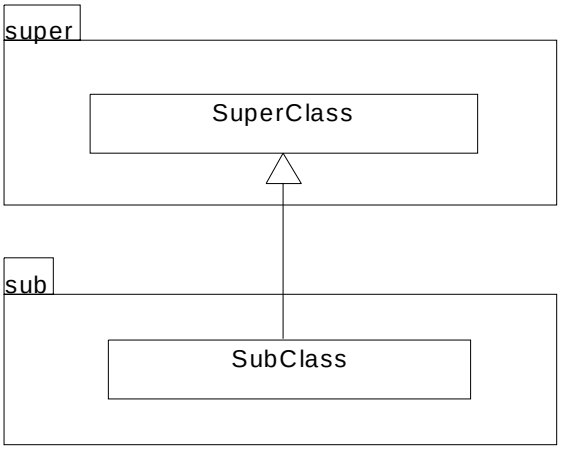


Figure 5: Class Inheritance

Interface Implementation

A Java class can have implementation relationships to one or several interfaces. The interface type is imported if the type is defined in another package. The interface generates an *implements* statement in the class declaration. Figure 6 shows a typical scenario, which creates the following source code:

```
    package application;
    import ifc.Writeable;
    import ifc.Printable;
    public class Editor implements Writeable, Printable{
        ...
}
```

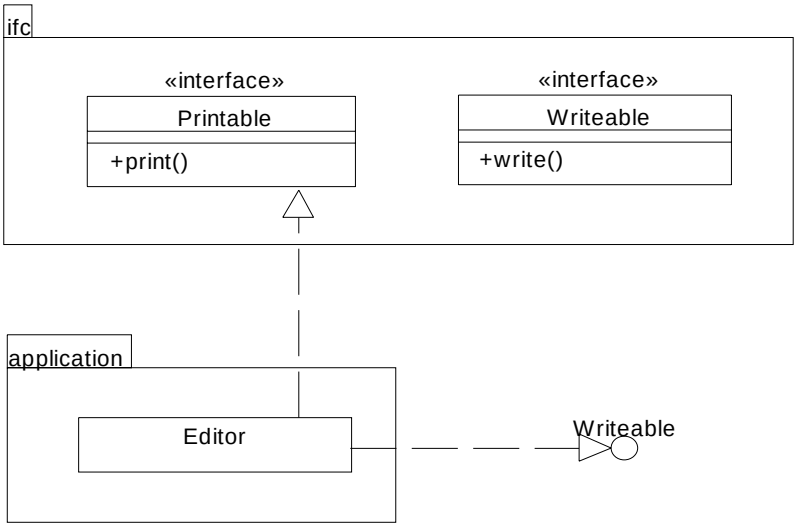


Figure 6: Interface Implementation

As you can see, you implement interfaces by modeling an interface (as class symbol with stereotype interface) and drawing a realization arrow either to the interface itself or to the lollipop representation of that interface.

Implementing Interface Methods

If a class implements one or several interfaces, the interface methods are inserted into the implementing class. The example in Figure 6 generates the following statements:

```
public class Editor implements Writeable, Printable {
    ...
    public void write() {
        ...
    }

    public void print() {
        ...
    }
```

```
}
```

Note: If the interfaces inherit superinterfaces, the methods of those superinterfaces are implemented recursively.

Inner Classes/Interfaces

In Java, classes are normally scoped to packages or are *toplevel*. However, there are situations where a class (in rare instances, an interface) is exclusively used inside another class. In this case, a class can be declared inside another class as a nested class. To be a nested class of some outer class, the Enclosing Scope item within the Class Definition note must be set to the name of the outer class. You set this item within the OAE for the nested class. Figure 7 shows a nesting hierarchy that results in the following class declarations:

```
package nested;
public class Outer {
  ...
    public class Middle {
        ...
        public class Inner {
            ...
        }
    }
}
```

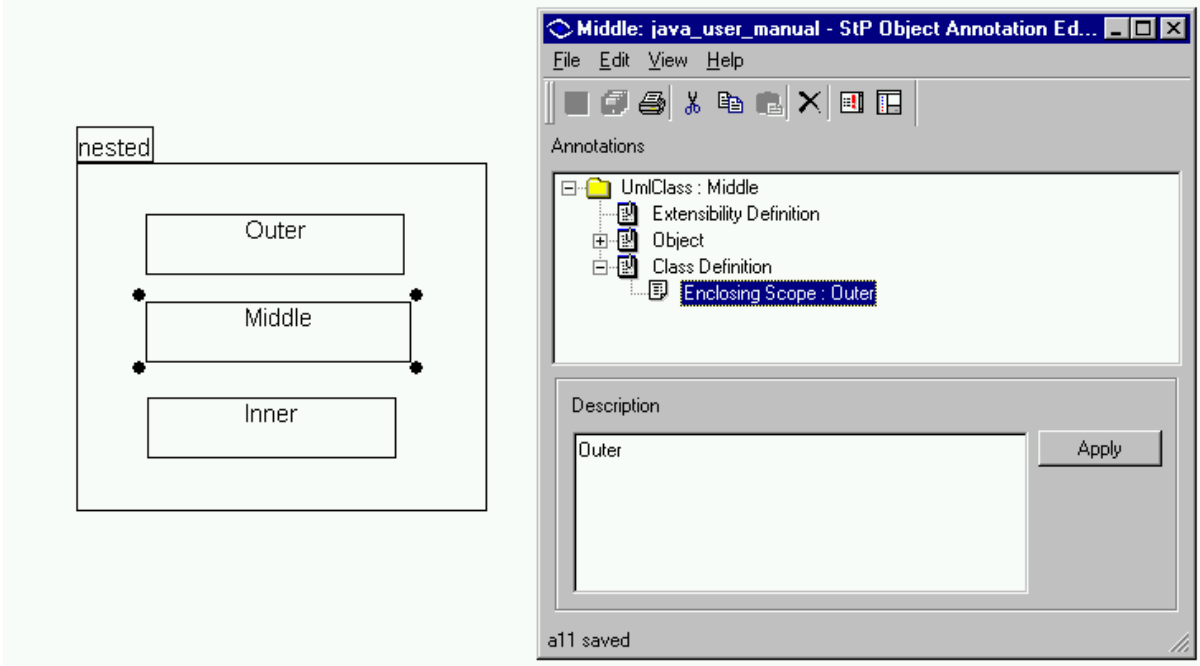


Figure 7: Inner Classes

Note: If you want to create static inner classes, you should annotate the particular class with the Static Class item in the Class Java Definitions note, which is reachable via the OAE, and set it to True.

Exceptions

Exceptions are used in Java to indicate an unusual situation or error condition that is not handled in the normal flow of control. In StP, Java exceptions are modeled as a class with stereotype <<exception>>. See Figure 8 below.

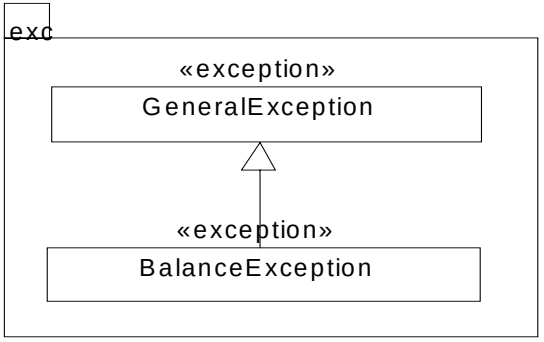


Figure 8: Exceptions

Although exceptions are normal Java classes and create exactly the same code, there are deviations from that rule:

- Exceptions do not create code for nested classes.
- Exceptions do not create code for state machines.
- Exceptions automatically create an additional constructor for the message argument.
- Exceptions are either a direct or indirect subclass of java.lang.Throwable.

Modifiers for Exceptions

Exceptions generally have the same modifiers as normal classes. Refer to "Modifiers for Classes" on page 2-14 for a complete discussion.

Additional Constructor

Every exception should have an additional constructor that contains the detailed message explaining the exception. The generator automatically creates this constructor for you. For example, the following code is inserted for BalanceException, depicted in Figure 8 above.

```
package exc;
public class BalanceException extends GeneralException {
```

```
    public BalanceException() { ... }
    public BalanceException(String message) {
        super(message);
    }
}
```

Exception Inheritance

Every exception in Java must be either a subclass of java.lang.Exception or java.lang.RuntimeException. Exceptions of the latter kind do not need to be caught in try-catch clauses.

You can manage your exceptions in the usual way by an inheritance relationship in the model. Alternatively, you can change the value of the global TDL variable [EXCEPTION_BASE_CLASS], which defaults to java.lang.RuntimeException. Change the variable, if all your exceptions inherit from a common exception in your model.

Attributes

Attributes describe the static features and the state of a class. In Java, they can be either (1) instance attributes, which belong to exactly one class instance and are exclusively used by this instance, or (2) static class attributes, which are initialized only once for every class. The attributes in a class are sorted by this criteria.

Furthermore, there can be *fixed final* attributes, which are used as constants. Other modifiers describe the visibility, the storage class (volatile), and whether the attribute can be serialized or not (transient).

The type of the attribute is imported when the attribute is a (non-primitive) reference type from the model and is not in the same package as the current class. (Refer to"Package and Import Statements" on page 2-8.) This is even true for arrays, where the *basetype* of the array attribute is imported according to the rules above.

Attributes can have a default value that is initialized in the attribute declaration. It is your responsibility as a modeler not to assign default values for derived attributes or attributes initialized in constructors. If a derived attribute also has a default value, the generator issues the following warning:

```
Warning: attribute <attribute_name> is derived, but has a
default value. This default value will be ignored in
initializations!
```

Note: The constants (e.g., *static final* attributes) *must* have a default value according to the Java Language Specification.

With ACD you can generate three kinds of model elements:

- *Real* attributes from model
- Attributes from associations
- Attributes from association classes

Attributes from Model

All possible declarations for a Java attribute can be expressed in StP. When creating an StP model, you build the attributes in class diagrams and add specific information such as type, visibility, etc., using the class table editor. Please refer to *Creating UML Models* and *Generating and Reengineering Code* for a more detailed overview. Figure 9 shows an example for various modeled attributes.

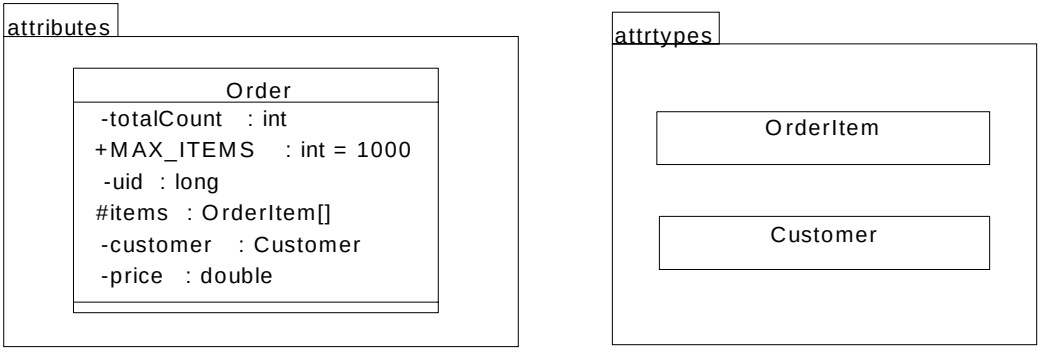


Figure 9: Attributes

Using the Class Table Editor

Use the sections of the class table editor to add detail modeling information to an attribute. For example:

Class member definitions

For every attribute, specify the name, the data type, and a possible default value.

Note: For constants (*final* attributes), you must supply default values.

Analysis items

For every attribute, specify the access privilege and whether or not it is a class attribute. The possible visibility values are the general values for UML: public, protected, private. You can also designate an attribute as *derived* here.

Java implementation items

For every attribute, specify one or more of the following type modifiers: final, transient, or volatile. You can override the visibility of Analysis Items with the visibility column in this section. As an additional value *package private* can appear here.

Table 1 below shows the class table for class Order shown in Figure 9 on page 2-22.

Table 1: Example for Attributes

Order			Analysis Items		Java Items			
Attribute	**Type**	**Default Value**	**Visibility**	**Class Attr?**	**Visibility**	**Final?**	**Transient?**	**Volatile?**
totalCount	int		private	True				
MAX_ITEMS	int	1000	public	True		True		
uid	long		private		package private			
items	OrderItem[]		protected					
customer	Customer		private					True
price	double		private				True	

If an attribute does not have modeled visibility (only possible when a class table for the current class does not exist), the attribute is automatically assigned the value of the global TDL variable [DEF_ATTR_VISIB]. This can be one of the predefined Java visibility values.

For class Order, the following source code is created:

```
package attributes;

import attrtypes.OrderItem;
import attrtypes.Customer;
public class Order {
      // ------------------------------------------------
      // static attributes
      // ------------------------------------------------
      private static int totalCount;
      public static final int MAX_ITEMS = 1000;

      // ------------------------------------------------
      // instance attributes
      // ------------------------------------------------
      long uid;
      protected OrderItem[] items;
      private volatile Customer customer;
      private transient double price;
```

}

Note: There is currently no support for multi-dimensional attribute types. Only one-dimensional (array) types are supported.

Attributes from Associations

The second source for attributes is (binary) associations. Classes, which have a navigable association from the current class to them, are referenced in the current class as an attribute.

In StP you model associations by drawing an association or aggregation link between two classes and filling in the various properties using the property sheet for associations. In StP/UML 8.3, you can alternatively draw a *navigable association*. The navigability can also be changed via the property sheet. Figure 10 below shows an example of various aspects of associations.

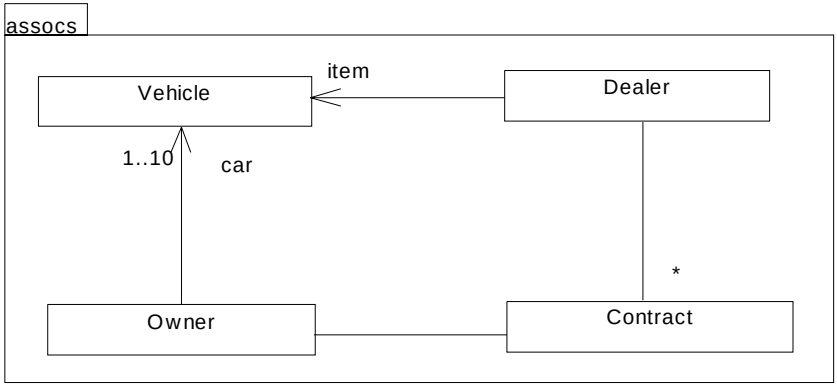


Figure 10: Associations

In the source code, the name of the reference attribute is determined either by the name of the opposite role or, if role names are missing, the name of the opposite class type with the first letter in lower case. The type of the reference attributes is imported if the attributes are outside the package of the referencing class (see "Package and Import Statements" on page 2-8).

If the multiplicity of the opposite role is > 1, the type of the reference attribute changes either to an array with the base type of the opposite class or to a collection class:

- An array type is chosen if the multiplicity is a *range* value.
- The collection class type is chosen if the value is *many*, i.e., the value is *n* or *

The type of the collection class is evaluated from the global TDL variable [DEF_COLL_CLASS], with a default value of Vector. You can influence that for individual roles with the tagged value "collection=<COLLECTION_CLASS>". This tagged value must currently be set from the OAE.

The visibility of the reference attribute is determined by global variable [DEF_ROLE_VISIB], which can be one of the possible Java visibility values. The value of this variable defaults to *protected*, so that subclass can also inherit these attributes. This can be changed for individual role ends by the using the OAE to change the Visibility item in the Role Java Definition note.

The classes shown in Figure 10 above generate the sample source code below. Class Vehicle is omitted because all its associations are directed to it, so it cannot have reference attributes.

```
package assocs;
import java.util.*; // because of „Vector“
public class Dealer {
   ...
   protected Vehicle item;
   protected Vector contract;
   ...
   }

package assocs;
public class Contract {
   ...
   protected Owner owner;
   protected Dealer dealer;
   ...
}

package assocs;
public class Owner {
```

```
    ...
    protected Vehicle[] car;
    protected Contract contract;
    ....
}
```

Note: Currently, no attempt is made to evaluate the *ordered* item of association ends. Qualifiers attached to a role are also not evaluated.

Note: No distinction is made in the source code between associations, aggregations, and compositions. Former generator implementations introduced initialization code for compositions, which was determined to be too specialized and restrictive.

Note: Attributes from associations are neither *static* nor *final*.

Note: If an attribute from an association leads to a name clash with a modeled attribute in the class, the code generator gives preference to the modeled attribute, emitting the following warning:

```
cannot create attribute <Ref_Attr>,because its already
modeled in class <class> manually.
```

Attributes from Association Classes

A third alternative for assigning attributes is the association class. An association class is linked to an association and, therefore, has properties of both model elements. The attribute is a class *and* an association at the same time.

An association class is exclusively linked to exactly one association whose properties it bears. However, an association class can have association to other classes (but not association classes itself). Figure 11 shows an example of an association class.

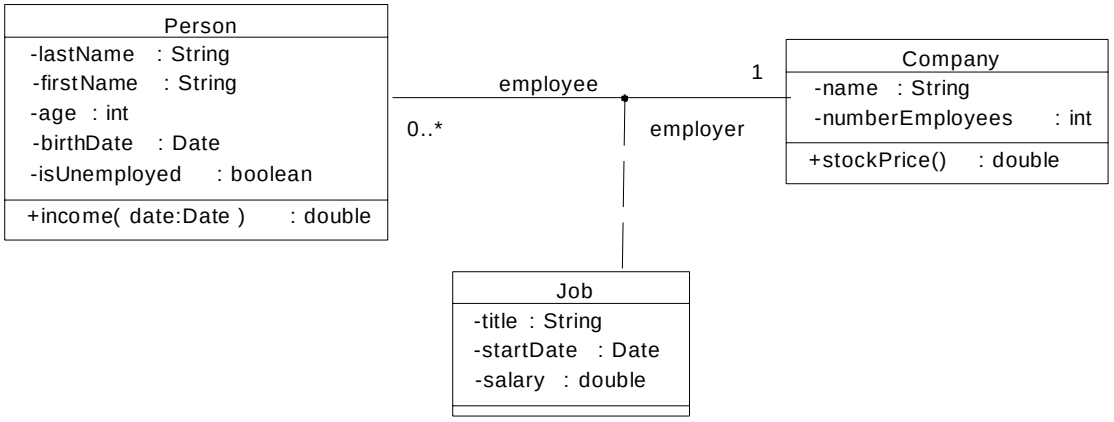


Figure 11: Association Class

In the source, association classes are broken into two separate binary associations between the original class and the association class. The multiplicity of the roles of the original classes is always 1. The multiplicity of the roles of the association class are that of the opposite role end of the original class, but mutually exchanged.

With regard to source code, these artificial associations are treated the same as real associations (see "Attributes from Associations" on page 2-25). The classes shown in Figure 11 above create the following code:

```
public class Person {
       ...
       private boolean isUnemployed;
       private Date birthDate;
       private int age;
       private String firstName;
       private String lastName;
       protected Job job;
       ...
       public double income(Date date) {
           ...
           return 0.0D;
           ...
       }
   }
```

```
import java.util.*;
public class Company {
    ...
    private int numberEmployees;
    private String name;
    protected Vector job;
    ...
    public double stockPrice() {
        ...
        return 0.0D;
        ...
    }
}

public class Job {
    ...
    private String title;
    private Date startDate;
    private double salary;
    protected Person employee;
    protected Company employer;
    ...
}
```

Operations

Operations or methods describe the behavioral features of a class and how classes react to messages from the outside. In Java, operations can be either (1) *instance* operations, which act upon a specific instance states, or (2) *static class* operations, which deal only with *static* class attributes.

Furthermore, there can be *final* methods, which cannot be overridden in subclasses. Other modifiers describe the visibility, whether the method is thread-safe (synchronized), whether it is abstract, and whether it must be implemented in a low-level language (native). The latter two modifiers lead to a declaration without implementation, i.e., the declarations are finished after the parameter list with a semicolon (;). Abstract methods must be implemented in a method of a subclass and native methods in a native library, i.e., in a library written in a low-level language.

Modeled operations lead to a *method declaration* in the source code. Operations can have zero, one, or several formal parameters, zero or one return values, and they can throw zero, one, or multiple exceptions.

The return types and the parameter types of operations are imported if they are a (non-primitive) reference type from the model and not in the same package as the current class (see "Package and Import Statements" on page 2-8). This is also true for arrays. In this case, the *base types* of the return type and/or parameter types are imported according to the rules above.

Using the Class Table Editor

As with attributes, you usually model operations in a class diagram and later add detailed information in the class table editor.

Class Member Definitions

For every operation, specify:

- Name
- Return type
- Operation parameters

If the return type is missing from the model, the operation is automatically assigned the return type *void*.

Analysis Items

For every operation, specify:

- Access privilege
- Whether it is *abstract*
- Whether it is a class operation
- What exception is thrown

If the visibility is not modeled, the operation is assigned as the visibility the value of the global TDL variable [DEF_OP_VISIB], which defaults to *public*.

Java Implementation Items

For every operation, specify one or more of the following modifiers:

- Final
- Synchronized
- Native

Using the OAE

You can provide a method body for an operation by adding a Java Source Code note to an operation by using the OAE and filling the description with your source code.

If operations do not have such a note, a default return value (corresponding to the return type *null* value) is provided. For example, the null value for a boolean type is False; for a char type the value is \u0000.

Constructors

Constructors basically are treated the same as other operations. However, they do not have a return type. Constructors appear in the source code before all other operations.

Examples

Figure 12 shows a class ShopController. The operations in this figure serve mainly as an example for visibility, type import, and exception handling. Static and instance operations also occur. The following code is produced:

```
package operations;
import optypes.Article;
import attrtypes.Customer;
import optypes.Address;
import optypes.InvalidAddressException;
public class ShopController {
       // ------------------------------------------------
       // methods
       // ------------------------------------------------
       public static void printBills() {
              return;
   }
```

```
        public int getStock(Article a) {
            return 0;
        }

        public void printBills(Customer c) {
            return;
        }

        public double getPrice(Article a) {
            return 0.0D;
        }

        protected Customer findCustomer(Address a) throws
            InvalidAddressException {
            return null;
        }

        double getPrice(Article a, int stock) {
            return 0.0D;
        }
}
```

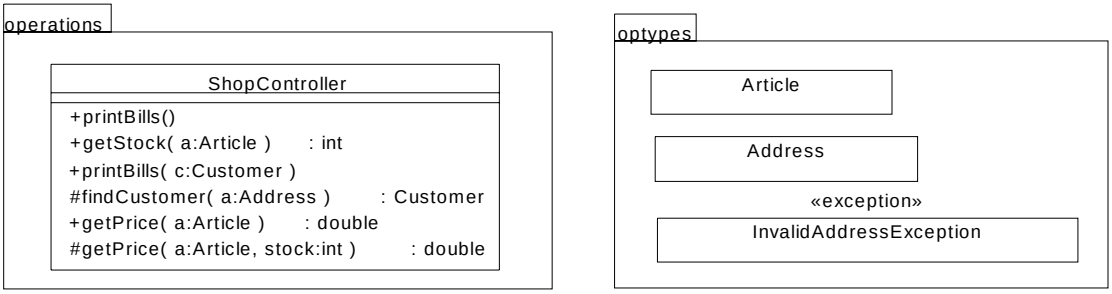


Figure 12: Operations 1

Class FileWriter in Figure 13 on page 2-34 shows other aspects of Java methods including abstract, native, synchronized, and final methods. Furthermore, the class contains constructors, which appear before the other methods. Note that the automatic default constructor is set to *protected*, because the class itself is *abstract* (see "Constructors" on page 2-31).

```
package operations2;
public abstract class FileWriter {

       // ---------------------------------------------------
       // constructors
       // ---------------------------------------------------
       public FileWriter(File f) {
          return;
       }

       public FileWriter(String fileName) {
          return;
       }

       /**
        * Default constructor
        */
       protected FileWriter() {
          return;
       }

       // ---------------------------------------------------
       // methods
       // ---------------------------------------------------
       public abstract void write(char[] cbuf);
       public final void close() {
          return;
       }

       public native void flush();
       public synchronized void write() {
          return;
          }
}
```

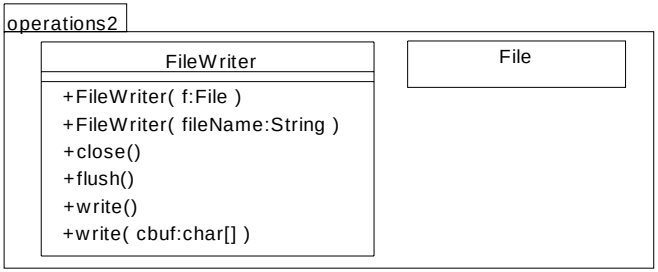


Figure 13: Operations 2

State Machines

In contrast to many code generators from other modeling tools, ACD has the ability to transform state chart diagrams into *finite state machines* (hereafter referred to as FSM). In the Java implementation, state charts (and tables) are translated into two possible variations of a FSM. In order to evaluate the impact of this feature, we have to clarify what the implementation is and what it is not.

UML Specification 1.3 says the following in Section 2.14.2 (execution semantics of a state machine) about a hypothetical state machine:

> In the general case, the key components of this hypothetical machine are:
>
> • an *event queue* which holds incoming event instances until they are dispatched
>
> • an *event dispatcher mechanism* that selects and de-queues event instances from the event queue for processing
>
> • an *event processor* which processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question.
>
> Because of that, this component is simply referred to as the “state machine” in the following text.

Currently only part three, the event processor, is created in the form of a method. This method accepts events in the form of integer constants. The constants are also written to the class, together with constants for states.

Parts one and two, the queue and the dispatcher, are omitted from the source. You must provide your own methods for collecting events and routing them to the event processing method.

State Machine Example

Consider the rather simplistic state machine in Figure 14 on page 2-38, which accepts a move event and says *tick* or *tock*. The class Pendulum has one method, print(s:String). Every time a move event occurs, this method is called with the appropriate argument. The argument depends on the state that receives the event.

For every state and every event, ACD creates a symbolic constant. Furthermore, ACD generates a takeEvent()method (the e*vent processor* in the terminology above), which accepts the events and switches the current state. The takeEvent also evaluates actions, depending on the state, that were active when the event occurred.

The state machine example generates the following code:

```
package state machines;
public class Pendulum {
// ---------------------------------------------------------
// constructors
// ---------------------------------------------------------
public Pendulum() {
   currentState = _NOTMOVING;
}
// ---------------------------------------------------------
// methods
// ---------------------------------------------------------
public void print(String s) {
       ...
       return;
       ...
   }

// ---------------------------------------------------------
// state machine (realized with nested switch statements)
// ---------------------------------------------------------

protected byte currentState;
```

```
// events
////////////////////////////////////////////////////////
/////
public static final byte _START = 0;
public static final byte _MOVE = 1;

// states
////////////////////////////////////////////////////////
/////
public static final byte _NOTMOVING = 0;
public static final byte _ISLEFT = 1;
public static final byte _ISRIGHT = 2;


/// Event taker functions
////////////////////////////////////////////
public void takeEvent(byte ev) {
       switch(currentState) {
       case _NOTMOVING:
          switch(ev) {

          case _START:
             print("Tock");
             currentState = _ISLEFT;
             isLeft_Action();
             break;
          default:
             System.out.println("Wrong event occurred "  +
             "Pendulum " + "NotMoving" + " " + ev);
             break;
          } // end event switch for state NotMoving
          break;
       case _ISLEFT:
          switch(ev) {

          case _MOVE:
             print("Tick");
             currentState = _ISRIGHT;
             isRight_Action();
             break;
          default:
             System.out.println("Wrong event occurred "  +
             "Pendulum " + "isLeft" + " " + ev);
             break;
          } // end event switch for state isLeft
```

```
            break;
        case _ISRIGHT:
            switch(ev) {

            case _MOVE:
                print("Tack");
                currentState = _ISLEFT;
                isLeft_Action();
                break;
            default:
                System.out.println("Wrong event occurred "  +
                "Pendulum " + "isRight" + " " + ev);
                break;
            } // end event switch for state isRight
            break;
    } // end state switch
    }

    /// State action functions//////////////////////
    protected void isLeft_Action() {
        ...
    }

    protected void isRight_Action() {
        ...
    }
}
```

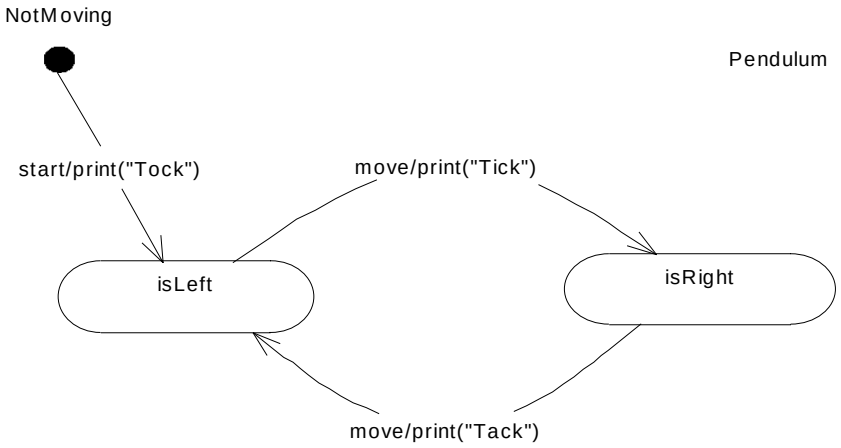


Figure 14: State Machine

Note: As outlined in "State Machines" on page 34, ACD provides only the event processing component of the state machine. The events must be generated and routed to the takeEvent() function by other functions, which call takeEvent() periodically with a move event.

Transitions

Probably the most important parts of a state chart diagram are the transitions. They rule when and to what state an object's lifecycle can change.

A transition has an event upon which the state change occurs. The event may be inhibited by a so-called *guard condition*. A guard condition is a boolean expression that can inhibit the transition, if it evaluates to False. An event may *trigger* an *action*, generally an operation, which the object can carry out. A transition may also trigger a *send event*, i.e., a message to another object.

Transitions may be forked or joined using *JunctionPoints* or *DynamicChoicePoints*. In the former case, all possible guard conditions are considered before an event can fire. In the latter case, a guard condition is evaluated and the result influences the guard conditions of transitions after the DynamicChoicePoint.

States

When we speak of the *state* of an object, we often think of the current arrangement of the object's internal attributes. However, in the state chart diagram, a state determines how the object reacts to an incoming event. An object with a trivial lifecycle, where the object reacts to incoming messages and events in the same manner all the time, does not need a state chart.

A state may have internal *entry* and *exit* actions. These actions are carried out by the state at every incoming or outgoing transition respectively, regardless of the exact nature of the transitions. This is a shortcut notation; otherwise, you would be forced to add the actions to every incoming or outgoing transition.

A different kind of action are the so called *do-activities*. As the name suggests, the do-activity is an ongoing action that is performed as long as the object is in that state or until the computation ends. Since the implementation is sequential, do-activities are put between entry and exit actions, if present.

Supported State Machine Elements

StP / ACD supports all UML state machine constructs in the metamodel. See *<StP>/documentation/ACD_METAMODEL*. The Java code generator supports most, but not all, of these constructs.

Note: Entry actions, exit actions, and do-actions must be defined on a single line in the appropriate sections of the State Table editor. Consecutive lines must be separated by semicolons.

Currently, you can evaluate initial states, final states, normal states (entry action, exit action, do-activity), pseudo states (DynamicChoicePoint, JunctionPoint), and transitions (events, guards, send events).

History states, deferred events, internal transitions, and composite states are not supported.

State Table Pattern

A different and more scalable state machine design than the *nested switch statement* approach above is the State Table pattern, introduced by Bruce Powell Douglas (Douglas, Bruce Powell, *Real-Time UML: Efficient Objects for Embedded Systems*.Reading, MA: Addison-Wesley-Longman, Dec. 1997).

As in the simple design above, a takeEvent() procedure processes incoming events, but it delegates its task to a *state table*, a matrix that holds the corresponding (incoming) transition for every state. States and transitions are modeled as inner classes that are assigned to the state table matrix.

The state table pattern is useful for larger state spaces and it can access a state in one single probe. However, a state table pattern is more difficult to set up. The state machine from Figure 14 on page 2-38 would be initialized as follows:

```
/**
* Default constructor
*/
public Pendulum() {
    ...
    stateTable = new StateTable();
    // This instance of State is needed to instantiate
    // the inner classes from State
    state = new State();

    State.NotMoving notmoving = state.new NotMoving();
    stateTable.assignState(notmoving, _NOTMOVING);

    State.isLeft isleft = state.new isLeft();
    stateTable.assignState(isleft, _ISLEFT);

    State.isRight isright = state.new isRight();
    stateTable.assignState(isright, _ISRIGHT);
```

```
   TransitionContainer();
trans_NotMoving.setTransInterface(trans_NotMoving.new
start(_ISLEFT));
   stateTable.assignTrans(trans_NotMoving, _NOTMOVING,
_START);

   TransitionContainer trans_isLeft = new
TransitionContainer();
    trans_isLeft.setTransInterface(trans_isLeft.new
move(_ISRIGHT));
    stateTable.assignTrans(trans_isLeft, _ISLEFT, _MOVE);

    TransitionContainer trans_isRight = new
TransitionContainer();
    trans_isRight.setTransInterface(trans_isRight.new
move(_ISLEFT));
    stateTable.assignTrans(trans_isRight, _ISRIGHT, _MOVE);

    //#end ACD#
}
```

Buildfiles

In order to automate the compilation of large and middle-sized projects, make tools (“build” tools) are an invaluable aid. Make tools create one or several targets, if and only if the target dependencies are unsatisfied. This way, you generally do not have to rebuild the whole application unless certain files have changed.

In the Java code generator, the make tool ANT, available from Apache Software Foundation, is currently supported. The generator creates *buildfile.xml* in the base directory of the code generation. You must set the global TDL variable [CREATE_MAKEFILES] to True and the variable [MAKEFILE_TYPE] must be *ant* or *all* to enable this feature.

Make Invocation

Simply change to the base directory of code generation and invoke *ant*. If ANT is correctly installed, it will create several directories, compile all source files, and create a JAR archive from the compiled source files.

Alternatively, you can choose to stop the make run after the compilation. Then type “ant build”, which invokes the build task. Typing “ant delete” deletes the compiled class files and the JAR archive.

In addition, you can create JavaDoc documentation with the command “ant doc”. Beneath the directory named *docs* there is a hierarchy of documentation in HTML format. The command “ant doc-clean” removes the generated documentation.

Additional Code Generation

In this section we discuss how you can create source code not originating from the model. With ACD, you can automate many repetitive programming tasks, such as generating default constructors, writing accessor methods for attributes or standard methods from java.lang.Object.

Note: All of the following additional code constructs are created *only* if not already modeled by hand. This way, you do not run into the situation where source code or descriptions that belong to a specific model element are lost in the source code.

Accessor Methods (Setter/Getter)

For every non-static method, a setter / getter method is generated automatically if the global [ACCESSOR_METHODS] is set to True. Individual attributes can be excluded if the tagged value *accessors* is set to False. Alternatively, you can set [ACCESSOR_METHODS] to False and enable the generation of certain attributes with tagged value *accessors* set to True.

The name of the method is built from the name of the attribute, with the first letter uppercased and the prefixes “set” and “get”. With the getter method, the return type is the attribute type and the method simply returns the attribute. With the setter method, the return type is *void* and the method features a parameter that is the exact name of the attribute and type. In the body the method assigns the parameter to the instance attribute. The visibility is always *public*.

For example, for an attribute *private double balance*, the methods

```
public void setBalance(double balance) {
    this.balance = balance;
}
public double getBalance() {
    return balance;
}
```

are created.

Note: If an attribute has the tagged value *readonly*, only a getter method is generated.

Implementation of Abstract Methods

Similar to the implementation of methods for implemented interfaces, the code generator automatically constructs implementations for abstract methods in superclasses. Setting the global TDL variable [CREATE_ABSTRACT_IMPL] to True enables the feature. Normally the superclass itself must also be abstract. You can automatically set superclasses, which contain abstract methods, to *abstract* by setting [SET_CLASS_ABSTRACT] to True.

Standard Methods

If [CREATE_STANDARD_OPS] is set to True, three standard methods from java.lang.Object are inserted automatically into the class. The methods are:

- `public boolean equals (Object o)`

  “equals(Object o)” compares an instance of class against another instance semantically. The comparison returns True when the other instance is of the same class and its non-static attributes have the same value as this instance.

- `public String toString ()`

  “toString()” returns a string representation of this instance status, i.e., the contents of its non-static attributes.

- `public Object clone ()`

  "clone()" returns a shallow copy of the current instance. This means that it creates a new instance of the class of which the instance is a member, copies all its fields with primitive Java types, and assigns a new reference to all the object types of the current instance.

All methods are developed with run-time errors in mind. For example, the equals method checks whether two compared reference attributes are all *null* or not *null*.

Constructors

If [CREATE_DEF_CTOR] is set to True in *java_globals.tdl*, a parameterless (no-arg) default constructor is generated. If [CREATE_FULL_CTOR] is set to True, a constructor, which initializes all non-static fields of the class, is created.

Abstract classes may also have default constructors, but they are set to *protected* visibility automatically to prevent fruitless initialization attempts.

Note: If this option is set to True, [CREATE_FULL_CTOR] is automatically also set to True. Creating constructors and not creating a no-arg constructor would cause the compilation to fail in many constructs.

Finalizers

By default, a finalizer (protected void finalize()) is created for a class. By using this method you can free resources that would otherwise not automatically be freed by the garbage collector of the Java Virtual Machine. The global TDL variable [CREATE_FINALIZER] must be set to False to deactivate this feature (default is True).

Patterns

The Java code generator supports some design patterns and predefined language constructs. These patterns can be utilized by providing either a stereotype or a specific tagged value. In particular, you can create code for:

- A *main* method
- A *singleton* pattern
- A *state table* pattern (See "State Table Pattern" on page 2-40.)

Main Method

If a class has a stereotype <<main>>, a main() method is automatically generated for that class. The usual modifiers and arguments are provided:

```
public static void main(String[] args) {
    ....
}
```

If you give a class the tagged value "pattern=singleton", the following code fragments / changes are created:

- The constructor's visibility is changed from public to private.
- An additional method, <CLASS> getInstance(), is generated.

What is Difficult to Generate?

In UML, you can model nearly everything that you can write by hand. However, there are some constructs that cannot be expressed easily in UML and can better be created by other, more specialized tools.

Local classes/Anonymous classes

Currently their is no good way to express anonymous classes; e.g., the Listener classes of the Java event handling mechanism. You can, however, write it down as the operations source code.

GUI modeling

In general, UML is weak in defining graphical user interfaces. Even if you manage to assemble a dialog or window from graphic parts such as buttons, lists, checkboxes, and so on, you cannot easily express the graphical layout or the user interactions related to that GUI element. GUI builders are much better at the job.

3 Generating Ada 95 Code

Introduction

This chapter discusses using ACD to generate Ada code from models created with StP / UML 8.x.

Besides standard elements such as code skeletons for classes, constructors, and operations, ACD also generates code for:

- Ada tasks and protected objects
- Implementation methods of abstract classes
- Accessor methods (*setter* and *getter* methods) for attributes and associations
- Implementation of state machines from state charts (!)
- Special variants of state machines to exploit Ada 95 real -time programming techniques
- Automatic memory management based on Ada controlled types

Furthermore, you can generate code selectively for single or multiple

- Class diagrams
- Class tables
- Packages
- Individual classes

The document describes the model constructs in StP and how they are mapped onto Ada 95 code. It also enumerates all stereotypes and tagged values considered by the ACD templates. These stereotypes are defined in stereotype diagrams contained in the Ada 95 example system,

Millenium_Clock_Ada95, which is delivered with the product. We suggest that you copy these diagrams into their own systems. Thus you can select the stereotypes interactively.

See also Appendix B, "Ada 95 Notes".

Prerequisites

The code generator is not based on a particular Ada 95 development environment It supports navigation from the StP model to the source code and vice versa, which is currently available only for Aonix ObjectAda.

Compatibility

With Language Specifications

The generated code conforms to the Ada 95 Standard ANSI/ISO/IEC-8652:1995.

With Previous Code Generators

The generated code is in general compatible with code generated by previous versions of the ACD based code generator for Ada 95 from StP 8.x. For these versions, there is no documentation available.

This code generator generates signal handler packages only if a tagged value is set (see "Signals" on page 3-31); previous versions generated them unconditionally.

Aspects of Mapping to Ada 95

This section discusses issues arising from the nature of the Ada 95 language.

UML Constructs That Determine Code Generation

The following UML constructs in your StP model determine what Ada 95 code is generated by ACD:

- Packages
- Classes
- Attributes
- Operations
- Associations, aggregations, compositions
- States and state machines
- Events
- Actions

Code generation can be controlled individually by use of stereotypes, tagged values, and annotations. All stereotypes and tagged values are handled in a case insensitive manner.

The following constructs are not currently supported:

- Bind relations
- Uses relations
- Import relations
- Object and object instantiation links

Case Sensitivity

Although StP is sensitive to upper and lower case, Ada 95 is not. Therefore, if there are two class symbols in StP with names that differ only in their case, e.g., Class_a and Class_A, duplicate declarations occur in Ada 95.

Declaration Order and Enclosing Package

Because the Ada 95 rules of visibility and the restriction that most defining occurrences need to be unique, placement and order of declarations in the generated Ada code is critical.

Ada does not allow declarations other than packages and subprograms to occur at the outermost level in a compilation unit. In UML, it is typical to declare a class without a package. As classes are normally mapped onto a type declaration (see "Mapping UML Classes to Ada Constructs" on page 3-10.), a package needs to be introduced by the code generator, at least if there is no package in the model enclosing the class. See "Implicit Packages" on page 3-8 for details.

A typical implementation of relations (see "Relations" on page 3-23.) found in UML is the use of pointers. This code generator uses the same approach. In Ada 95 the type construct for pointers is the access type. An access type must be declared before it is used. Moreover, the designated type needs to be declared in the same scope.

Here the scope is a package. For a relation navigated from class A to class B, the package containing B must be visible in the package containing A. If the relation can be navigated in both directions, the package with A must also be visible in the package B. Ada 95 does not allow a cyclic dependency between package specifications. There are two solutions to prevent the cycle:

1. A and B are declared in the same package.
2. A and B are declared as incomplete types in the package specification with their completion being located in the package body.

Although the second possibility sounds good, it has several drawbacks:

- There is always an indirection with the class; for example, it cannot be a member in a composition.
- The related class normally occurs in operations as well, e.g., in constructors. These operations cannot be declared at any place where they gain access to the type completion of the class without using unchecked conversions.

This code generator uses the first approach. You can control this directly by placing the classes with bi-directional navigable relations into one package in the model. Alternatively, the code generator implicitly generates a package containing all the classes that are part of the dependency cycle.

However, we suggest that you explicitly define the package. This way, you document the strong connection between the classes in the model. And you can give the package a reasonable name (the names of implicitly generated packages contain all the class names and thus are not very easy to read). See also "Implicit Packages" on page 3-8.

Realization of Protected Mode

The UML concept of protected access is not provided in Ada 95. In most cases, it should be appropriate to use visibility private, which allows visibility from child packages.

Evaluation of Expressions and Action Code in the Model

The code generator is able to handle expressions in state machines and action code associated with transitions even when these are written in C++/Java style.

More details about this transformation can be found in "Guard and Action Syntax" on page 3-34.

Mapping Model Type Names onto Ada Subtypes Indications

Class names from the UML model are mapped one-to-one to the names of classes (usually tagged types) in the generated Ada code. An exception is a class with stereotype Datatype (see "Data Type Class" on page 3-13). In addition, the class name with suffix "_ptr" is used to form the name of an access type for that class. In the same manner, the class name with suffix "_cptr" is used to form the name of an access type for the corresponding class-wide type. The generation of both access types can be controlled via tagged values; see Table 5 on page 3-17, which lists the tagged values for classes.

Base Classes

Every tagged Ada type generated from a root class in the model, i.e., one that does not have a base class in the model, is derived from either of two predefined base classes:

- Active_Base_Class for a class with an associated state machine
- Inactive_Base_Class for other classes

These base classes allow you to define a common infrastructure for the benefit of all classes generated by this code generator.

Memory Management

This section describes the memory management aspects of the dynamic creation of class instances and the data structures that implement associations and aggregations (called “relations” in this document). See also “Constructors and Destructors” on page 3-15.

Managing Class Instances

The Ada 95 templates support both the dynamic creation of class instances and their creation by object declarations. Class instances created by object declarations can be initialized by calling the implicitly generated operation Initialize. Dynamic creation of class instances is supported by the implicitly generated function Create, which allocates an object (on the heap) and then calls the Initialize operation.

In the same manner, there are implicitly generated operations Finalize and Free; Finalize does the cleanup and Free does the deallocation of the object itself. In particular, the generated cleanup code in Finalize deallocates the auxiliary data structures holding the associations.

Note: If a class is a generalization of Ada.Finalization.Controlled or Ada.Finalization.Limited_Controlled, there will be an implicit call to Initialize and Finalize, even if the class instance is created by an object declaration.

If desired, you can insert additional code into the Initialize and Finalize operations.

Note: For non-controlled types, you are responsible for calling the initialization routine for those class instances which are created by object declarations.

Managing Associated Class Instances

As is the case with C++, Ada 95 does not support automatic garbage collection. Therefore, some support is provided to automatically deallocate unused class instances. The details of these mechanisms are explained in "The Aggregation Stereotype: Owner" on page 3-24 and "The Aggregation Stereotype: Counted_Ref" on page 3-25.

Run-Time Environment

The code generated by the Ada 95 code generator depends on a framework that is implicitly generated in the subdirectory *ACD_Runtime*. It consists of the package ACD_Runtime plus several child units. This run-time environment consists of three major parts:

- The base classes for all classes from the model
- The signal mechanism for asynchronous events
- The generic Smart_Pointer_pkg, used to implement stereotype Counted_Ref

In addition, there is a procedure for reporting exceptions (from task bodies, in particular).

Packages

An Ada package is generated for UML packages in the model and for template and instance classes. Further, an implicit package is generated for a UML class in the model that is not enclosed in a UML package.

The code generator supports a stereotype (see “Stereotypes” on page 3-16.) for a UML package, which causes the package to represent a *subsystem*. A subsystem is not considered for generation of Ada packages. Instead, it causes the generated source code to be placed in subdirectories (see “Subsystems” on page 3-8).

Implicit Packages

A class that is not explicitly embedded in an Ada package is implicitly enclosed by a package. In the simplest case, each class is in its own package. The name of the package is the class name plus the suffix “_pkg”. If the class is derived from another class, the package is a child of the package where the parent class is defined.

The implicit package mechanism is used because Ada always requires such a package. On the other hand, it saves a lot of work if you do not need to model these packages explicitly.

If multiple classes have cyclic dependencies through associations, all classes of the cycle are embedded in one package. The package name includes the names of all classes in the cycle.

This mechanism allows rapid prototyping. For production use, it might be better to explicitly enclose the classes in an Ada package (see “Ada Packages” below). In particular, the explicit modeling of the package allows explicit naming of the package.

Ada Packages

A package that is not a subsystem (see “Subsystems” below) is considered as a simple Ada package that is used to enclose one or more classes.

Subsystems

UML packages with a stereotype *subsystem* (or segment) are treated in a special way by StP. They are used to logically group parts of the model and build the basis for configuration management.

These subsystem packages have no influence on the generated code, except for file naming conventions. For each subsystem, the source tree contains a subdirectory (with the same name as the subsystem) that contains all the files generated for that subsystem. Thus, the structure of the source tree reflects the subsystem structure of the model.

In addition, a subsystem can optionally define the tagged value FileNamePrefix. If defined, the value is prepended to all file names generated for the subsystem. This feature is useful in those cases where the files are viewed in a non-tree structured tool, e.g., file view of ObjectAda. When the prefix mechanism is used, the subsystem structure is maintained in an alphabetically sorted file list.

Stereotypes

Table 1 below lists the stereotypes for UML packages that are supported.

Table 1: Supported Stereotypes

Name	Meaning
generic	The generated Ada 95 package is made generic
segment	The package is a subsystem
subsystem	The package is a subsystem

Tagged Values

Table 2 below lists the tagged values for UML packages that are supported.

Table 2: Supported Tag Values

Name	Meaning
FileNamePrefix	The text is prepended to all file names in the subsystem

Annotations

Table 3 below lists the annotations for UML packages that are supported.

Table 3: Supported Annotations

Note	Item	Meaning
Package Definition	External Package	The package is external, i.e., no code is generated

Classes

This section describes the various mappings for UML classes followed by the general aspects that are applicable to all (or most) mapping variants.

Mapping UML Classes to Ada Constructs

Various mappings are available for UML classes. The mapping is selected from the set of tagged values, the stereotype, and the state machine of the class, if any.

A class with a state machine is called *active* here, *passive* otherwise.

A data type is a special case of class definition. Its only purpose is to define a name to be used for attribute or operation argument types. The data type has a tagged value that gives the subtype indication of that type. (See "Data Type Class" on page 3-13 for details.)

The code generator is aware of classes being derived from Ada.Finalization tagged types. For these classes, it generates the interfaces as described in "Constructors and Destructors" on page 3-15.

Tagged Type for a Passive Class

If no stereotype is given, classes are mapped to tagged records. This is the principal mechanism used to map classes to Ada. Stereotypes can be used to deviate from this standard mapping.

Protected Type for a Passive Class

Classes can be mapped to Ada protected objects by use of the stereotypes tagged_protected_type and protected_type. Please note that in Ada 95, the concepts of object orientation and protected types are not very well integrated. The stereotype tagged_protected_type combines both concepts.

The stereotype protected_type should only be used if you explicitly want a protected type and accept the restrictions imposed by the language.

The stereotype tagged_protected_type generates a tagged record, which encapsulates a protected object plus attributes, relations, and possible user-defined extensions. The protected object has an access discriminant to access the enclosing tagged record.

The stereotype protected_type directly generates a protected type. Please note that this implies some restrictions on the use of the class. The major restriction is that the class cannot be part of a generalization hierarchy. This is quite natural since tagged records are essential for implementing generalizations in Ada 95.

The attributes, relations, and possible user-defined extensions must be declared in the private part of the protected object. In general, when there is a problem using protected_type, you should evaluate the possibility of using the more general tagged_protected_type. You may encounter problems with initialization and finalization code in non-trivial cases. Using a more general stereotype may avoid some of these difficulties.

Task Type for a Passive Class

Passive classes are normally mapped to tagged records. However, when a class has a state machine with transitions triggered by time events, the class is implicitly mapped as for stereotype tagged_task.

Task Type for an Active Class

Classes can be mapped to Ada tasks by using the stereotypes tagged_task and *task*. The predefined stereotypes, thread and active, are synonyms for task. Please note that in Ada95, the concepts of object orientation and task types are not very well integrated. The stereotype tagged_task combines both concepts.

The stereotype *task* should only be used if you explicitly want a task type and accept the restrictions imposed by the language.

The stereotype tagged_task generates a tagged record that encapsulates a task object plus attributes, relations, and possible user-defined extensions. The task object has an access discriminant to access the enclosing tagged record.

The stereotype *task* directly generates a task. Please note that this implies some restrictions on the use of the class. The major restriction is that the class cannot be part of a generalization hierarchy. The attributes, relations, and possible user-defined extensions must be declared in the body of the task.

In general, if there is a problem when you use *task*, it is recommended that you evaluate the possibility of using the more general tagged_task.

Generic Package for Parameterized Class

The UML notion of a parameterized class is mapped to a generic package containing a class definition. Formal types and objects are supported. There is no support for generic formal subprograms because UML does not support that concept.

Formal Objects

A class parameter of the form “object_name:type_name” is mapped to a generic formal object.

Formal Types

A class parameter of the form “type_name or type_name:class” is mapped to a generic formal private type.

Generic Instance for Instance of Parameterized Class

An instance of a parameterized class is mapped to a generic instance of the template package. The actual generic parameters are taken from the bind relation between the class instance and the template class.

Data Type Class

A class with stereotype Datatype supports the type mapping mechanism as described in the *ACD Programming Guide* and the *UML Reference Manual*. During code generation the name of this class is replaced by the language-specific type name given by the tagged value Ada 95.

For example, if you model a class Real in UML, giving it the stereotype Datatype and the tagged value "Ada95=My_Types.Float_6", each reference to class Real is replaced by the name My_Types.Float_6. This method makes it very easy to change the type mapping for class Real. This is useful for referencing types that are not part of the model, e.g., the predefined types of Ada 95 or any hand-coded parts of the system.

Enumeration Type for a Passive Class

A class with stereotype *enumeration* is mapped to an enumeration type. The enumeration literals can be defined in one of two ways:

- If there are attributes without a default, the attribute names are taken as enumeration literals in an undefined order.
- The default values are interpreted as position numbers. The attribute names are taken as enumeration literals in the order of the position numbers. This method should be used if the order is relevant. The position numbers must be contiguous and start at 0.

  Note: This does not mean that a representation clause is generated.

Array Type for a Passive Class

A class with stereotype Array is mapped to an array type.

Such a class must have a single relation of type Aggregation or Composition to another class, which is used as element type. In case of a composition, the element type is the related class itself. For aggregations, the element type is a pointer to the related class.

The multiplicity of the relation determines the index type. If the multiplicity is static, the index is “range is 1..N”. Otherwise, the array type is indefinite and the index is “natural range <>”.

Storage Pool Class

A class with stereotype Storage_Pool is mapped to a type derived from System.Storage_Pools.Root_Storage_Pool. Such a storage pool implementation can be used by other classes, if they specify the name of the storage pool class in their tagged value Storage_Pool.

Visibility

The visibility of classes themselves (e.g., “type class_name is ...”) cannot be controlled. The classes are always considered public and the annotation Class Definition / Class Visibility is ignored.

The form of the visible type declaration can be controlled via annotation Class Ada_95 Definition / Class Type Privacy or the tagged type Class_Visibility. The tagged value is only evaluated if the annotation is not present.

The possible choices for visibility are as follows:

- `public: type ... is new ...with record ... end record`

  That is, the full type declaration is visible.

- `private: type ... is private`

  That is, the structure of the type is not at all visible.

- `implementation:`

  In the visible part, only a handle in the form “type xx_ptr is private” is generated; xx_ptr is an access to xx and the full type declaration of xx is in the package body.

- `tagged private: type ... is tagged private`

  That is, the property of being a tagged type is visible, but neither the base class nor the record extension.

- `private extension: type ... is new ...with private`

  That is, the property of being a tagged type derived from a certain base class is visible, but not the record extension.

The default is "private" for non-tagged class type and "private extension" for tagged class type.

The visibility choices, tagged private and private extension, are only applicable to classes mapped to tagged types.

Constructors and Destructors

For types derived from Ada.Finalization.Controlled and ...Limited_Controlled, the Initialize [Adjust] and Finalize subprograms are generated as defined for these root types, using *in out* parameters.

Note: The types derived never inherit the implementation of these routines. Instead, automatic calls to the operations from the superclass are inserted into the respective bodies.

For all noncontrolled types, a set of Initialize / Create / Finalize / Free routines is generated. Create allocates a class instance, initializes it, and returns a pointer to it. Initialize has a class-wide pointer as a parameter. Finalize uses an *in out* parameter of the class itself. Therefore, only Finalize is dispatching.

Initialize contains code for implicit initializations, e.g., for state machines. Finalize contains code for implicit cleanups, e.g., it deallocates the auxiliary data structures holding the associations.

In both subprograms, you can add your own code.

In both cases, there is a Free procedure that finalizes an instance and deallocates it.

Singleton Patterns

If the tagged value “pattern” has the value “singleton”, there exists exactly one instance of the class. The package contains a variable holding that instance and the constructor always returns that instance of the class.

Stereotypes

Table 4 below shows the stereotypes for UML classes that are supported.

Table 4: Supported Stereotypes for UML Classes

Name	Meaning
Array	Class is mapped onto an array type
Datatype	This is a datatype class
Enumeration	Class is mapped onto an enumeration type
Tagged_protected_type[a]	Class is mapped onto a tagged record type with a protected type component
Protected_type[a]	Class is mapped onto a protected type. **Note:** No other class must be derived from this one.
Storage_pool[a]	Class is mapped onto an implementation of a storage pool
Task Thread Active	Class is mapped onto an Ada 95 task type. All three names are synonyms.
Tagged_task[a]	Class is mapped onto a tagged record type with a task type component

a. Can be written with “_” or “-” in between the words.

Tagged Values

Table 5 below shows the tagged values for UML classes which are supported.

Table 5: Supported Tagged Values for UML Classes

Name	Type(default)	Meaning
Abstract	Boolean(false)	An abstract class
Child_Package	Boolean[a]	The generated package is a child package
Class_Pointer	Boolean(true)	Generate a class wide pointer type
Pointer	Boolean(true)	Generate a pointer type
Gen_Signal_Handler	Boolean(false)	Generate signal handler package (for active classes only)
Stream_Operations	Boolean(false)	Generate stream operations for the class type
Limited	Boolean(false)	Generate a limited class type
Attribute_Operations	Boolean(true)	Generate subprograms for attribute access. False implies that attributes are not externally visible.
Class_Visibility	[b] and [c]	Controls privacy of the class
Storage_Pool	""	The storage pool to be used for the access types of this class
Pattern	Singleton("")	Only one class instance exists
Subunit	Boolean(false)	The class must be a protected or task type. The body is realized as subunit.

a. True for a subclass; false for a class without superclass.
b. Enumeration (public, private, implementation, tagged private, private extension).
c. Private for non-tagged class type, private extension for tagged class type.

Note: The names of tagged values, with reference to capitalization, must be written as shown in Table 5.

In cases where an annotation as well as the corresponding tagged value is defined, the annotation takes precedence.

Annotations

Table 6 shows the annotations for UML classes which are supported.

Table 6: Supported Annotations for UML Classes

Item	Name	Meaning
Class Definition	Abstract Class	An abstract class
	External Class	The class is external, i.e., no code is generated for it
Class Ada_95 Definition	Generate Access Type to Class-Wide Type	Generate a class wide pointer type
	Generate Access Type to Class Type	Generate a pointer type
	UmlClassIsALimited (Root Class Only)	A limited class type is generated
	Class Type Privacy	Controls privacy of the class

Attribute

The component declaration for attributes is always generated in the private part of the package specification. There is no way to change the visibility of the components themselves. However, the visibility of the Get / Set subprograms can be controlled. (See "Visibility" below.)

Discriminants

An attribute is implemented as a discriminant of the (type of) class if it has the stereotype *discriminant*.

Visibility

There are three methods to control the generation of Get / Set subprograms:

- Tagged value for the whole class
- Class table visibility entries
- Individual tagged value for the attribute

If the class has the boolean tagged value Attribute_Operations set to False, none of its attributes are visible. They are all implemented as *implementation*, regardless of the class table entries or individual tagged values for the attribute. Therefore, Get / Set subprograms are not generated.

For a class table, visibility can be specified in columns Analysis Items / Visibility and Ada95 Items / CA Visibility. If the Ada 95 visibility is not empty, it supersedes the entry under Analysis Items. (Note that *protected* from the Analysis Items / Visibility is not implemented). Three visibility options can be selected, as shown in Table 7 below.

Table 7: Visibility Options

Visibility	Get and Set subprograms are declared
Public	In the visible part of the specification
Private	In the private part of the specification
Implementation	Not at all

The visibility from the class table can be overridden by the tagged value Visibility for the attribute. This allows fine control of the visibility. The value has the form “location-restriction”, where location determines the place where the subprogram specifications are generated according to Table 8 below.

Table 8: Visibility Overrides

Location	Get & Set subprograms are declared
Implementation	Not at all
Public	In the visible part of the spec
Private	In the private part of the spec
Public-child	In a public child package
Private-child	In a private child package

The restriction may be one of the following shown in Table 9 below.

Table 9: Restrictions

Restriction	
None	Equivalent to operation
Operation	Both Get and Set operations are generated
Read-only	Only a Get function is generated

The meaning of the tagged value Visibility is given in Table 10 below.

Table 10: Tagged Value Visibility

Visibility	Get	Set
Implementation	-	-
Public	Visible specification	Visible specification
Public-operation	Visible specification	Visible specification
Public-read-only	Visible specification	-
Private	Private part	Private part
Private-operation	Private part	Private part
Private-read-only	Private part	-

Public-child	In a public child	In a public child
Public-child-operation	In a public child	In a public child
Public-child-read-only	In a public child	-
Private-child	In a private child	In a private child
Private-child-operation	In a private child	In a private child
Private-child-read-only	In a private child	-

Stereotypes

The following stereotypes are supported for attributes:

- Discriminant implements the attributes as discriminant.

Tagged Values

The following tagged values are supported for attributes:

- Visibility - Controls where Get and Set operations are defined.

Annotations

No annotations are supported for attributes.

Operations

The code generator recognizes the syntax “classname*” for parameter types, and substitutes “classname_Cptr”, where asterisk (*) means a class-wide pointer, a non-dispatching operand for Ada 95.

Binding Techniques

Analogous to C++, Ada 95 supports three kinds of operations:

- Static

  Static operations are those that have no implicit operand of the class that defines them. The parameter profile is exactly as specified in the class table.
- Normal

  In normal operations there is an implicit operand of the class, but this operand is *not* controlling. For example, calls are never dispatching. The parameter type is a named access type.
- Virtual

  Virtual operations implicitly have a controlling operand “acc_this” of the class. The parameter is defined as an access parameter.

Choosing a Binding Technique

Selecting the binding technique for an operation is not very intuitive for Ada 95. The primary reason is that the decisions are made in the language independent part of the ACD code generator. This could only be changed by introducing a special version of the common part of ACD.

The binding of an operation is selected in the class table. Using columns Class Op? from Analysis Items and Virtual? from C++ Items with meanings as shown in Table 11 below.

Table 11: Binding Technique

Class Op?	Virtual?	Binding	Code
False (default)	False (default)	Normal	Op(acc_this : class_ptr; p1...)
True	False	Static	Op(p1...)
False	True	Virtual	Op(acc_this : access class; p1...)
True	True	Undefined	

Stereotypes

The following stereotypes are supported for operations:

- Entry - The operation is implemented as a protected entry.

Tagged Values

Table 12 shows the tagged values supported for operations.

Table 12: Supported Tagged Values for Operations

Name	Type(default)	Meaning
Timed_Call_Supported	Boolean(false)	Generate interface subprogram for timed calls

For details, see “Timed Entry Calls” in the section called “Entry Implementations” on page 3-29.

Annotations

No annotations are supported for operations.

Relations

There are three different kinds of relations between classes:

- Associations
- Aggregations
- Compositions

The name of the relation is based on the model according to the following rules, in the order given:

1. If their association end (at the *other* end) has a name, then this name is used
2. Else, if the relation has a name, this name is used
3. Else, the name of the associated class is used

For associations, the name is <*relation_name*>_asc. For aggregations and compositions, the relation is named <*relation_name*>_part.

For associations and aggregations, a record component is generated that is a pointer to the other class or a pointer to an array of such pointers, if the multiplicity of the relation is more than 1.

For compositions, the component is a class object rather than a pointer or an array of such objects, depending on the multiplicity.

The access of related class objects is supported by a set of subprograms. The place where these subprograms are declared depends on the selected visibility (see "Visibility" on page 3-25).

For relations with multiplicity 1, a pair of Set and Get operations similar to attributes is generated. For other multiplicities, the Set and Get operations have an additional index parameter. In addition, a set of operations to count the related objects and add and remove new associations is generated. This supports direct access to related objects and allows iterations with *for loops*.

The Aggregation Stereotype: Owner

The stereotype Owner for aggregations implies that the navigable end of the association is owned by the class at the other end. That is, when the owner is finalized, the other class is implicitly finalized and deallocated. Thus, an aggregation with stereotype Owner is similar to a composition, but not as restrictive. The big difference is that a class that is part of an aggregate can also exist outside this aggregate.

Note: For this concept to work properly, it is mandatory that no references to the owned class objects are kept other than from the owner. This is your responsibility as a modeler.

The Aggregation Stereotype: Counted_Ref

The stereotype Counted_Ref applied to an aggregation implies that the navigable end of the association implements a package that supports the concept of counted references. This kind of relation is implemented with this kind of reference instead of the access type.

These counted references (also called smart pointers) are controlled types that implicitly increment a reference counter when the counted reference is copied and decrement it when the reference is finalized. When the reference counter reaches 0, the referenced class object is itself finalized.

The concept of counted references can be used even if there is no relation with stereotype Counted_Ref. This is achieved by adding the tagged value Counted_Class_Ptr to the class definition. The value is a boolean and defaults to True.

The operation Create converts the access type normally used to reference the class objects into a counted reference. The operation Read can be used to the reverse conversion. The intent is that the result of Read should not be stored anywhere but should be used immediately as an input parameter in subprogram calls.

Note: The code generated for the class (implicit operations such as Create, as well as explicitly defined operations) uses the conventional access type as for other classes.

Visibility

The component declaration for a relation is generated in the tagged record, which is in the private part of the package specification. For tasks and protected types, where no enclosing tagged record exists, the component is a variable declaration in the corresponding task body or private part of the protected type, respectively.

There are two methods to control the generation of Get / Set subprograms:

- A tagged value for the whole class
- An individual tagged value for the relation's association end

If the class has the boolean tagged value Relation_Operations set to False, none of its relations are visible. They are all implemented as *implementation*, regardless of the individual tagged values for the relation. Therefore, no Get/Set subprograms are generated.

The use of the tagged value Visibility for the association end allows fine control of the visibility. The value is as described for attributes.

Note that for a relation from A to B, the Get/Set subprograms are generated in A, depending on the Visibility specified at the association end B.

Stereotypes

The following stereotypes are supported for relations:

- Owner - Implicit deallocation of aggregate upon finalization
- Counted_Ref - Reference counts used for aggregates

Tagged Values

The following tagged values are supported for relations:

- Visibility - Controls where Get and Set operations are defined.

For details, see "Visibility" above.

Annotations

Annotations are not supported for relations.

State Machines

In contrast to many code generators from other CASE tools, ACD has the ability to transform state chart diagrams into finite state machines. In the Ada 95 implementation, state charts (and tables) are translated into four possible variations of a finite state machine (FSM).

UML Specification 1.3 says in Section 2.14.2 (execution semantics of a state machine) about a hypothetical state machine:

> In the general case, the key components of this hypothetical machine are:
>
> • an *event queue* which holds incoming event instances until they are dispatched
>
> • an *event dispatcher mechanism* that selects and de-queues event instances from the event queue for processing
>
> • an *event processor* which processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question.
>
> Because of that, this component is simply referred to as the "state machine" in the following text.

Ada 95 itself does not incorporate asynchronous events or a message passing paradigm. With the Ada 95 code generator, the support of asynchronous events works as follows:

- For each of the classes with the tagged value "Gen_Signal_Handler=true", a child package class_name_signals is generated. This package contains the interface to send events to that class. (The default is not to generate this interface because the feature is not applicable for all state machines (see "Event Parameters" on page 3-31). If you make heavy use of asynchronous events, you might want to change the default in the templates.)
- The event queue is implemented by the packages ACD_Runtime.Signals, ACD_Runtime.Queued_Signals and ACD_Runtime.Signal_Queue which are always generated.
- The event dispatcher is a task in the body of the main package main_pkg; it is always generated.

If the signal handling described above does not fit your needs, you can implement your own methods to collect events and route them to the event processing method.

This way, ACD supports asynchronous events as well as synchronous (call) events.

But there are some features that are only applicable to call events: *out* parameters for events and the possibility of blocking calls with the Entry_Wait mechanism (see "Entry Implementations" on page 3-29). Hence, we can better support the real-time features of Ada 95 with synchronous events.

ACD supports the notion of composite states (substates, superstates) including shallow and deep history states. There is a limited form of activities (see "Unsupported Features" below).

Unsupported Features

We do not support the following features of UML state machines:

- Concurrent subregions
- Activities (as an independent, abortable flow of control, discussed in this section below)
- Inheritance of state machines

There is some limited implementation of activities. At each transition, some *activity code* is executed. This form of activity cannot be aborted in case another event comes in. Rather, the state machine is blocked until the activity is completed.

Inheritance of state machines is not really supported. If both a class X and one of its superclasses, S, have a state machine, class X will have two totally independent state machine implementations. We could say that class X inherits the whole state machine from S. But this will only work if they are totally independent. For more details, see "State Machines and Inheritance" on page 3-35.

Implementation Variants

The implementation of state machines can be selected by the tagged value State_Machine_Implementation. The Table 13 below illustrates the possible implementation techniques.

Table 13: Implementation Techniques

Table	One operation Take_Event with table access
Case	One operation Take_Event with case over State and nested case over event
Entry	(Protected Type / Task only) One entry per Event, discard events in wrong state
Entry_Wait	(Protected Type / Task only) One entry per Event, wait for state able to handle an event (violates UML semantic!)
None	Suppress generation of state machine

Table is incompatible with event parameters (Case is used).

Entry and *Entry_Wait* can only be used if the class is a protected type or task / tagged_task. Note also that the presence of a timeout event or an activity implies that the class is mapped to a task.

The technique Entry_Wait violates the UML rule requiring that unhandled events be silently discarded. In contrast, the calling task is blocked. The situation is similar to deferred events. In fact, this means that with Entry_Wait, all unhandled events are implicitly defined as being *deferred* (which is allowed by the UML definition).

The default is Case if there are event parameters, Table otherwise.

Entry Implementations

The Entry and Entry_Wait implementations generate a case over the current state and for each state a selective wait for all entries that are accepted.

Entry generates accepts for all events, even those that are unexpected by the state machine in the current state. An error message is produced in case such an event is received. Entry_Wait does not generate these accept alternatives, so that the code is usually much shorter and easier to read. However, the caller is blocked in that case, which means that the calls are being deferred.

Timed Entry Calls

With the Entry_Wait model, it makes sense to work with timed-entry calls. In order to make this easier, a tagged value, Timed_Call_Supported, is defined for operations mapped to call events. If this tagged value is set to true for an operation Op, a second subprogram Timed_Op is defined with the additional parameters “Timeout:Duration” and “Timed_Out:out Boolean”. The latter indicates whether the call timed out or the call was accepted. In case of operations mapped to functions, an access boolean parameter has to be used instead of the *out* parameter, which is an illegal construct.

For the *out* parameters of the operation, default assignments are implicitly generated in case the entry call was not performed. Where the type is known, null or <*type_name*>'last is used as a default. Otherwise, <*type_name*>_Default_Value is used.

Note: To avoid constraint errors during parameter passing, make sure that a suitable declaration is visible.

Action Mapping

You can select the way the actions presented in the state diagrams are mapped to the code by setting a tagged value Use_State_Machine_Code for the class. The default is Comment.

Table 14: Allowed Values for Action Mapping

Comment	Place the code in the model as comment into the action procedure which is filled by the user (mergeOut)
Ignore	Do nothing with the code in the model, but generate a stub for each action to be filled by the user (mergeOut)
Stub	Generate a stub for the actions with code in the model only, which is filled by the user (mergeOut)
Code	Use the code in the model directly as Ada code

Note: Stub is equivalent to Ignore, but avoids producing empty procedures.

Substate Entry Actions

Normally, actions are specified in the State Table. In some StP releases prior to version 8.3, this type of specification did not work for substates. Therefore, ACD supports the tagged value Entry_Action as an alternative method of specifying entry actions for a substate.

Note: The use of this feature is strongly discouraged since it is no longer supported.

Events

Events are defined in state diagrams and state tables (the signals received section of the class table is ignored by the code generator). In UML, there are two methods to send events to a class:

- Asynchronously, using the signal mechanism
- Synchronously, via call events

Event Parameters

There is currently no way to specify the parameter profile of an event directly. Events are declared through their occurrence in a transition. Therefore, events are, by default, considered as parameterless.

If an event and an operation have the same name and the event defines no explicit parameters, they are said to match and the event implicitly is assigned the same parameter profile as the operation. See “Call Events and Operations” below.

It is recommended that for each event, a matching operation be defined in the model with the desired parameter profile.

Signals

If the model contains an active class and the tagged value Gen_Signal_Handler is set to True, a child package is generated that implements signals. This implies a queuing mechanism that allows signals to be delivered asynchronously.

Note: With the Entry and Entry_Wait implementations, this is not necessary (and the child package can simply be ignored).

Note: This signal package is semantically incorrect if there is an event with an *out* parameter. Gen_Signal_Handler should be set to false in this case.

Call Events and Operations

If an event and an operation have the same name and parameter profile, the operation is interpreted as a call event. The operation is implemented as a synchronous event for the state machine. If the state machine is implemented as Entry or Entry_Wait, the operation is implemented by an entry call.

Note: Operations that do not match an event are only useful if they are class operations. In this case, the subprogram body can be supplied by the user and might contain things like asynchronous select statements or timed entry calls. But timed calls are supported directly (see "Entry Implementations" on page 3-29).

If you want to define dispatching operations that are not mapped to entry calls, they must be defined as class operations with an explicit parameter of the tagged type. This is particularly useful for internal operation that are only called from within the task body. See also "State Machines and Inheritance" on page 3-35.

Non-matching operations that are not class operations are of limited use. Since they are mapped to an entry where no accept statement is generated, it would be quite difficult to manually insert the proper accept statements into the structure generated for the state machine.

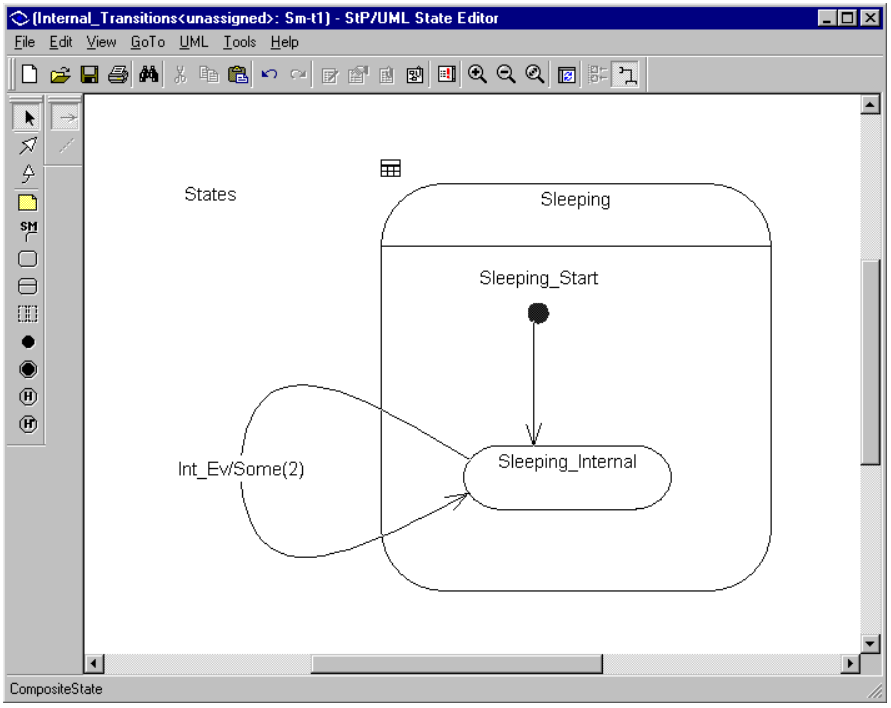
Completion Events and Time Events

To specify a completion event, simply omit the trigger event in the transition.

Time events can be specified with the syntax "after(duration_expr)". This means, of course, that no call event can have the name "after".

Internal Transitions

The current version of StP / ACD does not support internal transitions.



As a workaround, the following approach can be used. Transform the state into a composite state with one substate. An external transition from the substate to itself is equivalent to an internal transition of the superstate. If the original state has an activity, it must be moved to the substate.

Guard and Action Syntax

Guards and actions can be specified in notation similar to C++. There is a simple text transformation to change this to Ada syntax (TDL procedures genGuard, evaluateAction). The code generator transforms this program text into Ada 95 by applying these rules:

- Operator syntax

  C operators '=', '!=', '==', '&&', '||', '!' are mapped to the Ada operators ':=', '/=', '=', 'and', 'or', 'not'.

- Syntax of subprogram calls

  The C++ syntax 'role.op(p1,p2)' is transformed into 'role_pck.op(this.role,p1,p2)

- Access to attributes

  'getAttr' is replaced by 'getAttr (this)',

  'setAttr (Val)' is replaced by 'setAttr (this, Val)'.

- Access to associations

  'getRole' is replaced by 'getRole_Part(this)',

  setRole(Val)' is replaced by 'setRole_Part(this, Val)', and so on for the operations Get, Set, Count, Add, Remove, and Alloc.

The details of this evaluation can be found in TDL procedure evaluateOcl in file *ada_gensm.tdl*. If you are an advanced user, you may want to adapt this procedure to your specific needs.

This evaluation is strictly based on pattern matching. The reason for implementing this text transformation is that it allows you to generate code from the same model for different target languages. The transformation only works for simple cases. If the evaluation is not desired, it can be disabled by returning the argument string without any changes.

If the text is more difficult, the action mapping Comment can be used. Of course, it is also a good idea to call local subprograms that implement the more complicated actions.

Superstates

Composite states (superstates, substates) are supported, including both shallow and deep history states.

State Machines and Inheritance

Currently, no true inheritance mechanism is supported. If class B is derived from Class A, B inherits the state machine implementation from Class A, e.g., the task that implements the state machine.

If class B has a state machine as well, this machine is totally independent from the state machine inherited from A. Thus B would have two independent tasks. In most cases, this does not make sense.

If both the superclass and a subclass define a state machine, the templates do *not* implement UML semantics. Therefore, having state machines in both classes is discouraged.

There are, however, two useful schemes for using state machines together with inheritance.

Superclasses with State Machines

A state machine in a superclass can be used to define the common behavior of the subclasses. There is no way to modify the inherited state machine within the subclasses, but the subclasses can still have additional attributes and operations.

Note: These additional operations are not synchronized with state machine operations.

In particular, it is possible to define private dispatching operations which are called from the state machine. One problem here is that these operations are normally mapped to state machine operations in the superclass. If the superclass is abstract, this mapping can simply be ignored (although it might be good idea to suppress this in the templates).

Otherwise, it is a bit tricky. In order to avoid the mapping of these operations to state machine call events (entry calls), they must be defined as static (i.e., non-dispatching) operations in the class table but with a

parameter profile that makes them dispatching anyway. The parameter should be defined with a data type that is implemented as classname_pkg.classname. This assures that the code generator does not attempt to map class name to anything else.

Subclasses with State Machines

A common superclass can be used to define a common interface and common data, while the subclasses define their individual state machines. It would make sense to map some of the inherited operations to call events of the state machine in the subclass. But it is up to you to make sure that the subclasses show a consistent behavior.

4 Generating C++ Code

Introduction

This chapter discusses C++ code generation with ACD from models created with StP/UML 8.x.

Besides standard elements such as code skeletons for classes, with its attributes (including get and set methods, constructors, and operations), the code generator also supports implementation of state machines based on state diagrams.

All these additional features are optional and only created if not already in the model. Furthermore, in StP/UML 8.3, you can generate code selectively for single or multiple:

- Class diagrams
- Class tables
- Packages
- Individual classes

Compatibility

With Language Specifications

The generated code is, in general, compatible with the standard, Programming Languages - C++, ISO/IEC 14882:1998.

With Previous Code Generators

The generated code is, in general, not compatible with code created by previous versions of the C++ code generator. Particularly, some of the additional user-defined sections have changed their signatures, such that the merge process cannot merge source code produced by newer versions with code originating from older generators.

General Mechanisms

Before we look into classes and state machines in greater detail, we discuss:

- Which model elements determine code generation
- Invocation of the code generator
- How to enrich the model with additional information

Model Elements That Determine Code Generation

While ACD can access other model elements, e.g., Use Cases, Actors, and Requirements, only the elements listed below are relevant in C++ code generation:

- Classes
- Attributes
- Operations
- Associations, generalizations, dependencies
- State machines
- Annotations

Invoking the C++ Code Generator

The C++ code generator is invoked for either of the following:

- The whole model
- Individual model elements or files

To invoke for the whole model, select **Code > C++ > Generate C++ for Whole Model by ACD Template** from the StP Desktop.

To invoke for individual model elements or files, choose one or several model elements from either the class diagram view, class table view, package object view, or class objects view. Then select **Generate C++ for** <*element*> **by ACD Template**, <*element*> being one of class diagram, class table, package, or class.

Adding Information to the Model

We discuss here briefly, what input facilities you, as the modeler, can utilize.

In StP / UML there are four mechanisms to add information to a model:

- Class and state diagrams
- Class and state tables
- Various property sheets
- Object Annotation Editor (OAE)

Usually you begin with a diagram and then enrich your model elements with specific notes and items. For a class, you normally use the class table editor, with its general section (Analysis Items) and language specific sections (in this case, C++ Items).

For some code constructs, you may need to use the property sheet for an object to type set the appropriate stereotypes and tagged values.

Overview of the Templates

This section presents a brief overview of how the C++ templates generate code and how a UML model needs to be set up and annotated for C++ code generation.

Templates for C++ are shipped with StP. Although compilable and executable code can be generated using these templates out-of-the-box, you are encouraged to extend and modify the templates according to the specific needs of your project. Please refer to the *ACD Programming Guide* for details on the template definition language and the ACD technology.

A complete example, *Millennium_Clock_CPP*, is provided with the installation, together with pre-generated source code, including some minimal user-defined code fragments so that the Millennium_Clock application can be built and executed. Project files for Microsoft VisualStudio 6.0 are also provided. (See *<StP>/examples/Millennium_Clock_CPP/src_files.visual_cpp*.)

See also Appendix C, “C++ Notes”.

Common Code Fragments

This is an overview of code artifacts that are common to every C++ source file.

Default Files

The following support files are generated in general (without direct mapping from or influence by the model):

- *baseclasses.h*

  This file includes stub declarations for the common superclasses required by the state machine implementation (ActiveInstance and InactiveInstance). This file is included by every header file generated for the model.
- *classes.h*

  This file lists all classes defined by the model and is also included by every header file generated for the model

- *main.cpp*

  This file is a static default main() implementation:

```
#include <stdio.h>
#include <string.h>

//#ACD# M(UDIF) Include Files

    // user defined code to be added here ...

//#end ACD#

void Applic_Init() {

    //#ACD# M(UDSC) StartUp Code

        // user defined code to be added here ...

    //#end ACD#
}

int main(int argc, char * argv) {

    Applic_Init();

    //#ACD# M(UDMC) Main Program Code

        // user defined code to be added here ...

    //#end ACD#
    return 0;
}
```

Header Files

The header part of all **.h* files looks like this (assuming the class name is Class_A):

```
#ifndef _Class_A_h
#define _Class_A_h

#include "classes.h"
#include "baseclasses.h"
```

```
//#ACD# M(UDIF) includes

    // user defined code to be added here ...

//#end ACD#
```

Implementation Files

The header part of all **.cpp* files looks like this (assuming the class name is Class_A):

```
#include <iostream.h>
#include "Class_A.h"

//#ACD# M(UDIF) includes

    // user defined code to be added here ...

//#end ACD#
```

Classes - General File Structure

Header Files

Following is a list of the components of a header file for a C++ class generated with ACD (subsequent sections provide more detail):

- Class declaration statement
- Constructors and destructors, if modeled, or creation of default constructors and destructors
- Static or constant attributes, if present
- Instance attributes, if present
- Get / set methods for attributes
- Pointers for associated classes
- Modeled operations, if present

After the sections for attributes and methods, there are user-defined sections for inserting your own attributes and methods. User-defined sections are in general not modified when you generate code more than once (commonly referred to as incremental code generation).

Implementation Files

Implementation files generally define the method stubs together with user-defined sections, where user-defined code can be inserted. The code for state machines is also generated into implementation files.

Inheritance

A C++ class can have a generalization relationship to one or more superclass(es). The generalization to the superclass generates an inheritance statement in the class declaration. The privilege of the generalization as set in its properties is taken into account, as well as the flag for a virtual inheritance.

In order to support code generation for state machines, a class that does not have a modeled superclass automatically inherits from a default superclass InactiveInstance.

Dependency Relationships

Friend Classes

C++ classes can have a friend relationship to other classes. This is modeled by creating a dependency relationship and setting its stereotype to “friend”.

Note: According to the UML definition, the target class of the dependency relationship is the class for which the templates generate the friend statement.

Attributes

Attributes describe the static features and the state of a class. In C++, attributes can be:

- *Instance* attributes, which belong to exactly one class instance and which are exclusively used by this instance
- *Static* class attributes, which are initialized only once for every class

With ACD, you can generate attributes using two kinds of model elements:

- *Real* attributes from model
- Attributes from associations

Attributes from Model

All possible declarations for a C++ attribute can be expressed in StP. As usual, you model attributes in class diagrams and add specific information such as type, visibility, and so on, using the class table editor.

Use the sections of the class table editor to add detail modeling information to an attribute.

Class Member Definitions

For every attribute, specify the name, data type, and default value.

Analysis Items

For every attribute, specify the access privilege and whether or not it is a class attribute. The possible visibility values here are the general values for UML: public, protected, private. By default, attributes are non-static. To create static attributes, the class attribute property needs to be set to true.

C++ Implementation Items

Create *const* attributes by setting the Const? property to True. Additionally, you can override the visibility of Analysis Items with the visibility column in this section.

Attributes from Associations

Associations are the second source for attributes. Classes, which have a navigable association from the current class, are referenced in the current class as an attribute.

Note: In StP, associations are bidirectional by default.

In StP you model associations as usual, drawing an association or aggregation link between two classes and filling in the various properties with the property sheet for associations. Figure 1 below shows an example of various aspects of associations.

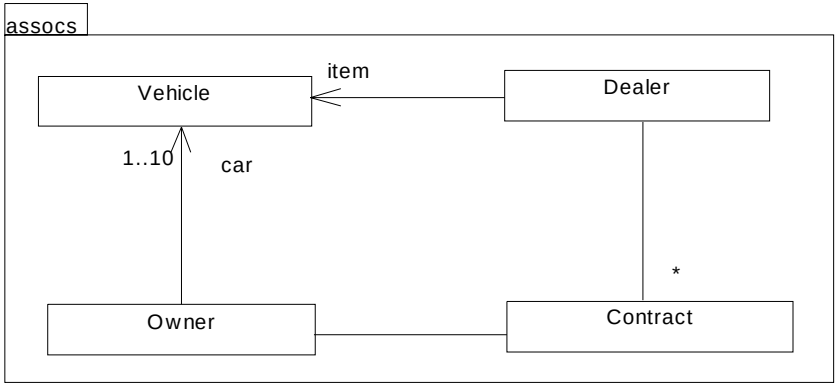


Figure 1: Associations

In the source code the name of a reference attributes is determined either by the name of the opposite role or, if role names are missing, the name of the opposite class. Both names are prepended by an underscore. Attributes generated for associations are always C++ pointers.

If the multiplicity of the opposite role is > 1, the type of the reference attribute changes to an array with the base type of the opposite class. The array dimension is set to the range of the multiplicity of the association role and defaults to a value of 100 if the range of the association role is set to "*" (meaning many).

Association pointers are always generated into the public part of the class declaration.

Note: In the source code, no differentiation is made among associations, aggregations, and compositions. Pointers are generated for all three relationships.

Operations

Operations or methods describe the behavioral features of a class and how the class reacts to messages from the outside. In C++, they can be either instance operations, which act upon a specific instances state, or static class operations, which deal only with static class attributes.

Modeled operations lead to a *method declaration* in the source code header files and implementation stubs in the implementation files. Operations can have zero, one, or several formal parameters and zero or one return values.

Using the Class Table Editor

As with attributes, you usually model operations in a class diagram and add detailed information in the class table editor later.

Class Member Definitions

For every operation, specify the name, the return type, and the operation parameters. Those defined with an *in* flag are generated as *const* arguments. If the return type is missing from the model, the C++ return type of the operation defaults to *void*.

Analysis Items

For every operation, specify access privilege, the operation is abstract and it is a class operation (which maps to an inline method). If the visibility is not modeled, the operation defaults to *public*.

C++ Implementation Items

For every operation, you can override the privilege and inline property from the analysis items section, as well as setting it to a *const* or *virtual* method.

Get and Set Methods

For each attribute, a public get method, get<*AttName*>, and a protected set method, set<*AttName*>, is generated. These methods are generated as inline methods.

Constructors and Destructors

Unless specifically defined in the model, default constructors and destructors appear in the class declaration code before all other operations. Destructors are always generated as *virtual*.

State machines

This section addresses UML state machines.

The current implementation maps a final state machine into a structure of nested switch / case statements that is implemented in a method called takeEvent.

Refer to the StP example system *Millennium_Clock_CPP* for a complete overview and example of how a UML state machine maps to C++ code in the current implementation of the templates. In particular, note the state diagram Clockwork__1 and the generated C++ file *Clockwork.cpp*.

See Appendix C, “C++ Notes” for an example diagram together with an implementation file.

Initialization of the State Machine

A state machine has a creation state that is used as an entry point. This state is not a real state but rather a pseudo state. That is why a static public method called classEvent is declared to trigger the initialization of the state machine. The task of this method is just to call the takeEvent method with the first event for the state machine.

States

Each UML state maps to an *enumerated value* which is used to define an *enum* type objStates. If a creation state has no name, an enumeration value, _Init, is added.

Example:

```
enum objStates
{
_Init,
State1,
State2
};
```

Each class, which has an associated state machine, inherits from the class ActiveInstance, which declares an attribute named currentState. This attribute is used to store the current state value in the takeEvent handler.

Events

Each UML event defined on a transition maps to an *enumerated value* that is used to define an *enum* type objEvent. If two transitions have the same event name, they appear only once as an enumerated value.

The takeEvent method is generated in order to process an event depending on the current state value. This method represents the core of state machine processing and is declared in the public part of the class definition.

Event Handling

This subsection addresses event handling, which is the core of the state machine. Event handling corresponds to the body of the method takeEvent. The event to handle is a parameter of this method. Its value is one of the enumerated values of the type objEvent. This is the pseudo code of the core method takeEvent:

```
void <ClassName>::takeEvent (const <ClassName>::objEvent& ev)
{
    switch(currentState) {
        case <ClassName>::<StateName>:
            switch(ev) {
```

```
                case <ClassName>::AutomaticTransition:
                    break;
                case <ClassName>::<EventName>:
                {
                    if (<Guard>) {
                        <Action1>;
                        <Action2>;
                        ....
                        currentState = <TargetStateOfTransition>;
                        <TargetStateOfTransition>_Action();
                    }
                }
                break;

                default:
                    cout << "illegal event " << ev << " in state " <<
                         currentState << endl;
                    break;

            } // end of switch event on SolidHourAtNight
            break;

...

        default:
            cout << "illegal state " << currentState <<endl;
            break;

    } // end of switch current state
}
```

For all states, all transitions starting from the state are generated. For non-automatic transition, a case is generated to compare the parameter *event* of the method takeEvent with the transition event name. Depending on the current state value, for each non-internal transition, the takeEvent method performs the following tasks:

- Compares the event (if any) defined on the transition with the one given as a parameter.
- Calls the actions defined on the transition depending on optionally existing guards.
- Sets the currentState attribute to the new target state.

- Executes the entry action defined on the new current state (target state).
- Calls the do-activity that is defined in the transition.

Guards on State Transitions

Guard conditions defined on state transitions can be used to prevent a transition. This happens when the boolean guard expression evaluates to False or the method call returns False.

Actions on State Transitions

Actions defined on a state transition are generated into the innermost part of the case statements (after all events and guards have been considered). A default case is generated to flag an illegal event as well as a default automatic transition case.

A default method is generated to support the entry action for each state. There is no specific generation for internal transition, exit, or action on transition.

A Java Notes

The Structure of the Code Generator

The Java code generator in ACD is highly modular and centers around the phases of generation: initialization and code creation.

All initialization files reside in a directory called *init*. The task of these files is to initialize elements of the ACD metamodel and their file names are derived from the metamodel element, e.g., *java_<MetaElement>.tdl*, e.g. *java_Mattribute.tdl.*

Files that create code reside in a directory called *code.* Names of these files are derived from the specific Java code element they create, e.g., *java_Interface.tdl* for creating an interface.

All helper and utility files reside in a directory named *util.*

Table 1 below lists all the Java code generator constructs together with a short description of what they do.

Table 1: Structure of Code Generator

File	Description
../tdl/std.tdl	Various basic procedures for file output, date/time functions and string handling. Language independent.
java_std.tdl	Various basic, Java-specific procedures
java_globals.tdl	Influences code generation via global variables
java_main.tdl	Main entry point for code generation

init/java_MClass.tdl	Initialization functions common to both MNormalClass and MInterface
init/java_MNormalClass.tdl	Initialization functions specific for an MNormalClass. Creation of elements at run time, like attributes, constructors, etc.
init/java_MInterface.tdl	Initialization functions specific for an MInterface
init/java_MAttribute.tdl	Initialization functions specific for an MAttribute (visibility, modifier, default values)
init/java_MOperation.tdl	Initialization functions specific for an MOperation (visibility, parameters, throws-statements)
code/java_Common.tdl	Generates the header of a class, interface, or exception. The header consists of general information like class name, copyright, package, and import statements.
code/java_Class.tdl	Generates a Java class and all its contained elements like attributes, methods, and inner classes
code/java_Interface.tdl	Generates a Java interface and all its contained elements like attributes, methods, and inner interfaces
code/java_Exception.tdl	Generates a Java exception (basically also a class) and all its contained elements like attributes and methods
code/java_Operation.tdl	Special code for operations (parameters, throws-clause, default return value)
code/java_Statemachine.tdl	Various utility and initialization functions for state machine code
code/java_Make.tdl	Currently empty, can be used for a traditional makefile
code/java_Ant.tdl	Creates makefile for ANT, the Java based make tool
util/java_import.tdl	Utility functions centered solely around import statements
util/java_MAssociation.tdl	Utility functions for associations (determines navigability, role visibility, etc.)
util/java_MStatemachine.tdl	Contains templates and procedure for the two state machine variants
util/java_string.tdl	String functions not found in *std.tdl*

Notes and Items used in Code Generation

Table 2 summarizes the notes and items used as annotations in code generation. Only those annotations that are actually used and can be reached exclusively by the OAE are listed.

Table 2: Notes and Items

Object	Note	Item	Item Value	Description
Class	Java Declaration	Note Description Dialog	Lets you add code for Java-specific constructs scoped to the class that are not represented by UML constructs. You add the text as a Note Description.	
	Class Definition	Enclosing Scope	Specifies the name of a package enclosing the parent class	
	Class Java Definition	Static Class	Specifies the class as a *static* inner class	
		Visibility	private / package private / protected / public	Lets you specify the Java specific visibility. The entry *package private* appears here in addition to the three values for UML.
		Final	Specifies if this class can be subclassed or not	
Operation	Java Source Code	Note Description Dialog	Lets you add Java source code for the operation	
Role	Role Definition	Multiplicity	Specifies the multiplicity of the role. Default is 1, if nothing is specified.	
		Role Navigability	True / False	If True, a reference to the class in the direction of the association is inserted in the source class
	Role Java Definition	Visibility	public / private / protected / package private	Lets you indicate the Java field access of constructs used to implement associations between classes

Stereotypes and Tagged Values

Stereotypes and tagged values can influence code generation. Stereotypes are often used to change larger aspects of code generation (patterns, class-types, and so on). Tagged values are primarily used to override global variables for individual elements.

Table 3 below summarizes all the stereotypes and tagged values used in Java code generation. The last column shows the allowed values (only applicable to tagged values), with the default value in bold type.

Table 3: Stereotypes and Tagged values

Model element	Extension	Type	Values
Class	interface	Stereotype	
Class	exception	Stereotype	
Class	main	Stereotype	
Attribute	readonly	Tagged Value	**false** / true
Attribute	access	Tagged Value	**true** / false
AssociationEnd	collection	Tagged Value	Any collection class

Global Variables for Code Generation

The code generator uses many global variables that can influence specific aspects of code generation.

In the Java code generator all these variables reside in the file *java_globals.tdl*, located in directory *<StP>/templates/uml/qrl/code_gen/java*.

All global variables that you can modify are all uppercase. Explanations of the variables can be found in the file *java_globals.tdl*.

Table 4 below summarizes all global TDL variables together with their default values.

Table 4: Global TDL Variables

Variable	Default value
AUTHOR	YOUR_NAME
COMPANY	YOUR_COMPANY
JAVA_EXTENSION	java
EXCLUSIONS	TRUE
ACCESSOR_METHODS	TRUE
DEF_ATTR_VISIB	private
DEF_OP_VISIB	public
DEF_CLASS_VISIB	public
DEF_ROLE_VISIB	protected
COMMENT_CLASSES	TRUE
COMMENT_ATTRIBUTES	TRUE
COMMENT_METHODS	TRUE
CREATE_DEF_CTOR	TRUE
CREATE_FULL_CTOR	FALSE
DEF_CTOR_VISIB	public

CREATE_FINALIZER	TRUE
CREATE_ABSTRACT_IMPL	FALSE
SET_CLASS_ABSTRACT	FALSE
CREATE_STANDARD_OPS	TRUE
DEF_COLL_CLASS	Vector
CREATE_MAKEFILES	TRUE
MAKEFILE_TYPE	ant
ANT_NAME	build
ANT_EXT	.xml
ANT_FILE_NAME	(derived from the previous two)

Source Code Example

This section provides an example diagram (see Figure 1 on page A-7) that shows all possible Java class types:

- Interfaces
- Classes
- Exceptions

The preamble and the user-definable sections are shown, which are normally omitted throughout the manual. Furthermore, inheritance and interface implementation is shown. Note the class Color. The constructor in it throws an exception WrongColorIndex, which is properly reflected in the code.

The generation was started with the following changed TDL global, [CREATE_ABSTRACT_IMPL], which was set to True. This ensures that the abstract methods in AbstractApp are automatically implemented in GraphEditor.

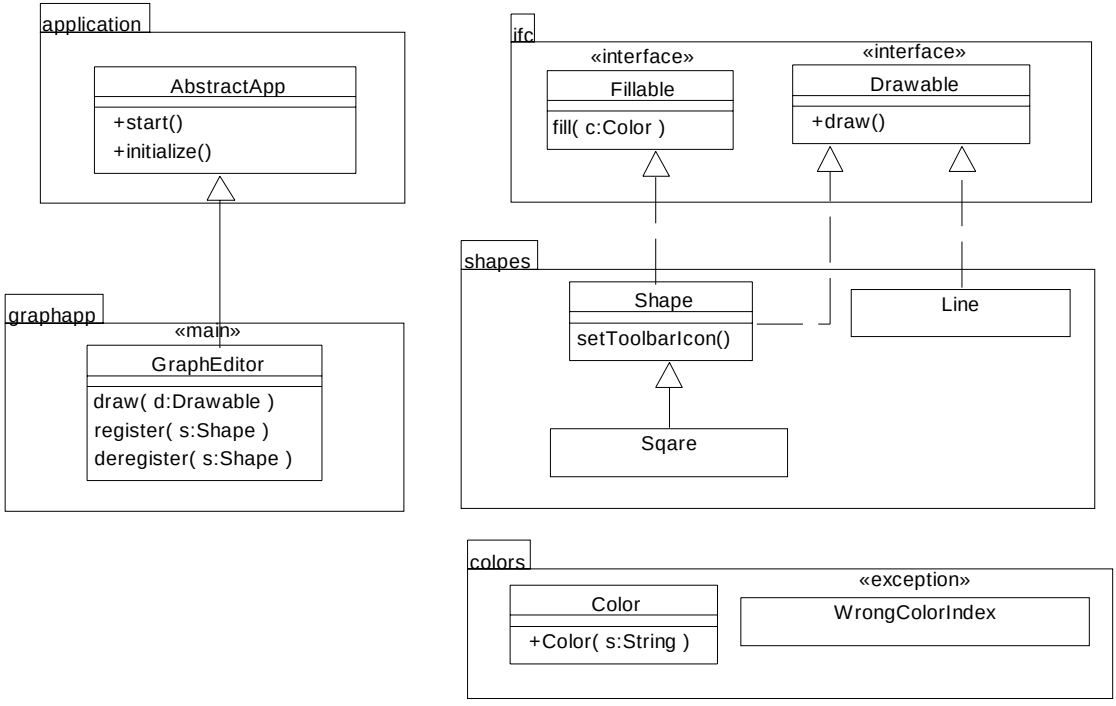


Figure 1: Class Diagram

The class diagram in the figure is constructed to show various aspects and features of the code generator.

An example of compilable and executable Java source code is available as part of the *Millennium_Clock_Java* StP example system. The code is completely generated based on this UML model and is located in *<StP>/examples/Millennium_Clock_Java/src_files.java.*

Provided that you have JDK 1.3 and the Apache/Jakarte ANT tool installed, you can compile this project using the generated *build.xml* file and then execute it in the build directory using the command:

```
java -cp . clockwork.Weight
```

B Ada 95 Notes

Source Code Example

An example of compilable and executable Ada95 source code is available as part of the *Millennium_Clock_Ada95* StP example system. The code is completely generated based on this UML model and is located in *<StP>/examples/Millennium_Clock_Ada95/src_files.ada95*.

Provided that you have an Ada 95 compiler, you can compile and link the *Millennium_Clock_Ada95* example code by compiling all the sources in the *src_files* directory tree.

Note that in the example source files we have provided user-defined code in the appropriate sections of the generated code. These code inserts are required for the example to execute correctly.

If you use the ObjectAda IDE on Windows, proceed as follows:

1. Create a new project.
2. Add the sources in *src_files.ada95/*.ad?* and *src_files.ada95/ACD_Runtime/*.ad?*.
3. Compile all sources.
4. Build the main program, main.

From the ObjectAda command line interface on both the Windows and UNIX platforms, use the commands:

```
cd <StP>/examples/Millennium_Clock_Ada95/src_files.ada95
adareg *.ad?
cd <StP>/examples/Millennium_Clock_Ada95/src_files.ada95/ACD_Runtime
adareg *.ad?
adabuild main
```

Structure of the Code Generator

Table 1 below shows the code generator template files.

Table 1: Code Generator Template Files

File	Purpose
std.tdl	Standard template file. See ACD manual.
ada_std.tdl	Support templates and procedures; stereotype and tagged value handling; package organization.
ada_main.tdl	Main procedure; loop over all classes
ada_genruntime.tdl	Generate ACD run-time environment
ada_genspec.tdl	Generate Ada 95 package specifications
ada_genbody.tdl	Generate Ada 95 package bodies
ada_genwith.tdl	Generate import of packages
ada_gentypes.tdl	Generate Ada 95 types for classes
ada_genconsdes.tdl	Generate predefined constructors and destructors
ada_genattr.tdl	Generate attributes and accessors
ada_genrel.tdl	Generate relations and accessors
ada_genop.tdl	Generate operations
ada_gensm.tdl	Generate state machine
ada_test.tdl	Test support for the templates
ada_gencom.tdl	Generate COM code

C C++ Notes

Source Code Example

An example of compilable and executable C++ source code is available as part of the *Millennium_Clock_CPP* StP example system. The code is completely generated based on this UML model and is located in *<StP>/examples/Millennium_Clock_CPP/src_files.visual_cpp.*

Provided that you have Microsoft Visual C++ 6.0 installed, you can compile and link the *Millennium_Clock_CPP* example code by loading the provided *Cuckoo.dsw* workspace file.

Note that in the example source files, we have provided user defined code in the appropriate sections of the generated code. These code inserts are required for the example to execute correctly.

State Machine Example

This section contains:

- A diagram of the state machine example discussed in Chapter 4, "Generating C++ Code"
- Sample code for the implementation of the takeEvent method in *Clockwork.cpp*

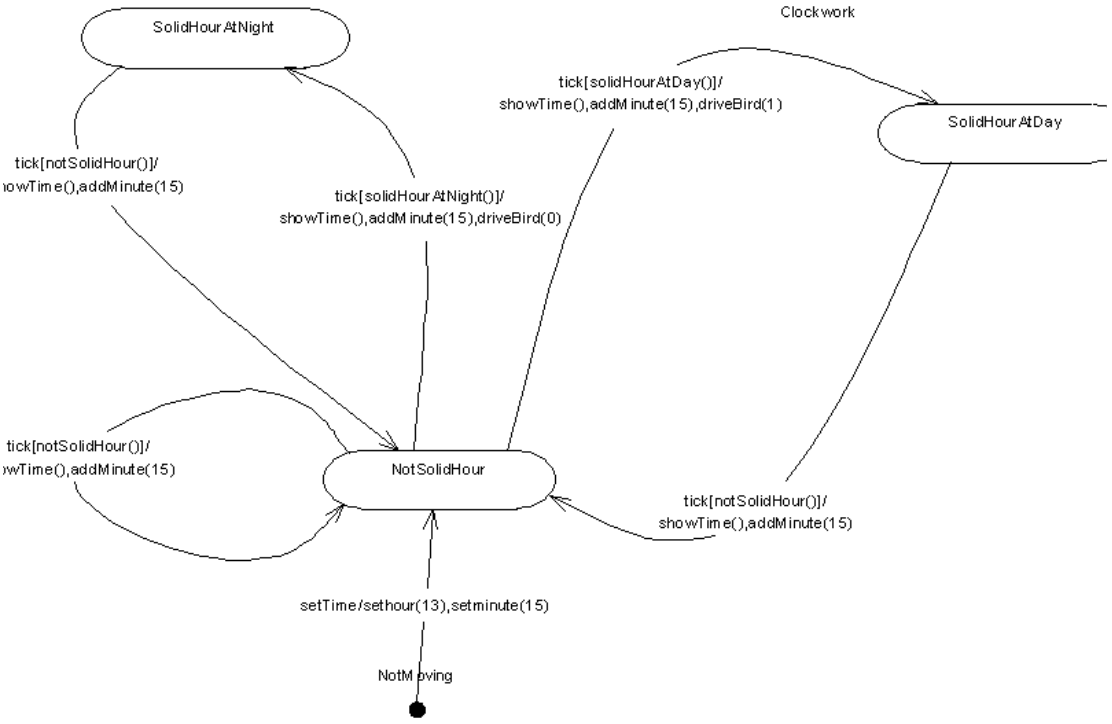


Figure 1: The UML State Diagram - Clockwork

Implementation of the takeEvent method in Clockwork.cpp:

```
void Clockwork::takeEvent (const Clockwork::objEvent& ev)
{
    switch(currentState)
    {
        case Clockwork::NotMoving:
            switch(ev) {
                case Clockwork::AutomaticTransition:
                    break;
                case Clockwork::setTime:
                {
                        sethour(13);
                        setminute(15);
                        currentState = NotSolidHour;
                        NotSolidHour_Action();
                }
                break;

                default:
                   cout << "illegal event " << ev << " in state " << currentState
<< endl;
                    break;

            } // end of switch event on NotMoving
            break;

        case Clockwork::SolidHourAtNight:
            switch(ev) {
                case Clockwork::AutomaticTransition:
                    break;
                case Clockwork::tick:
                {
                    if (notSolidHour()) {
                        showTime();
                        addMinute(15);
                        currentState = NotSolidHour;
                        NotSolidHour_Action();
                    }
                }
                break;

                default:
                   cout << "illegal event " << ev << " in state " << currentState
<< endl;
                    break;
```

```
            } // end of switch event on SolidHourAtNight
            break;

        case Clockwork::NotSolidHour:
            switch(ev) {
                case Clockwork::AutomaticTransition:
                    break;
                case Clockwork::tick:
                {
                    if (solidHourAtDay()) {
                        showTime();
                        addMinute(15);
                        driveBird(1);
                        currentState = SolidHourAtDay;
                        SolidHourAtDay_Action();
                    }
                }
                {
                    if (solidHourAtNight()) {
                        showTime();
                        addMinute(15);
                        driveBird(0);
                        currentState = SolidHourAtNight;
                        SolidHourAtNight_Action();
                    }
                }
                {
                    if (notSolidHour()) {
                        showTime();
                        addMinute(15);
                        currentState = NotSolidHour;
                        NotSolidHour_Action();
                    }
                }
                break;
                default:
                   cout << "illegal event " << ev << " in state " << currentState
<< endl;
                    break;

            } // end of switch event on NotSolidHour
            break;

        case Clockwork::SolidHourAtDay:
            switch(ev) {
                case Clockwork::AutomaticTransition:
                    break;
```

```
                case Clockwork::tick:
                {
                    if (notSolidHour()) {
                        showTime();
                        addMinute(15);
                        currentState = NotSolidHour;
                        NotSolidHour_Action();
                    }
                }
                break;

                default:
                   cout << "illegal event " << ev << " in state " << currentState
<< endl;
                    break;

            } // end of switch event on SolidHourAtDay
            break;

        default:
            cout << "illegal state " << currentState <<endl;
            break;

    } // end of switch current state
}
```