Software through Pictures®
Unified Modeling Language

Release 7.1

Generating and Reengineering Code

Software through Pictures Unified Modeling Language Generating and Reengineering Code

Release 7.1

Part No. 10-MN503/ST7100-01298/001

December 1998

World Headquarters
595 Market Street, 12th Floor
San Francisco, CA 94105
Phone: (800) 97-AONIX
Fax: (415) 543-0145
E-mail: info@aonix.com
http://www.aonix.com

Table of Contents

Chapter 3 Generating C++ Code

Preface

This manual describes how to generate code from models made using Software through Pictures / Unified Modeling Language (StP / UML).

This manual is part of a complete StP documentation set that also includes "Core" manuals: *Fundamentals of StP, Customizing StP, Query and Reporting System*, Object Management System and *StP Administration*. Basic information about the StP user interface is documented in *Fundamentals of StP*.

Intended Audience

The audience for this manual includes analysts and developers who wish to generate C++, IDL, Ada_95, TOOL, and Java code from StP / UML diagrams.

The information in this manual assumes you are familiar with:

- Fundamentals of Software through Pictures, 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) and *Unified Modeling Language Reference Manual* (Rumbaugh et al 1997)
- Your operating system directory and file structures
- Conventions of your UNIX window manager or Windows NT
- The target language for generating code

Refer to the appropriate documentation if you have any questions on these topics.

Typographical Conventions

This manual uses the following typographical conventions:

Table 1: Typographical Conventions

Convention	Meaning
Palatino bold	Identifies commands, buttons, and menu items.
`Courier`	Indicates system output and programming code.
`Courier bold`	Indicates information that must be typed exactly as shown, such as command syntax.
italics	Indicates pathnames, filenames, and ToolInfo variable names.
<angle brackets>	Surround variable information whose exact value you must supply.
[square brackets]	Surround optional information.

UNIX and Windows NT Conventions

StP is supported on UNIX and Windows NT operating systems. The following conditions apply:

- The terms “directory” and “folder” are synonymous.
- The terms “Enter” key and “Return” key are synonymous.
- This manual shows path names in UNIX format. Full Windows NT path names would include a drive letter at the beginning of each pathname.

Contacting Aonix

You can contact Aonix using any of the following methods.

Technical Support

If you need to contact Aonix Technical Support, you can do so by using the following email aliases:

Table 2: Technical Support Email Aliases

Country	Email Alias
Canada	support@aonix.com
France	customer@aonix.fr
Germany	stp-support@aonix.de
United Kingdom	stp-support@aonix.co.uk
United States	support@aonix.com

Users in other countries should contact their StP distributor.

Websites

You can visit us at the following websites:

Table 3: Aonix Websites

Country	Website URL
Canada	http://www.aonix.com
France	http://www.aonix.fr
Germany	http://www.aonix.de

Table 3: Aonix Websites (Continued)

Country	Website URL
United Kingdom	http://www.aonix.co.uk
United States	http://www.aonix.com

Reader Comments

Aonix welcomes your comments about its documentation. If you have any suggestions for improving *Generating and Reengineering Code*, you can send email to docs@aonix.com.

Related Reading

Generating and Reengineering Code is part of a set of Software through Pictures documentation. For more information about StP/UML and related subjects, refer to the sources listed in Table 4.

Table 4: Further Reading

For Information About	Refer To
Using the StP Desktop	*Fundamentals of StP, Creating UML Models*
Using StP editors	*Fundamentals of StP*
Task-oriented approach to creating UML models with StP/UML	*Creating UML Models*
StP/UML tutorial	*Getting Started with StP/UML*
Printing diagrams, tables, and reports	*Fundamentals of StP, Query and Reporting System*

Table 4: Further Reading (Continued)

For Information About	Refer To
Internal components of StP and how to customize an StP installation	*Customizing StP*
Using the Script Manager, writing Query and Reporting Language (QRL) scripts	*Query and Reporting System*
StP Object Management System (OMS)	*Object Management System*
Installing StP	*StP Installation Guide*
Interacting directly with the StP storage manager, Sybase SQL Server	*StP Guide to Sybase Repositories*
Managing an StP installation	*StP Administration*

1 Introduction

Generating and Reengineering Code presents a comprehensive, task-oriented approach to generating code from models you create with Software through Pictures / Unified Modeling Language (StP / UML). It provides instructions for:

- Generating code from your model in a variety of languages
- Using StP with supported programming environments
- Capturing specification-level information about classes in external libraries for reuse in your models

This manual assumes you are familiar with the use of StP / UML. Use this manual with *Creating UML Models*.

This chapter describes:

Supported Languages

You can generate code in the following languages for StP/UML:

- C++
- IDL (Interface Definition Language)
- Ada_95
- TOOL
- Java

Generating Code from Your Model

You can generate code for:

- All classes in an entire model, except external classes
- All classes in a selected package
- All classes in a selected diagram
- One or more selected classes in the model

StP/UML generates code from objects in the repository using a Query and Reporting Language (QRL) script that you can customize. For information about QRL scripts, see *Query and Reporting System*.

Navigation

Using StP/UML you can navigate to the C++, IDL, and Java source code. You cannot navigate to TOOL or ObjectAda source code.

Reverse Engineering

The StP / Object-Oriented-Reverse Engineering (StP / OO-RE) tool provides an automated method for capturing specification-level information about classes in external C++, IDL, Java, and TOOL libraries for reuse in your StP / UML model.

The StP / OO-RE utility must be purchased separately from StP / UML. For more information, contact your Aonix sales representative.

For a description of the StP / OO-RE utility, see Chapter 9, "Reverse Engineering."

Summary of Supported Features

Table 1 provides a summary list of supported languages and the code generation and reverse engineering features available for them.

Table 1: Supported Features

Language	Code Generation	Navigation	Reverse Engineering	Integration with Programming Environment
C++	X	Forward	X	Microsoft Developer's Studio (Visual C++ 6.0)
IDL	X	Forward	X	Microsoft Developer's Studio
Ada_95	X			
TOOL	X		X	
Java	X	Forward	X	Microsoft Developer's Studio (Visual J++ 6.0)

2 Generating Code From UML Models

StP/UML supports code generation based on classes and their relationships in class diagrams. You generate code from a combination of UML constructs, their annotations, and associated class tables.

This chapter describes the procedures for generating code common to all languages. It describes:

You can generate C++, IDL, Ada_95, TOOL, and Java code from StP/UML models. For details about generating code for each language, see one of the chapters listed in Table 1.

Table 1: Language-Specific Chapters

Language	Refer to:
C++	Chapter 3, "Generating C++ Code"
IDL	Chapter 4, "Generating IDL Code"
Ada_95	Chapter 5, "Generating Ada_95 Code"
	Chapter 6, "Mapping StP to Ada_95"

Table 1: Language-Specific Chapters

Language	Refer to:
TOOL	Chapter 7, “Generating TOOL Code”
Java	Chapter 8, “Generating Java Code”

Developing a Model for Code Generation

The following steps provide an overview of developing a model for code generation:

1. Create a class diagram using the Class Editor.
   See *Creating UML Models* for detailed instructions on creating these models.
2. Create a class table for each class with the Class Table Editor.
   For more information, see “Defining Classes for Code Generation” on page 2-3.
3. Complete the class definitions by adding the necessary annotations.
   For more information, see “Annotating Classes and Relationships for Code Generation” on page 2-7 for more information.
4. Generate code for one or more classes.
   For more information, see “Generating Code from the StP Desktop” on page 2-10
5. Compare the generated code with your requirements and refine the model, as necessary.
   For more information, see “Generating Code Incrementally” on page 2-11.

Defining Classes for Code Generation

To generate code for a class, a corresponding class table must exist even if the class has no attributes or operations. However, if a class has attributes and operations, you must define them in the class table. Code generation does not use attributes and operations only appearing in diagrams.

Using the Class Table Editor

Use the Class Table Editor (CTE) to create a class table. This section provides basic instructions for creating a class table; for complete instructions, refer to *Creating UML Models.*

To define attributes and operations in a class table:

1. Start the CTE for the class.
2. Display the class table sections using the **Hide/Show** command (available from the **View** menu).
3. Add the class member definitions for each attribute and operation, as described in "Adding Class Member Definitions" on page 2-3.
4. Add the analysis items for each attribute and operation, as described in "Analysis Items" on page 2-5.
5. Add the Implementation Items for the desired language, as described in the chapter for that language.
6. Choose **Save** from the **File** menu.
7. Choose **Exit** from the **File** menu.

Adding Class Member Definitions

The information provided in the Class Member Definitions section of the Class Table is common to all languages. In some cases, you must use language-specific syntax to provide class member definitions.

Table 2 lists and describes the columns in the Class Member Definitions section for attributes.

Table 2: Class Member Definitions for Attributes

Column Header	Description
Attribute	The name of the attribute (a string).
Type	(Required) The data type of the attribute (a string). This is the only class table item required for compilable code. Enter any type the specific language allows.
Default Value	The attribute's default value, if any (a string).

Table 3 lists and describes the columns in the Class Member Definitions section for operations.

Table 3: Class Member Definitions for Operations

Column Header	Description
Operation	The name of the operation (a string).
Arguments	The arguments of the operation, if any (a string). For StP/UML, you can enter the arguments in C++ or UML format. Regardless of the format in which you enter the arguments, the CTE displays them in UML format. StP then translates the arguments to C++.
Return Type	The operation's return type, if any (a string).

Analysis Items

Analysis items relate to UML methodology. Although code generation uses them, they are language independent.

Figure 1 shows the Analysis Items section of the Class Table Editor.

Figure 1: Analysis Items

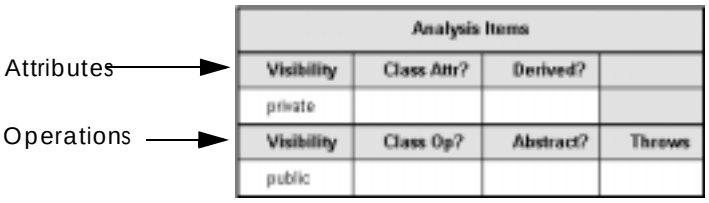


Table 4 lists and describes the columns in the Analysis Items section for attributes.

Table 4: Analysis Items for Attributes

Column Header	Description
Class Attr?	If True, designates the attribute as a class (or metaclass) attribute (True / False).
Derived?	Not used by code generation.
Visibility	Enables you to control access to a class's attributes. Options are: Public—Any other class can address this attribute. Protected—Only subclasses of this class can address this attribute. Private—No other class can address this attribute directly.

Table 5 lists and describes the columns in the Analysis Items section for operations.

Table 5: Analysis Items for Operations

Column Header	Description
Class Op?	If True, designates the operation as a class (or metaclass) attribute (True / False).
Abstract?	If True, generates the operation as virtual.
Throws	Specifies a list of exception names that the operation might throw (comma-separated string).
Visibility	Enables you to control access to a class's operations. Options are: Public—Any other class can address this operation. Protected—Only subclasses of this class can address this operation. Private—No other class can address this operation directly.

Implementation Items

Implementation Items are specific to the selected language. For descriptions of the implementation items for each language, see the chapter for that language.

Annotating Classes and Relationships for Code Generation

To prepare classes for code generation, annotate classes and their relationships with the appropriate notes and items. Annotations are either language-specific or language-independent. Language-independent annotations relate to UML methodology. Language-specific annotations relate to the language you are generating.

When entering annotations, use one of these tools:

- Property sheets
- Object Annotation Editor (OAE)

Using Properties Dialog Boxes to Enter Annotations

From an StP/UML editor, select an object and choose **Edit > Properties**. You may also click on the object with the right mouse button and choose **Properties**.

By using **Properties** dialog boxes, you can enter many annotations at one time. Conversely, with the Object Annotation Editor, you enter one annotation at a time.

For complete instructions on using the **Properties** dialog boxes, see *Creating UML Models.*

Using the Object Annotation Editor to Enter Annotations

Use the Object Annotation Editor (OAE) to enter the following types of annotations:

- Definition notes for classes, roles, and inheritance relationships
- Role <language> Definition notes
- <language> Declarations

- <language> Source code notes
- Comments

Descriptions of these annotations appear in the chapter for each language in this manual.

Adding Annotations

To enter notes and related items with the OAE:

1. Select the object to be annotated.
2. Choose **Edit > Object Annotation** or click on the object with the right mouse button and choose **Object Annotation**.
3. On the StP Object Annotation Editor, double click on the object to display the notes this object already comprises.
4. To add another note, choose the note from the note options list in the Set / Add area.
5. **Optional:** Add a note description in the field in the Set / Add area.
6. Click **Add**.

   The note (with its description if you added one) appears under the object name.
7. To add note items, select the appropriate note under the object name.
8. From the item options list in the Set / Add area, choose an item for the note.
9. In the field of the Set / Add area, choose a value for the item from the options list or insert your own value for the item if appropriate.
10. Click **Add**.
11. Choose **Exit** from the OAE **File** menu.

For a complete description of the OAE, see *Fundamentals of StP*.

Adding Source Code to Operations

You can add C++, IDL, Ada_95, TOOL, and Java source code to an operation.

To add source code to an operation:

1. Select a class operation in a diagram or an operation name cell in a class table.
2. Start the Object Annotation Editor.
3. From the options list in the Set / Add area, choose a source code note.
4. In the field of the Set / Add area, enter the source code.
5. Click **Add**.
6. Choose **Exit** from the OAE **File** menu.

Adding Comments

Add comments to a note by using the field of the Add / Set area.

To add comments to a note:

1. Select the object for which the note is written.
2. Start the Object Annotation Editor.
3. Under the name of the object, select the note.
4. In the field of the Set / Add area, enter the text of the comment.
5. Click **Set**.
6. Choose **Exit** from the OAE **File** menu.

Generating Code for External Classes

By default, StP does not generate code for classes annotated as external. Typically external classes originate in external libraries, as described in Chapter 9, “Reverse Engineering.”

If you want to generate code for a class captured from an external library, do one of the following:

- On the Generate <language> dialog, deselect the Ignore Classes Annotated as External option.

  For instructions on using the Generate <language> dialog, see the chapter for the language.
- Change its annotation from external to non-external

For instructions, see *Creating UML Models*.

Generating Code from the StP Desktop

You can generate code at any time during the development of your model. Although you do not need to complete your model before generating code, your model must contain certain annotations and table information to generate viable code.

This section provides instructions for generating code from the StP Desktop.

Generating Code

To generate code for StP / UML:

1. In the Model pane on the StP Desktop, open the appropriate category.
2. Based upon the code you want to generate, select a particular diagram, class(es), or package(s) from the objects pane.
3. Choose a **Generate <language> for...** command from one of the menus listed in Table 6 on page 2-11.
4. In the **Generate <language> for...** command properties dialog box, adjust property values.

   For information about setting the command properties for a specific language, see the chapter for that language.

   To return to the default settings after making changes, click **Reset**.
5. To generate code, click **OK** or **Apply**.

   While StP is executing the command, a status bar displays the command's progress. If you try to generate code for a class that has no class table, StP displays a warning message. By default, code generation puts code into files in the *<project>/<system>/src_files* directory.

Code Generation Commands

Table 6 provides an overview of the available commands for generating code.

Table 6: Code Generation Commands

To generate code for...	Choose this command...	For this Models category...	On this menu
All classes in the diagram	**Generate <language> for Diagram's Classes**	Diagrams>Class	**Code/<Language>** menu from the Desktop
			Class Diagrams shortcut menu
All classes in the model	**Generate <language> for Whole Model**	Diagrams (any) Tables (any) Model Elements>Classes	**Code/<Language>** menu from the Desktop
Selected classes	**Generate <language> for Class**		**Classes** shortcut menu
All packages in the diagram (Ada_95 only)	**Generate Ada_95 for Diagram's Packages**	Diagrams>Class	**Code/<Language>** menu from the Desktop
Selected packages (Ada_95 only)	**Generate Ada_95 for Package**	Model Elements>Packages	**Packages** shortcut menu

Generating Code Incrementally

StP generates code incrementally for all languages except TOOL. Incremental code generation lets you modify the generated source code files and have StP preserve your changes after subsequent code generations.

In order for incremental code generation to work, you must have created and saved a class table for the class.

After you modify a file and regenerate code, StP preserves the parts of the files that did *not* derive from the model during code generation.

To change code derived from your model:

1. Change the model.
2. Save the changes.
3. Regenerate the code.

If you generate all classes into one file or file set, you can choose whether to keep classes that were generated previously in the file, even if they are not part of the current set of classes for which code is being generated. For example, assume you previously generated code for classes A, B, and C into a file named *System*. Now you want to regenerate code for classes A and B into the same file. An option of the code generation command allows you to keep class C in the *System* file or delete all its code.

Each time code generates to an existing file, StP copies the existing file to a backup file. The backup file is named *<filename>.BAK*. If code you entered is deleted during a subsequent code generation, you can retrieve the code from the backup file.

For information required for incremental code generation by a specific language, see the chapter for that language.

3 Generating C++ Code

This chapter shows you how to generate C++ code from your StP/UML model. It describes:

Use this chapter with Chapter 2, "Generating Code From UML Models." Chapter 2 provides information for generating code that applies to all languages.

What is Generated for C++

For each class in an object model or class diagram that is not excluded, StP generates:

- Implementation file *(.cpp* file; also called body or source file)
- Interface file (*.h* file*;* also called header, include, or specification file)

The C++ code generated from your model conforms to the ANSI C++ 3.0 standard.

Implementation Files

Implementation (*.cpp*) files contain member function headers and bodies and any C++ code specified in the operation's Source Code note description. Member functions are derived from operations in the model.

Interface Files

Interface files contain definition and interface information for the class. Special information that appears in this file may include:

- References to superclasses and associated classes (generated as include files)

  See "Include Files," below.
- Descriptive notes for a class, its attributes, or its operations (generated as comments)

  See "Descriptive Notes" on page 3-3.
- Parameterized class declarations

  See "Generating C++ for Parameterized and Instantiated Classes" on page 3-4
- Inline functions

  See "Inline Functions" on page 3-14.
- Namespace declarations for packages

  See "Generating C++ for Packages" on page 3-16

Include Files

When a class is in a generalization or inheritance relationship to a superclass, StP places an include statement referring to the interface file of the superclass in the class's interface file.

For example, if Class C inherits from Class A and Class B, StP places these include statements near the top of the *C.h* file:

```
#include "A.h"
#include "B.h"
```

Similarly, when a class is in an association relationship with another class, and the association is navigable in the direction of the other class, StP places an include statement referring to the associated class in the class's interface file.

In both cases, include statements are written only when you use the command option to generate one set of files per class.

Descriptive Notes

You can add descriptive notes for classes, attributes, or operations using the Object Annotation Editor. When you generate C++ code, you can generate the notes as comments in the interface file by setting an option for the command.

The following code shows a comment generated for the *Message* class.

```
//  Class: Message
//    Description:
//    This is a comment
class Message : public Printable
```

When you generate code for the *Vector* class, the following code is placed near the top of the *Vector.h* file:

```
// stp class declarations
template<class T, int size> class Vector;
typedef Vector<int, 100> Numbers;
// stp class declarations end
```

Generating C++ for Parameterized and Instantiated Classes

Table 1 shows the C++ code that StP / UML generates for parameterized and instantiated classes.

Table 1: C++ Generated for Parameterized and Instantiated Classes

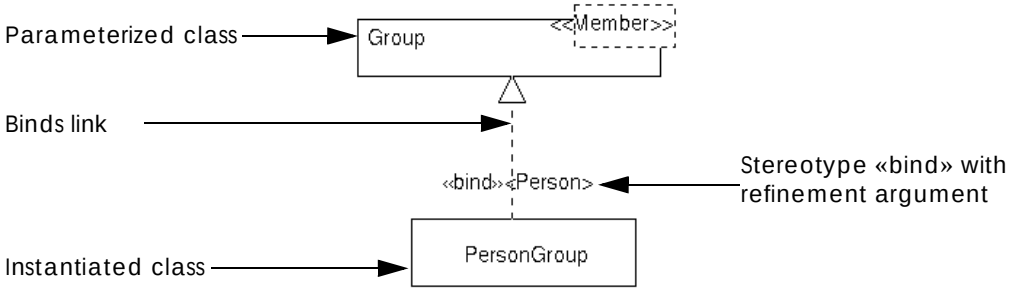
Class	C++ Construct	For more information, see
Parameterized	Template class	All of the subsections in this section
Named instantiated class	Typedef	"Named Instantiated Classes" on page 3-4
Anonymous instantiated class	Parts of other declarations	"Anonymous Instantiated Classes" on page 3-6

Named Instantiated Classes

Named instantiated classes are generated as typedefs in the same file as the corresponding parameterized class.

In Figure 1, *PersonGroup* is a named instantiated class. It is connected to *Group* by a binds link with the stereotype «bind» and the refinement argument *Person*, which is used to instantiate the class *Group*.

Figure 1: Named Instantiation



StP/UML generates the following C++ code for the diagram shown in Figure 1:

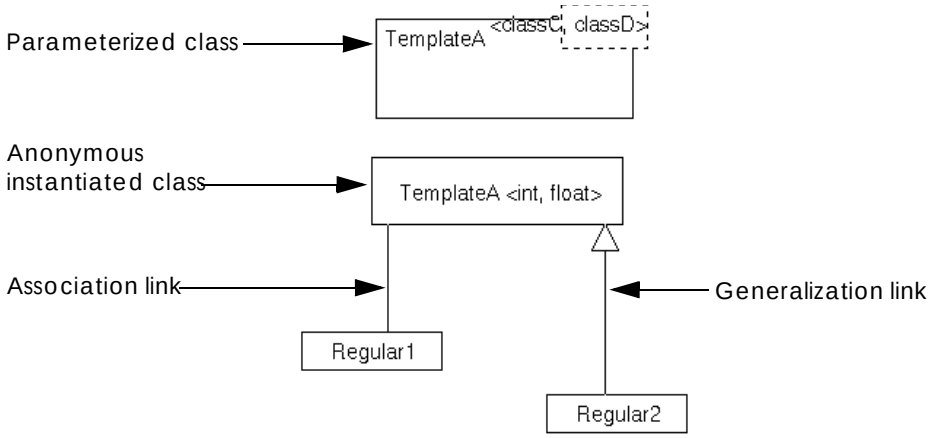
```
#ifndef _Group_h_
#define _Group_h_
// stp class declarations
template<<Member>> class Group;
typedef Group<Person> PersonGroup;
// end stp class declarations
// stp class definition 0
template<<Member>>
class Group
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition
// stp footer
#endif
// end stp footer
```

Anonymous Instantiated Classes

StP generates anonymous instantiated classes as parts of other declarations, depending on the relationships they are involved in. StP does not generate separate classes for the anonymous instantiated classes.

In Figure 2, *TemplateA<int, float>* is an anonymous instantiated class that is instantiated from the parameterized class *TemplateA*. *TemplateA<int, float>* is in an association relationship with *Regular1* and a generalization relationship with *Regular2*.

Figure 2: Anonymous Instantiation



StP/UML generates the following C++ code for this diagram:

```
#ifndef _Regular1_h_
#define _Regular1_h_

// stp class declarations
class Regular1;
// end stp class declarations

// stp class definition 0
class Regular1
{
// stp class members
public:
protected:
```

```
private:
// end stp class members
};
// end stp class definition

// stp footer
#endif
// end stp footer

#ifndef _Regular2_h_
#define _Regular2_h_

// stp includes 0
#include "TemplateA.h"
// end stp includes

// stp class declarations
class Regular2;
// end stp class declarations

// stp class definition 0
class Regular2 : public TemplateA<int, float>
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition

// stp footer
#endif
// end stp footer

--------------------------------------------------------------------------------

#ifndef _TemplateA_h_
#define _TemplateA_h_

// stp class declarations
template<classC, classD> class TemplateA;
// end stp class declarations

// stp class definition 0
template<classC, classD>
class TemplateA
{
```

```
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition

// stp footer
#endif
// end stp footer
```

Parameterized and Instantiated Class Relationships

In most cases, the code that StP/UML generates for relationships between parameterized and instantiated classes is similar to the code generated for regular classes. However, there are some differences, which are listed in Table 2.

Table 2: Parameterized and Instantiated Class Relationships

From	To	Relationship Type	Generated Construct
Instantiated Class	Regular or Parameterized Class	Association	Not generated
Regular or Instantiated Class	Parameterized class		
Regular class	Parameterized class	Binds with stereotype «bind» and refinement arguments	Inheritance
Parameterized class	Parameterized class		

Dependency Relationships between Classes

Table 3 describes the situations when dependency relationships are required for instantiated and parameterized classes.

Table 3: Dependency Relationships

From	To	Dependency is needed when:
Instantiated class	Regular class	The regular class is used as an actual argument, or the instantiated class is a friend of the regular class.
	Instantiated class	One instantiated class uses another instantiated class as one of its actual arguments.
	Parameterized class	The instantiated class is a friend of the parameterized class.
Regular or Parameterized class	Instantiated class	The instantiated class is used by the parameterized or regular class's declaration or implementation (such as an attribute's type or an operation's argument or return type) or in some other location in the code.
	Regular or Parameterized class	The regular or parameterized class is used by the other class's declaration or implementation, or one of the classes is a friend of the other.

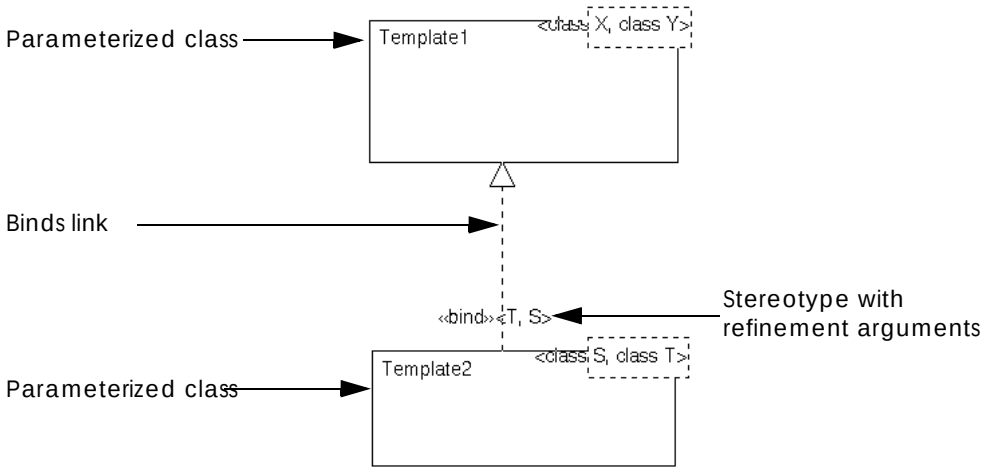
Examples

The following examples illustrate how StP generates code for relationships between parameterized and instantiated classes.

Generating Code for Inheritance Relationships

Figure 3 shows an alternative way of modeling inheritance from the one shown in Figure 1 on page 3-5. With this method, StP uses the formal arguments of one parameterized class (template) as the actual arguments of the other parameterized class.

Figure 3: Binds Relationship Between Parameterized Classes



StP / UML generates the following code for the diagram shown in Figure 3:

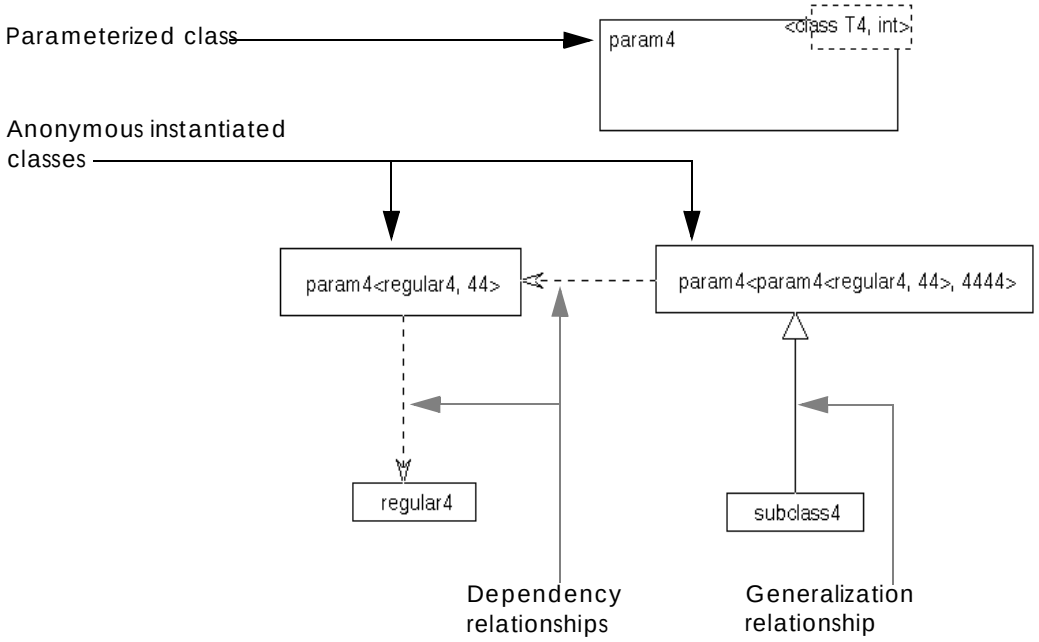
```
#ifndef _Template1_h_
#define _Template1_h_
// stp class declarations
template<class X, class Y> class Template1;
// end stp class declarations
// stp class definition 0
template<class X, class Y>
class Template1
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition
// stp footer
#endif
// end stp footer

#ifndef _Template2_h_
#define _Template2_h_
```

```
// stp includes 0
#include "Template1.h"
// end stp includes
// stp class declarations
template<class S, class T> class Template2;
// end stp class declarations
// stp class definition 0
template<class S, class T>
class Template2 : public Template1<T, S>
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition
// stp footer
#endif
// end stp footer
```

Generating Code for Chaining Dependency Relationships

The diagram in Figure 4 provides an example of the way in which StP generates C++ code for chaining dependency relationships.

Figure 4: Chaining Dependency Relationships



StP / UML generates the following C++ code for the diagram shown in Figure 4:

```
#ifndef _param4_h_
#define _param4_h_
// stp class declarations
template< class T4, int > class param4;
// end stp class declarations
// stp class definition 0
template< class T4, int >
class param4
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition
```

```
// stp footer
#endif
// end stp footer

-------------------------------------------------------------------------------

#ifndef _subclass4_h_
#define _subclass4_h_

// stp class declarations
class subclass4;
// end stp class declarations

// stp class definition 0
class subclass4 : public param4< param4<regular4, 44>,4444 >
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition

// stp footer
#endif
// end stp footer

-------------------------------------------------------------------------------

#ifndef _regular4_h_
#define _regular4_h_

// stp class declarations
class regular4;
// end stp class declarations

// stp class definition 0
class regular4
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition
```

```
// stp footer
#endif
// end stp footer
```

Inline Functions

There are three options for locating generated code for inline functions:

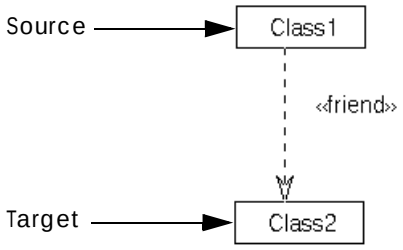
- In the interface file as stand-alone functions following the class declarations and definitions (default)
- In the interface file, embedded within the class declarations
- In a separate implementation file

You specify your choice of location using command options.

Generating C++ for Friend Relationships

In StP/UML, you create a friend relationship by drawing a dependency relationship and giving it the stereotype «friend», as shown in Figure 5.

Figure 5: Dependency Relationship with Friend Stereotype



The source of the dependency relationship becomes the friend of the target; the target is not a friend of the source. The target can be a regular class or parameterized class. The source can be one of the following classes:

- Regular class
- Parameterized class
- Named instantiated class
- Anonymous instantiated class
- Operation of a regular or parameterized class

For any dependency relationship, StP generates an #include directive in the file of the source class. For a friend relationship, StP also generates an #include directive in the file of the target class, and inserts the appropriate friend declaration into the target class.

StP/UML generates the following code for the diagram in Figure 5:

```
#ifndef _Class1_h_
#define _Class1_h_
// stp includes 0
#include "Class2.h"
// end stp includes
// stp class declarations
class Class1;
// end stp class declarations
// stp class definition 0
class Class1
{
// stp class members
public:
protected:
private:
// end stp class members
};
// end stp class definition
// stp footer
#endif
// end stp footer
```

```
#ifndef _Class2_h_
#define _Class2_h_
// stp includes 0
#include "Class1.h"
// end stp includes
```

```
// stp class declarations
class Class2;
// end stp class declarations
// stp class definition 0
class Class2
{
// stp class members
    friend class Class1;
public:
protected:
private:
// end stp class members
};
// end stp class definition
// stp footer
#endif
// end stp footer
```

Generating C++ for Packages

StP/UML generates packages into C++ as namespaces. Nesting of packages results in a corresponding nesting of namespaces. StP/UML does not generate code for dependency or inheritance relationships between:

- Packages
- Packages and classes
- Packages and operations

For each class contained by a package, StP encloses the class declaration by its own namespace declaration or by nested declarations, as needed. This includes any named instantiated classes (generated as typedefs).

StP generates a reference to any class declared within a namespace in its fully qualified form, whether or not the classes involved in the relationship are in the same namespace.

StP/UML does not generate code for C++ using declarations or using directives.

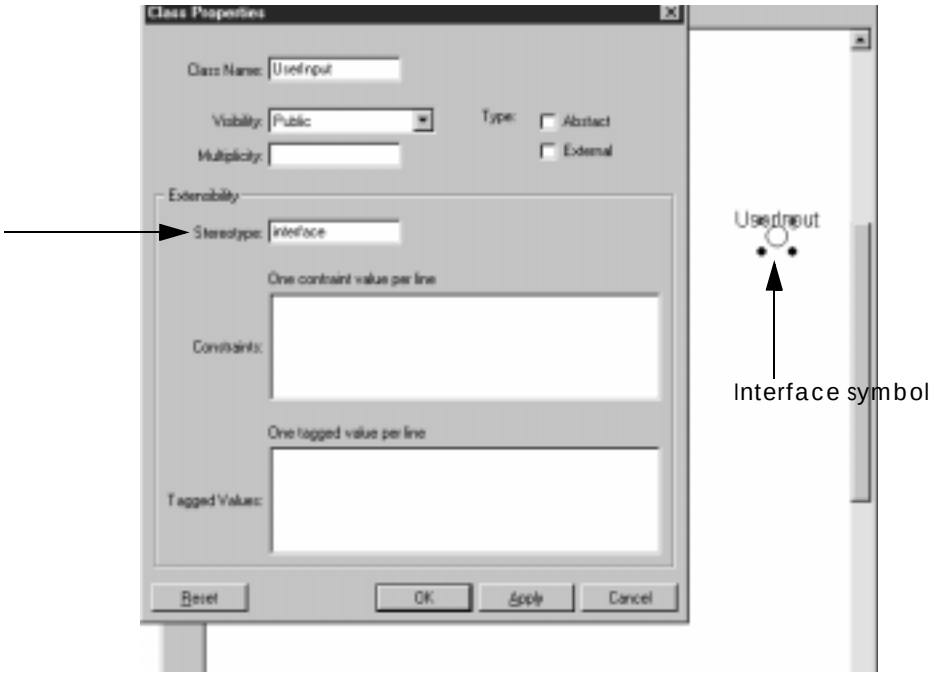
When StP generates code using one file name per class, an option provided on the code generation dialog enables you to include any enclosing package names in the generated file name for the class. Including package names as part of the class file name is a matter of preference; however, an advantage of not using package names is that if you move a class from one package to another, StP finds the class in the same file during incremental code generation. If you include the package name in the file name, you must manually transfer any code you enter (code *not* generated by StP) to the new file.

Generating C++ for Types and Interfaces

In StP / UML, a type or interface is a regular or parameterized class with the stereotype «type» or «interface». (The interface symbol that is available from the Class Editor is for graphical use only; to generate code properly, you must add the «interface» stereotype to the interface symbol, as shown in Figure 6.)

For code generation, types and interfaces are interchangeable. An interface can not have attributes. Any operations must be abstract and must have public visibility.

Figure 6: Adding the Interface Stereotype to an Interface Symbol



StP / UML generates types and interfaces as abstract classes. StP generates operations associated with types and interfaces as pure virtual functions, regardless of their designation in the Class Table Editor. Attributes associated with types are generated as designated in the Class Table Editor.

Turning off Code Generation for Interfaces

StP / UML allows you to turn off code generation for:

- All of a diagram's types and interfaces
- A selected type or interface
- Attributes associated with types and interfaces

Under certain circumstances, you may not want to generate code for a type or interface. For example, you may not want to generate code for a type or interface if other classes implement it on their own, or if you are using it purely for analysis. Alternatively, you may want to generate code

for a type, but not for its attributes, for example, when a type's attributes are used only as an abstract specification rather than as an actual implementation.

To turn off code generation for types and interfaces and their attributes, use one of these tools:

- **Generate C++ for Diagram's Classes** dialog box
- **Generate C++ for Class** dialog box
- Object Annotation Editor (OAE)

Using the Dialogs

The **Generate C++ for Diagram's Classes** and the **Generate C++ for Class** dialog boxes offer the following options for turning off code generation for classes with the stereotype «type» or «interface»:

- Generate Code for Types / Interfaces
- Generate Type Attributes

These options are described in Table 9 on page 3-31.

Using the Object Annotation Editor

The Object Annotation Editor (OAE) provides a note and items for turning off code generation for types and interfaces and their attributes. These notes and items are described in Table 4. For instructions on using the OAE to add notes and items for code generation, see "Adding C++ Annotations."

Table 4: Object Annotation Editor Notes and Items for Turning Off Code Generation

Note	Item	Description
Class C++ Definition		Lets you add items used for C++ code generation to a selected class
	Generate Code If Type	Lets you specify whether or not to generate code for this class if it has the stereotype «type» or «interface»
	Generate Type Attributes	Lets you specify whether or not to generate code for the attributes defined for this class if it has the stereotype «type» or «interface»

Relationships for Types and Interfaces

There are two relationships associated with types and interfaces:

- Implements
- Dependency (requires)

Implements Relationships

A class may implement an interface. StP generates this type of relationship into C++ as an inheritance relationship. The same properties and annotations are available for an implements relationship as for a generalization relationship.

Dependency Relationships

A class may require an interface from another class. StP draws this relationship as a dependency relationship from the class to an interface symbol. StP does not generate code for a dependency relationship.

For C++, the requiring class is not actually dependent on the interface, but on the class that is implementing the interface. You must draw this dependency separately; otherwise, when StP generates code, it would need to generate dependencies of the requiring class on all classes that implement that particular interface.

StP to C++ Mapping

Various elements of a model generate specific C++ constructs. Table 5 shows the correlation between UML objects and the C++ constructs that they generate.

Some C++ constructs, such as local typedefs or enums, are not represented by modeling constructs. You can add declarations for these constructs to the code generated for a class by using the Object Annotation Editor (OAE); for instructions, see “Using the Object Annotation Editor to Enter Annotations” on page 2-7.

Table 5: C++ Derived from StP Object-Oriented Models

Object	Generated C++ Construct
Class	Class
Parameterized class	Template class declaration
Named instantiated class	Typedef declaration
Anonymous instantiated class	Type specifier for data member
Regular or parameterized class with stereotype <<type>> or <<interface>>	Abstract class
Attribute	Data Member (default is private)
Operation	Member function (default is public)
Association	Data member (default is protected)
Aggregation	Data member

Table 5: C++ Derived from StP Object-Oriented Models

Object	Generated C++ Construct
Composition	Data member
Dependency relationship	#include directive
Role name	Name of the identifier for the association, aggregation, has relationship, or composition relationship
Association class	Class
Generalization	Inheritance (default is public)
Implements	Inheritance (default is public)
Package	Namespace
Object annotation for class	Class comments (optionally generated)
C++ Declaration annotation for class	C++-specific class elements
Object annotation for attribute	Attribute comments (optionally generated)
Object annotation for operation	Operation comments (optionally generated)
C++ source code annotation for operation	C++ source code for member function

Adding C++ Information to Class Tables

You define class attributes and operations using these sections of the Class Table Editor:

- Class member definitions
  See “Entering Class Member Definitions for C++.”
- Analysis items

See “Entering Analysis Items for C++.”

- C++ implementation items

  See “Entering C++ Implementation Items for C++.”

For general information about entering information in class tables for code generation, see “Defining Classes for Code Generation” on page 2-3.

Entering Class Member Definitions for C++

Most of the information entered in the Class Member Definitions section of a class table is common to all languages, as described in “Adding Class Member Definitions” on page 2-3. The information presented in this section is specific to C++ code generation.

To generate C++ code that can be compiled, you must provide a data type for each attribute. This information is the minimum that you must supply to generate compilable C++ code. Enter the data type in the Type column.

If an attribute’s data type is array, pointer, or reference, you must supply the appropriate array brackets ([]), array bounds, pointer(*), or reference (&) in the Type column for correct code to be generated.

If you leave the Arguments cell blank, the value of Arguments is void.

Entering Analysis Items for C++

The information that you must provide in the Analysis Items section of a class table is common to all languages. For instructions, see “Analysis Items” on page 2-5.

Entering C++ Implementation Items for C++

You enter C++ implementation items in the C++ Items section of a class table. To display the C++ Items section of a class table, follow instructions given in “Defining Classes for Code Generation” on page 2-3.

Entering C++ Items for Attributes

Table 6 lists and describes the attributes columns in the C++ Items section of the class table.

Table 6: C++ Items for Attributes

Header	Description
Const?	The attribute is a C++ constant (True / False).
Visibility	Enables you to control access to a class's attributes. Options are: Public—Any other class can address this attribute. Protected—Only subclasses of this class can address this attribute. Private—No other class can address this attribute directly.
Volatile?	Specifies whether or not the attribute is a "volatile" attribute in C++ (True / False).

Entering C++ Items for Operations

Table 7 lists and describes the operations columns in the C++ Items section of the class table.

Table 7: Class Table Operation C++ Items

Header	Description
Const?	The operation is a C++ constant (True / False).
Visibility	Enables you to control access to a class's operations. Options are: Public—Any other class can address this operation. Protected—Only subclasses of this class can address this operation. Private—No other class can address this operation directly.

Table 7: Class Table Operation C++ Items (Continued)

Header	Description
Virtual?	The operation is virtual (a subclass can implement the same operation) for C++ code (True / False).
Inline?	The operation is an inline operation for C++ code (True / False).
Ctor Init List	The initialization list if the operation is a constructor (a string).

Adding C++ Annotations

This section describes the C++ annotations available for UML. For general information about annotating objects, see “Annotating Classes and Relationships for Code Generation” on page 2-7.

StP does not check for illegal combinations of annotations.

C++ Annotations

Table 8 summarizes C++ annotations.

Table 8: Annotations for C++ Code Generation

Object	Note	Item	Item Value	Description
Class	C++ Declarations	Enter in Note Description dialog.	Lets you add C++-specific declarations not represented by UML constructs, such as local typedefs and enums. The code is prepended, as is, to the C+ class definition for the class.	

Table 8: Annotations for C++ Code Generation (Continued)

Object	Note	Item	Item Value	Description
Parameterized Class	Class Definition		Lets you add items that capture modeling information about a class. (The Parameters item is the only one used in C++.)	
		Parameters		Lets you add a list of parameters for a parameterized class.
Operation	C++ Source Code	Enter in Note Description dialog.	Lets you add C++ source code for the body of the operation. For instructions, see "Adding Source Code to Operations" on page 2-8.	
Role	Role Definition		Lets you add items that capture modeling information about one role in an association between classes.	
		Aggregation type	Composition Aggregation	Lets you indicate that the role is an aggregation, and if so, whether it is a regular aggregation or the stronger form of aggregation, called composition.
		Role Navigability	True/False	Shows whether or not a relationship can be traversed in a particular direction.
		Ordered	True/False	Lets you indicate that the objects at the many end of an association are explicitly ordered.
		Multiplicity	See *Creating UML Models* for examples of multiplicity.	Lets you indicate how many instances of a class may relate to another single class. Type the multiplicity value in the Value text field.

Table 8: Annotations for C++ Code Generation (Continued)

Object	Note	Item	Item Value	Description
Role	Role Definition	Qualifiers	See *Creating UML Models* for examples of qualifiers.	Lets you reduce the multiplicity of a one-to-many or a many-to-many association. Each qualifier distinguishes among the set of objects at the many end (this end) of an association. You can enter more than one qualifier for an association role.
	Role C++ Definition		Lets you add items for an association role that are used for C++ code generation. These items affect code for the association as it appears in the class at the opposite end of the association from the role being annotated.	
		Visibility	Private Protected Public	Lets you indicate the C++ access level of constructs used to implement associations between classes.
		Pattern Family	Default Template	Lets you specify which customizable pattern family to use in the implementation of roles for C++ code generation. For additional information, see "Specifying the Pattern Family" on page 3-30.
			Pointer / Instance	

Table 8: Annotations for C++ Code Generation (Continued)

Object	Note	Item	Item Value	Description
Generalization Refines relationship links	C++ Inheritance Definition		Lets you add items that capture C++-specific information about an inheritance relationship	
		Inheritance Visibility	Public Private Protected	Lets you indicate if the inheritance is public, protected, or private. (You can also set this property on the Properties dialog. For instructions on using the Properties dialog, see *Creating UML Models*.)
		C++ Inheritance is Virtual	True / False	Lets you indicate whether or not C++ inheritance is virtual. (You can also set this property on the Properties dialog. For instructions on using the Properties dialog, see *Creating UML Models*.)
Refines relationship link	Refinement Definition		Lets you add items that capture modeling information about the refinement relationship between classes.	
		Refinement Arguments		Lets you specify parameters when you are creating a Bound Element by refining a parameterized class.
Class, Attribute, or Operation	Requirement	Name	Lets you type C++ comments in the Note Description dialog or the Value field for the item. For instructions on using the Note Description dialog, see "Adding Comments" on page 2-9	
		Document		
		Paragraph		
	Object	Author		
		Generated		

Annotating Roles for Implementation of Associations

When annotating roles for implementation of associations in StP/UML you must specify the association's:

- Role navigability
- Pattern family

Specifying Role Navigability

The navigability of a role indicates the direction of an association.

By default, StP does not implement binary associations. To implement an association, you must specify role navigability.

To specify role navigability, you use one of these tools:

- **Properties** dialog box
- OAE

You can implement associations as either one-way associations (traversed in only one direction) or two-way associations (traversed in both directions).

To specify role navigability using the **Properties** dialog box:

1. Select an association.
2. From the **Edit** menu, choose **Properties,** or right click on the association and choose **Properties** from the object menu.
3. In the **Properties** dialog box, click Role Navigability beneath one or both roles.

   StP implements the association in the direction of the role whose navigability is set.

   To implement a one-way association, set Role Navigability beneath one role.

   To implement a bidirectional association, set Role Navigability beneath both roles.
4. Click **OK**.

Specifying Role Navigability with the OAE

To specify role navigability for an association:

1. Choose the **Role Definition** note.
2. Set the value of the **Role Navigability** item to **True**.
   To implement a bidirectional association, you must set both roles' navigability to True.

For general information on using the OAE, refer to *Fundamentals of StP*.

Specifying the Pattern Family

StP/UML lets you specify which pattern family should be used when roles are implemented for C++ code generation. The choices are:

- Default Template
- Pointer/Instance

You can customize existing and add new pattern families. Customizable patterns are listed and can be configured in the *uml/qrl/code_gen/cxx/association_patterns.inc file*.

To specify the pattern family, use one of these methods:

- Use the OAE, as described in this section
- Select the pattern family from the Generate C++ dialog box, as described in "Generating C++ Code" on page 3-31

To specify the pattern family for an association:

1. Select the appropriate role in the **Object Selector** dialog box.
2. Choose the **Role C++ Definition** note.
3. Set the value of the **Pattern Family** item to **Default Family**.
4. To change the Pattern Family, select **Pointer/Instance** from the item value options list

For general information on using the OAE, refer to *Fundamentals of StP*.

Generating C++ Code

This section shows you how to generate C++ code from the StP Desktop. To generate code for StP / UML:

1. In the Model pane on the StP Desktop, open the appropriate category.
2. Based upon the code you want to generate, select a particular diagram, or class(es) in the objects pane.
3. Choose a **Generate C++ for <object>** command from one of the menus listed in Table 6 on page 2-11.
4. In the **Generate C++ for <object>** dialog box, set the appropriate command properties, as described in Table 9.
5. Click **OK** or **Apply**.

Generate C++ Command Properties

Table 9 contains descriptions of the C++ code generation command.

Table 9: Generate C++ Command Properties

Property	Description
Directory	Specifies the pathname for the directory where output code is written. The default is *proj/sys/src_files*.
One File Name Per Class	Generates a separate pair of files for each class that is selected for code generation. The file names are taken from the class names when you select this option. This option is the default.
File Name	Active when the One File Name Per Class option is not selected. Generates code for all classes to one file and permits you to specify its name.
Use Package Name in File Names	Includes the name of the package containing the classes in the file name.
Preserve Code From Other Classes	Active when the One File Name Per Class is deselected. Retains or deletes any previously-generated code for classes and operations not in the current set of classes for which code is being generated. The default is Not Selected (Do Not Preserve).

Table 9: Generate C++ Command Properties (Continued)

Property	Description
Interface (Header) File Extension	Specifies the extension appended to the name of the interface file. The default is *.h*.
Implementation (Body) File Extension	Specifies the extension appended to the name of the implementation file. The default is *.cpp*.
Place Inline Functions Within Class Declarations	Puts inline functions in the interface file within the class declaration, rather than stand-alone in the interface file (default) or another file.
Inline Functions File Extension	Specifies the extension appended to the name of the inline functions file. The default is *.h*. To use this option, you must deselect the Place Inline Functions Within Class Declarations option.
Generate Code for Types / Interfaces	If ON, generates C++ for all types and interfaces on a diagram, or a selected type or interface. Default is ON.
Generate Type Attributes	If ON, generates C++ code for the attributes associated with all types on a diagram, or a selected type or interface. Default is ON. (This option is not available if the Generate Code for Types / Interfaces option is OFF.)
For Associations Use Pattern Family	Specifies the name of a pattern family to use when implementing associations. Options are Default Template or Pointer / Instance. You can select the pattern family on this dialog, or you can use the Object Annotation Editor. For additional information, see "About Association Pattern Families."
Ignore Classes Annotated As External	If selected, does not generate code for external classes.
Generate Comments For	Generates comment lines with the text of annotations for the selected objects. Options are Classes, Attributes, and Operations. The default is to be not selected.

About Association Pattern Families

You can tailor the data members of a class that are generated to implement associations in C++ by using code generation patterns. The pattern you use depends on the multiplicity and qualification of the

association. A complete set of these patterns, which includes every combination of multiplicity and qualification of an association, is called a pattern family, because every member of the set has something in common.

A pattern is a text string that usually contains one or more of the UML C++ string substitution symbols.

Using Pattern Families

The *association_patterns.inc file,* located in your *<stp_file_path>/uml/qrl/code_gen/cxx* directory, provides three pattern families:

- Default Template
- Pointer / Instance
- Rogue Wave, which is commented out by default
- A Rogue Wave pattern family, which is commented out by default

In addition, you can customize the patterns used for regular associations, aggregations, and compositions independently.

Generating C++ Incrementally

This section contains information that applies to incremental code generation for C++. For a general discussion of incremental code generation, see “Generating Code Incrementally” on page 2-11.

As part of code generation, StP generates all of a class’s operations into function headers or bodies that are put into an implementation file. If you generate code to the same file more than once, StP preserves some regions of the file. To add code to the file that StP will retain on subsequent code generations, you must add the code to these regions. For specific descriptions of these regions, see “Guidelines for Changing Files.”

If you add or delete operations in an implementation file from the class, you do so during incremental code generation.

Guidelines for Changing Files

You can edit a generated file anywhere except:

- In generated comments
- Between related lines or in single lines

If you wish to change parts of the code that cannot be edited, you must change the model, save the changes, and regenerate the code.

Table 10 provides editing guidelines.

Table 10: Editing Guidelines

Do not edit from this line:	To:
Interface File	
`// stp class declarations`	`// end stp class declarations`
`// stp includes nnnn`	`// end stp includes`
`// stp class definition nnnn`	`{`
`// end stp class definition`	
`// stp class members`	`// end stp class members`
`// stp footer`	/ / end stp footer
Implementation File	
`// stp attributes nnnn`	/ / end stp attributes
`// stp operation nnnn::mmmm`	Up to and including the function header
`// end stp operation`	
`// stp code`	`// end stp code`

4 Generating IDL Code

This chapter tells you how to generate IDL code from your StP / UML model. It describes:

Use this chapter with Chapter 2, "Generating Code From UML Models." Chapter 2 provides information for generating code that applies to all languages.

What is Generated for IDL

For each class in a class diagram that is not external, StP generates a source file. The default directory is the *src_files* directory in your project / system path. The default output file extension is *.idl*.

The generated code conforms to the CORBA standard.

Some constructs in IDL have clear mappings to UML objects; some are achieved through annotations or other means. Table 1 shows how the IDL constructs with clear mappings appear in StP / UML.

Table 1: IDL Constructs Mapped to UML Objects

StP/UML Object	IDL Construct
Class	interface <label>
Interface	interface <label>
Package	Not mapped to IDL construct
Parameterized class	interface <label>
Object	Ignored in IDL
Attribute	Attribute
Operation	Operation
Association	Data member (default is protected)
Generalization	interface <source>: <target>, <target>,...

IDL Constructs Not Mapped to UML

These IDL constructs are not directly represented by an object in UML models:

- Constants
- Types
- Exceptions
- Modules

When these constructs scope to a class, you can use the Object Annotation Editor to add declarations for them. When you generate IDL code, the declarations join the class definitions, as described in “Adding Class IDL Declarations” on page 4-7.

When these constructs scope to a diagram, you can add declarations manually into the generated code, as described in “Adding Class IDL Declarations” on page 4-7.

Adding IDL Information to Class Tables

When defining classes for IDL code generation, you provide IDL-specific information by using the Class Table Editor, the Class and Relationship dialogs, the Properties dialog, and the Object Annotation Editor.

You define class attributes and operations for IDL code generation using these sections of the Class Table Editor:

- Class member definitions
  See “Entering Class Member Definitions for IDL.”
- Analysis items
  See “Entering Analysis Items for IDL” on page 4-4.
- IDL implementation items
  See “Entering Implementation Items for IDL” on page 4-4.

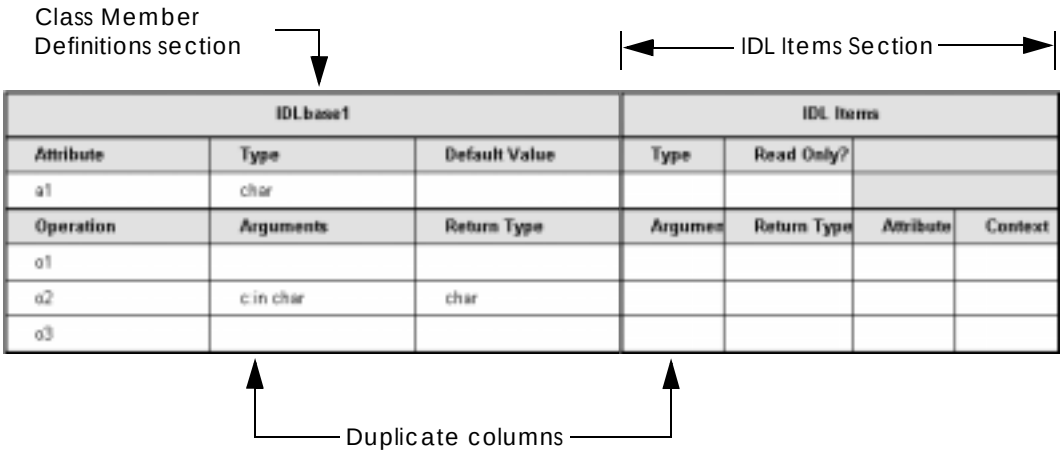
For general information about entering information in class tables for code generation, see “Defining Classes for Code Generation” on page 2-3.

Entering Class Member Definitions for IDL

The Class Member Definitions section of the Class Table contains two columns that duplicate columns in the IDL Items section, as shown in Figure 1:

- Attribute Type
- Operation Arguments

Figure 1: IDL Items in the Class Table



You can add IDL values into either section to produce valid IDL code. The advantage of adding them in the IDL Items section is that you can then reserve the standard section for defining the object (rather than the interface, which IDL generates).

When you generate IDL code, StP uses values defined in the IDL Items section. If none are supplied, it uses the values in the standard section.

See “Example of Generated IDL Code” on page 4-11 for an example of overloaded arguments and types.

Entering Analysis Items for IDL

The information you provide in the Analysis Items section of a class table is common to all languages, as described in “Analysis Items” on page 2-5.

Entering Implementation Items for IDL

You enter IDL implementation items in the IDL Items section of a class table. To display the IDL Items section of a class table, follow instructions given in *Creating UML Models*.

Entering IDL Items for Attributes

Table 2 lists and describes the attributes columns in the IDL Items section of the class table.

Table 2: Class Table Attribute IDL Items

Header	Description
Type	The attribute's type, in IDL terms (a string). Use this to declare an IDL-specific type that is different from the attribute's regular type.
Read Only?	If False, indicates the attribute can be modified (True / False). The default is False.

Entering IDL Items for Operations

Table 3 lists and describes the operations columns in the IDL Items Section of the class table.

Table 3: Class Table Operation IDL Items

Header	Description
Arguments	The operation's arguments, in IDL terms (a string). Use this to declare IDL-specific arguments that are different from the operation's regular arguments.
Return Type	The operation's return type, in IDL terms (a string). Use this to declare an IDL-specific return type that is different from the operation's regular return type.
Attribute	A specified property or characteristic of the operation (a string). The IDL standard defines only one value—"oneway."
Context	An IDL expression that specifies which elements of the client's context may affect performance of the operation (a string).

Adding IDL Annotations

This section describes available IDL annotations. IDL-specific annotations include:

- Class IDL Declarations
- Role IDL Definition
- Object descriptions on classes (can be generated as comments)

For general information about annotating objects, see "Annotating Classes and Relationships for Code Generation" on page 2-7.

For instructions on adding annotations common to all languages, such as multiplicity and navigability, refer to *Creating UML Models*.

Table 4: Annotations for IDL Code Generation

Object	Note	Item	Item Value	Description
Class	IDL Declarations	Note Description dialog	Lets you add IDL-specific declarations not represented by UML constructs, such as IDL-specific constants, types, or exceptions that are scoped to the class as notations. You add the text as a Note Description. When generated, these are placed in the interface definition. For more information, see "Adding Class IDL Declarations" on page 4-7.	
Role	Role IDL Definition	Read-Only	True	Causes an IDL get statement to be generated.
			False	Causes IDL get and set statements to be generated.
		Pattern Family	Default / Instance	Lets you indicate the set of association configuration patterns to use. Templates can be customized in *<stp_file_path>/templates/uml/qrl/code_gen/idl/association_patterns.inc*.

Table 4: Annotations for IDL Code Generation (Continued)

Object	Note	Item	Item Value	Description
Class, Attribute, or Operation	Object		Lets you add IDL comments for the object as a Note Description. If you generate the Note Description: • Class comments are placed above the interface declaration • Attribute or operation comments are placed above the attribute or operation definition	

Adding Class IDL Declarations

You can add code for IDL constants, types, and exceptions using the IDL Declarations note for the class. For instructions, see “Annotating Classes and Relationships for Code Generation” on page 2-7.

Annotating Roles for Implementation of Associations

StP / UML allows associations to be implemented in IDL on a case-by-case basis through IDL annotations of an association role. You use annotations to control the way StP generates code for the selected end of an association. You can:

- Suppress the code
- Indicate the set of association configuration patterns to use

You control the way associations and relationships are implemented by using one of these methods:

- Annotating the association or relationship using the OAE, as described in this section
- Setting properties of the **Generate IDL** command, as described in “Generating IDL Code” on page 4-9

The command properties method sets the default implementation. The OAE method sets the implementation and overrides the default implementation for individual associations or relationships.

Specifying Read Only on an Association Role

To generate an IDL get statement (but no set statement) on an association role:

1. From the OAE, choose the **Role IDL Definition** note.
2. Set the value of the **Read Only** item to **True**.

For general information on using the OAE, refer to *Fundamentals of StP*.

Specifying the Pattern Family

You can tailor the data members of a class generate to implement associations in IDL by using code generation patterns. The pattern used depends on the multiplicity and qualification of the association. A complete set of these patterns, which includes every combination of multiplicity and qualification of an association, is called a pattern family, because every member of the set has something in common.

A pattern is a text string usually containing one or more of the UML IDL string substitution symbols.

StP/UML lets you specify which pattern family to use when roles are implemented for IDL code generation.

Specifying a pattern family is not required.

The choices are:

- Default
- Instance

For IDL code generation, Default and Instance are identical.

You can customize existing and add new pattern families. Patterns you can customize and configure appear in the *<stp_file_path>/templates/uml/qrl/code_gen/idl/association_patterns.inc* file.

To specify the pattern family, use one of these methods:

- Select the pattern family from the Generate IDL dialog, as described in “Generating IDL Code” on page 4-9

- Use the Object Annotation Editor (OAE). See Table 4 on page 4-6 for information on the Role IDL Definition note and the Pattern Family item.

  For general information on using the OAE, refer to *Fundamentals of StP*.

Generating IDL Code

IDL code generation commands have associated command properties that enable you to specify output file directory, name, and extensions.

To generate IDL code from your model:

1. In the Model pane on the StP Desktop, open the appropriate category.
2. Based upon the code you want to generate, select a particular diagram or class(es) from the objects pane.
3. Choose a **Generate IDL for <object>** command from one of the menus listed in Table 6 on page 2-11.
4. In the **Generate IDL for...** dialog box, set the appropriate command properties, as described in Table 5.
5. Click **OK** or **Apply**.

For additional information, see “Generating Code from the StP Desktop” on page 2-10.

Generate IDL Command Properties

Table 5 contains descriptions of IDL code generation command properties.

Table 5: Generate IDL Command Properties

Property	Description
Directory	Specifies the pathname for the directory where output code is written. The default is *proj/sys/src_files*.

Table 5: Generate IDL Command Properties (Continued)

Property	Description
One File Name Per Class	Generates a separate file for each class (IDL interface) that is selected for code generation. The file names are taken from the class names when this option is selected. This option is the default.
File Name	Active when One File Name Per Class option is deselected. Generates code for all classes to one file and permits you to specify its name.
Preserve Code From Other Classes	Requires One File Name Per Class to be deselected. Retains or deletes any previously-generated code for classes and operations not in the current set of classes for which code is being generated. Default is not selected (do not keep).
IDL Output File Extension	Specifies the extension appended to the name of the output file. The default is *.idl*.
Generate Code for Types / Interfaces	If ON, generated IDL for all types and interfaces on a diagram, or a selected type or interface (from the Generate IDL for Class dialog). Default is ON.
Generate Type Attributes	If ON, generates IDL code for the attributes associated with all types on a diagram, or a selected type or interface. Default is ON. (This option is not available if the Generate Code for Types / Interfaces option is OFF.)
Ignore Classes Annotated As External	Does not generate code for selected class types. Options are External, Persistent, and Derived. Default is External only.
Generate Comments For	Generates comment lines with the text of Object annotations for Classes. Default is Not Selected.

Table 6: External Variables (IDL Only)

Variable	Type	Meaning
implement_ associations_ using	string	Specifies the way to implement associations. Options are to Suppress code or implement as embedded Instance. Default is Instance.

Table 6: External Variables (IDL Only) (Continued)

Variable	Type	Meaning
implement_ associations	boolean	If True, implements associations. Default is True.
implement_ aggregations	boolean	If True, implements . Default is True.

Example of Generated IDL Code

This example shows IDL code generated for the *Bank* class in the Bank Example, shown in Figure 2.

Figure 2: Bank Example Class Diagram

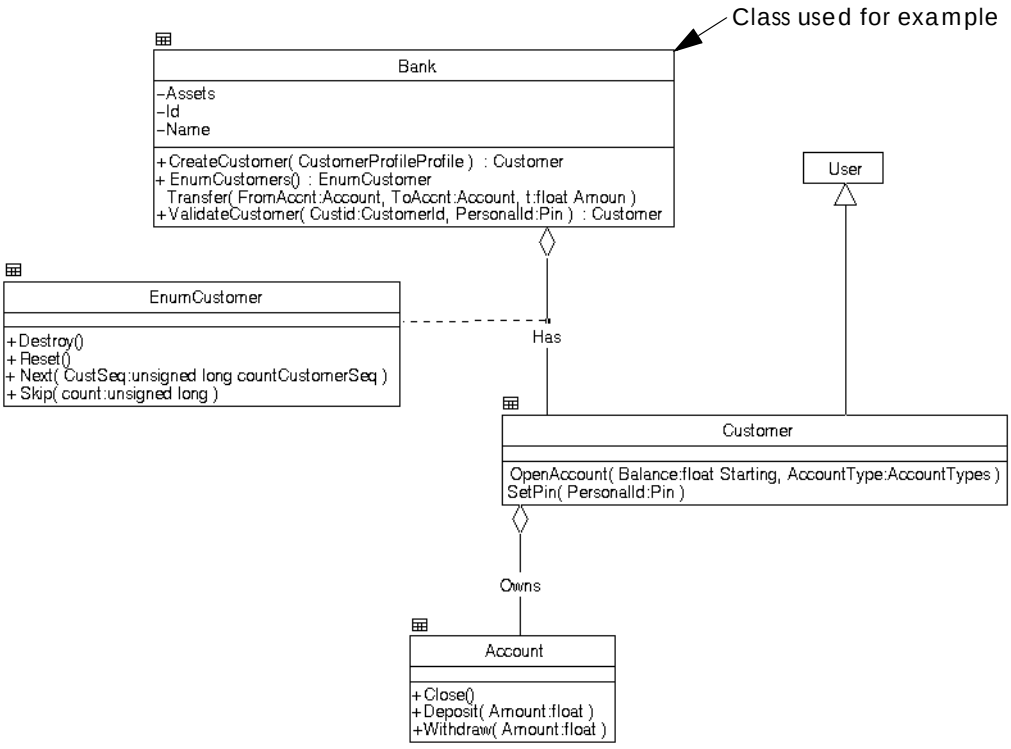


Figure 3 shows the class table for the *Bank* class.

Figure 3: Bank Class's Class Table

Bank			Analysis Items	
Attribute	**Type**	**Default Value**	**Class Attr?**	**Derived?**
Assets	float		False	False
Id	string		False	False
Name	string		False	False
Operation	**Arguments**	**Return Type**	**Class Op?**	**Abstract?**
CreateCustomer	CustomerProfile Profile	Customer	False	False
EnumCustomers		EnumCustomer	False	False
Transfer	Account FromAccnt, Account ToAccnt, float Amour	void	False	False
ValidateCustomer	CustomerId Custid, Pin PersonalId	Customer	False	False

Class table for Bank class: Standard section

IDL Items section

Bank	IDL Items			
Attribute	**Type**	**Read Only?**		
Assets		True		
Id		True		
Name		True		
Operation	**Arguments**	**Return Type**	**Attribute**	**Context**
CreateCustomer	in CustomerProfile Profile			"Bank Role"
EnumCustomers				"Bank Role"
Transfer	in Account FromAccnt, in Account toAccnt, in float Amount			
ValidateCustomer	in CustomerId Custid, in Pin PersonalId		readonly	

Three operations have overloaded arguments (arguments in both the standard definition section and in the IDL Items section): *CreateCustomer*, *Transfer*, and *ValidateCustomer*. When StP generates IDL code for this class, the values in the IDL Items section override the values in the standard section.

Bank.idl File

This section contains the output file generated for the *Bank* class using the command properties shown in Figure 2 on page 4-12.

```
// StP -- created on Mon Dec 18 14:54:52 1995 for janes@verdi
from system joecore20

#ifndef _Bank_idl_
#define _Bank_idl_

// stp class declarations
interface Bank;
// stp class declarations end

// stp class definition 25705
interface Bank
{
// stp class members
   readonly attribute float Assets;
   readonly attribute string Id;
   readonly attribute string Name;
   Customer CreateCustomer(in CustomerProfile Profile)
      context(\"Bank.Role\");
   EnumCustomer EnumCustomers() raises(NotAuthorized)
      context(\"Bank.Role\");
   void Transfer(in Account FromAccnt, in Account ToAccnt,
      in float Amount);
   readonly Customer ValidateCustomer(in CustomerId Custid,
      in Pin PersonalId);
// stp class members end
};
// stp class definition end

// stp footer
#endif
// stp footer end
```

About the Code Generated for the Bank Class

This section compares the code generated for the *Bank* interface with the class table (Figure 3) for the *Bank* class.

For attributes:

- Standard attribute types used because no IDL-specific types were provided
- All attributes declared read only, as specified in the class table under IDL Items

For operations:

- IDL-specific arguments overrode standard operation arguments
- Standard return types used because no IDL-specific return types were provided
- IDL-specific Raises and Context conditions shown

Generating IDL Incrementally

This section contains information that applies to incremental code generation for IDL. For a general discussion of incremental code generation, see “Generating Code Incrementally” on page 2-11.

If you manually add code for IDL constants, types, or exceptions into the output file of a class that was generated previously, the **Preserve Code From Other Classes** option retains them; see “Generate IDL Command Properties” on page 4-9.

5 Generating Ada_95 Code

This chapter shows how to generate Ada_95 code from an StP / UML model. It describes:

Use this chapter with Chapter 2, "Generating Code From UML Models." Chapter 2 provides information for generating code that applies to all languages.

What is Generated for Ada_95

For each class in a class diagram that is not external, StP generates:

- Package specification file (*.ads* default file extension)—Contains package specifications for the associated classes and generic instantiations making use of the class types
- Package body file (*.adb* default file extension)—Contains package body or bodies for the associated classes, including operations not implemented as subunits

- Subunit file (*.adu* default file extension)—Contains operations for the associated classes implemented as subunits

The default directory is the *src_files* directory in your <*project*>/<*system*> directory. The generated code conforms to the international ANSI / ISO / IEC-8652:1995 standard.

StP to Ada_95 Mapping

Various elements of the model generate specific Ada_95 constructs. Table 1 summarizes the correlation between UML elements and the Ada_95 constructs they generate. For complete mapping information, see Chapter 6, “Mapping StP to Ada_95.”

Table 1: Summary of StP/UML to Ada_95 Mapping

UML Construct	Generated Ada Construct
Package	Package
Class	Tagged type within a package
Attribute	Component field of class tagged type (declared outside a class type when designated as a class attribute)
Operation	Subprogram declared in the class package. Can be a procedure or a function.
Generalization Relationships	
Single inheritance	Tagged types - type extension
Multiple inheritance	One superclass chosen from which to inherit
Associations	
Associations	Additional component field of tagged type. The type is either an access type to a tagged type or a type exported from a generic instantiation.

Table 1: Summary of StP/UML to Ada_95 Mapping (Continued)

UML Construct	Generated Ada Construct
Multiplicity many	Uses generic set package
Multiplicity many, ordered	Uses generic list package
Qualified	Uses generic map package
Inverse qualified	Uses generic tuple package
N-ary	N binary one-way associations
Link attribute	Independent association class package and type
Association class	Independent association class package and type
Role name	Name of association reference class record field
Aggregation	Same as association
Composition	Same as association
Class attribute	Component field of a class package
Class operation	Subprogram in class package

Creating Classes for Ada_95 Code Generation

The following sections provide information to create classes for generating Ada_95 code.

Case Sensitivity

Although StP is sensitive to upper and lower case, Ada_95 is not. Therefore, if you create two class symbols in StP with names that differ

only in their case (such as *Class_A* and *class_a*), you will generate duplicate declarations in Ada_95.

Inheriting from Types in Library Units

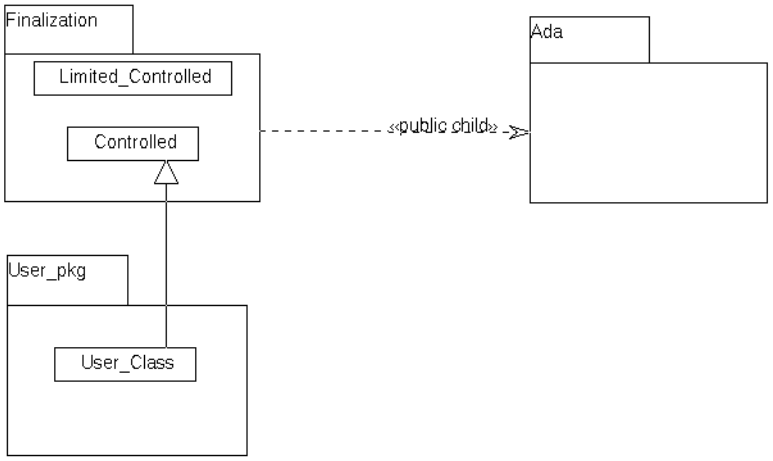
As StP / UML allows you to model an external class, you can model an external package for StP / UML-Ada_95 code generation. You can do this using the StP / UML Annotation Editor or the Properties dialog box available for the package from the Class Editor.

For example, in order to inherit from either the Controlled or Limited_Controlled classes in the Ada.Finalization package, you must first model the library structure with external packages and classes, then create the new class as a subclass. When you generate code, check the option to ignore classes and packages annotated as external using the Property dialog box, so code for the predefined types is not generated.

This mechanism is not limited to finalizations. You can use it to generate code for tagged types which extend tagged types in other library units.

Figure 1 shows an example model.

Figure 1: Modeling a Class as External



Adding Ada_95 Information to Class Tables

You define class attributes and operations using these sections of the Class Table Editor:

- Class member definitions
  See “Entering Class Member Definitions.”
- Analysis items
  See “Entering Analysis Items.”
- Ada_95 implementation items
  See “Entering Ada_95 Implementation Items” on page 5-6.

For general information about entering information in class tables, see “Defining Classes for Code Generation” on page 2-3.

Entering Class Member Definitions

Most of the information entered in the Class Member Definitions section of a class table is common to all languages, as described in “Adding Class Member Definitions” on page 2-3.

In Ada_95, for an operation to be a *primitive* operation of the class type, one of the arguments or the return type must be the class type. Neither of these types generates automatically.

If you specify a return type, the operation implements as a function.

Entering Analysis Items

Most of the information entered in the Analysis Items section of a class table is common to all languages, as described in “Analysis Items” on page 2-5. This section provides information about analysis items that is specific to Ada_95 code generation.

Table 2 lists and describes the attribute columns in the Analysis Items section that affect Ada_95 code generation.

Table 2: Class Table Attribute Analysis Items

Header	Description
Class Attr?	Specifies whether the attribute generates as a stand-alone variable rather than as a component of the class tagged type (True / False).
Derived?	Not used by Ada_95 code generation.

Table 3 lists and describes the operation columns in the Analysis Items section that affect Ada_95 code generation.

Table 3: Class Table Operation Analysis Items

Header	Description
Class Op?	Specifies whether or not an operation is a class operation, although there is no difference in the code generated for class operations other than their placement (True / False). Certain checks are made, such as insuring that a class operation is not also designated as abstract.
Abstract?	Specifies whether an **abstract** subprogram generates (True / False). Cases where that is not appropriate, such as when the class type is not abstract, are detected by code generation.

Entering Ada_95 Implementation Items

You enter Ada_95 implementation items in the Ada_95 Items section of the class table. To display the Ada_95 Items section of a class table, follow instructions given in *Creating UML Models.*

Table 4 lists and describes the attribute columns in the Ada_95 Implementation Items section.

Table 4: Ada_95 Class Table Attribute Information

Header	Description
Aliased?	Specifies whether or not the attribute generates as an aliased component (True / False).
CA Visibility	Specifies Class Attribute Privacy (Public, Private, Implementation, or unspecified). This applies only to attributes designated as class attributes. Public ones generate in the visible part of the class package specification, Private ones in the private part of the package specification, and Implementation ones in the package body. If unspecified, Public is used.

Table 5 lists and describes the operation columns in the Ada_95 Implementation Items section.

Table 5: Ada_95 Class Table Operation Information

Header	Description
Visibility	Specifies operation visibility (Public, Private, or Implementation). You declare public operations in the visible part of the class package specification; you declare private ones in the private part of the class package specification, and Implementation ones in the package body. The code for all visibility types appears in the package body, unless you override this by electing to use separate compilation units (*.adu* extension).
Subunit?	Specifies whether the subprogram for the operation generates as a separately compilable unit with the default extension "*.adu*." (True, False, or unspecified). If unspecified, the global subunit option takes effect.
Inline?	Specifies whether a pragma inline statement generates for the operation's subprogram (True / False).

Adding Ada_95 Annotations

This section describes the Ada_95 annotations available for UML. For general information about annotating objects, see “Annotating Classes and Relationships for Code Generation” on page 2-7.

StP does not check for illegal combinations of annotations.

Ada_95 Annotations

Table 6 provides a summary of Ada_95 annotations.

Table 6: Ada_95 Annotations

Object	Note	Item	Item Value	Description
Class	Class Ada_95 Definition	Class Type Privacy	Specifies how the generated class tagged type appears in its enclosing package specification.	
			Public	Full definition in visible part.
			Private Extension	Derivation in visible part, but extension in private part. Applies only to subclasses.
			Tagged Private	Tagged private in visible part, full definition in private part.
			Private	Private in visible part, full definition in private part.
			Implemen-tation	Incomplete type declaration in the private part. Full definition in package body. Allows the implementation of two-way associations as two one-way associations while avoiding cyclic compilation dependency between the associated class package specifications.
		Limited	Specifies that the class type is limited. This applies only to root classes.	
		Generate Access Type to Class Type	Specifies whether to generate an access type to the class tagged type in the class package specification.	
			False	Not generated
			True	Generated

Table 6: Ada_95 Annotations (Continued)

Object	Note	Item	Item Value	Description
Class	Class Ada_95 Implementation	Generate Access Type to Class-Wide Type	Specifies whether to generate an access type to the class-wide type associated with the class tagged type in the class package specification.	
			False	Not generated
			True	Generated
		Partial View Limited / Full View Not Limited	Specifies whether to not designate the full view of a class type as limited, if the partial view is limited and has a privacy of Private. Applies only to root classes.	
			False	Full view limited
			True	Full view not limited
		Partial View Abstract / Full View Not Abstract	Specifies whether to not designate the full view of a class type as abstract, when the partial view is abstract and has a privacy of either Private Extension or Tagged Private.	
			False	Full view abstract
			True	Full view not abstract
Operation	Ada_95 Source Code	Note Description dialog	Lets you type Ada_95 source code for the body of the operation in the Note Description. For instructions, see "Adding Source Code to Operations" on page 2-8.	
Class, Attribute, or Operation	Requirement	Name	Lets you type Ada_95 comments in the Note Description dialog or the Value field for the item. For instructions on using the Note Description dialog, see "Adding Comments" on page 2-9.	
		Document		
		Paragraph		
	Object	Author		
		Generated		
		(I18N) Generate as		

Generating Ada_95 Code

Ada_95 code generation commands have associated command properties that enable you to specify a number of global options. These properties appear on the Generate Ada_95 for <object> dialog.

To generate Ada_95 code from your model:

1. In the Model pane on the StP Desktop, open the appropriate category.
2. Based upon the code you want to generate, select a particular diagram, or package(s) from the objects pane.
3. Choose a **Generate Ada_95 for <object>** command from one of the menus listed in Table 6 on page 2-11.
4. In the **Generate <language> for...** properties dialog box, set the appropriate command properties, as described in Table 7.
5. Click **OK** or **Apply**.

For additional information, see “Generating Code from the StP Desktop” on page 2-10.

Generating Ada_95 Command Properties

Table 7 contains descriptions of Ada_95 code generation command properties.

Table 7: Generate Ada_95 Command Properties

Property	Description
Directory:	Specifies the pathname for the file directory where output code is written. The default is *proj/sys/src_files*.
One File Name Per Package	Generates a separate set of package specification, package body, and subunit files for each package selected for code generation. The file names are based on the Generated File Name Base when you select this option. The default is to be selected.

Table 7: Generate Ada_95 Command Properties (Continued)

Property	Description
Use Package Component for File Name	If selected, overrides the other file name options and uses the component information you specified for the package for the name of the generated file. The full name comprises the directory from the Output File Directory option, the name of the component, and the appropriate File Extension. The default is not to be selected.
File Name:	Active when you deselect the One File Name Per Package option. Generates code for all packages to one package specification file, one package body file, and one subunit file, and permits you to specify their base name. The generated package specifications are automatically ordered, if possible, to allow correct compilation.
Preserve Code From Other Packages	When you deselect One File Name Per Package, specifies whether to preserve the code already in the file previously generated for packages not included in the current group being generated. The default is to be not selected.
File Name Base:	Active when you select One File Name Per Package option. The file name base can either be the package name, or the fully qualified Ada unit name in which the class type is declared. The full Ada unit name includes all of the unit's parents in the hierarchical library. The full Ada unit name is required when using the free GNAT compile. You may want it in other cases as well. The default is to use the full Ada unit name.
Case:	When you select One File Name Per Package, this item specifies the case of the name of the generated file as: As Is, Lower, or Upper. The default is As Is.
Generate (with File Extension) Pkg Specs	Specifies whether to generate package specification files. Provides a text entry box for the file extension for package specification files. The default is *.ads*.
Generate (with File Extension) Pkg Bodies	Specifies whether to generate package body files. Provides a text entry box for the file extension for package body files. The default is *.adb*.
Generate (with File Extension) Subunits	Specifies whether to generate subunit files. Provides a text entry box for the file extension for subunits. The default is *.adu*.

Table 7: Generate Ada_95 Command Properties (Continued)

Property	Description
Root Class Type Privacy:	Specifies the privacy of the class tagged type in its enclosing package specification for classes with no superclasses. Choices are Public, Tagged Private, Private, or Implementation. The default is Tagged Private.
Root Class Type Limited:	Specifies whether the class type is limited or not for classes with no superclasses. The default is to be not selected.
Subclass Type Privacy:	Specifies the privacy of the class tagged type in its enclosing package specification for subclasses. Choices are Public, Private Extension, Tagged Private, Private, or Implementation. The default is Private Extension.
Implement Association / Aggregation / Composition	Indicates that Ada_95 code is generated to implement associations and aggregations / compositions between classes. The default is to be selected.
Generate Access Type For Class Type	Specifies whether to generate an access type to the class tagged type in the class package specification. The default is to be selected.
Generate Access Type For Class-Wide Type	Specifies whether to generate an access type to the class-wide type associated with the class tagged type in the class package specification. The default is to be selected.
For Operations Use Subunits	Indicates that operation subprograms are implemented as subunits. The default is to be not selected.
For Operations Generate Null Code	Specifies whether to generate a null statement in procedures, and a dummy return statement in functions, so that compilable code generates for operation subprograms without entering any source code either in the Operation Source Code note description or the source code files themselves. Useful in early development. The default is not to be selected.
Ignore External Packages / Classes	Does not generate code for external packages or classes depending on the option you select. The default is to be selected.
Generate Comments For:	Generates comment lines with the text of Object and Requirement annotations for the selected object(s). When selected, options are Classes, Attributes, and / or Operations. The default is not to be selected.

Table 7: Generate Ada_95 Command Properties (Continued)

Property	Description
Format to:	Modifies the number of comment characters per line. The default is 80.

Using Global Options and Annotations

A value set in the annotation of an individual entity in the model (such as a class, a relationship, or an association) takes precedence over the corresponding global option specified in the Command Properties dialog box. When you generate code for subsets of an entire model, use consistent values for global options and annotations for each subset generated. Inconsistent values can produce code that does not compile.

Avoiding Cyclic Compilation Dependencies

If you choose improper values for global options or annotations, you can generate code containing cyclic compilation dependencies among a group of package specifications.

You can use the Implementation class type privacy to avoid these cycles. The drawback of Implementation privacy is that the tagged type is not defined in a package specification, so you cannot define *primitive* operations on that tagged type.

If a component field of a class type refers to another class, a dependency (context clause) on that class exists. When the class type privacy is Implementation, the context clause is on the package body. In that way, no cycles can occur. For the other privacies, the context clause is on the package specification, so cycles can occur.

For example, if a binary association is traversed in both directions, and the class type privacy of neither of the associated classes is Implementation, the package specifications of the two classes must “with” each other, and there is a cyclic compilation dependency. Similarly, cycles can result through association traversal loops, and associations between superclasses and subclasses.

In general, when Implementation privacy is not used, dependencies generate in the following ways:

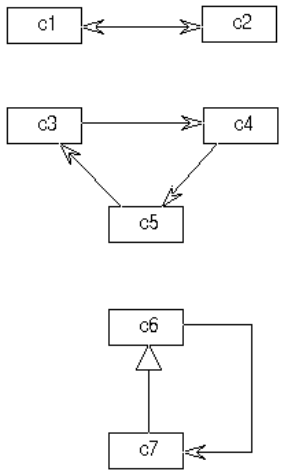
- For generalizations or inheritance relationships, a subclass depends directly on its immediate superclasses, leading to a chain of dependencies up to the root classes.
- For binary associations, given a direction of traversal, a source class depends on the destination class.
- For n-ary associations, all classes depend on the central association class, which in turn depends on all the other classes.

  Therefore, n-ary associations must always use Implementation privacy.

These dependencies are like a directed graph. Any cycle in the graph means a cyclic compilation dependency exists. Code generation issues a warning message when it detects this condition in the set of classes for which code is being generated. Other classes already in the files due to previous code generations are not included in this analysis.

Figure 2 shows examples of three models leading to cyclic compilation dependencies if you do not use Implementation privacy.

Figure 2: Cyclic Compilation Dependencies



Generating Ada_95 Incrementally

This section contains Ada_95-specific details for incremental code generation. For general information about incremental code generation, see “Generating Code Incrementally” on page 2-11.

Incremental code generation plays an important role in the development of an Ada_95 system using StP/UML. Because there are no Ada_95-specific features built into StP/UML, you must add some constructs into the generated code manually.

Uses of Incremental Code Generation

Use incremental code generation to:

- Insert necessary context clauses
- Define arbitrary Ada_95 types for use in attribute types, operation argument types and operation return types
- Insert source code for operation subprograms
- Declare exceptions

In addition to these uses, incremental code generation also allows you to enter Ada_95 source code manually.

Context Clauses

StP generates certain context clauses when it can determine the need for them from the model. For example, StP generates context clauses automatically when:

- A subclass depends on a superclass
- An association source class depends on a destination class

You can manually add a context clause to any generated unit and StP retains the clause on subsequent generations.

Ada_95 Types

Certain Ada_95 types generate automatically:

- Class type
- Access types to the class type and class-wide type

If you want to define other types for use in your programs, you can define them in the proper region of any desired unit, and StP retains them on subsequent code generations. You can enter context clauses to provide visibility to these types.

Source Code for Operation Subprograms

You can add the source code for an operation subprogram by:

- Using the OAE to enter an Ada_95 Source Code note description.

  With this method, StP automatically generates the code from the object model. For more information on using the OAE, see “Adding Ada_95 Annotations” on page 5-8.
- Entering the source code in the proper region for operation subprograms in the generated files.

  When you subsequently generate code, StP retains the code you entered.

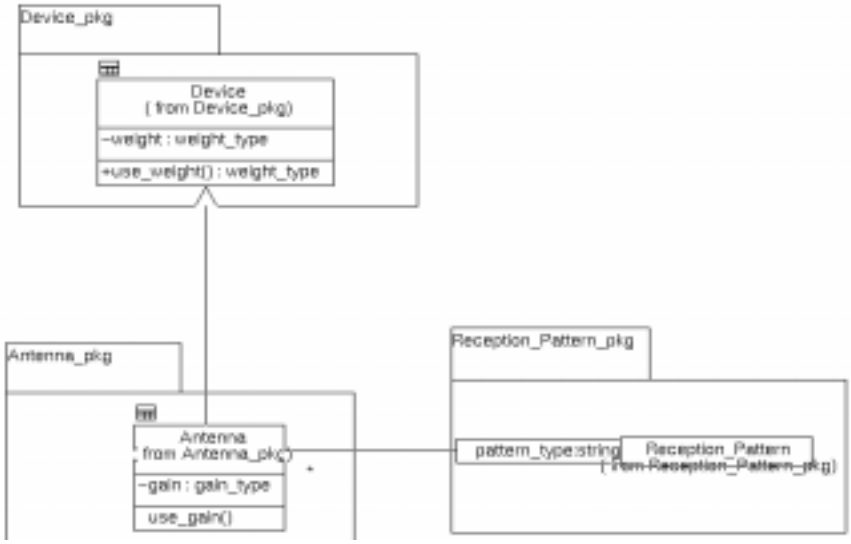
Ordering of Declarations

When you generate more than one class to the same package specification file, StP orders the package specifications correctly (if possible) to allow compilation. However, if you generate code incrementally, the package specifications in the resulting file may not be in the correct order, because all newly added specifications come after those that were already in the file.

To fix this problem, manually move the newly generated specifications to their proper positions. They will maintain that order on all subsequent generations. The same point is true of the ordering of generic instantiations associated with a particular package.

Figure 3 shows a model containing the class *Antenna*, which is a subclass that is involved in an association for which instantiations must be generated.

Figure 3: Incremental Code Generation



The following sections show the package specification file, package body file, and subunit file generated for the *Antenna* class, generated with Implementation privacy and a Public Child package. Arrows indicate the regions where you can add and retain source code on subsequent generations.

For specific descriptions of regions where you should not add source code, see “Guidelines for Changing Files” on page 5-21.

Package Specification File

```
--StP|
--StP|  Created on 06/18/98 12:17:18 for colin from system
       uml_ada
--StP|
```

```
--StP|  begin scope package  --  418

package Antenna_pkg is

   --StP|  begin block public forward declarations  --  5
   type Antenna_cptr is private;
   type Antenna_ptr is private;
   --StP|  end block public forward declarations  --
   --StP|  begin block public declarations  --  5
   function use_gain (self:Antenna) return gain_type;
   --StP|  end block public declarations  --

private

   --StP|  begin block private forward declarations  --  5
   type Antenna;
   type Antenna_cptr is access all Antenna'class;
   type Antenna_ptr is access all Antenna;
   --StP|  end block private forward declarations  --

end Antenna_pkg;

--StP|  end scope package  --

--StP|  begin block set  --  5
with generic_set;
with Antenna_pkg;
package Antenna_set is new generic_set
(Antenna_pkg.Antenna_cptr);
--StP|  end block set  --

--StP|  begin block map_to_set  --  string  5
with generic_map;
with Antenna_set;
package string_Antenna_set_map is new generic_map (string,
Antenna_set.set);
--StP|  end block map_to_set  --
```

Package Body File

```
--StP|
--StP|  Created on 06/18/98 12:17:18 for colin from system
```

```
       uml_ada
--StP|

--StP|  begin scope package body  --  418
with string_Reception_Pattern_tuple;
with Device_pkg;
package body Antenna_pkg is

   --StP|  begin block implementation declarations  --  5
   type Antenna is new Device_pkg.Device with record gain :
      gain_type;
      Reception_Pattern_asc :
      string_Reception_Pattern_tuple.tuple;
   end record;
   --StP|  end block implementation declarations  --

   --StP|  begin block operation stub  --  7
   function use_gain (self:Antenna) return gain_type is
      separate;
   --StP|  end block operation stub  --

end Antenna_pkg;

--StP|  end scope package body  --
```

Subunit File

```
--StP|
--StP|  Created on 06/18/98 12:17:18 for colin from system
uml_ada
--StP|

--StP|  begin scope operation  --  7

separate (Antenna_pkg)
function use_gain (self:Antenna) return gain_type is
begin

      --StP|  begin block code  --
      --StP|  end block code  --

end use_gain;
```

```
--StP|  end scope operation  --
```

Guidelines for Changing Files

Ada_95 code generation from UML generates code with two types of comment tags: those that define blocks and those that define scopes. In either case, the comment tags always come in pairs. All comment tags begin with the characters, `--StP|`. Block comment tags take the form, `--StP| begin block...` and `--StP| end block...`. Scope comment tags take the form `--StP| begin scope...` and `--StP| end scope...`.

This prefix, and the actual tag comments are QRL strings defined in the file *<base_dir>/templates/uml/qrl/code_gen/ada_95/customize.inc*. You can customize these strings, as described in "Tag Comments for Incremental Code Generation" on page 5-27.

You must not edit any generated code or add any new code between block comment tags. You must not edit any generated code between scope comment tags, but you may add new code between scope comment tags.

Typical places where you would want to do this are for manually adding context clauses or adding operation code.

You may nest scopes: you can place pairs of block comment tags or other scope comment tags between any particular pair of scope comment tags.

While editing code, you may move complete blocks as long as they remain within the same pair of scope tags. (However, you must not rearrange the code inside the block.) You may also move entire regions of code beginning and ending with matched scope comment tags so long as the nesting of scope comment tags remains the same. An example of needing to move code is that the addition of new code results in new dependencies between declarations which constantly must be reordered.

You can identify classes and operations by unique repository-independent identification numbers, which appear in the tag comments. This allows incremental code generation to behave correctly, even when you rename a class or operation using the StP Object Rename utility, because the renamed class or operation will still have the same ID.

The following sections describe the default scope and block comment tag pairs for the various files.

Package Specification File (Scopes)

```
--StP|  begin scope package  --  NNNN
--StP|  end scope package  --
```

Package Specification File (Blocks)

```
--StP|  begin block public forward declarations  --  NNNN
--StP|  end block public forward declarations  --

--StP|  begin block public declarations  --  NNNN
--StP|  end block public declarations  --

--StP|  begin block private forward declarations  --  NNNN
--StP|  end block private forward declarations  --

--StP|  begin block private declarations  --  NNNN
--StP|  end block private declarations  --

--StP|  begin block package instantiation  --  NNNN
--StP|  end block package instantiation  --

--StP|  begin block set  --  NNNN
--StP|  end block set  --

--StP|  begin block list  --  NNNN
--StP|  end block list  --

--StP|  begin block map  --  XXXX  NNNN
--StP|  end block  --

--StP|  begin block map_to_set  --  XXXX  NNNN
--StP|  end block map_to_set  --

--StP|  begin block map_to_list  --  XXXX  NNNN
--StP|  end block map_to_list  --

--StP|  begin block tuple  --  XXXX  NNNN
--StP|  end block tuple  --

--StP|  begin block set_of_tuple  --  XXXX  NNNN
--StP|  end block set_of_tuple  --

--StP|  begin block list_of_tuple  --  XXXX  NNNN
--StP|  end block list_of_tuple  --
```

```
--StP|  begin block map_to_tuple  --  XXXX  YYYY  NNNN
--StP|  end block map_to_tuple  --

--StP|  begin block map_to_set_of_tuple  --  XXXX  YYYY  NNNN
--StP|  end block map_to_set_of_tuple  --

--StP|  begin block map_to_list_of_tuple  --  XXXX  YYYY NNNN
--StP|  end block map_to_list_of_tuple  --
```

Package Body File (Scopes)

```
--StP|  begin scope package body  --  NNNN
--StP|  end scope package body  --

--StP|  begin scope operation  --  NNNN
--StP|  end scope operation  --
```

Package Body File (Blocks)

```
--StP|  begin block implementation declarations  --  NNNN
--StP|  end block implementation declarations  --

--StP|  begin block operation stub  --  NNNN
--StP|  end block operation stub  --

--StP|  begin block code  --  NNNN
--StP|  end block code  --
```

Subunit File (Scopes)

```
--StP|  begin scope operation  --  NNNN
--StP|  end scope operation  --
```

Subunit File (Blocks)

```
--StP| begin block code  --  NNNN
--StP|  end block code  --
```

Customizing Generated Ada_95 Code

This section provides information on customizing Ada_95 code generated from StP/UML.

Ada_95 Customization Declarations

Ada customization declarations appear in a QRL include file, *customize.inc*, located in the directory <*base_dir*>/*qrl*/*code_gen*/*ada_95*, where <base_dir> is the directory specified by the ToolInfo variable *stp_file_path* (or *IDE_PRODUCT_stp_file_path*).

The declarations divide into the following groups:

- Affixes for Ada identifiers based on class name
- File name generation
- Generic package and type names
- Formatting output
- Tag comments for incremental code generation

The next sections list the strings used for these different groups of declarations listed in the *customize.inc* file.

Affixes for Ada Identifiers Based on Class Name

Table 8 lists the strings used as affixes for Ada identifiers based on class name. In the generated code, these identifiers are:

```
<prefix> <class_name> <suffix>
```

Some affixes apply to the class name for which they are generated, while others apply to a referenced class name.

Table 8: Strings for Affixes for Ada Identifiers

QRL Name	Default Value	Purpose
PACKAGE_NAME_PREFIX		Prefix and suffix of class package names
PACKAGE_NAME_SUFFIX		
CLASS_TYPE_NAME_PREFIX		Prefix and suffix of class tagged types in class packages
CLASS_TYPE_NAME_SUFFIX		
ACCESS_TYPE_NAME_PREFIX		Prefix and suffix of access types to class types
ACCESS_TYPE_NAME_SUFFIX	_ptr	
CLASSWIDE_ACCESS_TYPE_NAME_PREFIX		Prefix and suffix of access types to class-wide types associated with class types
CLASSWIDE_ACCESS_TYPE_NAME_SUFFIX	_cptr	
ASSOCIATION_REFERENCE_PREFIX		Prefix and suffix of class tagged type fields referencing object(s) of another class in an association
ASSOCIATION_REFERENCE_SUFFIX	_asc	
AGGREGATION_WHOLE_REFERENCE_PREFIX		Prefix and suffix of class tagged type fields referencing object(s) on the whole side of an aggregation
AGGREGATION_WHOLE_REFERENCE_SUFFIX	_whole	
AGGREGATION_PART_REFERENCE_PREFIX		Prefix and suffix of class tagged type fields referencing object(s) on the part side of an aggregation
AGGREGATION_PART_REFERENCE_SUFFIX	_part	
COMPOSITION_WHOLE_REFERENCE_PREFIX		Prefix and suffix of class tagged type fields referencing object(s) on the whole side of a composition
COMPOSITION_WHOLE_REFERENCE_SUFFIX	_whole	
COMPOSITION_PART_REFERENCE_PREFIX		Prefix and suffix of class tagged type fields referencing object(s) on the part side of a composition
COMPOSITION_PART_REFERENCE_SUFFIX	_part	

File Name Generation

Table 9 lists the strings used in generating file names. File names are either based on the class name or on the fully qualified Ada unit name.

Table 9: Strings for Generating File Names

QRL Name	Default Value	Purpose
FILE_NAME_DOT_REPLACEMENT_CHARACTER	-(hyphen)	Character to replace dots in file names automatically generated from fully qualified Ada unit names

Generic Package and Type Names

Table 10 lists the strings used for generating instantiations of generic packages used in implementation of associations and aggregations.

Table 10: Strings for Generic Package and Type Names

QRL Name	Default Value	Purpose
SET_PKG_NAME	generic_set	Name of the generic package used for multiplicity many, unordered associations
SET_TYPE_NAME	set	Name of the type exported by the generic set package
LIST_PKG_NAME	generic_list	Name of the generic package used for multiplicity many, ordered associations
LIST_TYPE_NAME	list	Name of the type exported by the generic list package
MAP_PKG_NAME	generic_map	Name of the generic package used for qualified associations
MAP_TYPE_NAME	map	Name of the type exported by the generic map package
TUPLE_PKG_NAME	generic_tuple	Name of the generic package used for inverse qualified / keyed associations
TUPLE_TYPE_NAME	tuple	Name of the type exported by the generic tuple package

Formatting Output

Table 11 lists strings used for output formatting.

Table 11: Strings for Formatting Output

QRL Name	Default Value	Purpose
INDENT	<4 spaces>	Basic unit of indentation (can use spaces, tabs, or arbitrary string)
COMMENT_PREFIX	--StP |	String to begin comments. For Ada, must always start with -- (2 hyphens).

Tag Comments for Incremental Code Generation

Tag comments determine regions in source code you should not edit manually. These tags must be unique, to avoid matching a string in a source file.

There are two groups of tags: those required for all languages generated by StP/UML, and those required by Ada code generation. The declarations of both groups of tags appear in “Generating Ada_95 Incrementally” on page 5-16, and are not repeated here.

If you change these tags, incremental code generation will not work for any code you previously generated with different tags.

6 Mapping StP to Ada_95

This chapter describes the relationship between StP/UML constructs and the Ada_95 code you can generate from them.

The chapter describes:

For a summary of the mapping of StP/UML constructs to Ada code, see Table 1 on page 5-2.

You can customize the names of many generated constructs as explained in "Customizing Generated Ada_95 Code" on page 5-24. The examples in this chapter use default values.

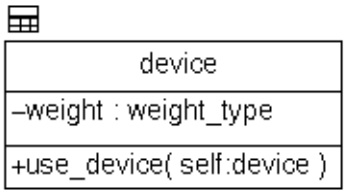
Mapping Overview

The fundamental mappings of StP / UML constructs to Ada code are:

- A class maps to a tagged type.
- A UML package maps to an Ada_95 package.
- An attribute maps to a component field of the tagged type.
- An operation maps to a subprogram declared in the package specification.

You can also generate an access type to the tagged type or to the class-wide type associated with the tagged type. A simple class with one attribute and one operation appears in Figure 1.

Figure 1: Simple Class



In its simplest form, the mapping for this class looks like:

```
package device_pkg is

   type device is tagged record
      weight: weight_type;
   end record;

   type device_cptr is access all device'class; -- optional
   type device_ptr is access all device; -- optional

   procedure use_device(self: device);

end device_pkg;
```

You would declare a static object of this class as:

```
my_device: device_pkg.device;
```

Filename Generation

The structure of the file names you generate depends upon whether you choose to use the default method or StP/UML components.

Using the Default Method

By default, each package generates its own file or set of related files. Each filename generated for a particular package is the StP/UML package name with one of three extensions appended to it. The three default filename extensions are:

`.adb` - (for package bodies)

`.ads` - (for package specifications)

`.adu` - (for package subunits)

For StP/UML packages that map to Ada_95 child packages, the default filename is the Ada_95 package name with the period "." replaced by a dash "-". For example:

`Ada-Text_IO.ads`

corresponds to the package specification for

`Ada.Text_IO`

Also, the default generates all subunits for a particular class to the same file.

You can customize all the above in QRL.

Using StP/UML Components

By using UML components, you can map packages to files. An annotation associates packages with a particular component. You can set the component associated with a package using the Properties dialog box available for the package from the Class Editor. The generated filename automatically adopts the component name and adds appropriate file extensions for the particular compiler.

Generating Packages

The unit of code generation in UML is the package, and a UML package maps directly to an Ada_95 package. Similarly, nested UML packages map to nested packages in Ada_95.

UML does not support code generation for a particular class. Therefore, you cannot generate code for a class outside of any UML package.

To model Ada_95 child packages, you create a UML dependency from a child to a parent package and add a stereotype to the dependency relationship. The stereotype should be one of:

`<<public child>>`

`<<private child>>`

If no stereotype is present (implying a straightforward package dependency), code generates a `with` clause.

In certain circumstances, code generation from UML will create generic Ada_95 packages. This happens implicitly with the default implementation of associations, and explicitly in the case of parameterized classes. For completeness, StP-Ada_95 code generation allows for the creation of generic Ada_95 packages with tagged types (for packages containing no classes). You specify this by adding a `<<generic>>` stereotype to the UML package.

You can specify whether a package is external using an annotation. This annotation is equivalent to the standard annotation available on UML classes and is used by Ada_95 code generation to control whether code is

generated for a particular package. You can specify the annotation using the Object Annotation Editor or the Properties dialog box available for the package from the Class Editor. For more information, refer to “Inheriting from Types in Library Units” on page 5-4.

Generating Classes

An StP/UML class maps to an Ada_95 tagged type. For any particular UML class there is an associated Ada_95 package. When you generate code, the tagged type definition is placed in that package.

StP-Ada_95 code generation derives from classes and their attributes and operations; you cannot generate code for a particular object.

You can generate an access type to the tagged type and/or to the class-wide type. You can do so on a per class or global basis. The default behavior generates access types.

Class Visibility

Class visibility indicates the access to the interface of a class. The following discussion of class visibility distinguishes two types of class: root classes (those having no superclasses), and non-root classes (those having superclasses).

Root Class Visibility

A class with no superclasses has four possible visibility levels for a tagged type:

- Public
- Tagged Private
- Private
- Implementation

The following sections show examples of Ada code for each visibility, for a class called *foo* with no attributes or operations. Note the tagged type may also be abstract (except for Private) or limited. Though not shown here, code generation also can generate cases where the partial view of a tagged type is abstract, but the full view is not, and where the partial view is limited, but the full view is not.

Public

```
package foo_pkg is
   type foo is [abstract] tagged [limited] null record;
end foo_pkg;
```

Tagged Private

```
package foo_pkg is
   type foo is [abstract] tagged [limited] private;
private
   type foo is [abstract] tagged [limited] null record;
end foo_pkg;
```

Private

```
package foo_pkg is
   type foo is [limited] private;
private
   type foo is tagged [limited] null record;
end foo_pkg;
```

Implementation

```
package foo_pkg is
   type foo_ptr is private;
private
   type foo;
   type foo_ptr is access all foo;
end foo_pkg;
```

```
package body foo_pkg is
   type foo is [abstract] tagged [limited] null record;
end foo_pkg;
```

Although not required, an access type appears for the Implementation visibility case to show the way a client of the class package can refer to the tagged type. Implementation visibility allows the generation of code that can be compiled for certain object models that would result in cyclic compilation dependencies if you used any of the other privacies. For more information on cyclic compilation dependencies, refer to “Avoiding Cyclic Compilation Dependencies” on page 5-14. Implementation visibility has the drawback that you can define no primitive operations (in the Ada_95 sense) for the tagged type. Furthermore, no child packages can see the tagged type.

Non-root Class Visibility

For a class with superclasses, there are five possible visibility levels for the tagged type: the four shown in “Root Class Visibility,”and Private Extension.

The following sections show examples of Ada code for each visibility for a class called *sub* that inherits from class *foo*. Due to Ada_95 rules on limited types, the keyword “limited” can appear for only Tagged Private or Private tagged types. Code generation can generate cases where the partial view of a tagged type is abstract, but the full view is not.

Public

```
package sub_pkg is
   type sub is [abstract] new foo with null record;
end sub_pkg;
```

Private Extension

```
package sub_pkg is
   type sub is [abstract] new foo with private;
private
   type sub is [abstract] new foo with null record;
end sub_pkg;
```

Tagged Private

```
package sub_pkg is
   type sub is [abstract] tagged [limited] private;
private
   type sub is [abstract] new foo with null record;
end sub_pkg;
```

Private

```
package sub_pkg is
   type sub is [limited] private;
private
   type sub is new foo with null record;
end sub_pkg;
```

Implementation

```
package sub_pkg is
   type sub_ptr is private;
private
   type sub;
   type sub_ptr is access all sub;
end sub_pkg;
```

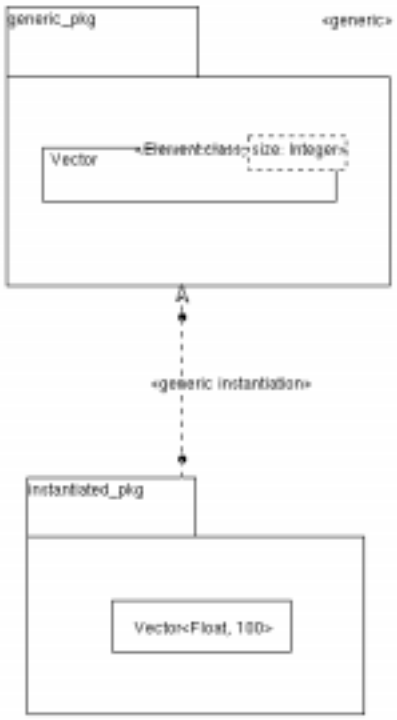
```
package body sub_pkg is
   type sub is [abstract] new foo with null record;
end sub_pkg;
```

Generating Parameterized and Instantiated Classes

A parameterized class in UML maps to a tagged type within a generic package in Ada_95. In order to generate Ada_95 code, you must identify the UML package that maps to the Ada_95 generic package by adding a <<generic>> stereotype to this package. UML code generation checks that every parameterized class appears in a package with the <<generic>> stereotype and that no instantiation of a class exists in the same package as the parameterized class itself.

Instantiated classes have no directly corresponding Ada_95 construct; you use them only to provide arguments to instantiate generic packages. To do this, you add a new package to the UML model and link it to the generic package with a dependency link. You then label the dependency link with the stereotype, `<<generic instantiation>>`. This new package maps to a generic instantiation of the Ada_95 generic package. You specify generic arguments by adding anonymous instantiations to the new package that correspond to the parameterized classes in the generic package. Figure 2 shows an example model.

Figure 2: Modeling a Parameterized and Instantiated Class



The model in Figure 2 maps to the following Ada_95 code.

Note: The model does not define attributes or operations.

```
generic
   type Element is private;
   size: Integer;
package generic_pkg is
   type Vector is private;
private
   type Vector is tagged null record;
end generic_pkg;
```

```
with generic_pkg;
package instantiated_pkg is new generic_pkg(
   Element => Float,
   size => 100);
```

You can show the class parameters in the parameterized class either in the standard UML format or directly as an Ada_95 generic parameter list. If the parameter string includes a semicolon, code generation interprets it as an Ada_95 generic parameter list and uses it directly in the generic specification. If the parameter string does not include a semicolon, it is interpreted as a UML parameter list, and code generation converts it to Ada_95 code. “T : Class” parameters map to Ada_95 code as `type T is private;`.

You can enclose the parameterized class in nested packages. In this case, the generic stereotype identifies which of the nested packages is the one that maps to the Ada_95 generic package.

Generating Dependency Relationships

UML allows you to specify dependency relationships from any one of:

- Package
- Class
- Operation

to any one of

- Package
- Class
- Interface

These all map to "`with`" context classes in Ada_95. Dependencies involving classes and operations use the package containing the tagged type or function. StP-Ada_95 code generation builds up a set of dependencies between packages for the entire model based on the explicit dependency relations listed above or implicit relationships (associations between classes, class inheritance, package inheritance). StP then generates the necessary context clauses when generating code for a particular package.

Generating Attributes

StP generates an attribute as a component field of a tagged type.

Using the Ada_95 items in the class table editor, you may specify an attribute as "aliased." You may also designate it as a class-scope attribute. In this case, it generates as a stand-alone variable and not as a component of the tagged type. You can set visibility for class-scope attributes on an individual basis, with the values Public, Private, or Implementation.

The following example code generates for the class *attr* with one aliased normal attribute, and one class attribute of each visibility, with the Private one being aliased.

```
package attr_pkg is
   type attr is tagged record
      normal_attr: aliased integer;
   end record;
   class_attr_public: integer;
private
   class_attr_private: aliased integer;
end attr_pkg;

package body attr_pkg is
   class_attr_implementation: integer;
end attr_pkg;
```

You can associate multiple classes with the same package in UML. Therefore, you may generate duplicate variable declarations for the same identifier in a single Ada_95 package. In this case, the code generator generates the code with the duplicate declarations and displays a warning in the StP Desktop window.

Generating Operations

An operation generates as a subprogram declared in the package associated with the operation's class. It can be a procedure or a function.

You may generate an operation as abstract, a subunit, or inline. You may also designate it as a class-scope operation which affects the placement of the generated code. This makes no difference in the generated code other than in its placement, and allows certain checks to occur. You can set visibility for all operations on an individual basis, with the values Public, Private, or Implementation.

For an operation to be a primitive operation of the tagged type in the Ada_95, the tagged type and the operation must have a visibility other than Implementation. You must specify at least one parameter type or the return type as either the tagged type or an access type to the tagged type.

```
-- package specification

package oper_pkg is

   type oper is abstract tagged null record;
   type oper_ptr is access all oper;

   function oper_public (self: oper_ptr) return Integer;
   function oper_inline (self: oper_ptr) return Integer;

      pragma inline (oper_inline)
   function oper_subunit (self: oper_ptr)return Integer;
   function oper_abstract (self: oper_ptr) return Integer is
      abstract;

-- class-scope operations identical to normal operations
-- except for position in package

function oper_class (self: oper_ptr) return Integer;
```

```
private
   function oper_private (self: oper_ptr) return Integer;
end oper_pkg;

-- package body

package body oper_pkg is
   function oper_implementation (self: oper_ptr) return
   Integer;

   function oper_public (self: oper_ptr) return Integer is
   begin
      return 0;
   end oper_public;

   function oper_private (self: oper_ptr) return Integer is
   begin
      return 0;
   end oper_private;

   function oper_implementation (self: oper_ptr) return
      Integer is
   begin
      return 0;
   end oper_implementation;

   function oper_inline (self: oper_ptr) return Integer is
   begin
      return 0;
   end oper_inline;

   function oper_subunit (self: oper_ptr) return Integer is
      separate;

   function oper_class (self: oper_ptr) return Integer is
   begin
      return 0;
   end oper_class;
end oper_pkg;

-- subunits

separate (oper_pkg)
   function oper_subunit (self: oper_ptr) return Integer is
   begin
      return 0;
   end oper_subunit;
```

You can associate multiple classes with the same package in UML. Therefore, you may generate duplicate function declarations for the same identifier in a single Ada_95 package. In this case, the code generator generates the code with the duplicate declarations and displays a warning in the StP Desktop window.

Generalization and Inheritance Relationships

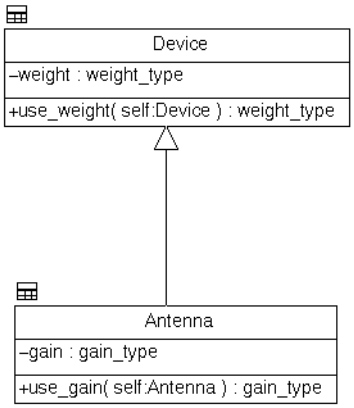
The generalization mapping implements single inheritance. Ada_95 does not directly support multiple inheritance. In this version of code generation, there are no special features to assist in its implementation. If a class has multiple superclasses, and a diagram exists in which all the superclasses appear, code generation chooses the left-most class as the primary superclass for type extension. However, StP-Ada_95 code generation does support generalization of multiple interface types.

Single Inheritance

Single inheritance uses tagged types and type extension. There is no need to specify the kind of package to be used for a subclass. This is controlled by which particular UML package is used. For more information, refer to “Generating Packages” on page 6-4.

Figure 3 shows a generalization relationship using a tagged type visibility of Public for each class. Assume the existence of the other types involved.

Figure 3: Single Inheritance



The superclass definition:

```
package Device_pkg is

   type Device is tagged record
      weight: weight_type;
   end record;

   function use_weight(self: Device) return weight_type;

end Device_pkg;
```

The subclass definition, with a Root class package:

```
with Device_pkg;
package Antenna_pkg is

   type Antenna is new Device_pkg.Device with record
          gain: gain_type;
   end record;

   function use_gain(self: Antenna) return gain_type;

end Antenna_pkg;
```

Generating Associations, Aggregations, and Compositions

Associations map into Ada in a variety of ways, depending on the existence and value of characteristics such as:

- Directionality
- Roles
- Multiplicity
- Ordering (when multiplicity > 1)
- Qualification
- Existence of link attributes / association classes

This version of code generation refers to classes involved in associations using only access types to class-wide types.

Associations in this discussion fall into two general groups:

- Binary / Reflexive
- N-ary, where n is greater than 2

Binary/Reflexive Associations

For a given direction of traversal, a binary or reflexive association involves a source class and a destination class. In the case of a reflexive association, the source and destination classes are the same. All binary associations (whether or not reflexive) generate as one way associations. You treat them as either a single one-way association (if traversed in only one direction), or as a pair of one-way associations (if traversed in both directions). You use the Role Navigability item on the Role Definition note to specify the traversal. For more information, see Table 6 on page 5-9.

To avoid cyclic compilation dependencies, the tagged type visibility must be Implementation for both classes involved in a binary two-way association, if they exist in different packages.

Binary associations are classified based on the following five specifications shown in Table 1.

Table 1: Association Specifications

Specification	Description	Code
Kind	Association / Aggregation / Composition	As / Ag / Co
Implementation	StP / User	S / U
Qualification	None / Single / Inverse / Double	N / S / I / D
Multiplicity	One / Many	1 / M
Ordering	Unordered / Ordered	U / O

When StP / UML generates the code for an association, it generates a tag detailing the type of the association using the code appearing in the third column of the above table, for example:

```
-- | StP Association As_S_N_1_U.
```

- Kind

  The code generated for aggregations and compositions follows the same general scheme as for regular associations. The difference with aggregations is in the suffix for the aggregation reference class record field. The difference with compositions is in replacing access types with actual values.

- Implementation

  You can divide associations as to whether they employ StP's default implementation scheme or whether they use a non-default user specified implementation. You specify user implementations by adding a note to the particular association. The note is a simple string specifying the appropriate component declaration. Therefore, the mapping for a user specified implementation is completely under the user's control. The remainder of this section describes mappings used in the default implementation scheme.

- Qualification / Multiplicity / Ordering

  Various combinations of these specifications result in twelve different kinds of association mappings. For more information, refer to "Default Implementation of Associations" on page 6-19.

Generic Specifications for Default Associations

The default association mappings assume the existence of four generic packages:

- generic_set
- generic_list
- generic_map
- generic_tuple

Sample specifications for these four generics are provided with the product in the directory *<base_dir>/qrl/code_gen/ada_95/generics* and are shown in Figure 4. To fully implement associations, you must either use the sample generic specifications, expanding them to meet your needs and supplying the package bodies; or, you must provide your own.

The only parts of these generics that are actually used in the generated code are the package name and the name of a type exported by the package, which are shown in bold. You can customize these names as described in "Customizing Generated Ada_95 Code" on page 5-24.

Figure 4: Generic Specifications for Associations

```
generic
   type item is private;
package generic_set is
   type set is private;
   procedure add       (s: set; i: item);
   procedure remove    (s: set; i: item);
   function is_member (s: set; i: item) return boolean;
private
   type set_type;
   type set is access set_type;
end generic_set;

generic
   type item is private;
package generic_list is
   type list is private;
private
   type list_type;
   type list is access list_type;
end generic_list;
```

```
generic
   type key is private;
   type value is private;
package generic_map is
   type map is private;
   procedure bind (m: map; k: key; v: value);
   procedure unbind   (m: map; k: key; v: value);
   function is_bound (m: map; k: key) return boolean;
   function value_of (m: map; k: key) return value;
private
   type map_type;
   type map is access map_type;
end generic_map;
```

```
generic
   type type1 is private;
   type type2 is private;
package generic_tuple is
   type tuple is record
      comp1: type1;
      comp2: type2;
   end record;
end generic_tuple;
```

Default Implementation of Associations

You implement a one-way association by adding a new component field to the source tagged type that references, directly or indirectly, an object or group of objects of the destination class. There are twelve cases to consider for UML as shown in Table 2.

Table 2: One-Way Associations

Case	Qualified	Multiplicity	Ordering	Generic Package			
				Set	List	Map	Tuple
1	No	1					
2		Many	No	X			
3			Yes		X		

Table 2: One-Way Associations (Continued)

Case	Qualified	Multiplicity	Ordering	Generic Package			
				Set	List	Map	Tuple
4	Yes	1				X	
5		Many	No	X		X	
6			Yes		X	X	
7	Inverse	1					X
8		Many	No	X			X
9			Yes		X		X
10	Double	1				X	X
11		Many	No	X		X	X
12			Yes		X	X	X

The mappings for these cases appear in the following figures.

The tagged type visibility in all these examples is Implementation. As mentioned before, Implementation visibility is provided to allow the implementation of arbitrary associations without cyclic compilation dependencies.

Only the package bodies for the source classes appear. The package specifications for Implementation visibility have appeared before, and there is no impact on the Ada representation of a destination class for a one-way association.

Figure 5: Association Case 1

```
with dest1_pkg;
package body source1_pkg is
   type source1 is tagged record
      dest1_pkg.dest1_asc: dest1_pkg.dest1_cptr;
   end record;
end source1_pkg;
```

Figure 6: Association Case 2


```
with generic_set;
with dest2_pkg;
package dest2_set is new generic_set (dest2_pkg.dest2_cptr);
```

```
with dest2_set;
package body source2_pkg is
   type class is record
      dest2_asc: dest2_set.set;
   end record;
end source2_pkg;
```

Figure 7: Association Case 3

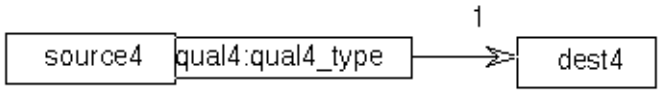
Note: There is no display mark for ordering.

Case 3 is exactly like Case 2, with “list” replacing “set” in all occurrences, marked by boldface here:

```
with generic_list;
with dest3_pkg;
package dest3_list is new generic_list
   (dest3_pkg.dest3_cptr);
```

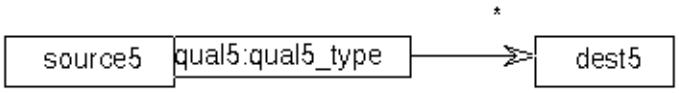
```
with dest3_list;
package body source3_pkg is
   type class is record
      dest3_asc: dest3_list.list;
   end record;
end source3_pkg;
```

Figure 8: Association Case 4




```
with generic_map;
with dest4_pkg;
package qual4_type_dest4_map is new generic_map (qual4_type,
   dest4_pkg.dest4_cptr);

with qual4_type_dest4_map;
package body source4_pkg is

   type class is record
      dest4_asc: qual4_type_dest4_map.map;
   end record;

end source4_pkg;
```

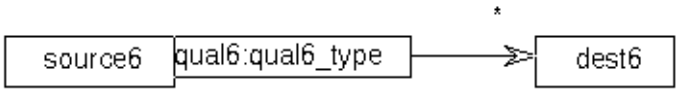
Figure 9: Association Case 5




```
with generic_set;
with dest5;
package dest5_set is new generic_set (dest5_pkg.dest5_cptr);

with generic_map;
with dest5_set;
package qual5_type_dest5_set_map is new generic_map
   (qual5_type, dest5_set.set);

with qual5_type_dest5_set_map;
package source5_pkg is
   type class is record
```

```
      dest5_asc: qual5_type_dest5_set_map.map;
   end record;
end source5_pkg;
```

Figure 10: Association Case 6



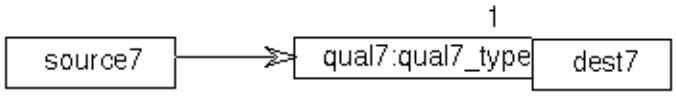
Note: There is no display mark for ordering.

```
with generic_list;
with dest6;
package dest6_list is new generic_list
(dest6_pkg.dest6_cptr);
```

```
with generic_map;
with dest6_list;
package qual6_type_dest6_list_map is new generic_map
   (qual6_type, dest6_list.list);
```

```
with qual6_type_dest6_list_map;
package source6_pkg is
   type class is record
      dest6_asc: qual6_type_dest6_list_map.map;
   end record;
end source6_pkg;
```

Figure 11: Association Case 7

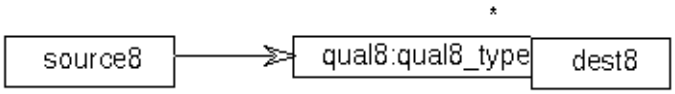



```
with generic_tuple;
with dest7;
package qual7_type_dest7_tuple is new generic_tuple
   (qual7_type, dest7_pkg.dest7_cptr);
```

```
with qual7_type_dest7_tuple;
package body source7_pkg is
```

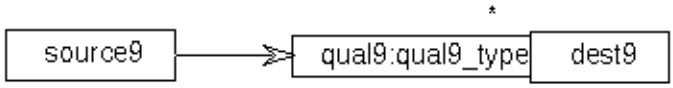
```
   type class is record
      dest7_asc: qual7_type_dest7_tuple.tuple;
   end record;
end source7_pkg;
```

Figure 12: Association Case 8




```
with generic_tuple;
with dest8;
package qual8_type_dest8_tuple is new generic_tuple
   (qual8_type, dest8_pkg.dest8_cptr);

with generic_set;
with qual8_type_dest8_tuple;
package qual8_type_dest8_tuple_set is new generic_set
   (qual8_type_dest8_tuple.tuple);

with qual8_type_dest8_tuple_set;
package body source8_pkg is

   type class is record
      dest8_asc: qual8_type_dest8_tuple_set.set;
   end record;
end source8_pkg;
```

Figure 13: Association Case 9

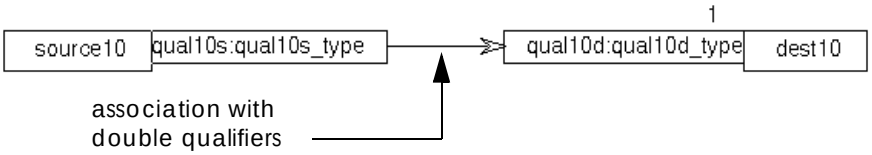


Note: There is no display mark for ordering.

```
with generic_tuple;
with dest9;
package qual9_type_dest9_tuple is new generic_tuple
   (qual9_type, dest9_pkg.dest9_cptr);

with generic_list;
with qual9_type_dest9_tuple;
```

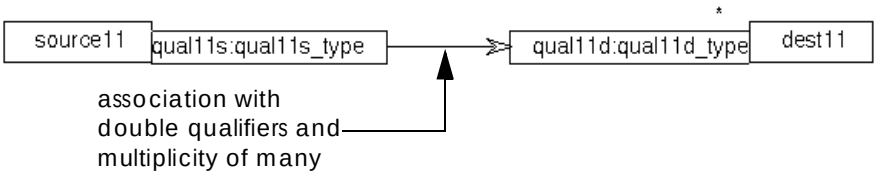
```
package qual9_type_dest9_tuple_list is new generic_list
   (qual9_type_dest9_tuple.tuple);

with qual9_type_dest9_tuple_list;
package body source9_pkg is

   type class is record
      dest9_asc: qual9_type_dest9_tuple_list.list;
   end record;
end source9_pkg;
```

Figure 14: Association Case 10



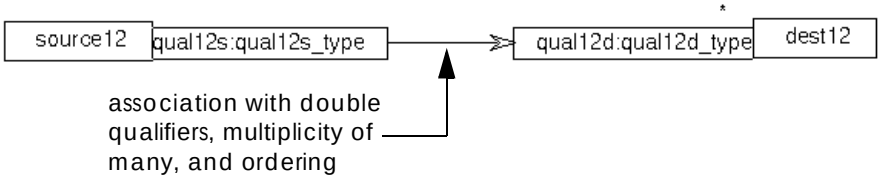

```
with generic_tuple;
with dest10;
package qual10d_type_dest10_tuple is new generic_tuple
   (qual10d_type, dest10_pkg.dest10_cptr);

with generic_map;
with qual10d_type_dest10_tuple;
package qual10s_type_qual10d_type_dest10_tuple_map is new
   generic_map (qual10s_type,
   qual10d_type_dest10_tuple.tuple);

with qual10s_type_qual10d_type_dest10_tuple_map;
package body source10_pkg is

   type class is record
      dest10_asc:
         qual10s_type_qual10d_type_dest10_tuple_map.map;
   end record;
end source10_pkg;
```

Figure 15: Association Case 11




```
with generic_tuple;
with dest11;
package qual11d_type_dest11_tuple is new generic_tuple
   (qual11d_type, dest11_pkg.dest11_cptr);

with generic_set;
with qual11d_type_dest11_tuple;
package qual11d_type_dest11_tuple_set is new generic_set
   (qual11d_type_dest11_tuple.tuple);

with generic_map;
with qual11d_type_dest11_tuple_set;
package qual11s_type_qual11d_type_dest11_tuple_set_map is
   new generic_map (qual11s_type,
   qual11d_type_dest11_tuple_set.set);

with qual11d_type_qual11d_type_dest11_tuple_set_map;
package body source11_pkg is

   type class is record
      dest11_asc:

qual11s_type_qual11d_type_dest11_tuple_set_map.map;
   end record;
end source11_pkg;
```

Figure 16: Association Case 12



Note: There is no display mark for ordering.

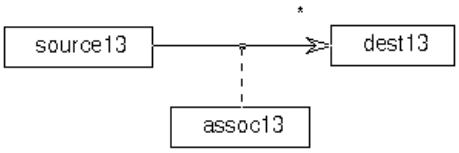
```
with generic_tuple;
with dest12;
package qual12d_type_dest12_tuple is new generic_tuple
   (qual12d_type, dest12_pkg.dest12_cptr);

with generic_list;
with qual12d_type_dest12_tuple;
package qual12d_type_dest12_tuple_list is new generic_list
   (qual12d_type_dest12_tuple.tuple);

with generic_map;
with qual12d_type_dest12_tuple_list;
package qual12s_type_qual12d_type_dest12_tuple_list_map is
   new generic_map (qual12s_type,
   qual12d_type_dest12_tuple_list.list);

with qual12d_type_qual12d_type_dest12_tuple_list_map;
package body source12_pkg is

   type class is record
      dest12_asc:

qual12s_type_qual12d_type_dest12_tuple_list_map.map;
   end record;
end source12_pkg;
```

Link Attributes/Association Classes

An independent tagged type generates for an association class.

The one-way association is made indirectly through this association class. Figure 17 shows an example of an association class. You implement other cases of multiplicity, qualification, and ordering analogously to the examples shown in “Default Implementation of Associations” on page 6-19.

Figure 17: One-Way Association Classes




```
with dest13_pkg;
package body assoc13_pkg is
   type assoc13 is tagged record
      dest13_asc: dest13_pkg.dest13_cptr;
   end record;
end assoc13_pkg;
```

```
with generic_set;
with assoc13_pkg;
package assoc13_set is new generic_set
(assoc13_pkg.assoc13_cptr);
```

```
with assoc13_set;
package body source13_pkg is
   type source13 is tagged record
      assoc13_asc: assoc13_set.set;
   end record;
end source13_pkg;
```

If an association with an association class is not implemented, the association class itself is still generated, but the references to and from it are not.

Role Names

If a role name exists for a binary association, it appears as the name of the association reference class record field. If not, code generation uses the name of the destination class followed by a customizable string. For reflexive associations, you must give role names to distinguish the two ends of the association.

Aggregations

You implement aggregations similarly to regular associations. The only difference is in the suffix for the aggregation reference class record field. This suffix is either _whole or _part, depending on the position of the aggregation mark. The mapping is as follows.

```
package Aggregation is

   type MailSystem is tagged private;
   type MailSystem_cptr is access all MailSystem'class;

   type Receiver is tagged private;
   type Receiver_cptr is access all Receiver'class;
private

   type MailSsytem is tagged record
      Aggregation_Receiver_part :
         Aggregation.Receiver_cptr;
   end record;

   type Receiver is tagged record
      Aggregation_MailSystem_whole :
         Aggregation.MailSystem_cptr;
end Aggregation;
```

For more information, see Table 8 on page 5-25.

Compositions

StP-Ada_95 code generation uses the same scheme to generate compositions as for associations and aggregations. However, actual values replace access types. The mapping is as follows.

```
package composition is

   type Collection is tagged private;
   type Collection_cptr is access all Collection'class;

   type Mailbox is tagged private;
   type Mailbox_cptr is access all Mailbox'class;
private

   type Collection is tagged record
      composition_Mailbox_whole :
```

```
            composition.Mailbox_cptr;
    end record;

    type Mailbox is tagged record
       composition_Collection_part :
            composition.Collection;
    end record;
end composition;
```

Generating N-ary Associations

StP/UML Ada code generation implements n-ary associations as n binary two-way associations, each between one of the involved classes and a central association class. An n-ary association could be implemented as a stand-alone entity; however, that mechanism is not currently provided.

When you add an association class to the model, it becomes the central class in the implementation. Each instance of the central class must reference one instance of the associated classes. Each instance of an associated class must be able to reference multiple central instances.

As with binary two-way associations, this mutual dependence requires Implementation tagged type visibility to avoid cyclic compilation dependencies.

To generate Ada_95 code, make sure an n-ary association is attached to an association class. You must nest the association class, as all other classes, inside some package (generic or otherwise).

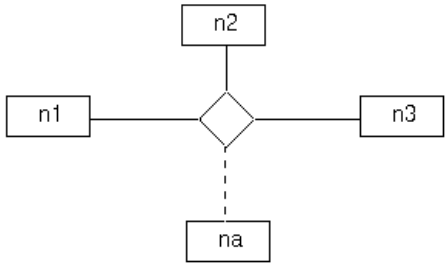
Inside the association package, a tagged type generates for the association class. It contains one field for each class involved in the association. A variable of that type is then an n-tuple of the values from the respective classes. A set (unordered association) or list (ordered association) of the association class tagged type is declared in the association package. References to the set or list are declared in the tagged types for the classes involved in the association.

When StP/UML generates the code, it generates a tag indicating that these constructs generated from an n-ary association.

Also, this mutual dependency requires Implementation tagged type visibility to avoid cyclic compilation dependencies.

Figure 18 shows a ternary association with an association class.

Figure 18: N-ary Association



The mapping is as follows, showing only package bodies:

```
with n1_pkg;
with n2_pkg;
with n3_pkg;
package body na_pkg is
   type class is record
      n1_asc: n1_pkg.n1_cptr;
      n2_asc: n2_pkg.n2_cptr;
      n3_asc: n3_pkg.n3_cptr;
   end record;
end na_pkg;
```

```
with generic_set;
with na_pkg;
package na_set is new generic_set (na_pkg.na_cptr);

with na_set;
package body n1_pkg is
   type n1 is record
      na_asc: na_set.set;
   end record;
end n1_pkg;

with na_set;
package body n2_pkg is
   type n2 is record
      na_asc: na_set.set;
```

```
   end record;
end n2_pkg;

with na_set;
package body n3_pkg is
   type n3 is record
      na_asc: na_set.set;
   end record;
end n3_pkg;
```

Role Names

If role names exist for an n-ary association, they are used as the names of the central association class record fields that reference the involved classes.

7 Generating TOOL Code

TOOL is an object-oriented fourth-generation language from Forté Software, Inc. This chapter assumes you have programming experience and have used the Forté Workshops to create classes.

This chapter tells you how to generate TOOL from your StP/UML models. It describes

Use this chapter with Chapter 2, "Generating Code From UML Models." Chapter 2 provides information for generating code that applies to all languages.

What is Generated for TOOL

For each class in a class diagram that is not external, StP generates a class definition in TOOL code.

The default directory is the *src_files* directory in your project/system path. The default output file extension is *.cex*.

The generated code conforms to the Forté standard.

StP to TOOL Mapping

StP generates TOOL code class definitions from UML constructs and their annotations, which you can then import into Forté projects. Some StP/UML constructs directly map to TOOL objects; you create other mappings using annotations.

Table 1 shows how StP constructs directly map to TOOL constructs.

Table 1: Mapping StP Objects to TOOL Constructs

StP Object	TOOL Construct
Class	Class
Interface	Interface <label> (for 3.x) class <label> (for 2.x)
Package	Not supported.
Instantiated class	Not supported
Anonymous instantiated class	Not supported
Parameterized class	Treated as a regular TOOL class. Parameter information is ignored.
Class TOOL annotations	Class properties
Placeholder external class	Forté-supplied class
Attribute with no default value	Attribute
Attribute with a get expression and (optionally) set expression	Virtual attribute
Attribute with default value, no type	Constant
Operation	Method

Table 1: Mapping StP Objects to TOOL Constructs (Continued)

StP Object	TOOL Construct
Operation TOOL Source Code note description	Method body
Operation with Kind set to Event	Event
Operation with Kind set to EventHandler	Event Handler
Association to <class> with multiplicity of one	Attribute of type <class>
Association to <class> with multiplicity of many	Attribute of type array with items of type <class>
Aggregation/Composition	Attribute of type <class>
Generalization	class <source> inherits <target>
Implementation	implements <target>,<target>, ... (3.x) class <source> inherits <target> (2.x)
Role	Attribute's name

Relationships

Associations and Aggregations/Compositions

StP/UML associations and aggregations/compositions map to attributes whose type is the associated class. If a role name exists, it becomes the attribute's name. If no role name exists, StP generates one.

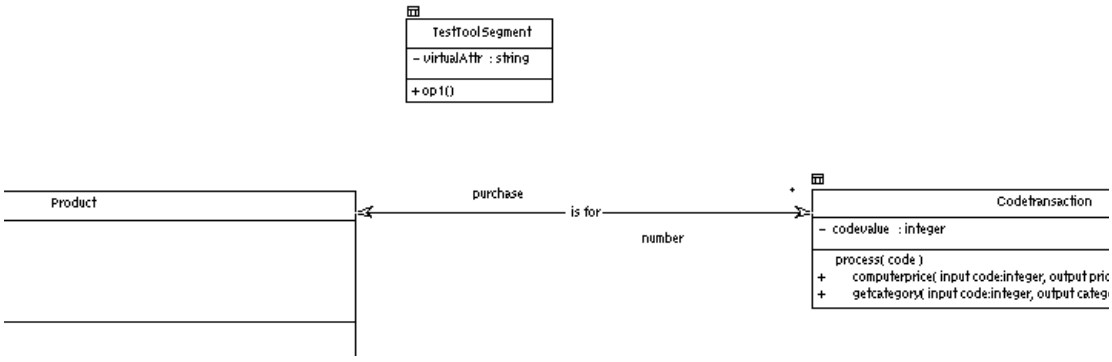
For each association and aggregation/composition, the generated attribute name is the name of the associated class with the prefix, `assn`.

If the multiplicity of an association is many, StP implements the association as an array whose item type is the associated class.

Association Example

An example association appears in Figure 1.

Figure 1: Association Example



Association with Multiplicity Examples

Figure 2 shows an excerpt of the generated code for the class components of the *Codedtransaction* class, which illustrates how StP generates an association with multiplicity of one.

Figure 2: Generated *is for* Association in Class *Codedtransaction*

```
// stp class components
    has private attribute codevalue : integer;
    attribute purchase : Product
    has property extended = (StPAttribute = 'association');
    has public method getcategory(input code:integer,
     output category:integer) : integer
    has property extended = StPSignature=inputcode:integer,
     output category:integer');
.
.
.
    // stp class components end
```

Attribute name = Role

e = Name of the associated class

Figure 3 shows an excerpt of the generated code for the class components of the *Product* class, which illustrates how StP generates an association with multiplicity of many.

Figure 3: Generated *is for* Association in Class *Product*

Aggregation Examples

```
// stp class components
     has private attribute category : string;
     has private constant minimum_stock = 300;
     has public attribute price : float;
     has private virtual attribute sale_price : float =
      (get = price + .20 * price);
     has private virtual attribute amount_inventory :
      integer = (get = 500+100);
     attribute number : Framework.Array of Codetransaction
     has property extended = (StPAttribute = 'association');
     has private method change_price(input oldprice:float,
      input discount:float, output newprice:float) : float
     has property extended = (StPSignature =
      'input oldprice:float, input discount:float, output
        newprice:float');
.
.
.
     // stp class components end
```

Attribute name = Role
Type = Array whose type is associated class

StP handles aggregations in the same way as it handles associations. Figure 4 shows a simple StP/UML aggregation.

Figure 4: Aggregation Example

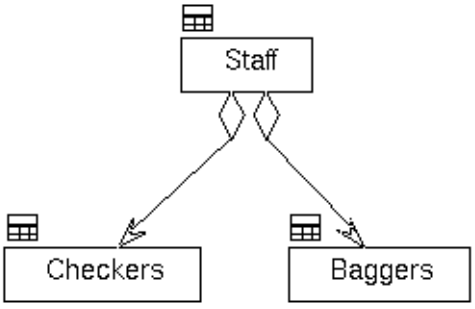
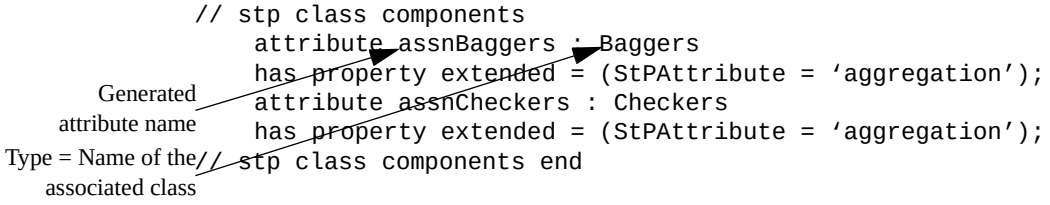


Figure 5 shows the generated code for the *Staff* class.

Figure 5: Generated Code for *Staff*



Generalizations

A generalization link between two classes maps to the TOOL class definition:

```
class <source> inherits <target>
```

Figure 6 is an example of a diagram featuring a generalization between classes.

Figure 6: Diagram of Generalization Between Classes

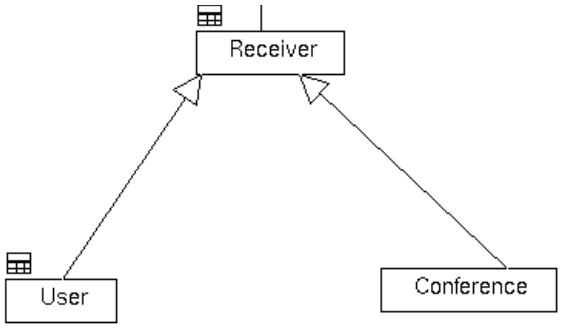


Figure 7 shows the generated TOOL class definitions for the Conference and User classes.

Figure 7: Generated TOOL Class Definitions

```
// stp class definition
class User inherits from Receiver
.
.
.

//stp class definition
class Conference inherits from Receiver
```

Stp does not support multiple inheritance. Stp does support `is mapped` inheritance from a Forté UserWindow class. Stp supports generalization links between interfaces by default. Figure 8 is an example of a diagram featuring a generalization between interfaces.

Figure 8: Diagram of Generalization Between Interfaces

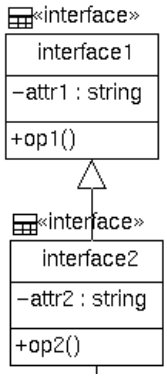


Figure 9 shows the forward declarations and class definition for a generalization between interfaces.

Figure 9: Code for Generalization Between Interfaces

```
forward interface interface1;
forward interface interface2;
.
.
.
// stp class definition
interface interface2 inherits from interface1
```

You can turn off code generation for interfaces by specifying that you are generating Release 2.x TOOL code. You do this by selecting the **Generate Code For Release 2.x** command property. For more information, see Table 8 on page 7-22.

Interfaces

In generating code for interfaces, StP:

- generates interface forward declarations
- generates interface definitions
- ignores method bodies for interface methods
- ignores interface TOOL class properties
- ignores UML associations involving interfaces
- supports generalizations between interfaces
- supports multiple implementation links

Implementations

For 3.x code, an implementation link between a class and an interface maps to the TOOL construct:

```
implements interface1
```

provided the classes are of stereotype “type” or “interface.”

For 2.x code, this same link maps to the construct:

```
class <source> inherits <target>
```

Creating Classes for TOOL Code Generation

The following sections provide information about creating classes in generating TOOL code.

Class Names, Case Sensitivity, and Scoping

TOOL is not case sensitive to names, but StP is. For example, *ClassA* and *classa* are the same class in TOOL, but they are different classes in StP/UML. If you

generate code for two StP classes whose names differ only by case, the second class you import into TOOL overwrites the first.

TOOL scopes class names to projects, while class names in StP are globally scoped. For example, in TOOL, there is a distinction between *Proj1.ClassA* and *Proj2.ClassA*, while in StP, both *ClassA*s are the same class. To ensure that the integration between StP and TOOL works, you must scope classes globally.

Creating Placeholder Classes

Classes with no superclass in the StP/UML model generate code with an `inherits from Framework.Object` clause in the class definition. If you want the clause to specify a class provided by Forté, create a placeholder class.

To create a placeholder class:

1. Create a class in a class diagram.
2. Label the class with the name of the Forté-supplied class.
3. Using the **Class Properties** dialog box, designate the class to be external.
4. Create a generalization/inheritance hierarchy from a class to the placeholder.

Figure 10 illustrates a diagram showing a placeholder class.

Figure 10: Diagram Showing a Placeholder Class

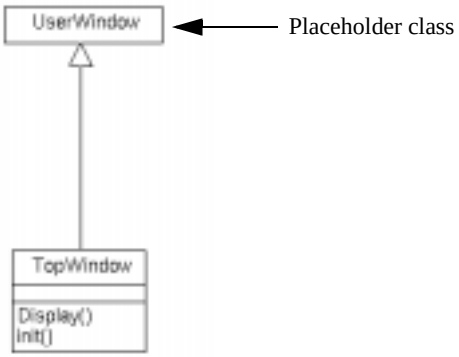


Figure 11 shows the generated code for a generalization involving a placeholder class.

Figure 11: Generated Code for a Placeholder Class

```
// stp class definition
class TopWindow is mapped inherits from
    DisplayProject.UserWindow
```

Adding TOOL Information to Class Tables

You define class attributes and operations using these sections of the Class Table Editor:

- Class member definitions

  See “Entering Class Member Definitions.”
- Analysis items

  See “Entering Analysis Items” on page 7-12.
- TOOL implementation items

  See “Entering TOOL Implementation Items” on page 7-12.

For general information about entering information in class tables for code generation, see “Defining Classes for Code Generation” on page 2-3.

Entering Class Member Definitions

Most of the information you enter in the Class Member Definitions section of a class table is common to all languages, as described in “Adding Class Member Definitions” on page 2-3. The information appearing in this section is specific to TOOL code generation.

Entering Class Member Definitions for Attributes

You use the Attributes section of a class table to define TOOL attributes, virtual attributes, and constants. Table 2 describes the attribute columns in the Class Members Definition section that affect TOOL code generation.

Table 2: Class Member Definitions for Attributes

Header	Description
Attribute	The name of the attribute or constant (a string).
Type	The data type of the attribute or constant. You can enter a simple data type or any class.
Default Value	You use this for constants only, in place of a data type. In TOOL code, attributes cannot have default values. If you enter a default value for an attribute, StP ignores the value and sends an error message when you generate code.

Entering Class Member Definitions for Operations

You define TOOL methods, events, and event handlers in the Operations section of the class table. Table 3 lists and describes the operations columns in the Class Member Definitions section.

Table 3: Class Member Definitions for Operations

Header	Description
Operation	The name of the method, event, or event handler (a string). For default Init or Display methods, enter **Init** or **Display** as the method name.
Arguments	The arguments of the method, event, or event handler, if any (a string). Use the correct TOOL syntax, exactly as you want it to appear in the generated code. If the cell is blank, the value of Arguments is void.
Return Type	For methods only. Specifies the method's return type, if any (a string).

Entering Analysis Items

The information you provide in the Analysis Items section of a class table is common to all languages. For instructions, see “Analysis Items” on page 2-5.

Entering TOOL Implementation Items

You enter TOOL implementation items in the TOOL Items section of a class table. To display the TOOL Items section of a class table, follow instructions given in *Creating UML Models*.

Table 4 lists and describes the attribute columns in the TOOL Items Section of the class table.

Table 4: TOOL Attribute Items in the Class Table

Header	Description
Privilege	The attribute or constant’s privilege (Public or Private).
Set Expr	For virtual attributes only. The expression evaluated by the program to set the attribute’s value (a string).
Get Expr	For virtual attributes only. The expression evaluated by the program to get the attribute’s value (a string).

Table 5 lists and describes the operation columns in the TOOL Items Section of the class table.

Table 5: TOOL Operation Items in the Class Table

Header	Description
Privilege	The method, event, or event handler’s privilege (Public or Private).
Return Event	The method’s return event, if any (a string).
Exception Event	The method’s exception event, if any (a string).

Table 5: TOOL Operation Items in the Class Table (Continued)

Header	Description
Copy Return?	If True, turns on the copy option for the method's return type (True/False).
Kind?	Identifies the operation type (Method, Event, or Event Handler).

Using TOOL Display Marks

Several TOOL items in the class table can cause display marks to appear in a class diagram. For example, a display mark indicates a TOOL event. For a complete list of StP/UML display marks, see *Creating UML Models*.

TOOL and C ++ Privilege Display Marks

By default, the privilege display marks for TOOL do not display on the diagram. To display them, set the display marks to Update on Demand or Continuous Update. For information on setting display marks, see *Fundamentals of StP*.

The C++ privilege display marks look the same as TOOL privilege display marks and are Update on Demand by default. To avoid confusion, change the C++ privilege display marks to Do Not Display: CXXPrivilegeMark for UML.

Defining Attributes and Virtual Attributes

To define an attribute or virtual attribute in the class table:

1. In the Attribute cell, type the attribute's name.
2. In the Type cell, declare the attribute data type.
3. Leave the Default Value cell blank.
4. In the Privilege cell, specify the attribute's privilege.
5. If the attribute is a virtual attribute, enter its get expression in the Get Expr cell.
6. Optional. In the Set Expr cell, enter the attribute's set expression.

Defining Constants

To define a constant in the class table:

1. In the Attribute cell, define the constant's name.
2. Leave the Type cell blank.
3. In the Default Value cell, type the constant's value.
4. In the Privilege cell, specify the constant's privilege.

Defining Methods

To define a method or event in the class table:

1. In the Operation cell, type the method's name.
2. In the Arguments cell, add arguments.
3. If the method returns a value, add a value in the Return Type cell.
4. In the Privilege cell, specify the method's privilege.
5. If the method has a completion option, add a value in the Return Event or Exception Event cell.

You use the Object Annotation Editor to create method bodies. For information, see "Defining Method Bodies" on page 7-19.

Specifying Default Methods

As does TOOL, StP/UML optionally provides default Init and Display method bodies for classes. (All classes can have default Init methods; only window classes have default Display methods.)

To provide a class with a default method, define an operation (method) named Init or Display in the class table.

Figure 12: Defining Default Methods

Top_Window			TOOL Items				
Attribute	Type	Default Value	Privilege	Set Expr	Get Expr		
			Private				
Operation	Arguments	Return Type	Privilege	Return Event	Exception Event	Copy Return?	Kind?
Init			Private			False	Method
Display			Private			False	Method

The code generated for the default methods illustrated in Figure 12 is:

```
// stp class declarations
forward Top_Window;
// stp class declarations end

// stp class definition
class Top_Window inherits from Framework.Object

// stp class components
   has Private method Int()
   has property extended = (StPSignature = '');
   has Private method Display()
   has property extended = (StPSignature = '');
// stp class components end

// stp class properties
// stp class properties end
end class;
// stp class definition end

//stp code

method Top_Window.Init
begin
super.init;
end method;
```

See "Generating Method Bodies" on page 7-20 for information on how to control generating the method bodies for default methods.

Defining Events and Event Handlers

To define an event or event handler in the class table:

1. In the Operation cell, type the event or event handler's name.
2. In the Arguments cell, add event or event handler arguments.
3. Leave the Return Type cell blank.
4. In the Privilege cell, specify the event or event handler's privilege.
5. Set the Kind cell value to Event or EventHandler.

Adding TOOL Annotations

You use the OAE to specify TOOL code generation features to:

- Set class properties, if different from the TOOL defaults
- Specify window class filenames for subclasses of *UserWindow*
- Define method bodies
- Specify when to generate method bodies
- Annotate associations

For general instructions on annotating objects, see "Annotating Classes and Relationships for Code Generation" on page 2-7. This section provides specific information about annotating objects for TOOL.

TOOL Annotations

Table 6 summarizes TOOL-specific annotations. Details appear in sections that follow the table.

Table 6: Annotations for TOOL Code Generation

Object	Note	Item	Item Value	Description
Class	Class TOOL Implementation	Shared Property	The class creates shared objects that allow multiple tasks to access and change the data	
			Disallowed	Allowed = off
			Allowed	Allowed = on; Default = off
			Default	Allowed = on; Default = on
		Subclass Override Shared Property	True/False	If True, subclasses of the class can override the class's setting
		Distributed Property	Applies only to shared objects: allows all partitions in the application to refer to the object, not just a copy of the object	
			Disallowed	Allowed = off
			Allowed	Allowed = on; Default = off
			Default	Allowed = on; Default = on
		Subclass Override Distributed Property	True/False	If True, subclasses of the class can override the class's setting
		Transactional Property	The class creates transitional objects that can participate in Forté transactions	
			Disallowed	Allowed = off
			Allowed	Allowed = on; Default = off
			Default	Allowed = on; Default = on
		Subclass Override Transactional Property	True/False	If True, subclasses of the class can override the class's setting

Table 6: Annotations for TOOL Code Generation (Continued)

Object	Note	Item	Item Value	Description
Class	Class TOOL Implementation	Monitored Property	For classes mapped to window widgets: updates the display whenever data is updated	
			Disallowed	Allowed = off
			Allowed	Allowed = on; Default = off
			Default	Allowed = on; Default = on
		Subclass Override Monitored Property	True/False	If True, subclasses of the class can override the class's setting
		Restricted Property	If True, the class belongs to a TOOL application intended to be integrated with restricted C routines	
		Window Class Filename	Specifies the filename of a window definition file created with Forté Window Workshop	
Operation (Method)	TOOL Source Code	Generate Method Body	Used only if the Generate Method Bodies option of the **Generate TOOL** command is set to Use Annotation: If True, the method body defined in the Note Description dialog generates for the method.	
		Note Description dialog	Defines the method body. For instructions on entering TOOL source code, see "Adding Source Code to Operations" on page 2-8.	
Association	Role TOOL Implementation	Privilege	Specifies the privilege of the association, whether Public or Private	
		Use Large Array	For associations of multiplicity = many; if True, the association is implemented with type LargeArray; if False, implemented with type Array	

Providing a Filename for a Window Class

StP automatically generates an `is mapped` clause as part of the class definition of each window class. A window class is a class that inherits from the *Display.UserWindow* class.

Provide a window definition filename for each Window class. You create a window definition file from a Forté window class using Forté's **Export** command. The filename must be in the format *<filename>.fsw*, where *<filename>* is any string you wish, and *.fsw* is the required extension.

The file you name in the annotation of an StP class must exist in the Forté environment when you import the code into Forté.

To provide a window class with a filename using the OAE:

1. Choose the **Class TOOL Implementation** note.
2. Set the value of the **Window Class Filename** item to **MainWind.fsw**.

For general information on using the OAE, refer to *Fundamentals of StP*.

The class definition generated for the *Top_Window* class is:

```
// stp class definition
class Top_Window is mapped inherits from UserWindow

// stp class components
   has public method Init();
   has public method Display();
// stp class components end

// stp class properties
   has file 'MainWindow.fsw';
// stp class properties end
end class;
// stp class definition end
```

Defining Method Bodies

You add source code for method bodies with the Note Definition of the TOOL Source Code note.

To add source code for a method body:

1. In the class table, select the cell of the method (operation) you wish to define.
2. Start the Object Annotation Editor.
3. Choose the **TOOL Source Code** note.

4. In the note description field of the Set/Add area, type the code for the method.

Generating Method Bodies

As you design your system, you often go through several iterations of code generation. For example, after you generate code, you may add classes, associations, attributes, and so forth to your class diagrams, and then regenerate TOOL code to reflect the changes. If you use TOOL for the development and test iterations of method bodies, you must prevent overwriting method bodies written in TOOL with the method bodies you created using StP.

Not all methods have method bodies. For methods with method bodies, you must control generation of the method bodies to prevent overwriting. There are two levels of control:

- Generate Method Body annotation item (True, False)
- Generate Method Bodies option of the **Generate TOOL** command (All, None, Use Annotation)

Table 7 indicates how the Generate Method Bodies command option interacts with the Generate Method Body annotation to generate or suppress code generation.

Table 7: Controlling Method Body Generation

Command Option Setting	Annotation Item Setting		
	True	False	No Annotation
All	Y	Y	Y
None	N	N	N
Use Annotation	Y	N	N

To annotate a method body for code generation using the OAE:

1. In the class table, select the cell of the method (operation) you wish to specify.

2. Start the Object Annotation Editor.
3. Choose the **TOOL Source Code** note.
4. Choose the **Generate Method Body** item.
5. In the item value field of the Add/Set area, choose **True** or **False** from the options list.

Generating Default Method Bodies

An option of the **Generate TOOL** command controls how StP generates the code of a default method body. The Generate Default Method Bodies option has two settings:

- Init—Generates default Init method bodies
- Display—Generates default Display method bodies (for window classes only)

The Generate Default Method Bodies option works the same as the Generate Method Bodies option in conjunction with the Generate Method Body annotation, as described in Table 7.

Note: StP never generates a default method body if a non-generic body exists for the method.

Annotating Role TOOL Implementation

There are two types of association annotations specifically for TOOL code:

- Privilege—Determines whether the association attribute is public or private
- Use Large Array—Determines whether to use Array or LargeArray types for associations with multiplicity of many

Annotating Implementation of a Role

You can choose whether to annotate the implementation of individual roles.

Note: Make sure you specify the navigability of a role before you annotate its role TOOL implementation. For more information, refer to “Generate TOOL Command Properties” on page 7-22.

To annotate implementation of a role for an association using the OAE:

1. Choose the **Role TOOL Implementation** note.
2. Choose a privilege level item for the association, and choose a value of **Public** or **Private** for the item.
3. To specify a LargeArray type for an association with multiplicity of many, add a **Use Large Array** item with a value set to **True**.

Generating TOOL Code

Tool code generation commands have associated command properties that enable you to specify output file directory, name, extensions, and other options. These properties appear on the Generate TOOL for <object> dialog.

To generate TOOL code from your model:

1. In the Model pane on the StP Desktop, open the appropriate category.
2. Based upon the code you want to generate, select a particular diagram or class(es) from the objects pane.
3. Choose a **Generate TOOL for <object>** command from one of the tools listed in Table 6 on page 2-11.
4. In the **Generate TOOL for <object> Properties** dialog box, set the appropriate command properties, as described in Table 8.
5. Click **OK** or **Apply**.

For additional information, see “Generating Code from the StP Desktop” on page 2-10.

Generate TOOL Command Properties

Table 8 contains descriptions of TOOL code generation command properties.

Table 8: Generate TOOL Command Properties

Property	Description
Directory	Specifies the pathname for the file directory where output code is written. The default is *<project>/<system>/src_files*.

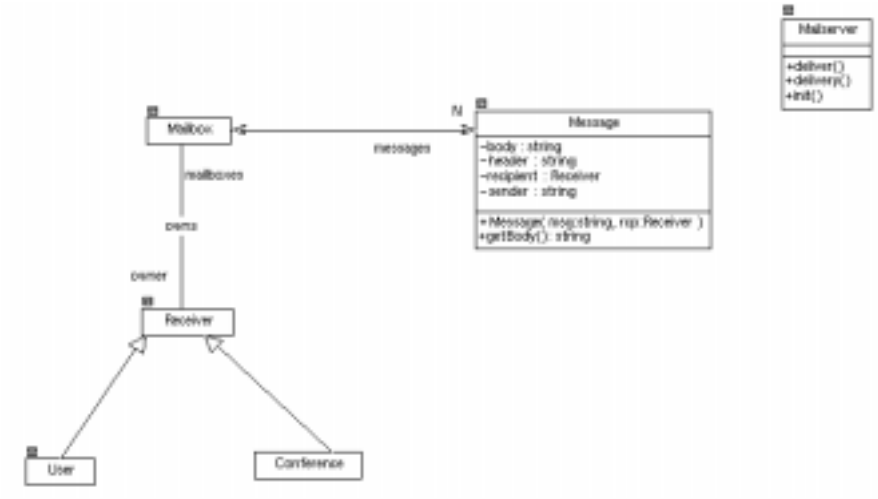
Table 8: Generate TOOL Command Properties (Continued)

Property	Description
One File Name Per Class	Generates a separate file for each class (TOOL interface) that is selected for code generation. The file names come from the class names when you select this option. This option is the default.
File Name	Active when One File Name Per Class option is not selected. Generates code for all classes to one file and permits you to specify its name.
TOOL Output File Extension	Specifies the extension appended to the name of the output file. The default is *.cex*.
Generate Code for Types/Interfaces	Generates TOOL code for all types and interfaces on a diagram, or a selected type or interface. This option is the default.
Generate Type Attributes	Generates TOOL code for the attributes associated with all types and interfaces on a diagram, or a selected type or interface. This option is the default. This option is not available if the Generate Code for Types/Interfaces option is not selected.
Generate Code For Release 2.x	If selected, does not generate code for interfaces, but will generate code for the interface as a regular class. The default is not to be selected.
Ignore Classes Annotated As External	Does not generate code for external class types. This option is the default.
Generate Method Bodies	Permits you to limit generation of method bodies to all, none, or only method bodies whose methods have the Generate Method Bodies item set to True.
Generate Default Method Bodies	Not active if you set Generate Method Bodies to None. If you set Generate Method Bodies to All, generates default Init and Display method bodies for all classes with Init and Display methods, unless you define a method body. If you set Generate Method Bodies to Use Annotation, generates default Init and Display method bodies for Init and Display methods unless you define a method body or annotate the method with a Generate Method Body item set to False.

Example of Generated TOOL Code

This example shows TOOL code generated for the classes shown in Figure 13.

Figure 13: Class Diagram for Example



In this example:

- The classes in this example contain attributes, methods, and events.
- You specify attributes and methods as public or private.
- The association of the *Receiver* class to the *Mailbox* class has a multiplicity of many.
- StP provides no placeholder class for any class.
- An init method is defined for the *MailServer* class

Class Tables for the Example

Four classes in the diagram have class tables, which are shown in Figure 14, Figure 15, Figure 16, and Figure 17.

Figure 14: *Mail Server* Class Table

MailServer			TOOL Items				
Attribute	**Type**	**Default Value**	**Privilege**	**Set Expr**	**Get Expr**		
			Private				
Operation	**Arguments**	**Return Type**	**Privilege**	**Return Event**	**Exception Event**	**Copy Return?**	**Kind?**
deliver			Public	delivered		False	Method
delivery			Public			False	Event
init			Public			False	Method

Figure 15: *Message* Class Table

Message			TOOL Items				
Attribute	**Type**	**Default Valu**	**Privilege**	**Set Expr**	**Get Expr**		
body	string		Private				
header	string		Private				
recipient	Receiver		Private				
sender	string		Private				
Operation	**Arguments**	**Return Type**	**Privilege**	**Return Event**	**Exception Event**	**Copy Return?**	**Kind?**
Message	msg: string, rcpt: Receiver		Public			False	Method
getBody		string	Public			False	Method

Figure 16: *Receiver* Class Table

Receiver			TOOL Items				
Attribute	**Type**	**Default Value**	**Privilege**	**Set Expr**	**Get Expr**		
address	string		Private				
Operation	**Arguments**	**Return Type**	**Privilege**	**Return Event**	**Exception Event**	**Copy Return?**	**Kind?**
recv		Message	Private			False	Method

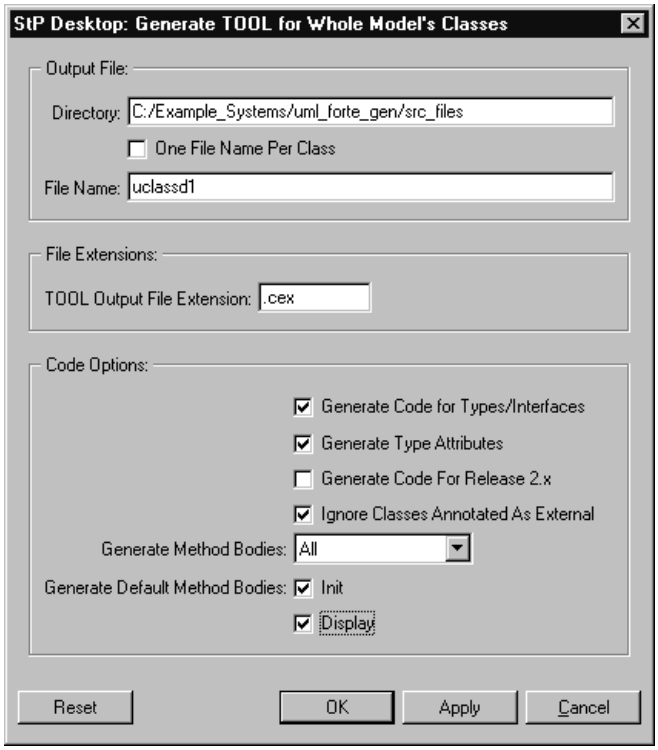
Figure 17: User Class Table

User			TOOL Items				
Attribute	**Type**	**Default Value**	**Privilege**	**Set Expr**	**Get Expr**		
Mail_system			private				
Operation	**Arguments**	**Return Type**	**Privilege**	**Return Event**	**Exception Event**	**Copy Return?**	**Kind?**
join_conf			private			False	Method

Generated Code for the Example

Figure 18 shows the options StP uses to generate the code for the subsystem *MailSystem*.

Figure 18: Code Generation Options



This is the generated code from *<project>/src_files/uclassd1.cex*:

```
// stp class declarations
forward Receiver;
forward User;
forward Conference;
forward Mailbox;
forward Message;
forward Mailserver;
// stp class declarations end
```

```
// stp class definition
class Receiver inherits from Framework.Object

// stp class components
   has private attribute address : string;
   has private method recv() : Message
   has property extended = (StPSignature = '');
// stp class components end

// stp class properties
// stp class properties end
end class;
// stp class definition end

// stp class definition
class User inherits from Receiver

// stp class components
   has private has private method join_conf()
   has property extended = (StPSignature = '');
// stp class components end

// stp class properties
// stp class properties end
end class;
// stp class definition end

// stp class definition
class Conference inherits from Receiver

// stp class components
// stp class components end

// stp class properties
// stp class properties end
end class;
// stp class definition end

// stp class definition
class Mailbox inherits from Framework.Object

// stp class components
   has private attribute address : string;
   attribute messages : Framework.Array of Message
   has property extended = (StPAttribute = 'association');
   has private method recv() : Message
   has property extended = (StPSignature = '');
// stp class components end
```

```
// stp class properties
// stp class properties end
end class;
// stp class definition end

// stp class definition
class Message inherits from Framework.Object

// stp class components
   has private attribute header : string;
   has private attribute recipient : Receiver;
   has private attribute sender : string;
   has private attribute body : string;
   attribute assnMailbox : Mailbox
   has property extended = (StPAttribute = 'association');
   has public method getBody() : string
   has property extended = (StPSignature = '');
   has public method Message(msg:string, rcp:Receiver)
   has property extended = (StPSignature = 'msg:string,
      rcp:Receiver');
// stp class components end

// stp class properties
// stp class properties end
end class;
// stp class definition end

// stp class definition
class Mailserver inherits from Framework.Object

// stp class components
   has private event delivery()
   has property extended = (StPSignature = '');
   has private method init()
   has property extended = (StPSignature = '');
   has private method deliver()
   where completion = (return = delivered)
   has property extended = (StPSignature = '');
// stp class components end

// stp class properties
// stp class properties end
end class;
// stp class definition end

// stp code
method Mailserver.Init
```

```
begin
super.init;
end method;

// stp code end
```

Renaming Objects and Forté

In StP, the **Rename Object Systemwide** command allows you to change the name of an object without changing the object's identity (without losing the object's annotations and relationships). If you want to rename an object for which StP has generated TOOL code and imported it into Forté, keep StP models in sync with the Forté components by either:

- Renaming the object in both StP and Forté
- Deleting the object from Forté, renaming it in StP, generating code for it, and importing it back into Forté

  This option works only if you have not edited pertinent method bodies on the Forté side; otherwise, they will be lost.

Changing a method's signature or arguments is analogous to renaming, in that you are changing an identifying attribute of the object. Therefore, if you wish to change a method's signature or return type, and the method has a body edited using TOOL, edit the arguments and/or return type in both places before you generate code, or the method body will be lost.

8 Generating Java Code

This chapter tells you how to generate Java code from your StP / UML model. It describes:

Use this chapter with Chapter 2, "Generating Code From UML Models." Chapter 2 provides information for generating code that applies to all languages.

What is Generated for Java

For each class in a class diagram that is not external, StP generates a source file. The default directory is the *src_files* directory in your project / system path. The default output file extension *.java.*

The generated code conforms to the Java standard.

StP to Java Mapping

Some constructs in Java have clear mappings to UML objects; you map others through annotations or other means. Table 1 shows how the Java constructs with clear mappings are represented in StP / UML.

Table 1: Java Constructs Mapped to UML Objects

Object	Java Construct
Class	class <label>
Interface	interface <label>
Package	package <label>
Parameterized class	class <label>
Object	Ignored in Java
Attribute	Field
Operation	Method
Association	Field
Generalization	class <source> extends <target>
Implementation	class <source> implements <target>, <target>, ...
Dependency	import <target>

Adding Java Information to Class Tables

You define class attributes and operations using these sections of the Class Table Editor:

- Class member definitions
- Analysis items

- Java implementation items

Entering Class Member Definitions and Analysis Items

The information that you must provide in the Class Member Definitions and Analysis Items sections of a class table is common to all languages. For instructions, see “Adding Class Member Definitions” on page 2-3 and “Analysis Items” on page 2-5.

Entering Implementation Items for Java

You enter Java implementation items in the Java Items section of a class table.

Entering Java Items for Attributes

Table 2 lists and describes the attributes columns in the Java Items Section of the class table.

Table 2: Class Table Attribute Java Items

Header	Description
Visibility	The field access of the attribute in Java terms (public / private / protected).
Final?	Specifies whether or not the attribute is a “final” attribute in Java (True / False). The default is False. For details, see the Java Language Specification manual.
Transient?	Specifies whether or not the attribute is a “transient” attribute in Java (True / False). The default is False.
Volatile?	Specifies whether or not the attribute is a “volatile” attribute in Java (True / False). The default is False.

Entering Java Items for Operations

Table 3 lists and describes the operations columns in the Java Items Section of the class table.

Table 3: Class Table Operation Java Items

Header	Description
Visibility	The field access of the operation in Java terms (public / private / protected / private protected).
Final?	Specifies whether or not the operation is a "final" attribute in Java (True / False). The default is False.
Native?	Specifies whether or not the operation is a "native" operation in Java (True / False). The default is False.
Synchronized?	Specifies whether or not the operation is a "synchronized" operation in Java (True / False). The default is False.

Adding Java Annotations

Java-specific annotations include:

- Class Java Declaration
- Class Java Definition
- Operation Java Source Code
- Role Java Definition
- Object descriptions on classes, attributes, and operations (can be generated as comments)

For general instructions on annotating objects, see "Annotating Classes and Relationships for Code Generation" on page 2-7. This section provides specific information about annotating objects for Java.

Java Annotations

Table 4 summarizes Java-specific annotations and the type of code they generate. Details are given in sections that follow the table.

Table 4: Annotations for Java Code Generation

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. For more information, see "Adding Java Declarations" on page 8-6.	
	Class Java Definition	Enclosing Scope	Specifies the name of a package enclosing the parent class.	
Operation	Java Source Code	Note Description dialog	Lets you add Java source code for the operation. For instructions, see "Adding Source Code to Operations" on page 2-8.	
Role	Role Java Definition	Visibility	Public / Private / Protected	Lets you indicate the Java field access of constructs used to implement associations between classes.
		Initial Value	Lets you specify an optional assignment statement to follow the field declaration.	
		Pattern Family	Generic / Instance	Lets you indicate the set of association configuration patterns to use. You can customize templates in *<stp_file_path>/templates/uml/qrl/code_gen/java/association_patterns.inc*. For more information, see "Adding Customized Pattern Families" on page 8-8.

Table 4: Annotations for Java Code Generation (Continued)

Object	Note	Item	Item Value	Description
Class, Role, Attribute, or Operation	Requirement	Name	Lets you type Java comments in the Note Description dialog or the Value field for the item. For instructions on using the Note Description dialog, see "Adding Source Code to Operations" on page 2-8.	
		Document		
		Paragraph		
	Object	Author		
		Generated		
		(I18N) Generate as		

Adding Java Declarations

You can add code for Java constructs that are scoped to a class by:

- Using the Java Declaration note
- Adding the required code to the generated Java code

If other Java constructs, such as import statements, are scoped to a file, module, or operation, you must add the code manually to the generated Java code.

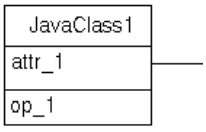
For instructions on adding a declarations note, see "Annotating Classes and Relationships for Code Generation" on page 2-7.

This section gives an example of entering a Java declaration with the Object Annotation Editor.

Example of Java Declarations

The class used in the example appears in Figure 1.

Figure 1: *JavaClass1* Class



At code generation, StP extracts the following code from the repository and places it in the interface definition. The code generated for this class is shown as the *JavaClass1.java* file.

JavaClass1.java File

```
// StP
// Created on 11/19/97 12:35:54 for dougl@HAWK
//   from system Z:/Core_2.4/StP_V2.4/project/dlarkin

// stp class definition 29
class JavaClass1
{
// stp class members
[something Javalike here]

    private volatile int attr_1;

    // stp method 29::30
    public void op_1( )
    {
        // stp code
        // end stp code
    }
    // end stp method

// end stp class members
}
// end stp class definition
```

Specifying a Pattern Family

The data members of a class that StP generates to implement associations in Java can be tailored by using code generation patterns. The pattern that is used depends on the multiplicity and qualification of the association. A complete set of these patterns, which includes every combination of multiplicity and qualification of an association, is called a pattern family, because every member of the set has something in common.

A pattern is a text string that usually contains one or more of the UML Java string substitution symbols.

StP/UML lets you specify which pattern family to use when roles are implemented for Java code generation.

Specifying a pattern family is not required.

The pattern family choices are:

- Generic
- Instance

You can customize existing and add new pattern families. Customizable and configurable patterns appear in the *<stp_file_path>/templates/uml/qrl/code_gen/java/association_patterns.inc* file. For more information on customizing the *association_patterns.inc* file, see "Adding Customized Pattern Families" below.

To specify the pattern family, use one of these methods:

- Select the pattern family from the Generate Java dialog, as described in "Generating Java Code" on page 8-15.
- Use the Object Annotation Editor (OAE). See Table 4 on page 8-5 for information on the Role Java Definition note and the Pattern Family item.

  For general information on using the OAE, refer to *Fundamentals of StP*.

Adding Customized Pattern Families

Optionally, you can customize existing or add new pattern families by editing the *<stp_file_path>/templates/uml/qrl/code_gen/java/association_patterns.inc* file.

The new pattern family name automatically joins the Value pull-down menu in the Object Annotation Editor (OAE). To add the pattern family name to the Generate Java Command dialog box, see "Adding Pattern Families to the Java Dialog Box" on page 8-13.

To customize existing or add new pattern families, edit the *association_patterns.inc* file. In brief, you need to:

1. Add a declaration and initialization statement for a new *uml_java_pattern_descriptor* type.

2. Add a new initialization function for the declaration described in Step 1.
3. Add a declaration and initialization statement for a new *uml_java_binary_pattern_descriptor* type.
4. Add a new initialization function for the declaration described in Step 3.
5. Add a new initialization function for the *list* declaration to add a pattern name to the Object Annotation Editor (OAE) pull-down menu.

You can do each step by copying existing code in the *association_patterns.inc* file and modifying identifier names within the statements.

The following is an example of adding a new pattern family, named TestPattern, in the *association_patterns.inc* file:

1. Locate the declaration and initialization statement for the default *uml_java_pattern_descriptor* type similar to the following statement:

```
uml_java_pattern_descriptor
default_uml_java_pattern_family =
     initialize_default_uml_java_pattern_family();
```

2. Cut and paste the statement from Step 1 and change the identifiers for the variable name and the initialization function:

```
uml_java_pattern_descriptor
TestPattern_uml_java_pattern_family =
   initialize_TestPattern_uml_java_pattern_family();
```

3. Locate the initialization function for the default *uml_java_pattern_descriptor* type similar to the following:

```
uml_java_pattern_descriptor
initialize_default_uml_java_pattern_family()
{
   uml_java_pattern_descriptor uml_java_pattern_family;
   uml_java_pattern_family.name = "Generic";

   uml_java_pattern_family.associations =
      java_generic_patterns;
    uml_java_pattern_family.aggregations =
      java_generic_patterns;
    uml_java_pattern_family.compositions =
      java_generic_patterns;
```

```
    uml_java_pattern_family.nary_pattern =
         "Assn_${target_class.name} ${identifier_name}";
    return uml_java_pattern_family;
}
```

4. Change the function name, pattern family name field, and pattern family identifiers:

```
uml_java_pattern_descriptor
initialize_TestPattern_uml_java_pattern_family()
{
    uml_java_pattern_descriptor uml_java_pattern_family;
    uml_java_pattern_family.name = "TestPattern";

    uml_java_pattern_family.associations =
       TestPattern_java_patterns;
    uml_java_pattern_family.aggregations =
       TestPattern_java_patterns;
    uml_java_pattern_family.compositions =
       TestPattern_java_patterns;

    uml_java_pattern_family.nary_pattern =
       "Test_Assn_${target_class.name}
          ${identifier_name}";

    return uml_java_pattern_family;
}
```

5. Locate the declaration and initialization statement for the pattern family identifier type *uml_java_binary_pattern_descriptor*:

```
uml_java_binary_pattern_descriptor
java_generic_patterns =
    initialize_java_generic_patterns();
```

6. Cut and paste the statement and change the identifier and initialization function name:

```
uml_java_binary_pattern_descriptor
TestPattern_java_patterns =
    initialize_TestPattern_java_patterns();
```

7. Locate the initialization function *initialize_java_generic_patterns*:

```
uml_java_binary_pattern_descriptor
initialize_java_generic_patterns()
{
    uml_java_binary_pattern_descriptor patterns;

    patterns.unqualified_single_pattern =
```

```
        "${target_class.name} ${identifier_name}";
    patterns.unqualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.unqualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.qualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.qualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.qualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.inverse_qualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.inverse_qualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.inverse_qualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.double_qualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.double_qualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.double_qualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";

    return patterns;
}
```

8. Cut and paste the function and change the name:

```
uml_java_binary_pattern_descriptor
initialize_TestPattern_java_patterns()
{
    uml_java_binary_pattern_descriptor patterns;

    patterns.unqualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.unqualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.unqualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.qualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.qualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.qualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";
```

```
    patterns.inverse_qualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.inverse_qualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.inverse_qualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.double_qualified_single_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.double_qualified_unordered_pattern =
        "${target_class.name} ${identifier_name}";
    patterns.double_qualified_ordered_pattern =
        "${target_class.name} ${identifier_name}";

    return patterns;
}
```

9. Modify the dollar (“$”) variables.

   The target_class.name variable is the name of the class symbol that corresponds to the Java abstract type.

   The identifier_name variable is the role name that corresponds to the Java object name.

10. Locate the initialization function for the *list* type similar to the following:

```
list
initialize_uml_java_pattern_families()
{
    list uml_java_pattern_families =
list_create("uml_java_pattern_descriptor", 0);

    list_append(uml_java_pattern_families,
default_uml_java_pattern_family);
    list_append(uml_java_pattern_families,
simple_uml_java_pattern_family);

    return uml_java_pattern_families;
}
```

11. Cut and paste a list_append identifier and change the identifier pattern name:

```
    list_append(uml_java_pattern_families,
TestPattern_uml_java_pattern_family);
```

12. Save and close the file.

Adding Pattern Families to the Java Dialog Box

After adding a new pattern family to the *association_patterns.inc* file, you can add the pattern family name to the Generate Java Command dialog box. The name appears on the For Associations Use Pattern Family pull-down menu.

The following is an example of adding the pattern family created in the previous example to the Generate Java Command dialog box:

1. Use an editor to open the *<stp_file_path>/templates/uml/rules/stp_code_gen.rules* file.
2. Locate the JavaImplementAssociations property.
3. Add a new choice item:

```
{ ChoiceItem
   { Label "TestPattern" }
   { Value "-x
implement_associations_using_pattern_family
'TestPattern'" }
}
```

4. Save and close the file.

Generating Package Statements

StP generates a package statement for a file containing a class if either of the following occurs:

- The class that you are generating code for is enclosed in a package by using the **Add Class to Enclosing Package** command. For more information, see Chapter 4, “Creating a Class Diagram” in *Creating UML Models*.
- The Enclosing Scope annotation for the class (or interface) is the name of a package. Form more information, see Chapter 4, “Creating a Class Diagram” in *Creating UML Models*.

The format for the package statement is:

```
package <package_name>;
```

Generating Import Statements

An import statement generates for a file containing a class if:

- The class has a dependency relationship with a package or another class
- The class has a generalization relationship with another class (or interface) and has an Enclosing Scope that is the name of a package
- The class has an association relationship with another class or interface, role navigability is set for at least one association role, and the class has an Enclosing Scope that is the name of a package

The two formats for import statements are:

```
import <package_name>.<class_name>;
import <package_name>.*;
```

For information on setting the Enclosing Scope, see Chapter 4, “Creating a Class Diagram” in *Creating UML Models*.

Designating Inner Classes

StP / UML supports designating top-level and member classes or interfaces.

To designate a nested top-level or member class or interface, enter the name of the parent class or interface in the inner class’s Enclosing Scope annotation Chapter 4, “Creating a Class Diagram” in *Creating UML Models*.

The code generated for the inner class nests in the parent class’s code below its member methods.

Generating Java Code

Java code generation commands have associated command properties that enable you to specify output file directory, name, and extensions.

To generate Java code from your model:

1. In the Model pane on the StP Desktop, open the appropriate category.
2. Based upon the code you want to generate, select a particular diagram or class(es) from the objects pane.
3. Choose a **Generate Java for <object>** command from one of the menus listed in Table 6 on page 2-11.
4. In the **Generate Java for <object> Properties** dialog box, set the appropriate command properties, as described in Table 5.
5. Click **OK** or **Apply**.

For additional information, see “Generating Code from the StP Desktop” on page 2-10.

Generate Java Command Properties

Table 5 contains descriptions of Java code generation command properties.

Table 5: Generate Java Command Properties

Property	Description
Directory:	Specifies the pathname for the directory where output code is written. The default is *proj/sys/src_files*.
One File Name Per Class	Generates a separate file for each class (Java interface) that is selected for code generation. The file names are taken from the class names when you select this option. This option is the default.
File Name:	Active when One File Name Per Class option is deselected. Generates code for all classes to one file and permits you to specify its name.

Table 5: Generate Java Command Properties (Continued)

Property	Description
Preserve Code From Other Classes	Requires One File Name Per Class to be deselected. Retains or deletes any previously-generated code for classes and operations not in the current set of classes for which code is being generated. Default is not selected (do not Keep).
Java Output File Extension:	Specifies the extension appended to the name of the output file. The default is *.java*.
For Associations Use Pattern Family:	Specifies the set of association configuration patterns to use. Options are Generic or Instance.
Ignore Classes Annotated As External	Does not generate code for classes annotated as external.
Generate Comments For:	Generates comment lines with the text of Object annotations for Classes, Fields (attributes), and Methods (operations). The default is to be not selected.
Format to:	Modifies the number of comment characters per line. The default is 80.

Example of Generated Code

This example shows Java code generated for the *ScribbleFrame* class in the Scribble Tablet Example, shown in Figure 2.

Figure 2: Scribble Tablet Example Class Diagram

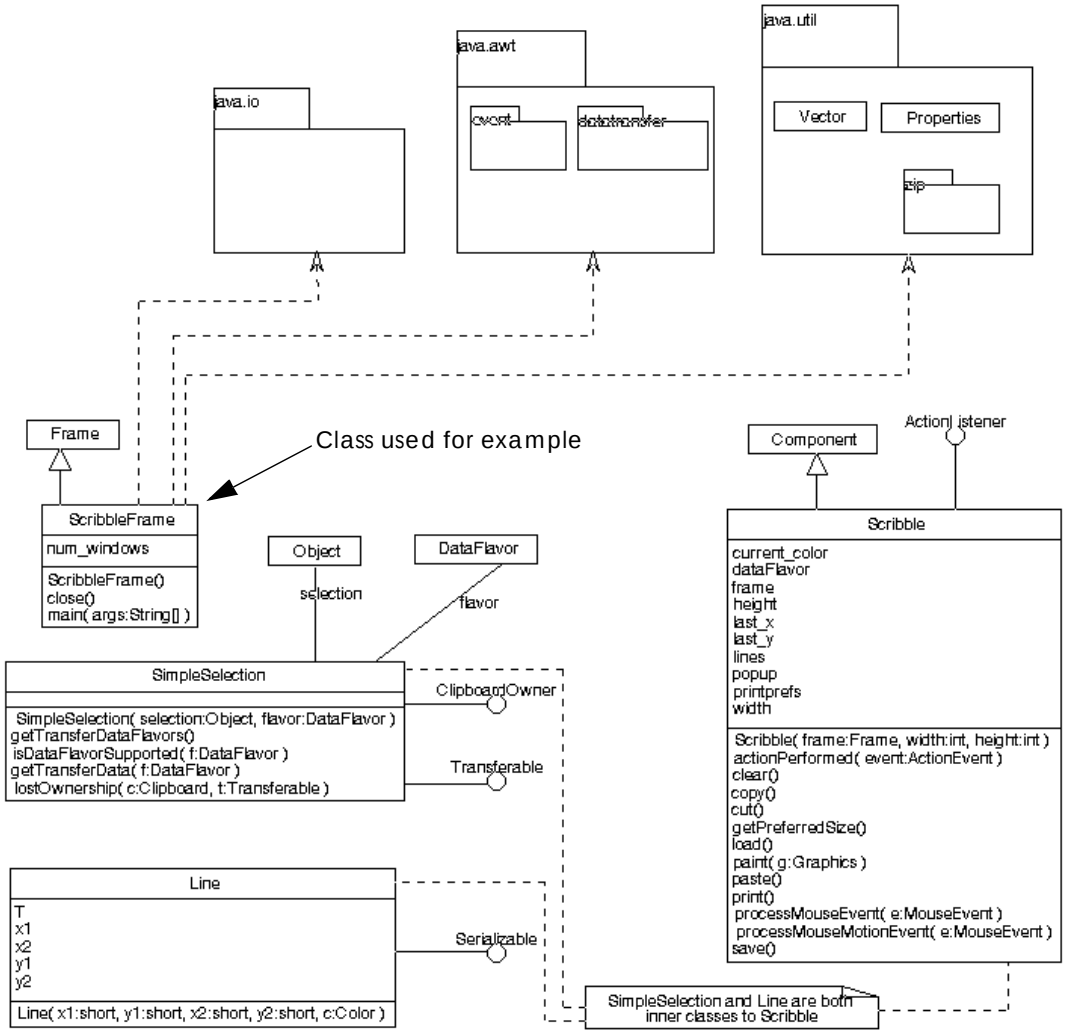


Figure 3 shows the class table for the *ScribbleFrame* class.

Figure 3: ScribbleFrame Class's Class Table

ScribbleFrame			Java Items			
Attribute	Type	Default Value	Visibility	Final?	Transient?	Volatile?
num_windows	int	0	protected			
Operation	Arguments	Return Type	Visibility	Final?	Native?	Synchronized?
main	args:String[]	void	public			
ScribbleFrame			public			
close		void				

This section contains the output file generated for the *ScribbleFrame* class.

```
// StP
// Created on Fri Nov 21 14:27:51 1997 for dougl@verdi
//   from system /home/horn/project/scribble

// stp imports 1
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import java.util.*;
import java.util.Vector;
import java.util.Properties;
import java.util.zip.*;
// end stp imports


// stp class definition 1
public class ScribbleFrame extends Frame
{
// stp class members

    protected static int num_windows = 0;

    // stp method 1::2
    public static void main( String[] args )
    {

        // stp code
```

```
            new ScribbleFrame();
            // end stp code

        }
        // end stp method

        // stp method 1::3
        public  ScribbleFrame( )
        {

            // stp code
            super("ScribbleFrame");
            num_windows++;

            ScrollPane pane = new ScrollPane();
            pane.setSize(300, 300);
            this.add(pane, "Center");
            Scribble scribble;
            scribble = new Scribble(this, 500, 500);
            pane.add(scribble);

            MenuBar menubar = new MenuBar();
    this.setMenuBar(menubar);
            Menu file = new Menu("File");
            menubar.add(file);

            MenuItem n, c, q;
            file.add(n = new MenuItem("New Window",
                                      new
    MenuShortcut(KeyEvent.VK_N)));
            file.add(c = new MenuItem("Close Window",
                                      new
    MenuShortcut(KeyEvent.VK_W)));
            file.addSeparator();
            file.add(q = new MenuItem("Quit",
                                      new
    MenuShortcut(KeyEvent.VK_Q)));

            n.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e)
                { new ScribbleFrame(); }
            });

            c.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e)
```

```
            { close(); }
        });

        q.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            { System.exit(0); }
        });

        this.addWindowListener(new WindowAdapter() {
            public void actionPerformed(ActionEvent e)
            { close(); }
        });

        this.pack();
        this.show();
        // end stp code

    }
    // end stp method

    // stp method 1::4
    public void close( )
    {

        // stp code
        if (num_windows-- == 0)
            System.exit(0);
        else
            this.dispose();
        // end stp code

    }
    // end stp method

// end stp class members
}
// end stp class definition
```

Generating Java Incrementally

This section contains information applying to incremental code generation for Java. For a general discussion of incremental code generation, see “Generating Code Incrementally” on page 2-11 in *Generating and Reengineering Code*.

As part of code generation, StP generates all of a class’s operations into function headers or bodies that appear in an implementation file. If you generate code to the same file more than once, StP preserves some regions of the file. To add code to the file that StP will retain on subsequent code generations, you must add the code to these regions. For specific descriptions of these regions, see the following section, “Guidelines for Changing Files.”

If you add or delete operations in an implementation file from the class, you do so during incremental code generation.

Guidelines for Changing Files

You can edit a generated file anywhere except:

- In generated comments
- Between related lines or in single lines

If you wish to change parts of the code that cannot be edited, you must change the model, save the changes, and regenerate the code.

Table 6 provides the editing guidelines.

Table 6: Editing Guidelines

Edit between this line:	And this line:
`{` (Designates the beginning of a block of code.)	`// stp <start tag>` (Designates the beginning of StP code generation.)
`// end stp <end tag>` (Designates the end of StP code generation.)	`}` (Designates the end of a block of code.)

The following is an example of where you can add code that is retained in subsequent generations:

```
public class ScribbleFrame extends Frame
{
    *ADD USER CODE HERE
    // stp class members
    protected static int num_windows = 0;
    *DO NOT ADD USER CODE HERE
        // stp method 7::8
        public static void main( String[] argv )
       {
            *ADD USER CODE HERE
            // stp code
            *DO NOT ADD USER CODE HERE
            // end stp code
            *ADD USER CODE HERE
        }
        // end stp operation
    *DO NOT ADD USER CODE HERE
    // end stp class members
    *ADD USER CODE HERE
    }
```

9 Reverse Engineering

You can use the StP/Object-Oriented Reverse Engineering tool to generate a new StP/UML model or update an existing model from C++ and Java source code files. With TOOL code, you create the model from the TOOL repository. (You can also create or update a model from IDL source code files, but this feature is not supported by Technical Support.)

Reverse Engineering is purchased separately from StP/UML. Contact your Aonix sales representative for details.

This chapter describes:

Overview

Reverse Engineering provides the following tools:

- Source code parser—Extracts code from multiple source files to produce a comprehensive semantic model of your software. (This feature is not applicable for TOOL source code files.)
- Comment extractor—Associates comments extracted by the parser with classes, members, and functions. (This feature is not applicable for TOOL source code files.)
- Model generator—Uses the semantic model to automatically create the diagrams and annotations that compose a graphical model of your software.

  For TOOL code, you use the TOOL repository to create the graphical model.

Using Reverse Engineering

The reverse engineering process generates both a semantic and a graphical model of your software.

Note: The reverse engineering process does not create a semantic model from TOOL code.

The semantic model contains information extracted from source code and is used to generate the graphical model. The graphical model consists of:

- A class table for each class
- An annotation file for each class, attribute, and operation
- A generalization diagram
- An association relationship diagram
- An aggregation relationship diagram (Aggregation diagrams are not generated for models created from Java or IDL.)
- A nesting diagram (Nesting diagrams are not generated for models created from TOOL.)

StP stores the diagrams, tables, and annotations created by reverse Engineering in the repository for the current project and system. All other reverse engineering files appear in the */revc_files* directory for the current project and system, as described in "Reverse Engineering Directory Structure (C++)" on page 9-55.

StP stores the diagrams, tables, and annotations created by Reverse Engineering of TOOL code in the repository for the current project and system. All other reverse engineering files for TOOL appear in the *uclassd*, *uclasst*, *ant_files*, and *src_files* directories.

An Overview of the Reverse Engineering Process

To use Reverse Engineering:

1. From the **Code** menu on the StP Desktop, choose **Reverse Engineering**.
2. Choose **Parse Source Code** to create a semantic model of your system. For additional information, see "Parsing the Source Files" on page 9-5.

   If you are generating a model from TOOL code, choose **Generate Model from TOOL Code**. For additional information, see "Generating a Model from TOOL Code" on page 9-36.
3. If you want source code comments to appear as object annotations in your graphical model, choose **Extract Source Code Comments** after parsing is complete.

   For additional information, see "Extracting Comments" on page 9-25.
4. Choose **Generate Model from Parsed Source Files** to create the diagrams, tables, and annotations that compose the graphical model of your system.

   For additional information, see "Generating a Model from Parsed Files" on page 9-30.

Using Reverse Engineering Menus and Commands

The **File**, **View**, **Code**, **Report**, **Repository**, **Tools**, and **Help** menus on the StP Desktop contain the standard selections available in all StP products.

These menus and the associated commands are described in *Fundamentals of StP*. Table 1 describes only those commands specific to reverse engineering, all of which appear on the **Reverse Engineering** submenu of the StP Desktop **Code** menu.

Table 1: Reverse Engineering Commands

Command	Description	For Details, See
Parse Source Code	Extracts all information needed to generate design documentation from the source code and stores it in a semantic model. The semantic model is separate from the object repository.	"Parsing the Source Files" on page 9-5
Extract Source Code Comments	Associates comments extracted by the parser with classes, members, and functions.	"Extracting Comments" on page 9-25
Generate Model from Parsed Source Files	Automatically creates a graphical model of the reverse engineered software, consisting of class diagrams, class tables, and annotations. This model is saved to the repository.	"Generating a Model from Parsed Files" on page 9-30
Check RE Semantic Model Locks	Checks to find out the lock status of a semantic model.	"Checking Semantic Model Locks" on page 9-23
Check RE Semantic Model Consistency	Checks to find out if a semantic model has been corrupted.	"Checking the Semantic Model Consistency" on page 9-24
Generate Model from TOOL Code	Automatically creates a graphical model of the reverse engineered software, consisting of class diagrams, class tables, and annotations. This model is saved to the repository. Use this command for TOOL code only.	"Generating a Model from TOOL Code" on page 9-36

Parsing the Source Files

The source code parser extracts code from multiple source files and produces a semantic model of your software. The semantic model is tightly bound to the source code and emphasizes, at a low level, the information flow through the software. All other Reverse Engineering commands require access to the semantic model.

The parser parses ANSI C or C++, Java (1.0 and 1.1), and IDL source code files. (The ccparser is used to parse the source code files for all of these languages.)

You can parse all or selected source code files for your software. You can also parse the files in incremental mode to update an existing semantic model with any source code changes made since the semantic model was last generated. For more information, see "Using Incremental Mode" on page 9-11.

The parser uses a utility called the Makefile Reader, which reads a makefile to find target and prerequisite files. (The target file is the executable program that the source files build, and the prerequisite files are the files that are parsed.) If you do not use the Makefile Reader, you need to supply the source files to parse on your own. For more information, see "Using the Makefile Reader" on page 9-13.

Using the Parse Source Code Dialog Box

You use the **Parse Source Code** dialog box and its related dialogs to parse the source code files.

The **Parse Source Code** dialog box options appear in Table 2.

Table 2: Parse Source Code Dialog Box Options

Specification	Description	For Details, See
Filter button	Applies the filter pattern in the filter text field to the Directories search box.	"Deciding Which Files to Parse" on page 9-9
Filter text field	Displays the current directory or user-specified pattern for determining the contents of the Directories window.	
Directory Listing	Provides a file / directory window from which you choose file(s) to be parsed.	
Files to Parse	Contains the files to be parsed. When you select the files to parse, you can mix languages.	
Read Selected Makefile button	Reads the selected makefile and displays the appropriate information in the Parse Source Code window.	"Using the Makefile Reader" on page 9-13
Makefile Options button	Displays the Makefile Reader Options dialog, where you can change the default makefile options.	
Incremental Mode	Parses only those specified files that have been modified since the last time they were parsed.	"Using Incremental Mode" on page 9-11
Use Microsoft Extensions	If on, parses files with Microsoft extensions. The default is on.	"Using Microsoft Extensions" on page 9-12
Preprocessor Options	Displays the Preprocessor Options dialog, where you can change the default preprocessor options.	"Using the Parser Preprocessor Options Dialog Box" on page 9-16

Parsing the Files

To parse the source code files:

1. From the **Code** menu, choose **Reverse Engineering > Parse Source Code**.
2. In the Filter text field of the **Parse Source Code** dialog box, type the directory path of the file(s) you want to parse.
3. Click the **Filter** button.

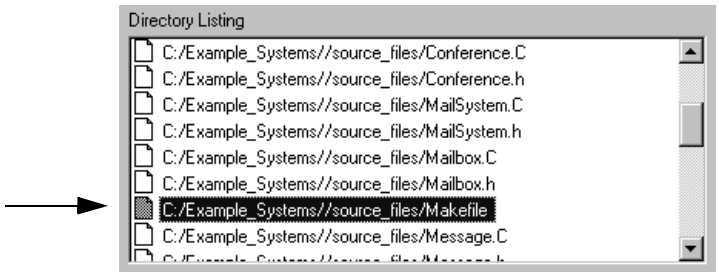
   The directory's contents are displayed in the Directories scrolling text field.

   To display the contents of a subdirectory, double click on a directory name in the scrolling list.

   To display the contents of the parent directory, double click Parent Directory, which appears at the top of the scrolling list.
4. Take *one* of the following actions:
   - Select a makefile, as shown in Figure 1. Go to Step 5.
   - Select the source file(s), and use the right arrow button to move them into the Files to be Parsed scrolling list. Go to Step 8.

   For instructions on selecting more than one file, see "Selecting Multiple Files in a List" on page 9-9 for instructions.

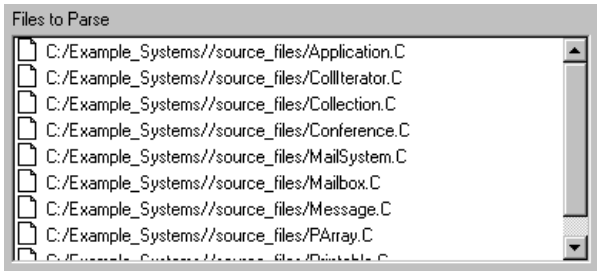
Figure 1: Selecting a Makefile



5. Click the **Makefile Options** button.
6. On the **Makefile Options** dialog box, fill in the makefile options (or accept the defaults) as described in "Using the Makefile Reader" on page 9-13.
7. Click the **Read Selected Makefile** button.

The Makefile Reader reads the makefile, finds the first target and its prerequisites, and displays them in the Files to Parse scrolling list. It also finds the user include and system include directories to be searched during preprocessing, and displays them in the **Preprocessor Options** dialog box.

Figure 2: Filled Files to Parse Scrolling List



8. To run the parser in incremental mode, select Incremental Mode.
   For more information, see "Using Incremental Mode" on page 9-11.
9. To parse files with Microsoft extensions, ensure that the Use Microsoft Extensions option is selected.
   See "Using Microsoft Extensions" on page 9-12.
10. To change the default preprocessor options, click **Preprocessor Options**.
    The **Parser Preprocessor Options** dialog box appears.
11. Fill in the preprocessor options (or accept the defaults) as described in "Using the Parser Preprocessor Options Dialog Box" on page 9-16.
12. Click **OK** or **Apply** to run the parser.

Processing time depends on the amount of code being parsed and whether or not the code is being preprocessed. To improve the speed of the parser, see "Increasing the Speed of the Parser" on page 9-21.

If you encounter problems when parsing files, see "Common Parsing Problems" on page 9-22. After the source files parse without error, you can execute any of the other Reverse Engineering commands.

Selecting Multiple Files in a List

When you use the **Parse Source Code** dialog box, you can select all or a range of files in the Files to Parse or Directory Listing windows.

To select all or a group of the files in a list:

1. Scroll to the top of the list, and select the first source file.
2. Scroll to the last file in the list or to the last file in the group of files that you wish to select.
3. While pressing **Shift**, use the left mouse button to select the last file.

To add selections to or delete selections from a list, press **Control** while selecting or deseleting a file.

Deciding Which Files to Parse

When you choose the source files to be parsed, you determine the content of the diagrams produced by StP. You can parse an individual source file, several selected source files, or all source files used to build the executables that form an entire software system.

Processing several or all executables at once allows you to create a model for a particular part or all of the software system. Generally, you should parse as a group all related executables that share common data definitions in include (header) files. This action causes identical information for different executables to be diagrammed only once, rather than reproduced in identical diagrams and definitions for each executable.

The source files to be parsed can reside anywhere on the file system. They are not necessarily all in the same directory. Although you can parse files that are incomplete or have syntax errors, you may receive warning messages.

If you inadvertently omit necessary source files from the list of source code files to parse, the generated model may be incomplete.

StP does not extract the source code from version control systems. If your code resides within a version control system, you must manually extract readable copies of the source into a directory where StP can process them.

If you choose to run StP incrementally with such code, make sure that only changed files are copied over their previous instances.

Mixing Languages

When you create or update a model, you can mix languages. For example, you can parse C++, Java, and IDL source code files at once, or you can add information from Java source code files to a model that was created from C++ source code files.

C and C++ Suffixes

The parser recognizes the following suffixes for C and C++ files:

- .C
- .h
- .hh
- .cc
- .def
- .cpp

If you want to parse a C or C++ source file that does not have one of these suffixes, you must enter the full list of suffixes into the *RE_CXX_suffixes* ToolInfo variable. For example, to parse files with the '.cxx" suffix, add this line to your *ToolInfo* file:

```
RE_CXX_suffixes=.cxx:.C:.h:.hh:.cc:.def:.cpp
```

In addition to adding these suffixes to your ToolInfo variable, you may need to add them to the Source File Pattern list in the **Makefile Options** dialog box. See "Using the Makefile Options Dialog" on page 9-13 for additional information.

The following suffixes are not allowed for C++. If you enter any of these suffixes into the list in your ToolInfo variable, the parser issues an error message at startup:

- .idl
- .java
- .jav

Java Suffixes

When parsing Java files, the parser recognizes the .java suffix only. If you want to parse a Java file with a different suffix, you must enter the full list of suffixes (including the .java suffix) into the *RE_Java_suffixes* ToolInfo variable.

Since the parser cannot read Java .class files, you can use it on source code files only.

Microsoft Extensions have no effect on Java parsing.

IDL Suffixes

When parsing IDL files, the parser recognizes the .idl suffix only. If you want to parse an IDL file with a different suffix, you must enter the full list of suffixes (including the .idl suffix) into the *RE_IDL_suffixes* ToolInfo variable.

Using Incremental Mode

Select **Incremental Mode** to parse only those specified files that have been modified since the last time they were parsed. The parser uses the file system modification date stamp to determine when the files were last modified.

When you select **Incremental Mode**, a file is parsed only if one of the following apply:

- Has been modified since last being parsed
- Failed to pass prior parsing correctly
- Includes an #include file that has been re-parsed incrementally
- Has just been added to the source file list

Generally, you should use Incremental Mode for parsing files except when:

- Many files have changed
- A frequently included file has changed

In these cases, parsing in non-incremental mode is usually faster. However, the results of parsing incrementally and non-incrementally are the same.

Using Microsoft Extensions

If you select the Use Microsoft Extensions option, it enables the parser to parse C++ files containing these extended reserved words:

- __asm
- __based
- __cdecl
- __declspec
- __except
- __fastcall
- __finally
- __fortran
- __inline
- __int16
- __int32
- __int64
- __int8
- __leave
- __multiple_inheritance
- __oldcall
- __pascal
- __single_inheritance
- __stdcall
- __syscall
- __try
- __unaligned
- __virtual_inheritance

Each of the extensions listed above includes a double underscore. For compatibility with previous releases of Microsoft's compilers, a single underscore is also allowed in these extensions.

Microsoft extensions have no effect on the parsing of Java source code files.

To parse Microsoft IDL (MIDL) source code files, you must enable Microsoft Extensions. When parsing Microsoft IDL (MIDL) source code files, the parser ignores all parts of declarations that are contained within square brackets ([]).

Using the Makefile Reader

The Reverse Engineering tool provides a feature called the Makefile Reader, which automatically finds the files needed to model your software program using the information contained in a Nmake, GNU, or UNIX makefile.

The Makefile Reader uses the same rules as the UNIX **make**, Microsoft **Nmake**, or Free Software Foundation (FSF) GNU **make** command to read the makefile and determine all the prerequisites for the primary target or any subordinate targets specified in the **Makefile Options** dialog box.

It then displays the:

- Prerequisite files in the Files to Parse window on the **Parse Source Code** dialog box
- Directories containing user-defined and system-defined header files in the User Include Search Directories and System Include Search Directories scrolling list on the **Preprocessor Options** dialog box

You can preprocess and parse the files as displayed or edit the information before parsing.

Using the Makefile Options Dialog

If you wish to change the default makefile options, you use the **Makefile Options** dialog box. These options are described in Table 3.

Table 3: Makefile Options Dialog Box Specifications

Specification	Description
Makefile type	Identifies the type of makefile you are using (Generic, Nmake, or Gnumake).
Overwrite current settings	Clears the existing files from the Files to Parse window and all fields on the Preprocessor Options dialog. If you do not select this option, the Makefile Reader concatenates new information with existing information.
Make Depth Bound	Number of directory levels that the Makefile Reader searches to find prerequisite files. For more information, see “Reading Recursive Makes” on page 9-15.
Make Targets	Subordinate targets for the makefile. For more information, see “Selecting Subordinate Targets” on page 9-15.
Source File Pattern	Pattern that identifies a source file contained in the compiler command. Changing this pattern affects the Makefile Reader only; it does not affect the suffixes that the parser accepts. If a source file has an odd suffix, add it to this list to ensure that the Makefile Reader extracts the correct source files from the Makefile. See “C and C++ Suffixes” on page 9-10 for additional information.
Compiler Command Pattern (case insensitive)	Pattern that identifies the name of the executable that runs the compiler.
Compiler Flags for Search Directories	Flags that identify search directories for header files that are included in the source files with #include directives.
Compiler Flags for System Search Directories	Flags that identify search directories for system-defined header files that are included in the source files with #include directives.
Compiler Flags for Preprocessor Definitions	Flags that identify preprocessor definitions.

Table 3: Makefile Options Dialog Box Specifications (Continued)

Specification	Description
Exclude files matching	Pattern that identifies files that you want the Makefile reader to ignore. For example, to exclude yacc generated files from the Makefile run, type *yy*.c in this field.

Reading Recursive Makes

The Makefile Reader can trace prerequisite files through a hierarchy of directories when **make** is invoked recursively. By default, there is no limit on the number of directory levels the Makefile Reader searches; however, you can specify a limit using the Depth Bound option. Select 1, 2, 3, 4, or 5 directory levels, or use the default setting of No Limit.

Selecting Subordinate Targets

By default, the MakeFile Reader finds the files needed to model the primary target or executable identified in the makefile. You can override the default by specifying one or more subordinate targets in the Make Target text entry field. Whenever this field has a value, the MakeFile Reader generates prerequisites for the specified targets only.

A subordinate target can be any object in the makefile that has a dependency line. If you specify more than one target, make sure to insert a blank space between each name.

In addition to specifying targets, you can also define macros in the Make Targets field. Type macros in the form <NAME> = <value>. For example:

```
OBJECTS="db_ant.o ant_files.o ant_main.o ant_ids.o"
```

or

```
BIN=$(HOME)/bin
```

Using the Parser Preprocessor Options Dialog Box

You use the **Parser Preprocessor Options** dialog box to change the default preprocessor options.

Both C++ and IDL files are preprocessed; Java files are not preprocessed. However, you still need to provide the locations of the user and system source files for Java. For details, see "Specifying Include File Search Directories" on page 9-17. The **Parser Preprocessor Options** dialog box is described in Table 4.

Table 4: Parser Preprocessor Options Dialog Box Specifications

Specification	Description	For Details, See
No preprocessing	If selected, Reverse Engineering parses files without preprocessing them. Parsing is faster without preprocessing. Default is for preprocessing to occur.	"Increasing the Speed of the Parser" on page 9-21
Undefine Platform-Specific Defines	If selected, Reverse Engineering ignores the standard define directives.	
User Include Search Directories	Directories containing user-defined header files	"Specifying Include File Search Directories" on page 9-17
System Include Search Directories	Directories containing system-defined header files	
Defines	Preprocessor flags that direct the compiler to create a semantic model for the exact version of the software built by the compiler.	"Specifying Compiler Directives as Defines" on page 9-18

Specifying Include File Search Directories

When you click the Read Makefile button on the **Parse Source Code** dialog box, the Makefile Reader fills the User Include Search Directories and System Include Search Directories scrolling lists.

Generally, you should specify directories as absolute paths, although you can specify relative pathnames or a combination of both types. Relative paths are evaluated from the current working directory from which StP was started.

The specified directories are searched in top to bottom order.

If you wish, you can change the directories provided. You can include as many of the include file search directories as you want.

Specifying Include File Search Directories for C++

For C++, user include files contain the **#include ""** statement, and system include files contain the **#include <>** statement

On Windows NT, you do not need to specify any files contained in the INCLUDE and Include environment variables, because StP automatically adds their paths to the end of the System Include Search Directories list.

Specifying Include File Search Directories for Java

To treat a package as a system file, put its path in the System Include Search Directories window.

To parse the standard Java class libraries, put their path in the System Include Search Directories window. For example, if the source files are in */java/src*, then enter */**java/src** in the System Include Search Directories window. Note that this action slows down parsing.

Specifying Include File Search Directories for IDL

For IDL, there is only one **#include** notation, and there is no concept of system files. Put all search directories in the User Include Search Directories window.

Specifying Compiler Directives as Defines

If C++ or IDL source code was compiled with preprocessor flags specifying the compilation of a particular version of the software, you may want to specify compiler directives for parsing the code. Specifying the directives tells the Reverse Engineering code parser to create a semantic model for the exact version of the software built by the compiler. These flags are usually passed to the compiler from the makefile, which holds all the preprocessor flags needed to build each version of the software.

To specify the preprocessor flags, enter a string containing the necessary compiler directives, separated by spaces, into the Defines field. This creates a file called *Defines* in the workspace directory. If the directive includes spaces or punctuation, enclose it within double quotes. For example:

```
VERSION="6.0 Beta" MACHINE=SUN DEBUG
```

Normally, you do not need to use the Defines field. Enter compiler directives only if necessary.

Parsing Source Code Containing Embedded Code

If the dialect of source code used in your organization supports the embedding of SQL, assembler, or other languages, you can configure the Reverse Engineering parser to search for and ignore the embedded code. This action allows you to reverse engineer the original source code without special preprocessing that would create extra code you may not want in your source code.

To tell the Reverse Engineering tool to ignore embedded code, you specify in a file the starting and ending tokens that surround the code to be ignored. The tokens should be used by the embedded language to signify sections of embedded code, such as declaration sections and SQL commands. The Reverse Engineering parser then looks for the user-specified tokens in the source code and ignores all code found between the matching starting and ending tokens.

For example, when embedded in C++ code, some dialects of SQL use the token EXEC to indicate the beginning of an embedded SQL command, and a semicolon (;) to indicate the end of the embedded command. By specifying EXEC as the start token and a semicolon as the end token, you instruct the Reverse Engineering tool to search for and ignore each embedded SQL command.

To specify tokens, you add them to a file that the *re_token_file* ToolInfo variable points to. You can set this variable in the master ToolInfo file so the parser uses the specified tokens whenever anyone parses code in any StP system. See *Customizing StP* for more information on ToolInfo variables.

Token Search Patterns

The token strings you choose and the order in which you enter them in the file affect the results of the parser search. Precedence of start tokens is by the order in which the token pairs occur in the file.

For example, suppose your token definition file contains these token specifications:

```
"EXEC SQL BEGIN DECLARE SECTION;" "EXEC SQL END DECLARE SECTION;"
"EXEC" ";"
```

When parsing the following SQL code embedded in C++ code, the Reverse Engineering parser first finds and ignores all of the SQL declarations in the declaration section; then it finds and ignores the embedded SQL command.

```
some C++ code here...
EXEC SQL BEGIN DECLARE SECTION;
     some SQL declarations here...
EXEC SQL END DECLARE SECTION;
more C++ code here...
EXEC embedded SQL command;
additional C++ code here...
```

However, suppose you entered the tokens in the file in the reverse order:

```
"EXEC" ";"
"EXEC SQL BEGIN DECLARE SECTION;" "EXEC SQL END DECLARE SECTION;"
```

In this case, the Reverse Engineering parser finds the first EXEC token at the beginning of the `BEGIN DECLARE SECTION` in the preceding code and ignores code only until it encounters the semicolon token at the end of the `BEGIN DECLARE SECTION` on the same line. It then stops ignoring code and tries to parse the SQL declarations between the first end token and the next EXEC token in the `EXEC SQL END DECLARE SECTION`.

Likewise, the Reverse Engineering parser would not be able to identify a nested token pattern, such as this:

```
some C++ code here...

EXEC embedded SQL command
     EXEC embedded SQL command;
     more SQL code
     ;

more C++ code here...
```

The parser would ignore all code between the first EXEC token and the first semicolon (;) end token it finds, thus ignoring the nested EXEC as well. Having ignored the second EXEC token, the parser would not look for its paired semicolon end token either. Thus, the parser would try to parse the SQL code between the two semicolons.

The preceding examples show how important it is to choose appropriate tokens and enter them in the token file in an appropriate order, so the parser parses the source code correctly, while ignoring the embedded language code.

Specifying the Embedded Tokens

You can specify as many pairs of tokens as needed to identify all of the embedded code to be ignored.

Token matching is case sensitive. User-specified tokens must exactly match the tokens used by the embedded language, or the parser cannot identify the code to ignore.

To specify the start and end tokens:

1. Create a file (for instance, *embedded_tokens*) to contain the token specifications.
2. Add a new line to the file for each pair of start and end tokens you want the Reverse Engineering parser to use.

   Enclose each token within double quotes and separate the start token from the end token with a single space. For example:

   ```
   "EXEC SQL BEGIN DECLARE SECTION;" "EXEC SQL END DECLARE SECTION;"
   "EXEC" ";"
   ```

3. In your ToolInfo file, set the *re_token_file* ToolInfo variable to point to the file containing the token specifications (see *Customizing StP*).

   For example, if your specifications are in the *embedded_tokens* file, set the value of the *re_token_file* ToolInfo variable as follows:

   ```
   re_token_file=/usr/stp/template/embedded_tokens
   ```

When you run the Reverse Engineering parser, it looks for the file and the tokens you have specified.

Increasing the Speed of the Parser

If you wish to increase the speed of the parser, you can:

- Parse header files only
- Switch off preprocessing

By parsing header files only, you can increase the speed of the parser by ten to fifteen percent. Since the majority of the information required for building a model is extracted from header files, you can eliminate source files from parsing without losing a significant amount of information. However, by parsing header files only, you cannot include operation bodies during model generation (see "Generating a Model from Parsed Files" on page 9-30).

Additionally, associations that represent a local variable of another class type that are declared within an operation body of another class do not appear on the generated diagrams. These associations can be created only if the source code containing the operation definitions is parsed.

You can also speed up the parser by not preprocessing the files to be parsed; however, if your code is complex, it may be essential to preprocess it.

To increase the likelihood of successful parsing without preprocessing:

- Add *.h to the Source File Patterns text entry field on the Makefile Options dialog.
  This action causes the Makefile Reader to place all header files associated with the specified target in the Files to Parse scrolling list on the **Makefile Options** dialog box.
- Change the order of the files in the Files to Parse scrolling list so that header files that are depended on by other files are parsed first (before the dependent files).

Terminating the Parser

If you need to abort the parser for any reason, press Ctrl+C with the cursor in the StP Desktop Execution Window. After a few second's delay while saving its current state, the parser terminates normally.

Note: Never terminate the parser by killing the process directly, as this can corrupt the semantic model. For more information, see "Checking the Semantic Model Consistency" on page 9-24.

Common Parsing Problems

You will have problems when parsing source code if:

- There are locks on the semantic model
- Some files fail preprocessing
- The parser cannot locate required source or header files
- Case and spelling of source filenames is incorrect
- Directory of source files is not specified as a path
- Paths to #include<> files are specified incorrectly

Locks on the Semantic Model

If there are read or write locks on the semantic model, the parser cannot update the database. For details about getting information on lock status, see "Checking Semantic Model Locks" on page 9-23.

Preprocessing Problems

By default, files are preprocessed before they are parsed. Sometimes preprocessing produces the problem that prevents the files from parsing correctly. To see the preprocessed source code, look in the *revc_files/code_dir* directory in the current system. A copy of all files that fail parsing is left in this directory.

Missing Source and Header Files

If the parser cannot find the required source files, error messages appear in the StP Desktop Execution window. If it cannot find the required header files, warning messages appear.

To correct these problems:

1. Specify the locations of the missing files in the **Parser Preprocessor Options** dialog box.

   See "Using the Parser Preprocessor Options Dialog Box" on page 9-16 for information about this dialog.

2. Reparse the files.

If only source files are missing, you can reparse incrementally. If both source and header files were missing, you must reparse in non-incremental mode.

Checking Semantic Model Locks

Reverse Engineering uses locking to prevent users from simultaneously performing operations on the same data in the semantic model database. For example, you cannot have two parsers executing at the same time, updating the same semantic model.

Due to the massive number of records stored in the semantic model and the large numbers of contiguous records that are removed and replaced during parsing, it is not appropriate to lock individual records. Instead, complete tables are locked exclusively by all Reverse Engineering processes.

Semantic model locks do not affect users of the StP repository, which is always available to multiple readers and writers.

To find out the lock status of a semantic model:

1. On the StP Desktop, click on the **Code** menu.
2. Choose **Reverse Engineering > Check RE Semantic Model Locks**.

The command removes any redundant locks and reports active locks by listing the name of the person and the machine holding the lock, as well as the time the lock was applied.

Checking the Semantic Model Consistency

It is possible for the semantic model's indexes to become corrupted if a process that is writing it is killed before completing the task. Also, if the data on the disk is corrupted due to a disk crash, for example, the data fields in the semantic model may be invalid or out of range.

If you suspect that the semantic model has been corrupted, check it using the following procedure**:**

1. On the StP Desktop, click on the **Code** menu.
2. Choose **Reverse Engineering > Check RE Semantic Model Consistency**.

When completed, the command displays a summary line stating whether the semantic model can be used or if it has been corrupted. If it is corrupted, you should destroy the semantic model by rerunning the parser on all the source files in non-incremental mode. This action removes the current semantic model and rebuilds an entirely new one.

Extracting Comments

Comments represent an important source of information that can enhance the final design documentation. The source code parser alone does not sufficiently handle comments; it stores comments, but it does not associate them with classes and functions. To place source code comments appropriately in your semantic model, you must run the source code comment extractor.

The comment extractor tries to associate source code comments with data structures, functions, and global variables by using search parameters that you provide. It then concatenates the comments and writes them to the semantic model. The model generator uses the comments in the semantic model to create annotations for diagrams in the graphical model.

The comment extraction phase is optional. If you do not extract comments from the source code, they do not appear as annotations in your model.

Using the Extract Source Code Comments Dialog Box

The **Extract Source Code Comments** dialog box enables you to specify search parameters for finding and extracting comments from source code.

You specify comment extraction parameters by setting non-exclusive options for each of the source code object categories in the dialog box:

- Class Comment settings—options for extracting comments related to classes
- Member Comment settings—options for extracting comments declared within a class
- Function Comment settings—options for extracting comments related to function definitions (including operation definitions)

These options are described in Table 5.

Table 5: Extract Source Code Comments Dialog Box Options

Option	Description	For Details, See
Above Definition	Extracts comments found above the definition of the source code object.	"Comment Position Settings" on page 9-27
Below Definition	Extracts comments found below the definition of the source code object.	
Right of Definition	Extracts comments found to the right of the definition of the source code object.	
Class Gap Member Gap Function Gap	Specifies within how many non-comment lines of a source code object a comment must occur, in order to be associated with that object. Legal values are 1 to 4.	"Gap Settings" on page 9-27
Use Comment Delimiters	Allows you to specify start and end delimiters (as strings) for extraction of formatted comments.	
Copy Text Exactly Left Justify Text Fully Justify Text	Provides options for formatting comment text.	"Formatting Options" on page 9-28
Line Length	Maximum line length allowed in reformatted comment text. (Available only with **Fully Justify Text** option).	

Extracting Comments from Source Code

To extract source code comments from your source code:

1. On the StP Desktop, click on the **Code** menu.
2. Choose **Reverse Engineering > Extract Source Code Comments**.

3. In the **Extract Source Code Comments** dialog box, make selections for the source code object categories and fill in the fields, as needed (for details, see Table 5).
4. Click **OK** or **Apply**.
   This action runs the comment extractor and automatically saves the current dialog box option settings.

Comment Position Settings

The position settings indicate where you think the comments for that object are most likely to occur within the source code (above, below, or to the right of the object's definition in the code).

The position specifications are cumulative. If you click all three position options for a comment (above, below, and to the right of an object definition), the comment extractor extracts all comments for that object that appear within the specified gap, no matter where they are positioned relative to the definition.

For example, by selecting options on the **Extract Source Code Comments** dialog box, you can direct the comment extractor to:

- Look for all comments positioned above a member definition, as well as all comments positioned to the right of the definition.
- Ignore any comments found if there is a gap of more than one blank line between the comment and the object definition.

Gap Settings

The **Gap Setting** option determines the number of non-comment lines (containing either code or white space) that constitute a legal gap between a comment and its associated object. For example, a comment separated from a function by two lines would be associated with that function only if the gap setting is two lines or more; a gap setting of one line would not associate the comment with this function. If you set too large a gap, you may extract irrelevant comments.

The gap setting determines spacing for comments both below and above the target object, as indicated by the position setting. Different gaps can be set for classes, members, and functions.

The gap for functions is measured from the line on which the function appears. For example, the gap is measured from the second line in the following code fragment:

```
int
main( int argc, char **argv )
```

The gap for named classes is measured from the line containing the name; for unnamed classes, the gap is measured from the line containing the curly bracket ({) that occurs after the class declaration.

In the following example, the gap for the stock_item class is measured from the second line (which contains the class's name, *stock_item*):

```
class
       stock_item
       {
            int code;
```

In the next example, the gap for the unnamed class is measured from the second line, which contains the curly bracket:

```
class
       {
            int code;
```

Formatting Options

You can select one of the following formatting options:

- Copy Text Exactly—Copies comment text exactly as it appears in the code, without reformatting.
- Left Justify Text—Reformats comments by left-justifying the text with a ragged right margin.
- Fully Justify Text—Reformats comments by justifying both left and right margins.

  If you select this option, enter the maximum line length allowed in the Line Length field. This number determines where line breaks occur.

Evaluating and Improving Comment Extraction

Unless you wrote the source code being parsed, you do not know where comments will be found. You may need to experiment with the Extract Source Comment option settings to see what combination yields the best results.

Each time you run the comment extractor, StP displays diagnostic statistics in the StP Desktop Execution Window. These statistics indicate the percentage of user-defined classes and functions with associated comments, given your search specifications.

If the percentages are not satisfactory, you can rerun the comment extractor with new options to increase the percentage of comments found. Comment extraction replaces any comments in the semantic model with the latest ones found.

The percentage of comments is based only on the classes, members, and functions found in the source files and in header files that are included using the `#include "`*`filename.h`*`"` notation. Comments extracted from system files are not included in the diagnostic calculation.

You can also use the comment extraction program to check comment usage standards within an organization. For example, if every function should have a comment above its definition, make sure that you achieve a 100% rate of extracting comments for functions.

Detection of Inappropriate Language in Comments

If inappropriate language appears in any of the comments, Reverse Engineering replaces it with a random collection of #@!&* characters and informs you of the number of words it replaced.

Generating a Model from Parsed Files

The **Generate Model from Parsed Source Files** command creates a graphical model from the semantic model produced by the parser.

The generated model is stored in the repository for the current StP project and system. When you generate a model where a particular class already exists, you can either:

- Regenerate the class
- Retain user modifications

Generating a Model

To generate a model:

1. On the StP Desktop, click on the **Code** menu.
2. Choose **Reverse Engineering > Generate Model from Parsed Source Files**.
3. On the **Generate Model from Parsed Source Files** dialog box, adjust values or accept the defaults, as desired (for details, see Table 6).
4. On the **Filter Source Information** dialog box, indicate those files and classes you want in the model and those you want to leave out (for details, see Table 10 on page 9-36).
5. To generate the model, click **OK** or **Apply**.
   Progress and results display in the StP Desktop Message Log.

Generate Model from Parsed Source Files Dialog Box

The **Generate Model from Parsed Source Files** dialog box enables you to specify the information to be generated. Table 6 describes the functions of each command property.

Table 6: Generate Model Dialog Box Options

Property	Description
Annotate Classes as External	If selected, annotates all captured classes as external. The default is to be selected.
Include Operation Bodies	Allows you to include operation bodies. (This option has no effect on IDL, since it does not have system files or operation bodies.) The default is to be selected.
Ignore Standard Library Definitions.	If selected, does not generate code for standard library definitions. The default is to be selected. If you define a filter, this option is ignored.
Generate Repository	Synchronizes information about the reverse engineered classes with information in the StP repository. The default is to be selected.
If Class Already Exists:	If you select ReGenerate, you regenerate an existing class. The default is to be selected. If you select Retain User Modifications, you regenerate existing class tables and retain user modifications. If you do not select Retain User Modifications, you regenerate existing class tables but lose user modifications. The default is Retain User Modifications. For more information, see "Retaining User Modifications if a Class Exists."
Filter Source Information	Clicking this button summons the **Filter the Model** dialog box, which allows you to indicate those classes you want in the model and those you want to leave out.
Generate	Provides options for generating diagrams, tables, and annotations. The default is to generate all three objects; to change the default, deselect one or more objects.

Retaining User Modifications if a Class Exists

The following information pertains to using the **If Class Already Exists** option of the **Generate Model from Parsed Source Files** dialog box.

If a class exists, and you do not select the **Regenerate** option, the existing class table, diagram, and annotations remain as they are before you used the **Generate Model from Parsed Source Files** command.

If you do select **Regenerate**, and do not select **Retain User Modifications**:

- The **Generate...** command completely rewrites the class table and removes any added rows that are not in the source code. The command also blanks out any extra columns. For more information, see Table 7.
- The command completely rewrites the class diagram. Therefore, any modifications to all the existing <classname>_RE_ [AGGR | ASSOC | NESTED | INH] diagrams will be lost.
- The command updates annotations, or creates them if they do not exist.

If you select both **Regenerate** and **Retain User Modifications**:

- The command updates the class table. Any user modifications to places that were not created by the reverse engineering tool will still exist. If you have modified parts of the table that the reverse engineering tool has created, these will be overwritten by the correct values in the source code.
- The command creates a class diagram if it does not already exist. If the diagram does exist, it is neither recreated nor updated.
- The command updates annotations, or creates them if they do not exist.

See Table 7 for information on UML Uclasst columns populated for attributes.

Table 7: Columns Populated for Attributes

Language	Column Numbers
Mandatory for all languages	1, 2, 4, 5
C++ source code	8, 9, 10
Java source code	3, 16, 17, 18, 19

Table 7: Columns Populated for Attributes

Language	Column Numbers
IDL source code	20 (if RE_update_all_idl TI is set), 21
TOOL source code	24, 25, 26

See Table 8 for information on UML Uclasst columns populated for operations.

Table 8: Columns Populated for Operations

Language	Column Numbers
Mandatory for all languages	1, 2, 4, 5, 6, 7
C++ source code	8, 9, 10, 11, 12
Java source code	16, 17, 18, 19
IDL source code	Both 20 + 21 (if RE_update_all_idl TI is set), 22, 23
TOOL source code	24, 25, 26, 27, 28

Table 9 shows rows for attributes and operations, and their corresponding names.

Table 9: Attribute and Operation Column Names

Column	Attribute Column Name	Operation Column Name
1	attribue	operation
2	type	arguments

Table 9: Attribute and Operation Column Names

Column	Attribute Column Name	Operation Column Name
3	default value	return type
4	visibility	visibility
5	class attr?	class op?
6	derived?	abstract?
7		throws
8	const?	const?
9	visibility	visibility
10	volatile?	virtual?
11		inline?
12		ctor init list
16	visibility	visibility
17	final?	final?
18	transient?	native?
19	volatile?	synchronized?
20	type (IDL specific)	arguments
21	read only?	return type
22		attribute
23		context
24	privilege	privilege
25	set expr	return event
26	get expr	exception event
27		copy return?
28		kind?

Filtering Source Information

Before you generate a model from parsed source files, you can indicate those files and specific classes you want in the model generation and those you want to leave out. You do this by constructing filters using the **Filter the Model** dialog box.

Before you construct any filters, system files / classes (those included with #include <> notation) appear in the Do not include pane of the dialog box. User files / classes (those included with #include "" notation) appear in the Include pane. Source files that do not contain classes will not appear in the panes.

You include or exclude files / classes by using the -> and <- arrows to move them from one pane to the other.

Moving a file from one pane to another affects all the classes defined in that file. Moving a group of classes by moving the file that contains them acts as a shortcut feature. However, you can include a file in a filter but not include one or more of its classes. Ultimately, only what is in the list of classes to include gets put into the model.

When you create a filter, you can save it as the default filter for anyone who has not specified a personal filter. Or, you can save it as your personal filter. This filter will then appear for you each time you return to the dialog box. (If you are working on a part of a large model, this feature allows you to update only your part.) If you have not saved a personal filter, the dialog box will load the default filter. If no default filter has been saved, the dialog box will appear in its initial state.

To filter source information:

1. On the **Generate Model from Parsed Source Files** dialog box, click on the **Filter Source Information** button.
2. Designate files to be included or excluded in the filter by moving them from one pane to another.
3. When the file selection is correct, click on the **Classes** option button to do the same with specific classes.
4. Click on the **All Users** option button to designate the filter as the default or click on the **Only Me** option button to designate the filter as your personal filter.
5. Click **OK** to save your filter.

Table 10: Filter the Model Dialog Box Options

Property	Description
Files	Selecting this option button allows you to see the files to be included or left out of the model generation.
Classes	Selecting this option button allows you to see specific classes to be included or left out of the model generation.
Do not include: / Include:	Files / classes in the Do not include window will be left out of model generation. Those in the include window will be a part of model generation.
Reset to Defaults	Clicking this button resets the windows to their initial setting (before the creation of any filters).
Save Filter for:	Clicking the All Users option button designates this filter as the default filter everyone will use who has not specified a personal filter. Clicking the Only Me option button designates this filter as your personal filter.
Delete my Filter File	Clicking on this button deletes your personal filter and returns you to the All Users filter.
Reset	Clicking this button returns the windows to their settings before the current session's changes.

Generating a Model from TOOL Code

Generating a model from TOOL code requires that you choose only the **Generate Model from TOOL Code** command. You do not need the command series, **Parse Source Code**, **Extract Source Code Comments**, and **Generate Model from Parsed Source Files**.

Defining Forté Environment Variables

Before you can generate a model from TOOL code you must first define two environment variables:

- the Forté repository name variable
- the Forté workspace variable

Defining the Forté Repository Name and Workspace Variables

To define the Forté repository name and workspace variables:

1. Choose **My Computer** > **Control Panel** > **System** >**Environment** tab.
2. In the Variable field, type ***FORTE_REPOSNAME***.
3. In the Value field, type <**repository name**>**.**
4. Click **Set**.
5. Return to the Variable field and type ***FORTE_WORKSPACE***.
6. Return to the Value field and type <**workspace name**>.
7. Click **OK**.

Generating a Model

To generate a model from TOOL code:

1. On the StP Desktop, click on the **Code** menu.
2. Choose **Reverse Engineering > Generate Model from TOOL Code**.
3. In the **Class Capture** dialog box, select the name of the project you want to import into StP.

   An Execution window appears listing the generated class diagrams, class tables, and annotations.

Class Diagrams Created by Reverse Engineering

During the reverse engineering process, StP searches for a class's associations, aggregations, generalizations, and nesting relationships. If these relationships exist, StP creates the following class diagrams:

- <*classname*>_ASSOC_RE—Shows association relationships
- <*classname*>_AGGR_RE—Shows aggregation relationships (for reverse engineered C++ source code files only)
- <*classname*>_INH_RE—Shows generalization relationships
- <classname>_NESTED_RE—Shows nesting relationships (not applicable for TOOL source code files)

  When Reverse Engineering finds a nesting relationship, it creates an annotation item, as described in "Annotations Created by Reverse Engineering" on page 9-50.

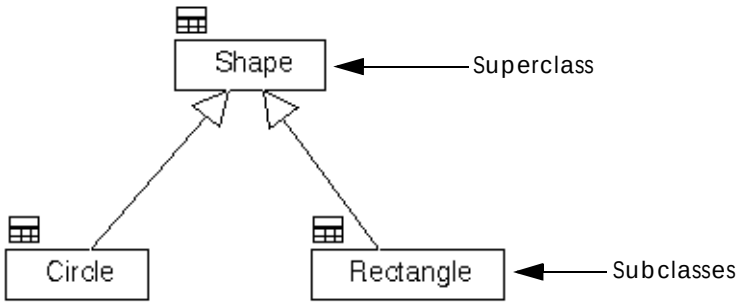
When you use Reverse Engineering, StP places generalization, association, and aggregation diagrams in the *uclassd_files* subdirectory of the current System / Project directory and includes them in the list of files you can load into the Class Editor.

If the reverse engineering class is a Window class, StP places files with the *.fsw* suffix in the *src_files* subdirectory.

Generated Generalization Diagrams

StP creates a class diagram for each distinct generalization hierarchy that exists in the classes captured with Reverse Engineering. Each generated diagram is a simple drawing using orthogonal arcs of the subclasses inheriting from the superclass, as shown in Figure 3.

Figure 3: Generalization Diagram



Inheritance information is placed in the repository as an annotation specified on the inheritance link joining the subclass and the superclass.

Reverse Engineering can create a generalization diagram even if you capture only a subclass. For example, if you capture just the *Circle* class in Figure 3, StP creates a file named *Shape_INH.ome* (*Shape_INH.uclassd* for UML) showing *Circle* and *Shape* in an inheritance relationship. If you then capture just *Rectangle*, the *Shape_INH.ome* (or *Shape_INH.uclassd*) file is enhanced to show both subclasses. However, the *Shape* class attributes and operations are not defined in the repository until you capture the class.

Generated Association Diagrams

StP creates association diagrams when:

- An attribute points to another class (as in UsesB3 in the example code shown below)
- An attribute is the return type of a member function or a parameter of a member function (as in the second constructor method for A)
- A local variable of another class type is declared within any member function (as in A::A in the example code below)

Association diagrams created by Reverse Engineering show role names and multiplicity, as described in:

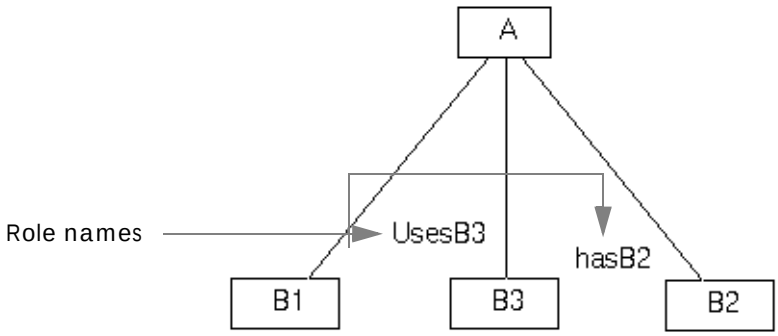
- "Roles in Reverse Engineering Diagrams" on page 9-43
- "Multiplicity in Reverse Engineering Diagrams" on page 9-43.

Association Diagram Example

The following C++ code was used to produce the association diagram shown in Figure 4:

```
class B1
{
};
class B2
{
};
class B3
{
};
class A
{
  public:
    A();
    A(B1 &usesB1);

  private:
    B3 *UsesB3;
    B2  hasB2;

};

A::A()
{
   B2 usesB2;
}
```

Figure 4: Association Diagram



Generated Aggregation Diagrams

StP creates aggregation diagrams from C++ and TOOL source code files; it does not create aggregation diagrams from IDL or Java files. StP creates aggregation diagrams when:

- A class contains attributes that are instances of another class (if the attribute is a pointer to an object, then the relationship is considered an association).
- An attribute is typed as being an array of elements, each of which is another class.
- An attribute's type is a recursive pointer to the class in which it is defined, as shown in the following C++ code example:

```
class IntList
{
    IntList *next;
    int      data;
};
```

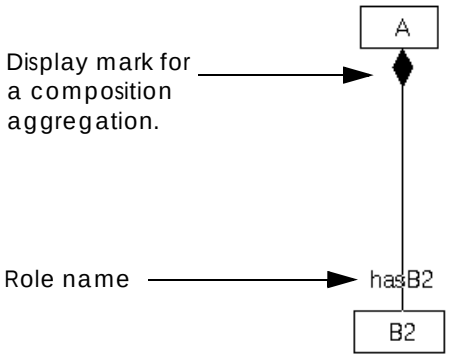
StP/UML supports both aggregations and composition aggregations. Reverse Engineering can create both types of aggregations. Composition aggregations appear as a solid diamond; aggregations appear as a hollow diamond. For information about aggregations and composition aggregations, see *Creating UML Models.*

Aggregation diagrams created by Reverse Engineering show role names and multiplicity, as described in:

- "Roles in Reverse Engineering Diagrams" on page 9-43
- "Multiplicity in Reverse Engineering Diagrams" on page 9-43.

The code shown in "Generated Association Diagrams" on page 9-39 was used to create the aggregation diagram shown in Figure 5.

Figure 5: Aggregation Diagram with Role Name



Generated Nested Diagrams

Reverse Engineering creates a nested diagram when it finds:

- C++ namespaces
- Java packages
- IDL modules

In StP/UML, when a class is nested within another class, Reverse Engineering does not create a diagram, because this action breaks the UML syntax checks.

Instead, Reverse Engineering represents these relationships by using the Enclosing Scope item of the Class Definition note, described in "Annotations Created by Reverse Engineering" on page 9-50.

When creating nested diagrams, Reverse Engineering:

- Draws C++ namespaces, Java packages, and IDL modules as packages in class diagrams

- Completes the Enclosing Scope item of the Class Definition note
- Does not create class tables for these classes

Roles in Reverse Engineering Diagrams

Role names automatically appear on association and aggregation diagrams created by Reverse Engineering. The role name is taken from an attribute name, where the attribute is mapped into an aggregation or association.

Role names are placed in the repository as contexts on links.

If you do not want to show role names on your diagram, you can:

- Use a filter to hide them from view, as described in *Fundamentals of StP*
- Select and then delete them from the diagram

Figure 5 on page 9-42 shows a role name on an aggregation diagram created by Reverse Engineering. For a more detailed example, see "Example of Roles and Multiplicity" on page 9-44.

Multiplicity in Reverse Engineering Diagrams

Multiplicity display marks automatically appear on association and aggregation diagrams created by Reverse Engineering if the multiplicity display mark is set for display. For information about setting display marks, see *Fundamentals of StP*.

Information about multiplicity is placed in the repository as an annotation on the link.

Rules for Determining Multiplicity

Reverse Engineering uses the following rules to determine multiplicity for aggregations:

- An aggregation that is an array (but not an array of pointers) is created with Many multiplicity

- An aggregation created from a recursive pointer is created with Optional Multiplicity
- All other aggregations are created as Exactly One multiplicity

All associations are created with the default multiplicity (Exactly One), unless the association is a pointer to an array or a double pointer (or triple pointer or more). These associations are created as Many multiplicity.

If Reverse Engineering finds more than one identical association or aggregation within a class, it merges their multiplicity. The basic rules for merging are:

- Many + <anything else> = Many
- One or More + Exactly One = Many
- One or More + One or More = One or More
- One or More + Numerically Specified = Many
- Numerically Specified + Optional = Many
- Numerically Specified + Exactly One = Many
- Optional + Exactly One = One or More
- Exactly One + Exactly One = Numerically Specified (with a value of 2)

Additionally, if one role name has less than three characters and the other has three or more characters, Reverse Engineering uses the longer name.

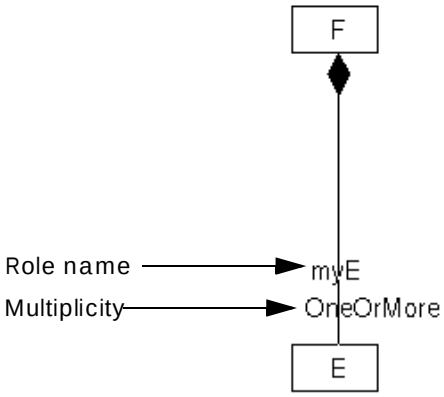
Example of Roles and Multiplicity

The following code was used to create the aggregation diagram in Figure 6, which shows both role names and multiplicity:

```
class E
{
};
class F
{
   E myE[100];
};
```

Notice that the role name, *myE*, appears on the end of the link opposite the class from which the role name was extracted (*F*).

Figure 6: Aggregation Diagram with Role Name and Multiplicity



Interfaces in Reverse Engineering Diagrams

Reverse Engineering generates interfaces from TOOL and Java source code files. Interfaces appear as class symbols with a stereotype of «Interface» on StP / UML association, generalization, and nested diagrams. For each class, the stereotype item of the Extensibility Definition note is set to interface.

ViewPoints in Reverse Engineering Diagrams

When Reverse Engineering creates an StP / UML class diagram, it attaches the ViewPoints to nodes on diagrams, as described in Table 11. Since ViewPoints are invisible, they do not alter the appearance of the diagram.

For additional information about nodes, see *Object Management System*. For additional information about ViewPoints, see *Creating UML Models*.

Table 11: Viewpoints on Reverse Engineered Diagrams

Viewpoint	Attached to:
All Aggregation	All nodes on <classname>_AGGR_RE diagrams

Table 11: Viewpoints on Reverse Engineered Diagrams

Viewpoint	Attached to:
All Generalization	All nodes on <classname>_INH_RE diagrams
All Association	Root node of any <classname>_ASSOC_RE diagrams
Package Containment	All nodes, except the root node, on <classname>_NESTED_RE diagrams

Class Tables Created by Reverse Engineering

StP creates a class table for each class that is created during the reverse engineering process. Table 12 and Table 13 describe the attributes and operations provided by Reverse Engineering for each class.

Table 12: Class Table Attribute Information Provided by RE

Class Table Section	Attribute Information Item	Value
Class Members	Attribute	Attribute name
	Type	Attribute type
	Default Value	Initialization (Java only)
Analysis Items (These attributes are provided for classes created from C++ and Java source code files only.)	Visibility	(Public, Private, or Protected)
	Class Attr?	True / False Set to True if the attribute is static

Table 12: Class Table Attribute Information Provided by RE

Class Table Section	Attribute Information Item	Value
C++ Items	Const?	True / False
	Visibility	C++ visibility
	Volatile?	Public / Protected / Private
Java Items	Visibility	Public / Protected / Private Java default visibility is private.
	Final?	True / False
	Transient?	
	Volatile?	
IDL Items	Type	Not provided when the class table is initially generated from Reverse Engineering
	Read Only?	True / False
TOOL Items	Privilege	Public / Private
	Set Expression	The expression to set the attribute's value.
	Get Expression	The expression to get the attribute's value.

Table 13: Class Table Operation Information Provided by RE

Class Table Section	Operation Information Item	Value
Class Members	Operation	Operation name
	Arguments	Operation arguments
	Return type	Operation return type
Analysis Items (These items are provided for classes created from C++ and Java source code files only.)	Visibility	Public / Protected / Private
	Class Op?	True / False Set to True if the operation is static
	Abstract?	True / False Set to True if the operation has abstract in its declaration
	Throws	Exceptions that this function throws
C++ Items	Const?	True / False
	Visiblity	Public / Protected / Private
	Virtual?	True / False
	Inline?	True / False
Java Items	Visibility	Public / Protected / Private Java default visibility is private
	Final?	True / False
	Native?	True / False
	Synchronized?	True / False
IDL Items	Argument	Operation arguments
	Return Type	Operation return type

Table 13: Class Table Operation Information Provided by RE

Class Table Section	Operation Information Item	Value
TOOL Items	Privilege	Public/Private
	Return Event	The method's return event, if any.
	Exception Event	The Method's exception event, if any.
	Copy Return?	True/False
	Kind	Method, Event, or Event Handler

Class Table Example

Given the following C++ class definition:

```
class Shape
{
public:
   Shape(int x, int y);
   virtual void draw() = 0;
   virtual void erase() = 0;
protected:
   int locx, locy;
};
```

Reverse Engineering creates the UML class table for the Shape class, as shown in Figure 7. (Note that the table in Figure 7 shows only partial views of the Analysis Items and C++ items for the *Shape* class.)

Figure 7: Class Table from Reverse Engineering

Shape			Analysis Items		C++ Items		
Attribute	Type	Default Value	Visibility	Class Attr?	Const?	Visibility	Volatile?
locx	int		protected	False	False	protected	False
locy	int		protected	False	False	protected	False
Operation	Arguments	Return Type	Visibility	Class Op?	Const?	Visibility	Virtual?
Shape	x:int, y:int		public	False	False	public	False
draw		void	public	False	False	public	True
erase		void	public	False	False	public	True

Annotations Created by Reverse Engineering

During the reverse engineering process, when StP finds the appropriate information, it creates annotations for each class. Table 14 shows the annotations created for StP / UML.

Table 14: StP/UML Annotations

Note	Item	Description	Created for:
Object	Appears in Note Description dialog	Contains source code comments for a class, attribute, or operation.	C++, Java, IDL
C++ Declarations		Contains locally defined typedefs, friend classes, or friend functions.	C++
		Contains source code for operations.	
IDL Declarations		Contains typedefs declared in the IDL interface.	IDL
Class Definition	Enclosing Scope	Indicates nested classes. This item contains the name of the parent class that the annotation class is nested in.	C++
		Indicates the package in which the class resides.	Java
		Indicates the module in which the class resides.	IDL
	Abstract Class	Indicates that a class is abstract.	C++, Java
	Parameters	Contains formal template arguments of a parameterized class.	C++

Table 14: StP/UML Annotations (Continued)

Note	Item	Description	Created for:
Class TOOL Implementation	Shared Property	Properties captured from TOOL source code	TOOL
	Subclass Override Shared Property		
	Distributed Property		
	Subclass Override Distributed Property		
	Monitored Property		
	Subclass Override Monitored Property		
	Transactional Property		
	Subclass Override Transactional Property		
	Restricted		
	Window Class Filename		
	Interface Stereotype		
Class Java Definition	Final	Indicates that the class was declared “final” when implemented in Java.	Java
	Visibility	Indicates Java visibility (public, private, protected). If visibility is set in the Class Definition, this setting overrides it.	
C++ Inheritance Definition	Inheritance is Virtual	Indicates if the C++ inheritance is virtual.	C++
	Inheritance Visibility	Indicates if the inheritance is public, protected, or private.	
Extensibility Definition	Stereotype	Specifies a stereotype name for an object.	C++, Java, IDL

Table 14: StP/UML Annotations (Continued)

Note	Item	Description	Created for:
Role Definition	Aggregation Type	Indicates if the relationship is an aggregation or a composition aggregation.	C++
	Multiplicity	See "Multiplicity in Reverse Engineering Diagrams" on page 9-43.	

Checking on Generated Annotations

To check on annotations you have generated from TOOL code:

1. In the Model pane on the StP Desktop, open the **Tables** category and select the **Class** component type.
2. Select a class table from the objects pane.
3. From the CTE **Edit** menu, choose **Cell Annotation**.
4. In the Object Annotation Editor, click on the object.
5. Choose **Class TOOL Implementation** from the note options list in the Set / Add area.
   In the item field of the Set / Add area is a list of the items captured.

Updating a Class from the Editors

You can use the Reverse Engineering commands to update a class or create a generalization hierarchy from the Class Editor or Class Table Editor. These commands, which are described in Table 15, are greyed out except when a semantic model exists.

Table 15: RE Commands for Updating Classes

Command	Editor	Description
Construct From Reverse Engineering	Class Diagram Editor	Gets the definition of the selected class from the source code files and updates the attributes and operation for the selected class. See "Updating from the Class Diagram Editor" on page 9-54.
Create Subclass Hierarchy from Reverse Engineering		Builds a generalization hierarchy consisting of classes beneath the selected class. See "Creating a Generalization Hierarchy" on page 9-55.
Create Superclass Hierarchy from Reverse Engineering		Builds a generalization hierarchy consisting of classes that the selected class inherits from. See "Creating a Generalization Hierarchy" on page 9-55.
Create Whole Hierarchy from Reverse Engineering		Builds a generalization hierarchy consisting of classes above and below the selected class. See "Creating a Generalization Hierarchy" on page 9-55.

Updating from the Class Diagram Editor

When you update a class from the Class Diagram Editor, StP refreshes the drawing, but does not build a complete class definition. Use the Class Table Editor to build a complete class definition.

To update a class from the Class Editor:

1. Select a class on your class diagram.
2. From the **UML** menu, choose **Attributes and Operations > Construct from Reverse Engineering**.

   The class's attributes and operations appear in the class.

Creating a Generalization Hierarchy

If you created a semantic model, you can use the following commands to create a generalization hierarchy from the Class Editor, using one of the **Create...Hierarchy** commands described in Table 15 on page 9-54.

To construct a generalization hierarchy from the Class Editor:

1. Select a class on your class diagram.
   The selected class must be the parent class in the generalization.
2. Choose **UML > Generalization Hierarchy**.
   Note: If you perform this procedure following the very first time you have used the Reverse Engineering tool, the Generalization Hierarchy item may not appear on the **UML** menu. In this case, shutdown and restart the Class Editor. The Generalization Hierarchy item will then appear on the menu.
3. Choose **Create Subclass Hierarchy from Reverse Engineering**, **Create Superclass Hierarchy from Reverse Engineering**, or **Create Whole Hierarchy from Reverse Engineering**.
   StP draws the generalization hierarchy for the selected class.

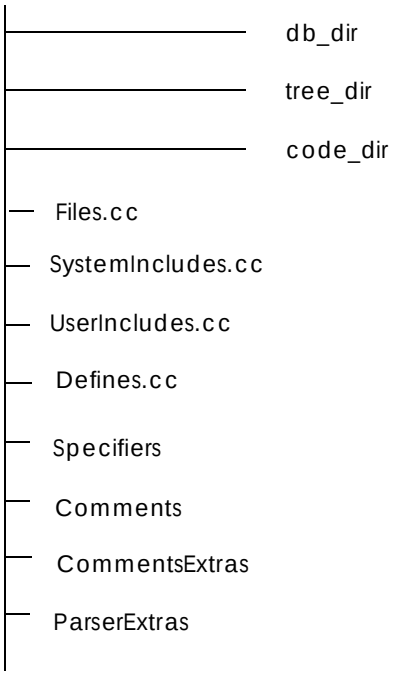
   Note: When you use these commands, Reverse Engineering removes and recreates the selected symbol. When it removes the symbol, it also removes any links attached to the symbol.

Reverse Engineering Directory Structure (C++)

The */revc_files* directory for the current project and system contains the reverse engineering files, as shown in Figure 8. Diagrams and annotations created by StP / UML are stored in the repository.

Figure 8: Reverse Engineering Directory Contents

/<project>/<system>/revc_files Directory



The db_dir Directory

The *db_dir* directory holds the semantic model database files for the current project and system. Lock files are text files and can be viewed; they show the owner and mode of each lock held on the file.

The tree_dir Directory

The *tree_dir* directory holds ascii files that correspond to a syntactic and semantic parse tree for every function defined in the source code. You should never need to access this directory.

The code_dir Directory

The *code_dir* directory holds temporary copies of each source file as it is being preprocessed and parsed by Reverse Engineering. If a source file fails parsing, a copy of the fully expanded file can be found in this directory. This file can be useful if you need to determine why a file will not parse.

Files.cc File

The *Files* file contains the name of each source file that is part of this system.

SystemIncludes.cc File

The *SystemIncludes* file holds the paths for system-included files. These files are included with the `#include <...>` notation. Paths can be either absolute or relative to the current project and system. The default path */usr/include* is added automatically to the end of this list.

UserIncludes.cc File

The *UserIncludes* file holds the paths for user-included files. These files are included with the `#include "...."` notation. Paths can be either absolute or relative.

Defines.cc File

The *Defines* file contains each of the defines entered in the Defines field in the Parse Source Code dialog box. The spaces separating the defines in the dialog box are converted to newline characters in the *Defines* file.

Comments File

The *Comments* file holds an encoded description of the comment template for this system. It is read and written by the comment gathering process and is not meant to be accessed manually.

CommentsExtras File

The *CommentsExtras* file holds the information saved from the **Extract Source Code Comments** dialog box.

ParserExtras File

The *ParserExtras* file holds the information saved from the **Makefile Options** dialog box.

10 Navigating with StP

You can navigate from StP/UML to C++, IDL, and Java source code files for a selected object using the Microsoft Developer's Studio programming environment, Windows Notepad, or the default StP editor.

This chapter describes:

Navigating from StP to Source Code

Navigating from StP to source code first involves setting a default editor in which the code will appear, then choosing the proper source code navigation command.

Setting the Default Editor

By default, when you navigate from StP to source code, the code displays in the default StP editor, where you can view it but you cannot edit it. However, by setting the *source_editor* ToolInfo variable, you can view and edit source code from:

- Microsoft Developer's Studio
- Windows Notepad

To set up the default editor for the source code navigation, enter one of the lines shown in Table 1 in your ToolInfo file.

Table 1: Setting up the Source Code Navigation

To view the code from:	Add to your ToolInfo file:
StP code viewer	This is the default; you do not need to add anything to your ToolInfo file.
Microsoft Developer's Studio (Visual C++ 5.0 or 6.0, Visual J++ 6.0)	`source_editor=MSDev`
Windows Notepad	`source_editor=notepad [FILE]`

Enter the lines exactly as shown; do not substitute anything for `[FILE]`.

Using the Navigation

When you navigate from StP to source code, use one of the commands shown in Table 2.

Table 2: Source Code Navigation Commands

Command	Use to navigate
Source Code	From a class or an attribute to source code.
Source Code Declaration	From an operation to its declaration.
Source Code Definition	From an operation to its definition. If a semantic model does not exist, the **Source Code Definition** command does not appear on the **Go To** menu.

Navigating to Source Code When a Semantic Model Exists

When you use one of the source code navigation commands listed in Table 2, StP searches for a semantic model in the current system. If StP finds a semantic model and the selected object in the semantic model database, the source code displays in the selected source viewer or editor.

To navigate from StP to source code when a semantic model exists:

1. Select an object in the StP / UML Class Diagram Editor or Class Table Editor.

   See Table 3 on page 10-4 for details.

2. From the **Go To** menu, choose one of the source code navigation commands listed in Table 2.

   The source code appears in the selected source viewer or editor.

Navigating to Source Code When a Semantic Model Does Not Exist

When you use one of the source code navigation commands in Table 2, StP searches for a semantic model. When StP does not find a semantic model in the current system, it displays a window where you can enter the source code's directory.

StP assumes that the source code was created with StP's code generation facility. If the code was not generated with StP, the navigation may not be accurate.

If you do not have Reverse Engineering, you can still use this option to navigate to source code.

To navigate from StP to source code when a semantic model does not exist:

1. Select an object in the StP / UML Class Diagram Editor or Class Table Editor.

   See Table 3 on page 10-4 for details.

2. From the **Go To** menu, choose one of the source code navigation commands listed in Table 2.

   A pop-up window appears.

3. Type the directory of the source code to which you want to navigate.

   The next time you run StP and try to navigate, StP searches for this directory. If it cannot find the source code in this directory, the pop-up window appears so you can enter a new directory.

If you want, you can also provide the name of the source code file; however, if you do, StP uses this file for all subsequent navigations in the current system, until you change it.

4. Click **OK**.
   StP tries to find the source file that matches the class name of the selected object. For example, if you select a class named *Mailbox*, StP tries to find *Mailbox.h* in the specified directory. If StP finds *Mailbox.h*, it navigates to the line where "class Mailbox {" appears and displays the source code in the selected source editor or viewer.

Available Navigations

Table 3 summarizes the navigation possibilities available from StP / UML to source code. See *Creating UML Models* for more information on navigations.

Table 3: Navigating from StP/UML to Source Code

StP/UML Editor	Selected Symbol	Source Code
Class Editor	Class Parameterized Class Instantiated Class	Class definition
	Attribute	Attribute definition
	Operation	Method declaration or definition
	Interface	Interface definition
Class Table Editor	Class name in table cell	Class definition
	Attribute name in table cell	Attribute definition

Index

A

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

W