Getting Started with StP/UML

Click on any Table of Contents entry to jump to the corresponding page.

Preface

Chapter 1 Creating an StP/UML Model

Chapter 2 Beginning the Email Model

Chapter 3 Creating a Class Diagram

Chapter 4 Completing the Class Diagram

Chapter 5 Using the Class Table Editor

Chapter 6 Creating a State Diagram

Chapter 7 Creating an Activity Diagram

Chapter 8 Creating Interaction Diagrams

Chapter 9 Creating Implementation Diagrams

Chapter 10 Generating C++ Code

Preface

This manual presents an introduction to Software through Pictures/Unified Modeling Language (StP/UML) for the first-time user. After completing this manual, you should:

- Be comfortable using the StP/UML user interface
- Be acquainted with many important features of StP/UML
- Understand basic Unified Modeling Language concepts and terminology
- Have created a basic object model
- Know where to find more information about 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 use StP/UML to create object, dynamic, functional, and use case model diagrams.

The information in this manual assumes that you are familiar with:

- Fundamentals of Software through Pictures, including using diagram editors, the Object Annotation Editor, and annotation templates
- Object-oriented modeling, as documented in *Unified Modeling Language User Guide* (Booch et al 1998) and *Unified Modeling Language Reference Manual* (Rumbaugh et al 1998)
- Your operating system directory and file structures
- Conventions of your UNIX window manager or Windows NT

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 uses screenshots from StP for UNIX. If the equivalent Windows NT dialog box or window is substantially different, it is shown as well.
- This manual shows pathnames in UNIX format. Full Windows NT pathnames would include a drive letter at the beginning of each pathname.

Reader Comments

Aonix welcomes your comments about our documentation. If you have any suggestions for improving *Getting Started with StP/UML*, please complete the Reader Comments form included with your StP product documentation and send it to us.

Related Reading

Getting Started with StP/UML 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 2.

Table 2: 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*
Generating C++ code from StP/UML models	*Generating and Reengineering Code*
Printing diagrams, tables, and reports	*Fundamentals of StP*, *Query and Reporting System*
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 SQL Server*
Managing an StP installation	*StP Administration*
Annotation items	Object Annotation Editor Help menu

Table 2: Further Reading (Continued)

1 Creating an StP/UML Model

This tutorial guides you through the creation of an StP/UML model for an electronic mail system. It introduces you to basic tasks involving:

- StP/UML diagram and table editors
- The StP repository
- Diagram checking facilities
- Code generation

The tutorial also introduces basic concepts of the Unified Modeling Language. For a comprehensive introduction to UML modeling and semantics, refer to *Unified Modeling Language User Guide* (Booch et al 1998) and *Unified Modeling Language Reference Manual* (Rumbaugh et al 1998).

A complete version of the Email sample system is included on your StP/UML installation media. You may find it useful to review the system after completing this tutorial.

About StP/UML Models

UML models represent real-world objects and their interrelationships. There are five major, overlapping components of UML models:

- A requirements model—A behavioral view of system requirements interacting with external agents
- A logical model—A conceptual view of the entire domain of activity that is not concerned with physical implementation

- A physical model—A material view of the details of implementation, such as the hardware and software components of a system
- A static model—A descriptive view of the characteristics of parts of the domain of activity
- A dynamic model—A behavioral view of the parts of the domain of activity

StP/UML Diagrams

Each of the five components of a UML model represents a view of the entire model. In StP/UML, each view is represented by one or more of the following types of diagrams:

- Use Case diagram
- Class diagram
- Interaction diagrams (collaboration and sequence)
- State diagram
- Activity diagram
- Implementation diagrams (component and deployment)

Use Case Diagrams

Use case diagrams model the interactions between the system and external agents. They provide an overview of user requirements and system behavior that is helpful in the early phases of the system development life cycle.

Class Diagrams

A class diagram is a graphical representation of the classes in the real-world domain and the relationships between these classes. Class diagrams can have associated class tables, which allow you to capture the full definitions of a class's attributes and operations. Class diagrams and class tables are key components in the logical-static view of a system.

Interaction Diagrams

Interaction diagrams show groups of objects interacting through message passing. The two types of interaction diagrams—sequence and collaboration—show different views of object interaction. Sequence diagrams show a dynamic view; collaboration diagrams show a static view.

Sequence Diagrams

A sequence diagram shows the order of messages sent between objects. It emphasizes control flow and control information such as conditions and iterations. Sequence diagrams complement state diagrams in presenting a logical-dynamic view of a system.

Collaboration Diagrams

A collaboration diagram is a graphical representation of the objects in an interaction and the message pathways between them. It emphasizes message types, concurrency, visibility, and return values. Collaboration diagrams complement class diagrams in the logical-static view of a system.

State Diagrams

A state diagram represents the potential states of an object and the events that transform the object from one state to another. State diagrams can have associated tables, which provide detailed information about individual states. State diagrams and state tables present a logical-dynamic view of the system.

Activity Diagrams

An activity diagram is a variation of a state diagram used to model the procedural flow of a task. Using activity diagrams, you can model the steps required to perform an operation, a use case, or a behavior spanning several use cases. Activity diagrams present a logical-dynamic view of the system.

Implementation Diagrams

Implementation diagrams show the physical design of the system. There are two types of implementation diagram: component diagrams, which show software structure, and deployment diagrams, which show hardware configuration. Component and deployment diagrams present a physical-static view of the system.

Component Diagrams

A component diagram shows the binary, executable, and source code components of the system and the dependencies between these software components.

Deployment Diagrams

A deployment diagram shows the allocation of software to hardware and the interconnections between the hardware components.

Summary of UML Diagrams

Table 1 provides a summary of UML diagram types.

Table 1: StP/UML Diagram Types

Diagram	Purpose	Logical/Physical/ Requirements	Dynamic/ Static
Use Case	Shows interactions between the system and its users	Requirements	Static
Class	Shows classes and their relationships	Logical	Static
Sequence	Shows the order of message passing among interacting objects	Logical	Dynamic
Collaboration	Shows the objects in an interaction and their message pathways	Logical	Dynamic and Static

Table 1: StP/UML Diagram Types (Continued)

Diagram	Purpose	Logical/Physical/ Requirements	Dynamic/ Static
State	Shows the potential states of an object, the events that cause a transition from one state to another, and the actions that result from the transition	Logical	Dynamic
Activity	Traces the execution of a task or method; can show parallel processing	Logical	Dynamic
Component	Shows software structure and dependencies	Physical	Static
Deployment	Shows hardware configuration and dependencies	Physical	Dynamic and Static

Extending UML Models

The Unified Modeling Language includes a standard set of modeling elements such as packages, classes, components, and dependencies. You can extend the semantics of these elements using constraints, tagged values, and stereotypes.

Constraints allow you to restrict the properties of elements in your model; tagged values allow you to add properties. Stereotypes allow you to define new modeling elements based on the existing set.

StP/UML supports all of the Unified Modeling Language's predefined constraints, tagged values, and stereotypes. In addition, StP/UML supports user-defined extensions. You use the Stereotype Editor to define stereotypes and stereotype hierarchies. You use property sheets to define tagged values and constraints.

Generating Code for a UML Model

Using a valid UML class diagram and class table, you can generate C++ code. In this tutorial, you generate C++ code for a class in the Email application.

For detailed information about generating code for UML models, see the *Generating and Reengineering Code* manual.

Continuing the Tutorial

In this chapter, you received an overview of StP/UML. You learned about:

- The components of a UML model: requirements, logical, physical, static, and dynamic
- The types of StP/UML diagrams and tables
- The mechanisms for extending UML modeling elements: constraints, tagged values, and stereotypes
- Generating code from UML models

In Chapter 2, you begin modeling the Email application.

2 Beginning the Email Model

In this chapter, you examine the problem statement provided for the Email application, set up your system directory, and begin your Email model by creating a use case diagram. The use case diagram provides an overview of user requirements and system behavior.

Requirements for the Email Application

This section contains a list of the requirements for the Email application. Review these requirements now. Later, as you create your Email model, you refer to these requirements to determine its classes and relationships.

The requirements for the Email application include:

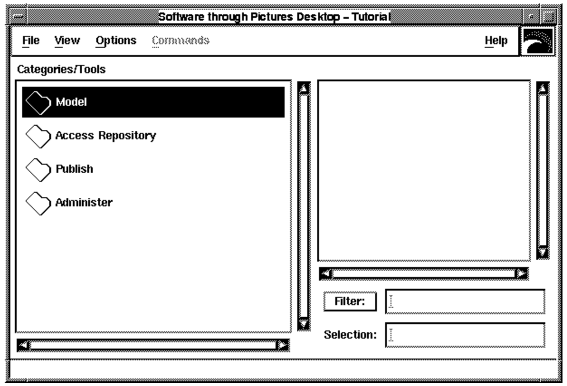
- The Email system must provide an electronic means of sending textual messages to users.
- Each user must have a name and a mailbox that can contain any number of messages.
- When logged in, a user must be able to ask to see the contents of his or her mailbox and, in response, the system should present a list of message subject lines.
- The user must be able to send a message to another user. The Email system must prompt the user for the name of the recipient, a subject line, and the body of the message.
- The Email system must check the address of the recipient. If the address is not valid, the system must display an error message.
- The user must be able to forward a message to another user.

- The Email application must support “conferences,” which allow users to post messages to electronic bulletin boards.
- Conferences must have subject names.
- When logged in, a user must be able to join any conference.
- The logged-in user must be able to ask for a list of users and conferences.

Using the StP Desktop

The Software through Pictures (StP) Desktop, shown in Figure 1, provides access to StP/UML editors and commands, StP Core commands, and StP utilities.

Figure 1: The StP Desktop



In this tutorial, you use the StP Desktop to:

- Start the StP/UML editors
- Generate C++ code for the Email system

For a complete description of the StP Desktop, refer to *Creating UML Models*.

Starting the StP Desktop from Windows NT

To start the StP Desktop from Windows NT:

1. Click on the Start menu.
2. Choose the Programs menu.
3. Choose **StP UML** from the Aonix Software through Pictures menu.

Starting the StP Desktop from UNIX

To start the StP Desktop from UNIX, type the following command on the command line:

```
stp [&]
```

Creating a System for Your Model

Before beginning the exercises in this tutorial, you must create and set a system in which to work. A system contains the database and files associated with your model.

If you do not have the privileges needed to create a system, ask your system administrator to create a system for use with this tutorial and to add you as a repository user.

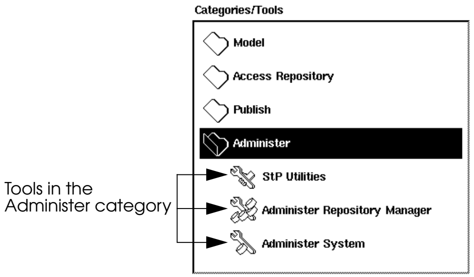
For more information about systems, see *StP Administration*.

Creating a System

To create a system:

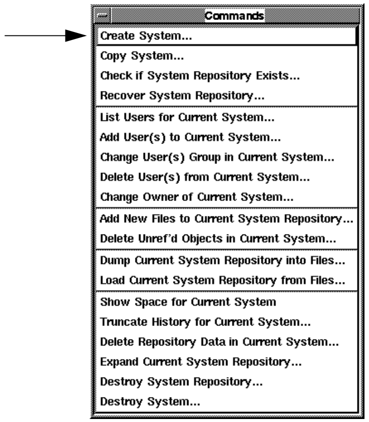
1. Select the Administer category on the StP Desktop.
   A list of tools appears.

Figure 2: Administer Category



2. Select the Administer System tool.
3. Choose **Create System** from the Commands menu.

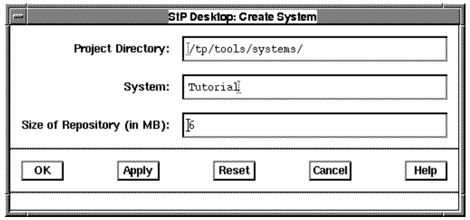
Figure 3: **Create System Command**



The Create System dialog appears.

4. Type `Tutorial` in the System input box, as shown in Figure 4.

Figure 4: **Create System Dialog**



Project Directory: /tp/tools/systems/

System: Tutorial

Size of Repository (in MB): 6

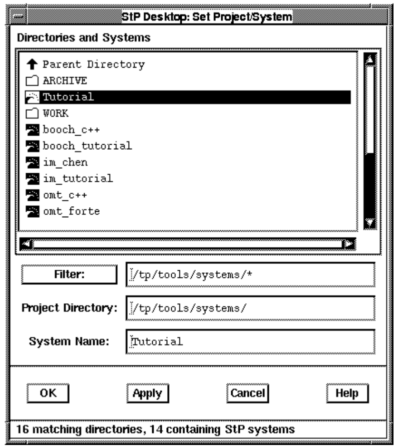
5. Click **OK**.
A new system called *Tutorial* is created.

Setting the System

To set the current system to the *Tutorial* system:

1. Choose **Set Project/System** from the File menu on the StP Desktop.
The Set Project/System dialog appears.

Figure 5: **The Set Project/System Dialog**



Filter: /tp/tools/systems/*

Project Directory: /tp/tools/systems/

System Name: Tutorial

16 matching directories, 14 containing StP systems

2. Select *Tutorial* from the Directories and Systems scrolling list.
3. Click **OK**.
The current system is set to *Tutorial* and the dialog is dismissed.

You are now ready to begin creating your model.

About the Use Case Editor

You begin your model by creating a use case diagram for the Email system. Use case diagrams model the interactions between the system and external agents. By beginning your model with a use case diagram, you gain an understanding of how the system will be used.

You use the Use Case Editor to create the diagram. For a complete description of the Use Case Editor, see *Creating UML Models*.

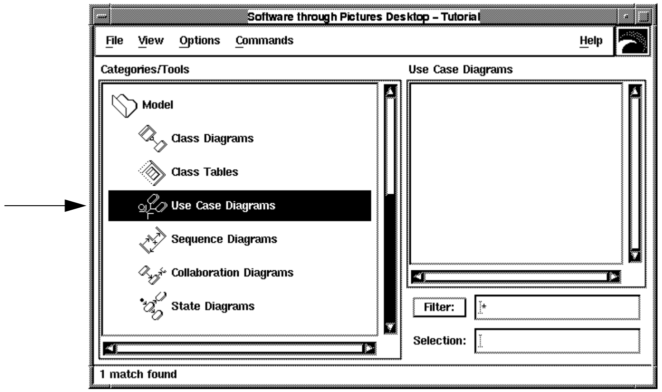
Starting the Use Case Editor

You can start the StP/UML editors from the StP Desktop or, if you are a UNIX user, from the command line. In this tutorial, you start all the editors from the StP Desktop. For instructions on starting an StP/UML editor from the UNIX command line, refer to the chapter that describes the editor in *Creating UML Models*.

To start the Use Case Editor:

1. Select the Model category on the StP Desktop.
   A list of tools appears.
2. Select Use Case Diagrams, as shown in Figure 6.

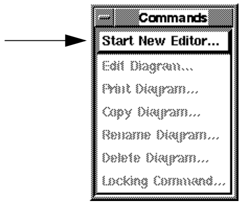
Figure 6: Starting the Use Case Editor



1 match found

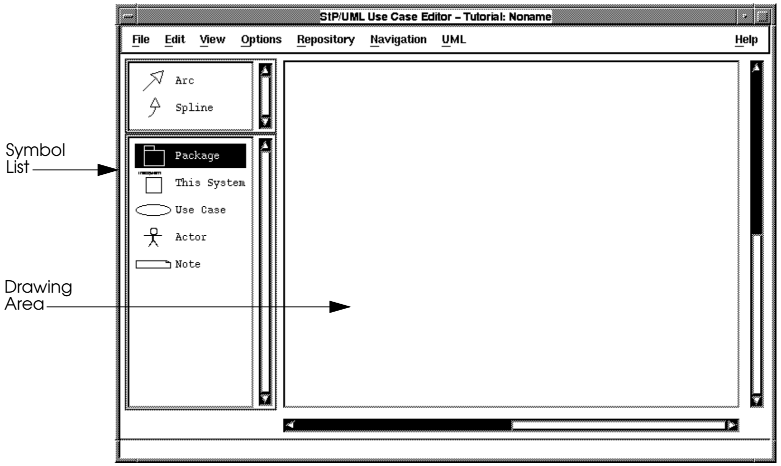
3. Choose **Start New Editor** from the Commands menu.

Figure 7: Start New Editor Command



The Use Case Editor window appears, as shown in Figure 8.

Figure 8: StP Use Case Editor Window



The name of the system and the diagram appear in the title bar in the format *system name: diagram name*. Since you have not yet named the diagram, the term *Noname* appears instead of the diagram name.

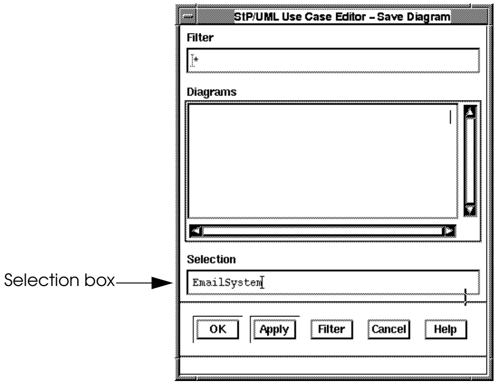
Naming the Diagram

You use the Save Diagram dialog to name and save the diagram.

To name the diagram:

1. Choose **Save As** from the File menu.
   The Save Diagram dialog appears.
2. Type **`EmailSystem`** in the Selection box, as shown in Figure 9.

Figure 9: Naming the Diagram

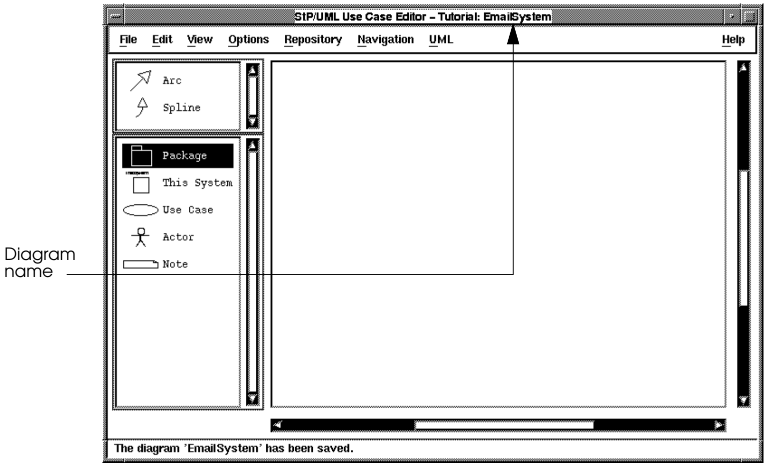


Filter: *

Selection: EmailSystem

3. Click **OK**.
   The name of the diagram, *EmailSystem*, appears in the Use Case Editor title bar, as shown in Figure 10.

Figure 10: Use Case Editor with Diagram Name



The diagram 'EmailSystem' has been saved.

Creating a Use Case Diagram

A use case diagram describes interactions between actors and the system. An actor is an entity outside the system that uses the system to perform various tasks. Each task is represented by a use case. For example, a user (the actor) might use the Email application to send a message (the use case).

By beginning your model with a use case diagram, you gain an understanding of how the Email system will be used. Use case diagrams help you ensure that your model captures the requirements of the application's users.

For this tutorial, you model two use cases:

- Sending a message
- Forwarding a message

In a complete UML model, you would try to identify all possible actors and use cases for the system. Additional use cases in an Email system might include:

- Saving a message
- Deleting a message
- Joining a conference

Inserting Actors

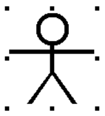
In this example, the actor is the user who is sending or forwarding the message. Begin your use case diagram by inserting the actor *User.*

To insert and label an actor:

1. Drag an actor symbol into the drawing area.

   An actor symbol appears in the drawing area, surrounded by selection handles, as shown in Figure 11.

   Figure 11: An Actor Symbol

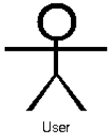
   

2. Type the label `User` on the selected actor symbol.
3. Click the left mouse button to terminate the label editing mode.

   For the remainder of the tutorial, whenever you label a symbol, use the left mouse button to terminate editing.

   Your diagram should resemble Figure 12.

Figure 12: Labeled Actor



Modeling Use Cases

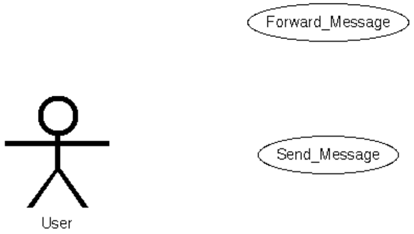
A use case symbol represents one or more ways that an actor can use a system. In this tutorial, you model use cases called *Send_Message* and *Forward_Message*.

To model a use case:

1. Drag a use case symbol from the symbol list into the drawing area, placing it to the right of *User*.
2. Label the use case **`Send_Message`**.
3. Place a second use case symbol above *Send_Message* and label it **`Forward_Message`**.

   Your diagram should resemble Figure 13.

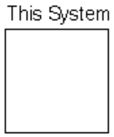
Figure 13: Labeled Use Cases



Modeling the System

The *User* actor is external to the Email application, while the *Send_Message* and *Forward_Message* use cases are functions contained within the application. You can represent the boundaries of the Email system using the This System symbol.

Figure 14: This System Symbol

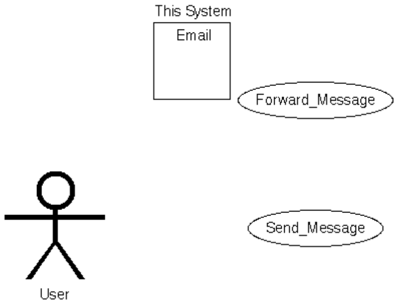


To represent the system:

1. Drag a This System symbol into the drawing area and place it to the left of *Forward_Message*.
2. Label the symbol *Email*.

   Your diagram should resemble Figure 15.

Figure 15: Labeled System Symbol



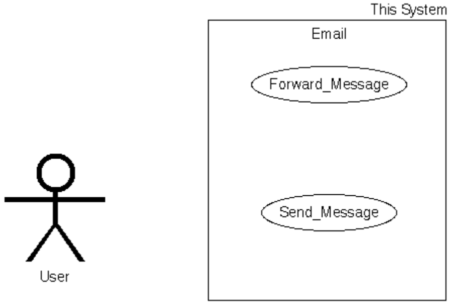
Scaling Symbols

You need to scale the This System symbol so that it encloses the *Forward_Message* and *Send_Message* use cases.

To scale a symbol:

1. Select the This System symbol.
   Selection handles appear on the symbol.
2. Press the left mouse button on the lower right selection handle.
3. Drag the bounding box to the desired size.
4. Release the mouse button.
   Your diagram should resemble Figure 16.

Figure 16: Scaled System Symbol



Modeling Communication

To show that an actor interacts, or “communicates,” with a use case, you draw an arc from the actor to the use case symbol.

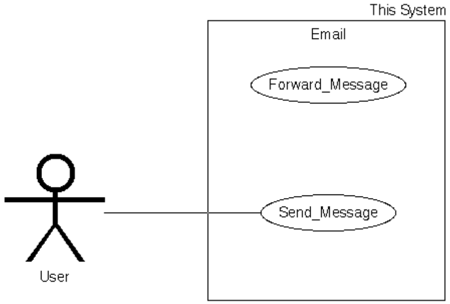
To insert a communication link:

1. Using the left mouse button, drag an Arc symbol into the *User* actor symbol.

---

2. Release the mouse button.
   This action attaches one end of the arc to the actor symbol.
3. Move the pointer into the *Send_Message* use case symbol.
4. Click to attach the opposite end of the arc to the use case symbol.
   Selection handles appear on the arc.
5. Click the left mouse button to dismiss the selection handles.
   Your diagram should resemble Figure 17.

Figure 17: Communication Link



Alternate Method for Inserting Arcs

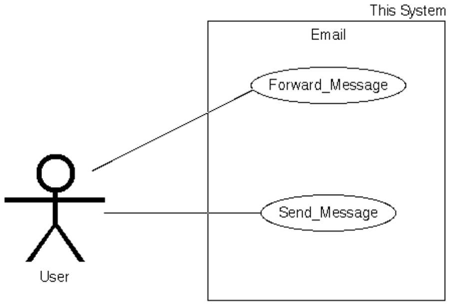
You might find it useful to experiment with a shorter method for inserting arcs.

To insert an arc using the alternate method:

1. Move the pointer to the *User* actor symbol.
   When the pointer is over the symbol, it changes to a circle.
2. While pressing the Control key on your keyboard, click on the *User* actor symbol.

3. Release the Control key.
   This action attaches one end of the arc to the actor symbol.
4. Drag the arc into the *Forward_Message* use case symbol.
5. Click to attach the arc to the use case symbol.
   Your diagram should resemble Figure 18.

Figure 18: Alternate Method for Inserting Links



Modeling Relationships between Use Cases

Two kinds of relationships can exist between use cases:

- “Extends” relationships, in which one use case may, under special circumstances, include the behavior of another use case.
- “Uses” relationships, in which one use case always includes the behavior of another use case.

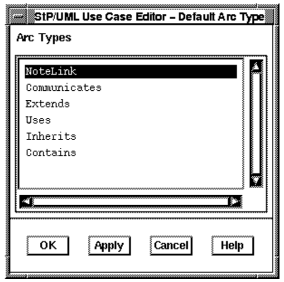
Setting the Default Arc Type

You can select the type of arc you want to draw (uses or extends) using the Default Arc Type dialog. Alternatively, you can change an arc that you have already drawn from one type to another using the **Replace** command. You use both methods in this tutorial.

To set the default arc type:

1. Choose **Default Arc Type** from the Options menu.
   The Default Arc Type dialog appears.

Figure 19: Default Arc Type Dialog



2. Choose **Uses** from the Arc Types list.
   The arc type is highlighted.
3. Click **OK**.

Indicating a “Uses” Relationship

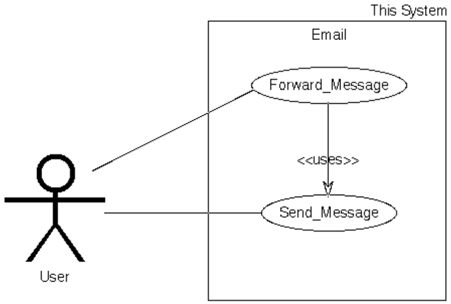
The requirements state that a user must be able to forward an email message to another user. For the system, forwarding a message involves copying an existing message into the body of a new message, highlighting the copied text, and sending the new message.

In sending the new message, *Forward_Message* uses the same behavior as the *Send_Message* use case. Rather than duplicating the specifications for sending an email message, you can indicate that *Forward_Message* “uses” the *Send_Message* behavior.

To model the uses relationship, draw an arc from *Forward_Message* (the client use case) to *Send_Message* (the use case that is used). StP automatically adds a pointer to the target use case and adds the label «uses» to the relationship arc.

Your diagram should resemble Figure 20.

Figure 20: Uses Relationship



The guillemets (« ») surrounding the label indicate that the uses relationship is a variant, or stereotype, of a standard UML modeling element (in this case, a dependency relationship). Stereotypes can be assigned to a number of UML modeling elements to indicate special semantics. You will learn more about stereotypes in a later chapter.

Indicating an "Extends" Relationship

In the Email system, if a user attempts to send a message to an invalid address, the system displays an error message. The error message does not occur in a typical *Send_Message* transaction; it is displayed only when an invalid address is used. You can model this behavior as a separate use case, *Invalid_Address*, that extends the behavior of the *Send_Message* use case in certain circumstances.

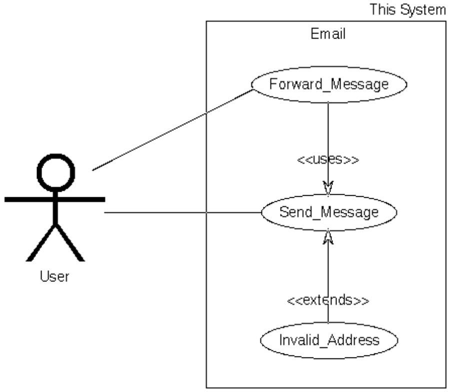
To model an extends relationship:

1. Place a use case symbol below *Send_Message* and label it `Invalid_Address`.
   You may have to scale the system symbol to make room for the new use case.

2. Draw an arc from the *Invalid_Address* use case to the *Send_Message* use case.
   An arc appears with the label «uses».
3. With the arc still selected, choose **Replace** from the Edit menu.
   The label on the arc changes to «extends».

   Your diagram should resemble Figure 21.

Figure 21: Extends Relationship



Saving the Diagram

When you save a diagram (or table, in subsequent chapters), it is committed to the repository, which is the central store of all the information associated with your diagram. By storing diagram and table information in the repository, you make it available to other diagrams and programs such as the StP Query and Reporting System (QRS). The repository also enables checking programs to make sure the elements defined in your model are used properly elsewhere in the system. For more information about the repository, see *Object Management System*.

To save your diagrams and tables, use one of the following methods:

- Choose **Save** from the editor File menu.
- Use the accelerator Ctrl+s.
- Display the File menu and type S, the mnemonic for the **Save** command.

Using one of these methods, save your use case diagram now. For the remainder of the tutorial, whenever you save a diagram, you can use any of these methods. (For a complete description of accelerators and mnemonics, see *Fundamentals of StP*.)

After saving a diagram or table, you can exit the editor and recall the diagram or table at a later time. If you exit an editor without saving your work, any changes made since the last **Save** are lost.

Exiting the Editor

Exit the Use Case Editor by choosing **Exit** from the File menu.

Continuing the Tutorial

In this chapter, you:

- Reviewed the requirements for the Email application
- Started the StP Desktop
- Created and set up your Tutorial system
- Created a use case diagram

In the next chapter, you create a class diagram for the Email system.

---

3 Creating a Class Diagram

In this chapter, you create a class diagram for the Email application.

About the StP/UML Class Editor

The StP/UML Class Editor enables you to draw classes and the relationships between them. In this chapter, you model the classes and relationships for the Email application.

For this tutorial, you identify and model all of the Email application's packages, classes, attributes, operations, and relationships at one time. Typically, this process is an iterative one; you would identify and model classes, attributes, and operations throughout the modeling process.

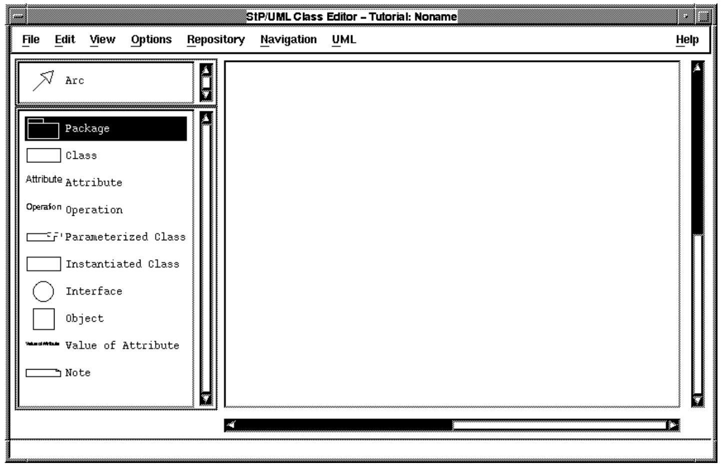
Starting the Class Editor

You start the Class Editor from the StP Desktop.

To start the Class Editor:

1. Select the Model category on the StP Desktop.
   A list of tools is displayed.
2. Select Class Diagrams.
3. Choose **Start New Editor** from the Commands menu.
   The Class Editor appears, as shown in Figure 1.

Figure 1: The Class Editor



Naming the Diagram

Use the Save Diagram dialog to name the diagram *Email*. If necessary, refer to "Naming the Diagram" on page 2-9.

Setting the Default Arc Type

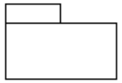
Use the procedure described in "Setting the Default Arc Type" on page 2-17 to set the Class Editor's default arc type to **Association**.

Identifying Class Packages

The first step in creating a UML model is to identify the application's classes. There is more than one way to approach this task. You can identify the application's classes directly using the information available to you in the requirements and in use case diagrams. Another approach is to first determine categories of classes based on the application's architecture or services, and then identify the classes in each category. In a class diagram, you represent this kind of logical grouping as a "package" (the category) with contents (the classes).

A package is a general purpose symbol used in many different StP/UML diagrams to show containment. It is represented by a folder icon.

Figure 2: Package Symbol



In this tutorial, you begin by identifying the Email application's packages. Since this is a simple application, it has only two packages:

- *ReuseLibrary*—Contains classes that come from a reuse library
- *Application*—Contains the classes that are part of the Email application

Although you identify all of the Email application's classes and packages at the beginning of this tutorial, typically this process is iterative. You can identify and model classes and packages at any time during the development of your model.

Modeling Packages

To model packages:

1. Drag a package symbol into the drawing area.
2. Label the symbol **`Application`**.

3. Drag another package symbol into the drawing area, and place it to the right of the *Application* package symbol.
4. Label this symbol `ReuseLibrary`.

   Your diagram should resemble Figure 3.

Figure 3: Email Class Diagram with Packages

Modeling Dependencies Between Packages

The *Application* package relies on the *ReuseLibrary* package for services. You model this dependency by drawing an arc from the package requiring services (*Application*) to the package providing services (*ReuseLibrary*).

The relationship between packages is always a dependency relationship.

To model a dependency between packages:

1. Drag an Arc symbol into the *Application* package and release the mouse button.

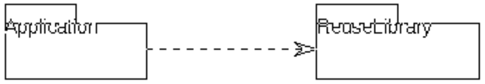
   This action attaches one end of the arc to the *Application* package.
2. Move the pointer into the *ReuseLibrary* package and click the left mouse button.

   This action attaches the opposite end of the arc to the *ReuseLibrary* package. Selection handles appear on the arc.
3. Choose **Replace** from the Edit menu to change the arc from an association to a dependency.

   The arc changes to a dashed line with an arrow pointing to the *ReuseLibrary* package.

   Your diagram should resemble Figure 4.

Figure 4: Packages with Dependency Relationship



Adding Contents to Packages

In the following sections, you add contents to the packages. Package contents can be classes or other packages. The *Application* and *ReuseLibrary* packages contain classes only.

Adding Contents to the ReuseLibrary Package

The *ReuseLibrary* package contains two classes:

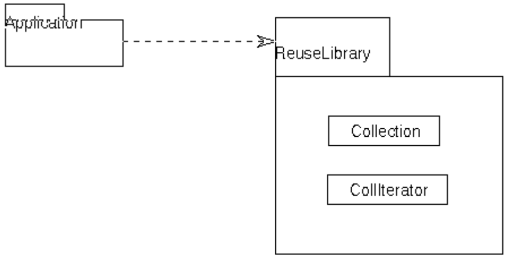
- *Collection*—This class holds a collection of objects. In the Email application, it holds a collection of messages in a user’s mailbox.
- *CollIterator*—This class iterates through a collection of objects. In the Email application, it iterates through a collection of messages in a user’s mailbox in order to print them.

To add contents to the *ReuseLibrary* package:

1. Scale the *ReuseLibrary* package so it can accommodate two classes.
2. Drag a class symbol into the package.
3. Label the class `Collection`.
4. Drag a second class symbol into the package.
5. Label the class `CollIterator`.

   Your diagram should resemble Figure 5.

Figure 5: ReuseLibrary package with Contents



6. Choose **Save** from the File menu to save your diagram.

Adding Contents to the Application Package

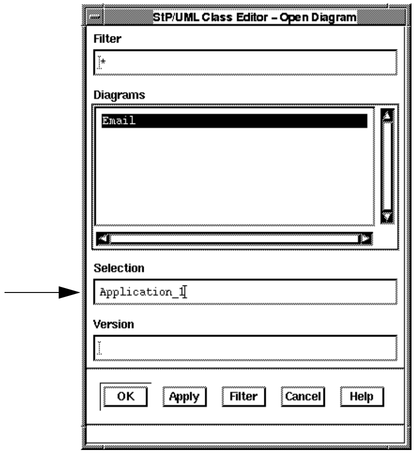
The *Application* package contains all of the classes that are part of the Email application. To avoid crowding the *Email* diagram, you draw the contents of the *Application* package on a separate class diagram.

Creating a New Diagram

To create a new diagram from within the Class Editor:

1. Choose **Open** from the File menu.
   The Open Diagram dialog box appears.
2. Type `Application_1` in the Selection text field, as shown in Figure 6.

Figure 6: Open Diagram Dialog


StP/UML Class Editor – Open Diagram

Filter: *

Diagrams: Email

Selection: Application_1

Version:

3. Click **OK**.
   The dialog is dismissed and a new class diagram named *Application_1* appears in the editor window.
4. Choose **Save** from the File menu.
   You are now ready to begin creating your class diagram.

Modeling Classes

The *Application* package contains all of the classes that are part of the Email application. Try to identify the classes in this package by reviewing the requirements in Chapter 2. If you need help, refer to the following list:

- Conference
- Mailbox

- MailSystem
- Message
- Receiver
- User

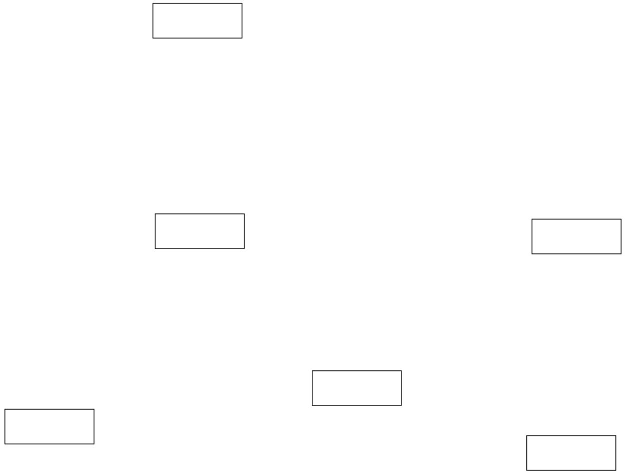
Inserting Class Symbols

Begin the class diagram by inserting a symbol for each class in the *Application* package.

To model classes:

1. Select the Class symbol in the Symbol List.
   The Class symbol is highlighted.
2. Move the pointer into the drawing area.
3. While pressing Alt+Ctrl, click the left mouse button.
   This accelerator enables you to insert the currently selected symbol (in this case, a class symbol) in your diagram.
4. Insert five additional class symbols, placing them so that your diagram resembles Figure 7.
   A total of six class symbols should appear in the drawing area. You may need to enlarge your drawing area to fit all the symbols. For instructions, refer to the following section, "Using Zoom."

Figure 7: Inserting Classes



Using Zoom

Use the Zoom features to enlarge your view of the drawing area and provide more space for inserting objects. The Zoom features allow you to enlarge the drawing area by a specific percentage or to zoom up or down by set increments.

To use Zoom:

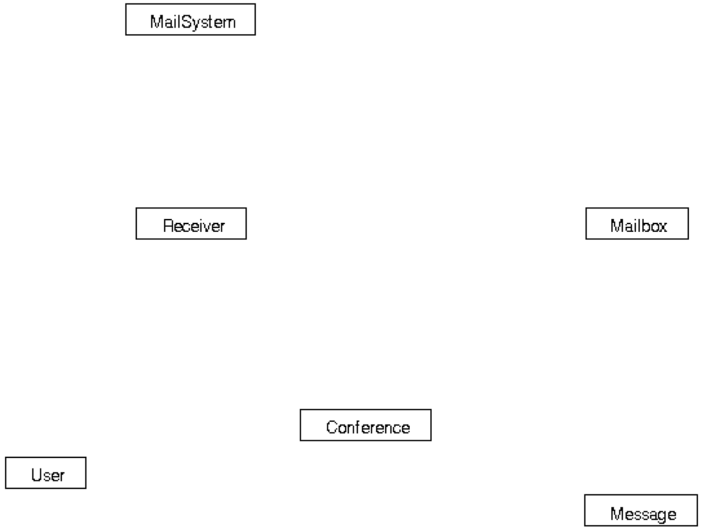
1. Display the View menu.
2. Drag the pointer to **Zoom** and across to the arrow to display its cascade menu.
3. Choose a selection from the Zoom cascade menu to enlarge the view of the drawing area.

Labeling Classes

To label class symbols:

1. Select the topmost class symbol on your diagram and type `MailSystem`.
2. Label the remaining class symbols so that your diagram resembles Figure 8.

Figure 8: Labeled Class Symbols



Modeling Attributes of Classes

Attributes represent the data held by objects in a class.

For example, each message must have a subject line and a body. Therefore, *subject* and *body* are modeled as attributes of the *Message* class.

Try to determine the attributes of the Email application’s classes from the requirements. If you need help, refer to Table 1.

Table 1: Email Model Class Attributes

Class	Attributes
MailSystem	conferences users
Receiver	name mail_system mailbox current_sender
Mailbox	name messages
User	name mail_system mailbox current_receiver
Conference	subject
Message	body subject from to

Inserting Class Attributes

There are three methods for inserting class attributes:

- Inserting attribute symbols using the Symbol List
- Using the Alt+Ctrl+a keyboard accelerator
- Using the Class Table Editor (CTE)

The instructions in this chapter tell you how to insert attributes using the Symbol List. If you prefer, you can use the keyboard accelerator. For more information about using keyboard accelerators, see *Fundamentals of StP*.

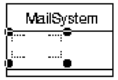
In Chapter 5, you use the CTE to add additional details about the Email application's attributes.

Inserting Attributes Using the Symbol List

To insert attributes on your diagram using the Symbol List:

1. Drag an attribute symbol from the symbol list into the *MailSystem* class.
   The class symbol resizes to fit the attribute. Selection handles appear on the attribute symbol.

   Figure 9: Attribute Symbol with Selection Handles

2. Label the attribute symbol `conferences`.
   The MailSystem class should resemble Figure 10.

   Figure 10: Labeled Attribute

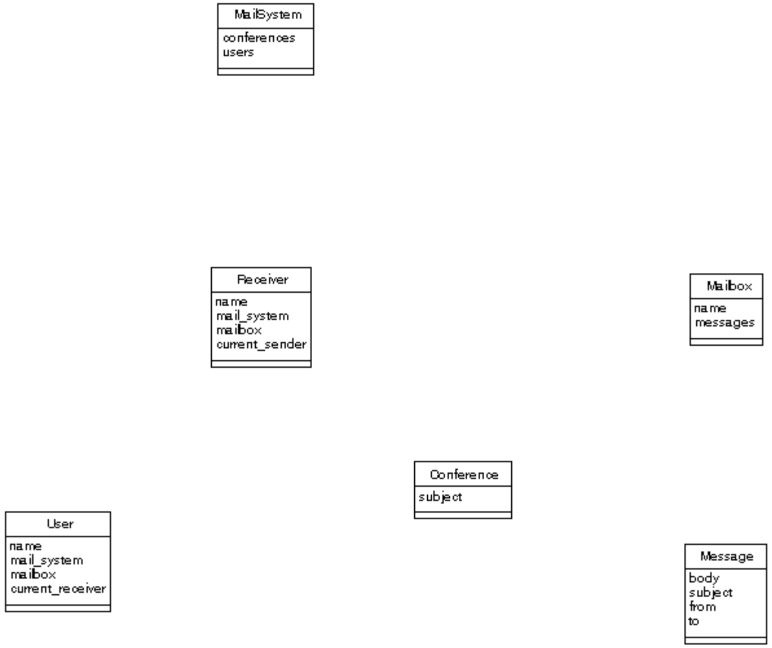
3. Drag another attribute symbol into the MailSystem class.

   The attribute symbol automatically positions itself under the attribute you inserted in step 1.

4. Label the attribute `users`.
5. Using Table 1 as a guide, specify the attributes of the remaining classes.
   Be sure to insert a new attribute symbol for each attribute in the class (rather than entering multiple lines on one attribute symbol).

   Your diagram should resemble Figure 11.

   Figure 11: Completed Attributes for the Email Model



Moving Symbols

As your class symbols grow, you may find it necessary to reposition them. Use the procedures in this section to move symbols individually or in groups.

Moving a Symbol

To move a symbol:

1. Click on the symbol.
   Selection handles appear.
2. Press the left mouse button and drag the symbol to the desired position.

Selecting a Group of Symbols

You can add to a selection by pressing the Shift key and selecting other symbols, or you can use a bounding box to select all of the symbols in an area of the diagram at one time.

To use a bounding box:

1. Move the pointer to a corner of the area you want to select.
2. Press the left mouse button.
3. Drag the pointer diagonally across the area to be selected.
   A bounding box appears around the perimeter of the selected area.
4. Release the left mouse button.
   All the symbols and arcs within the area display handles.

Moving a Group of Symbols

To move a group of selected symbols:

1. Move the pointer over one of the selected symbols.
   The pointer changes to a circle.
2. Press the left mouse button and drag the symbol.
   As you drag the symbol, all of the selected symbols move in tandem.

Modeling Operations

An operation is an action performed by objects in a class. For example, a user can send messages and join conferences. Therefore, *send_msg* and *join_conf* are modeled as operations of the *User* class.

Try to determine the operations of the Email application's classes from the requirements. When you are finished, compare your list against the one in Table 2.

Table 2: Email Model Class Operations

Class	Operations
MailSystem	find_user find_conf login_user delete_user_or_conf create_user_or_conf start
Receiver	read_msg print_mbx receive_msg
Mailbox	add_msg print_mbx print_msg empty
User	string start send_msg join_conf
Conference	string
Message	send_to_rec print

Inserting Class Operations

There are three methods for inserting class operations:

- Inserting operation symbols using the Symbol List
- Using the Alt+Ctrl+a keyboard accelerator
- Using the Class Table Editor (CTE)

The instructions in this chapter tell you how to insert operations using the Symbol List. If you prefer, you can use the keyboard accelerator.

In Chapter 5, you use the Class Table Editor to add details about the Email application's operations.

Inserting Operations Using the Symbol List

To insert operations on your diagram using the Symbol List:

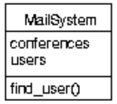
1. Drag an operation symbol from the symbol list into the *MailSystem* class.

   The class symbol resizes to fit the operation. Selection handles appear on the operation symbol.

2. Label the operation **`find_user`**.

   StP automatically appends parentheses to the label to represent the operation's arguments. Later in this tutorial, you define operation arguments using the Class Table Editor.

   Figure 12: Labeled Operation

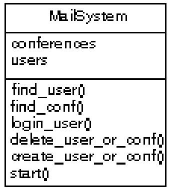
   

3. Using Table 2 as a guide, insert the remaining operations for the *MailSystem* class.

   Be sure to insert a new operation symbol for each operation in the class.

   The *MailSystem* class symbol should resemble Figure 13.

Figure 13: MailSystem Operations

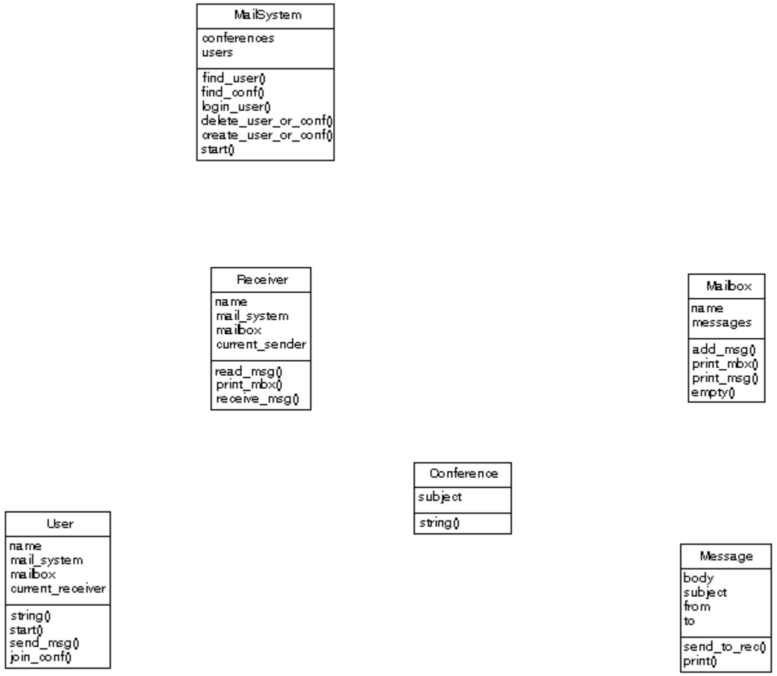


4. Using Table 2 as a guide, specify the operations for the remaining classes.

   As your diagram grows, you may find it necessary to zoom to a lower percentage or to use the **Fit Diagram in Window** command. These options are available from the View menu.

   Your diagram should resemble Figure 14.

Figure 14: Completed Operations for the Email Model



Modeling Associations

In addition to classes, you must also identify and understand the associations, or relationships, between classes. For example, users of the Email application can send messages. Therefore, an association exists between the *User* class and the *Message* class. Associations are represented by connection arcs on your diagram.

Refer to the requirements to determine the associations between the classes in your diagram. If you need help, refer to Table 3.

Table 3: Email Model Associations

Class	Is Associated With	Arc Label
MailSystem	User	*Is_Logged_Onto_By*
User	Conference	*Joins*
	Message	*Sends*
Receiver	Mailbox	*Owns*
Mailbox	Message	*Holds*

Inserting An Association Arc

To insert an association arc:

1. Drag an Arc symbol into the *User* class and release the mouse button.
   This action attaches one end of the association arc to the *User* class.
2. Move the pointer into the *Message* class and click the left mouse button.
   This action attaches the opposite end of the association arc to the *Message* class. Selection handles appear on the arc.

Figure 15: Association Arc

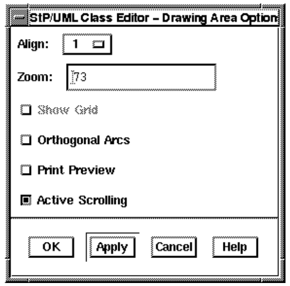
Inserting Orthogonal Arcs

Orthogonal arcs consist of horizontal and vertical line segments. An orthogonal arc makes it possible for you to draw arcs without intersecting other symbols in the diagram.

To set orthogonal arcs:

1. Choose **Drawing Area** from the Options menu.
   The Drawing Area Options dialog appears.

Figure 16: Drawing Area Options Dialog



Align: 1

Zoom: 73

☐ Show Grid

☐ Orthogonal Arcs

☐ Print Preview

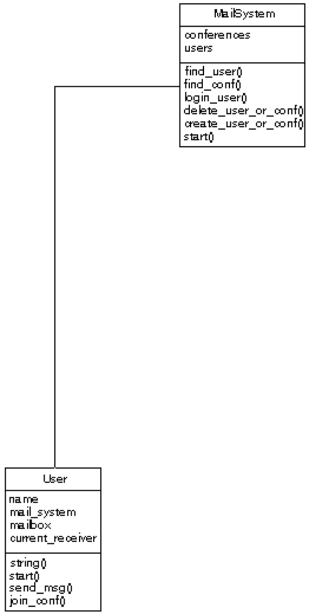
☒ Active Scrolling

2. Select **Orthogonal Arcs**.
3. Click **OK** to apply the change and dismiss the dialog box.

To insert an orthogonal arc:

1. Drag an Arc symbol into the *User* class and release the mouse button.
2. Drag the arc up and to the right into the *MailSystem* class and click the left mouse button.
   This action creates an orthogonal arc between the *User* and *MailSystem* classes, as shown in Figure 17.

Figure 17: Orthogonal Arc



You do not need to insert another orthogonal arc in your diagram. To deselect orthogonal arcs:

1. Redisplay the Drawing Area Options dialog.
2. Deselect the **Orthogonal Arcs** option.
3. Click **OK**.

Inserting the Remaining Associations

Using Table 3 as a guide, insert the remaining associations into your diagram. If you wish, you may draw the associations using the accelerated method of inserting arcs, as described on page 2-16.

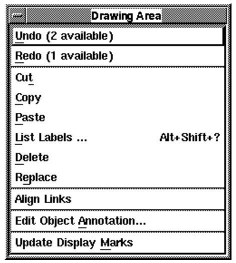
Aligning the Links on Your Diagram

The **Align Links** command provides a quick method for straightening the arcs on your diagram.

To use this command:

1. Display the Drawing Area menu by clicking the right mouse button anywhere in the drawing area.
   The Drawing Area menu appears, as shown in Figure 18.

   Figure 18: Drawing Area Menu

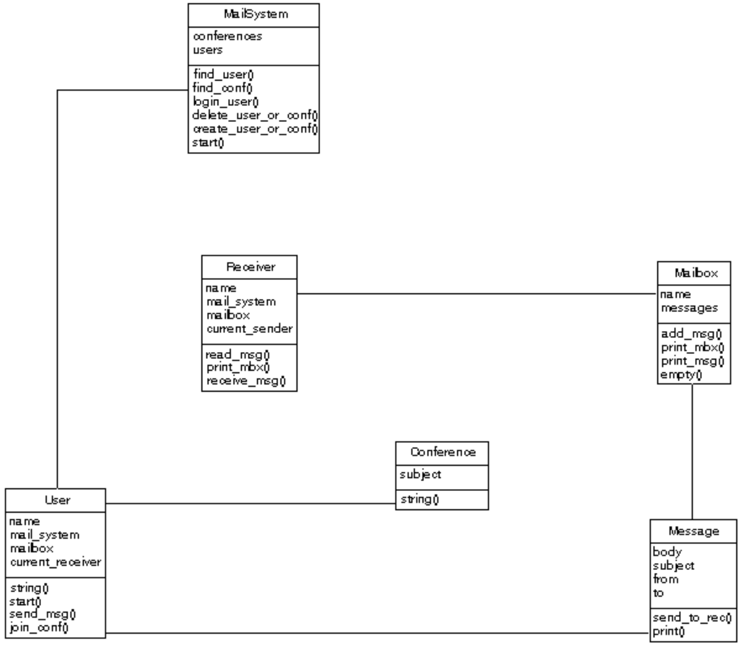
   

   For additional information about the Drawing Area menu, see *Fundamentals of StP*.

2. Choose **Align Links**.
   The links on the diagram are aligned.

   Your diagram should resemble Figure 19.

Figure 19: Inserting Association Arcs



Labeling Associations

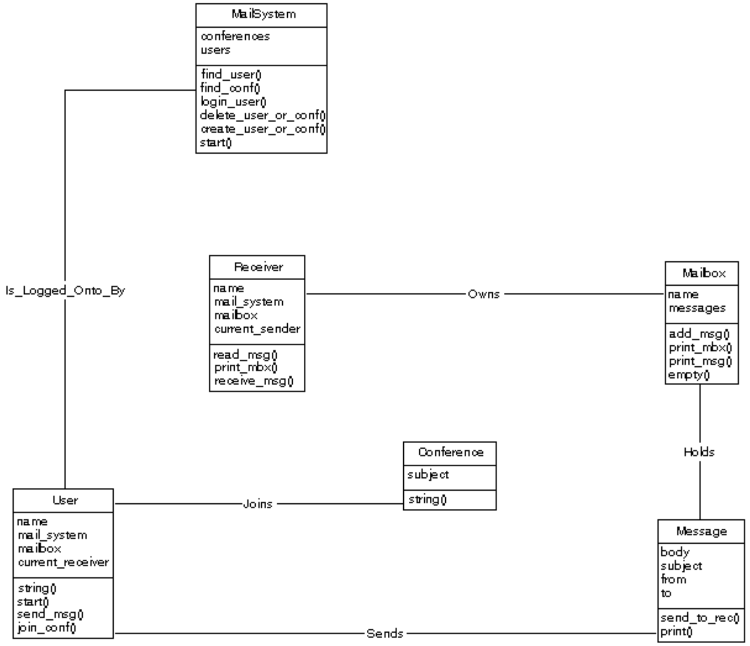
Associations are bidirectional. The label of an association can imply a direction, but it can also be read to show the other direction. For example, a user *sends* a message, and conversely, a message *is sent by* a user. Either direction is acceptable, but to improve readability, try to use active verbs when labeling associations.

Draw your diagram so that association labels read from left to right or top to bottom. To have all the labels for this diagram read from left to right or top to bottom, you need to use one label, *Is_Logged_Onto_By*, that is not an active verb.

To label the association arc between the *MailSystem* and *User* classes:

1. Select the vertical portion of the association arc between *MailSystem* and *User*.
2. Type **`Is_Logged_Onto_By`**.
   The label appears on the arc.
3. Click the left mouse button to terminate the label-editing mode.
4. Label the remaining association arcs on your diagram, using Table 3 as a guide.
   Your diagram should resemble Figure 20.

Figure 20: Labeling Associations



Setting Display Mark Options

Display marks are used to indicate a variety of information. In the remainder of this chapter, you add information to your object model that has associated display marks.

You can set options for display marks to decide which marks are:

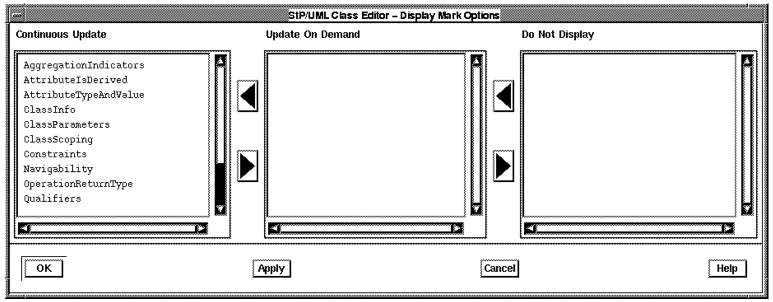
- Updated automatically
- Updated on demand (by choosing **Update Display Marks** from the Edit menu)
- Not displayed

You set display mark options using the Display Mark Options dialog box. For this tutorial, you set the display marks to be updated automatically.

To set display mark options:

1. Choose **Display Marks** from the Options menu.
   The Display Mark Options dialog appears.
2. Move all of the display marks into the Continuous Update window by using the appropriate arrows.
   The Display Mark Options dialog should resemble the one shown in Figure 21.

Figure 21: The Display Mark Options Dialog



AggregationIndicators
AttributeIsDerived
AttributeTypeAndValue
ClassInfo
ClassParameters
ClassScoping
Constraints
Navigability
OperationReturnType
Qualifiers

3. Click **OK** to apply the change and dismiss the dialog.

For more information about the Display Mark Options dialog, refer to *Fundamentals of StP*.

Defining Association Properties

An association can have various properties, including:

- Role Names
- Multiplicity
- Qualifiers
- Navigability
- Constraints

In the following sections, you add these properties to associations in your diagram using UML menu commands and the Properties dialog box. The Properties dialog box enables you to change the characteristics of a selected relationship and apply all the changes at once.

For a complete description of the association properties that can be defined using the Properties dialog, see Creating UML Models.

Naming Association Roles

Each end of an association relationship is a role. A role is the function that one end of a relationship assumes with respect to the other end of the relationship. Role names are optional; you can provide each role with a role name, but you are not required to do so. The advantage of providing role names is that they clarify the meaning of the relationship.

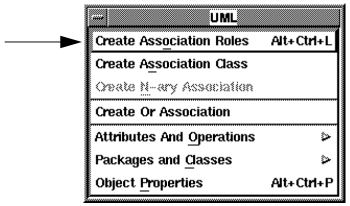
In the *Email* application, you give role names to the relationship between the *User* and *Message* classes. In this relationship, *User* assumes the role of sender and *Message* assumes the role of mail message.

Specifying Role Names

To specify association role names:

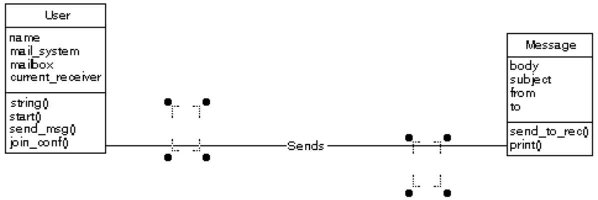
1. Select the arc between the *User* and *Message* classes.
   Selection handles appear on the arc.
2. Choose **Create Association Roles** from the UML menu.

Figure 22: The Create Association Roles Command



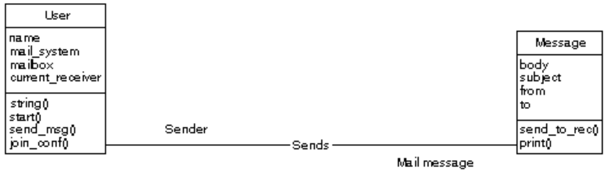
Text frames with selection handles appear on each side of the arc.

Figure 23: Text Frames



3. Select the text frame next to the *User* class.
4. Type the role name `Sender`.
5. Click the left mouse button to terminate the label editing mode.
6. Select the text frame next to the *Message* class.
7. Type the role name `Mail Message`.
   Your diagram should resemble Figure 24.

Figure 24: Role Names on the Diagram



Indicating Multiplicity on Roles

In an association, a single instance of one class can relate to instances of the other class one or more times. The number of instances that can relate at one time is called the “multiplicity” of the association.

Table 4 lists the conventional notation for different types of multiplicity.

Table 4: Multiplicity Types

Type	Description	Notation
Unspecified	Default	None
Exactly One	A single instance of a class participates in the association	1
Many	More than one instance of a class participates in the association	*
Optional	A single instance of a class may or may not participate in the association	0..1
One or More	One or more instances of a class may participate in the association	1..*
Explicit Multiplicity (single value)	A specific number of instances of a class may participate in the association	N (where N is an integer)

Table 4: Multiplicity Types (Continued)

Type	Description	Notation
Explicit Multiplicity (range)	The number of participating instances of a class falls within a specified range	N..N (where N is an integer)

Assigning Multiplicity to Roles

In this example, you assign a multiplicity value of one to the *Sender* role and a multiplicity value of many to the *Mail Message* role, because a *Sender* can send many *Mail Messages*, but a *Mail Message* can be sent by only one *Sender*.

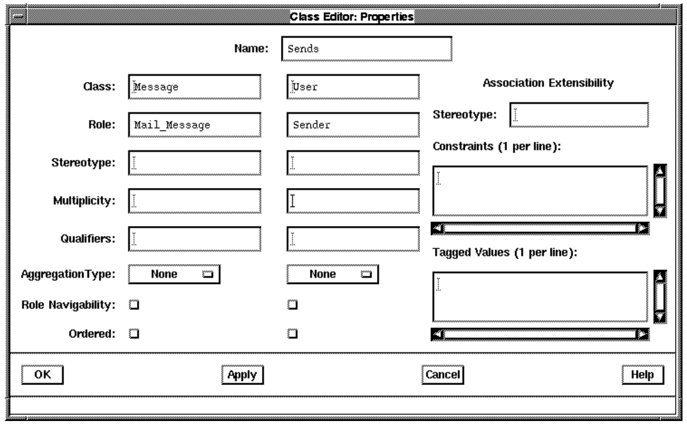
You use the Properties dialog box to specify multiplicity.

To assign multiplicity to an association:

1. Select the arc between the *User* and *Message* classes.
   Selection handles appear on the arc.
2. Use one of the following methods to open the Properties dialog box:
   - Choose **Edit Properties** from the UML menu
   - Double-click on the selected relationship
   - Use the accelerator Alt+Ctrl+P

   The Properties dialog box appears. The name of the association appears in at the top of the dialog box. The names of the classes joined by the association as well as the association role names appear in read-only fields beneath the association name.

Figure 25: The Properties Dialog for Associations



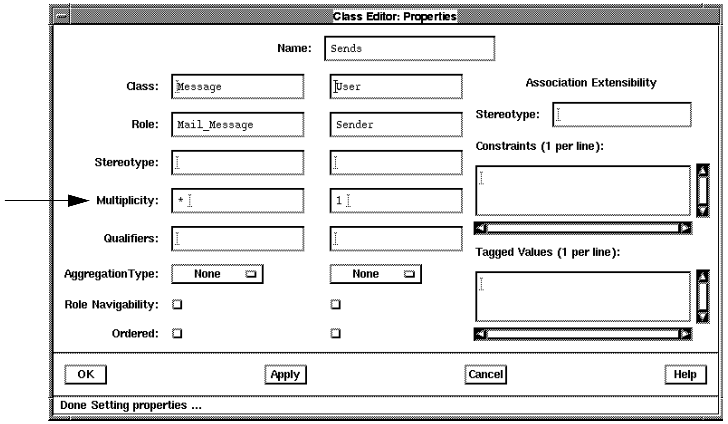
Name: Sends

Class:	Message	User
Role:	Mail_Message	Sender
Stereotype:		
Multiplicity:		
Qualifiers:		
AggregationType:	None	None
Role Navigability:	☐	☐
Ordered:	☐	☐

3. Click on the Multiplicity text field for *Message*.
4. Type `*`.
   This establishes a multiplicity of many for the *Mail Message* role.
5. Click on the Multiplicity text field for *User*.
6. Type `1`.
   This establishes a multiplicity of exactly one for the *Sender* role.

   The Properties dialog should resemble the one shown in Figure 26.

Figure 26: Specifying Multiplicity



Class Editor: Properties

Name: Sends

Class:	Message	User
Role:	Mail_Message	Sender
Stereotype:		
Multiplicity:	*	1
Qualifiers:		
AggregationType:	None	None
Role Navigability:	☐	☐
Ordered:	☐	☐

Association Extensibility

Stereotype:

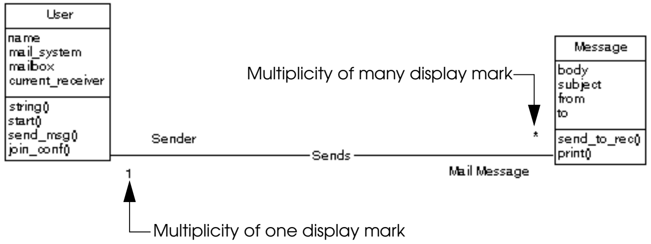
Constraints (1 per line):

Tagged Values (1 per line):

Done Setting properties ...

7. Click **OK**.
   The dialog is dismissed and the properties are applied. The multiplicity display marks appear on the diagram, as shown in Figure 27.

Figure 27: Multiplicity Display Marks



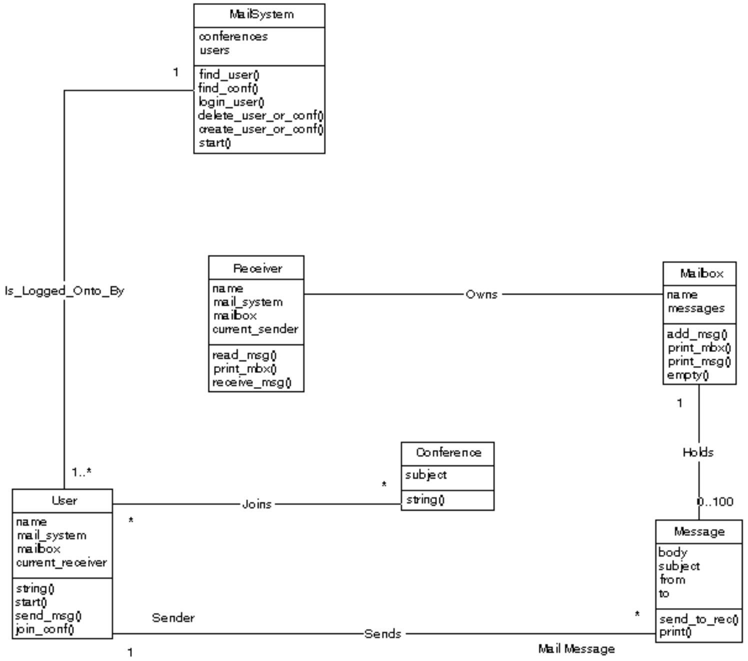
Specifying Additional Multiplicity

Add additional multiplicity to the Email diagram to indicate:

- A user can have only one mail system, but a mail system can have one or more users (multiplicity of exactly one to one or more).
- A user can join many conferences, and a conference can have many users (multiplicity of many to many).
- A message can have only one mailbox; a mailbox can contain up to a hundred messages, or it may be empty (multiplicity of one to a specified range).

When you are finished, your diagram should resemble Figure 28.

Figure 28: Assigning Multiplicity to Associations



Qualifying an Association

A qualifier is an attribute whose unique value identifies a single object instance. For example, in the relationship between *MailSystem* and *User*, the attribute that uniquely identifies a user of the mail system is *name*.

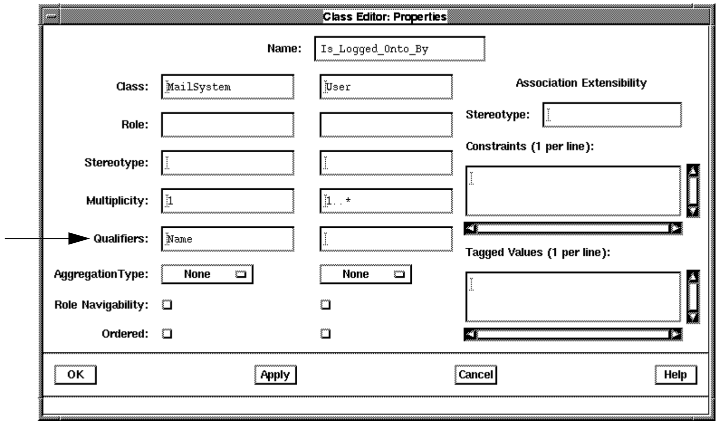
A qualifier appears as a display mark on a diagram. It is a labeled box on the end of the association arc opposite the class it qualifies.

Inserting a Qualifier

To insert a qualifier:

1. Select the arc between the *MailSystem* and *User* classes.
   Selection handles appear on the arc.
2. Choose **Edit Properties** from the UML menu.
   The Properties dialog box appears.
3. Click on the Qualifier text field for the *MailSystem* class.
4. Type `Name`.
   The Properties dialog should look like the one in Figure 29.

Figure 29: Designating Qualifiers



Class Editor: Properties

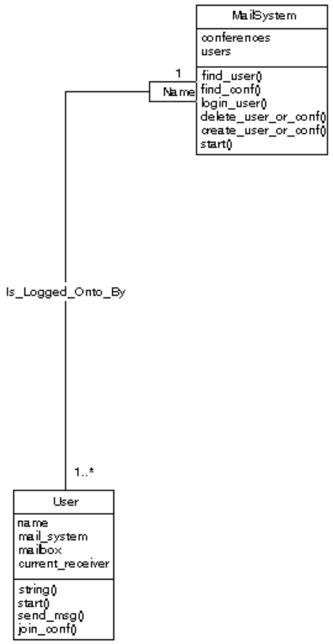
Name: Is_Logged_Onto_By

Class:	MailSystem	User	
Role:			
Stereotype:			
Multiplicity:	1	1..*	
Qualifiers:	Name		
AggregationType:	None	None	
Role Navigability:	☐	☐	
Ordered:	☐	☐	

5. Click **OK**.

The dialog is dismissed and the property is applied. The qualifier appears in the diagram next to the *MailSystem* class, as shown in Figure 30.

Figure 30: Inserting a Qualifier



Indicating Navigability

An association can be bidirectional (used in both directions) or unidirectional (used in only one direction). You indicate the direction of an association by specifying the “navigability” of each of its roles. In bidirectional associations, both association roles are navigable. In unidirectional associations, only one association role is navigable.

For example, in the Email system, the *User* class requests services from the *Conference* class, but the *Conference* class does not request services from the *User* class. In this unidirectional association, only the role representing *Conference* (the class that receives service requests) is navigable.

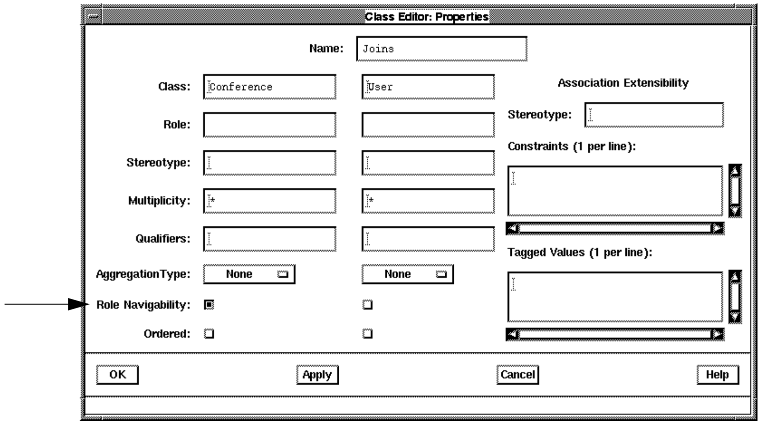
Bidirectional and unidirectional associations are implemented differently, so it is important to specify role navigability if you plan to generate code from your StP/UML model. For complete information about annotating your model for code generation, see *Generating and Reengineering Code*.

To indicate navigability on an association:

1. Select the arc between the *User* and *Conference* classes.
   Selection handles appear on the arc.
2. Choose **Edit Properties** from the UML menu.
   The Properties dialog box appears.
3. Click on the Role Navigability option for the *Conference* class.
   The option is highlighted.

   The Properties dialog should look like the one in Figure 31.

Figure 31: Indicating Role Navigability


Class Editor: Properties

Name: Joins

Class:	Conference	User
Role:		
Stereotype:		
Multiplicity:	*	*
Qualifiers:		
AggregationType:	None	None
Role Navigability:	☒	☐
Ordered:	☐	☐

Association Extensibility

Stereotype:

Constraints (1 per line):

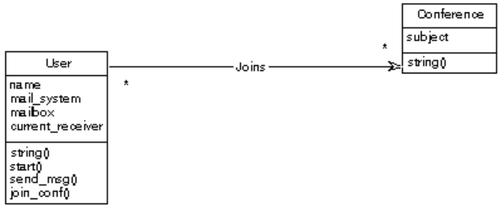
Tagged Values (1 per line):

4. Click **OK**.

The dialog is dismissed and the property is applied. The display mark for navigability is an arrow pointing to the class that receives service requests.

Your diagram should resemble Figure 32.

Figure 32: Role Navigability Display Mark



Indicating Relationship Constraints

Constraints are one of the mechanisms provided by the Unified Modeling Language to extend the semantics of standard modeling elements. A constraint on a relationship restricts the values that the relationship can have.

Some relationship constraints are predefined in UML; you can also define your own.

Constraints are not displayed on the class diagram.

Assigning a Constraint

In the Email application, each mailbox has a name that corresponds to the name of the mailbox owner. In this example, you place a constraint on the *Owns* relationship specifying that the name attributes of a *Mailbox* and its associated *Receiver* must match.

Unlike the properties you defined in the previous sections, the constraint property applies to the association itself rather than to one of its roles.

To place a constraint on the *Owns* relationship:

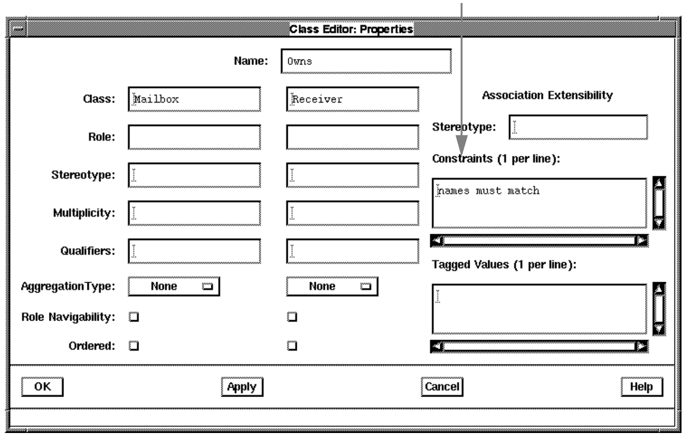
1. Select the arc between the *Receiver* and *Mailbox* classes.

   Selection handles appear on the arc.
2. Choose **Edit Properties** from the UML menu.

   The Properties dialog box appears.
3. Click in the Constraints text field.
4. Type `names must match`.

   The Properties dialog should look like the one in Figure 33.

Figure 33: Indicating Constraints



Name: Owns

Class: Mailbox | Receiver

Role:

Stereotype:

Multiplicity:

Qualifiers:

AggregationType: None | None

Role Navigability: ☐ ☐

Ordered: ☐ ☐

Stereotype:

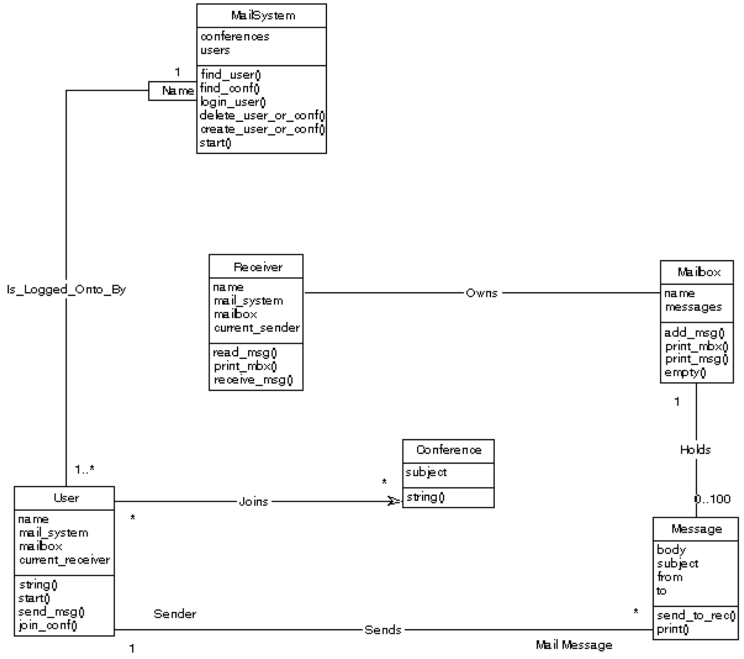
Constraints (1 per line): names must match

Tagged Values (1 per line):

5. Click **OK**.
   The dialog is dismissed and the constraint is applied.

You have now completed a basic class diagram for the Email system. Your diagram should resemble Figure 34.

Figure 34: Class Diagram for the Email Application



Saving the Class Diagram

To save your diagram, choose **Save** from the File menu. Your diagram should be error-free, so you should not receive error messages. If you do, correct the error and save the diagram again.

Continuing the Tutorial

In this chapter, you:

- Started the Class Editor from the StP Desktop
- Created a basic class diagram for the *Email* system
- Used the Properties dialog box to define association characteristics

In the next chapter, you expand the model to include special class and relationship types.

4

Completing the Class Diagram

In this chapter, you expand your class diagram by adding specialized relationships and classes. In addition, you define a stereotype for an Email class.

Modeling Specialized Relationships

Some classes participate in special types of relationships. These relationships include:

- Generalization
- Aggregation
- Composition

In the following sections, you model each of these relationships. You model the relationships by drawing them on the diagram or by using the Object Annotation Editor (OAE).

Modeling Generalizations

A generalization is the relationship between a class, called a superclass, and one or more specializations of the class, called subclasses. Each subclass inherits attributes, operations, and associations from its superclass. In addition, a subclass can have its own attributes and operations that add to or override those of its superclass.

In the Email system, you model one generalization. Since the *Receiver* of a message can be either a *User* or a *Conference*, you model this set of relationships as a generalization. *Receiver* is the superclass, and *User* and *Conference* are the subclasses.

A generalization is often called an “is-a” relationship: a *User* is a *Receiver*, and a *Conference* is a *Receiver*.

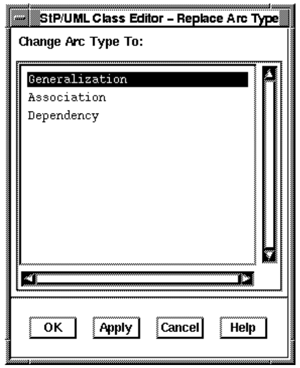
Inserting a Generalization

You draw a generalization arc from the subclass to the superclass.

To insert a generalization:

1. Insert an arc from the *User* class to the *Receiver* class.
   Selection handles appear on the arc.
2. Choose **Replace** from the Edit menu.
   The Replace Arc Type dialog appears.

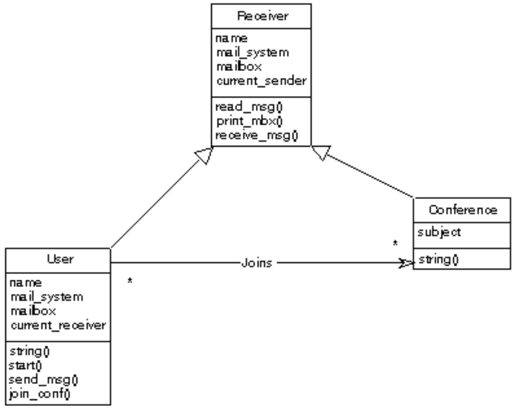
Figure 1: The Replace Arc Type Dialog



3. Choose **Generalization** from the scrolling list.
4. Click **OK**.
   A generalization relationship arc appears, with an arrow pointing to the superclass, *Receiver*.

5. Repeat the procedure to create a generalization relationship between the *Conference* class and the *Receiver* class.
   Your diagram should resemble Figure 2.

Figure 2: Inserting a Generalization



Modeling Aggregations

An aggregation is a special form of association between a whole (or assembly) and its parts (or components). Instances of the component classes can be assembled into a single instance of the assembly class.

For example, the mail system consists of users and conferences. In your class diagram, *User* and *Conference* are subclasses of *Receiver*. Therefore, you model the relationship between *MailSystem* (the assembly class) and *Receiver* (the component class) as an aggregation.

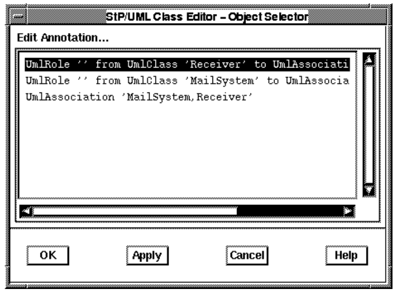
To create an aggregation, you first draw an association, then you add an annotation to change the association to an aggregation. Annotations are notes and items containing descriptive text and values. You add them using the Object Annotation Editor (OAE).

Adding the Aggregation Annotation Item

To change an association to an aggregation:

1. Insert an arc from the *MailSystem* class to the *Receiver* class.
   Selection handles appear on the arc.
2. Choose **Edit Object Annotation** from the Edit menu.
   The Object Selector dialog appears.

Figure 3: The Object Selector Dialog



StP/UML Class Editor – Object Selector

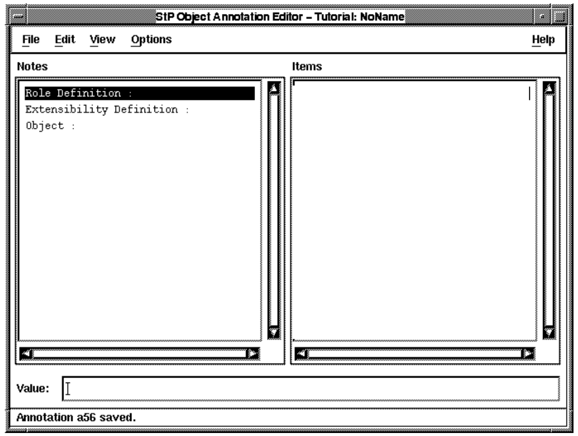
Edit Annotation...

```
UmlRole '' from UmlClass 'Receiver' to UmlAssociati
UmlRole '' from UmlClass 'MailSystem' to UmlAssocia
UmlAssociation 'MailSystem,Receiver'
```

In the StP repository, an association arc maps to three objects: the association relationship and the two roles representing the ends of the relationship. You choose which object to annotate using the Object Selector. To create an aggregation, you annotate the role representing the assembly class.

3. Select **UmlRole '' from UmlClass 'MailSystem'**.
4. Click **OK**.
   The OAE appears. The Role Definition note is highlighted in the Notes scrolling list.

Figure 4: The Object Annotation Editor (OAE)



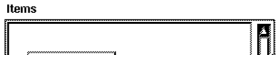
StP Object Annotation Editor – Tutorial: NoName

```
Role Definition :
Extensibility Definition :
Object :
```

Annotation a56 saved.

5. To view the annotation items for the Role Definition note, press the right mouse button anywhere in the Items scrolling list and display the Add Item cascade menu.

Figure 5: The Add Item Cascade Menu



Navigating to a Stereotype Definition

You can navigate from a stereotype in the Class Editor to its definition in the Stereotype Editor.

To navigate from the Class Editor to the Stereotype Editor:

1. Select the *TextInput* class.
2. Choose **Stereotype Definition** from the Navigation menu.

   The Stereotype Editor appears, with the *UserInput* stereotype symbol flashing.

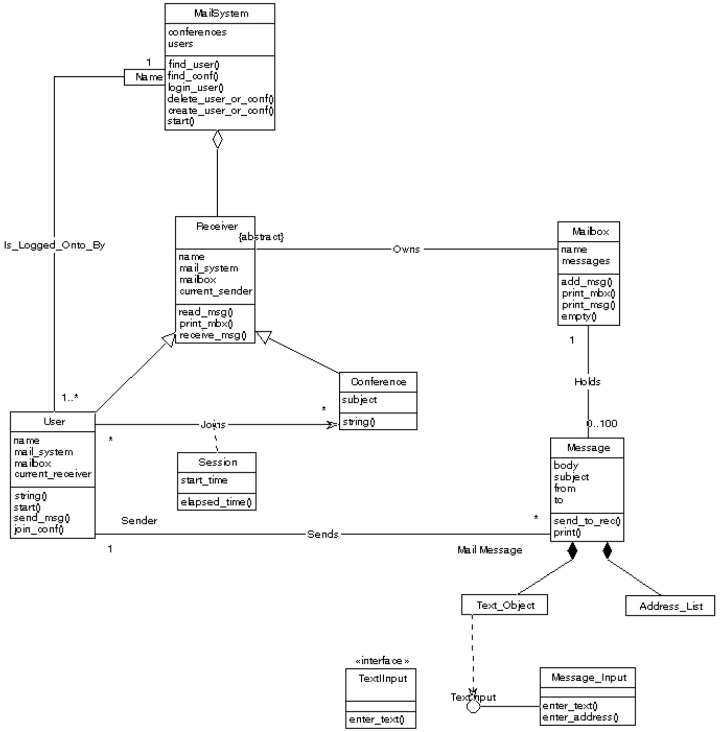
You do not use the Stereotype Editor again in this tutorial, so use one of the following methods to exit the editor:

- Choose **Exit** from the File menu.
- Use the Ctrl+E accelerator.

Completing the Class Diagram

You have now completed the *Application_1* class diagram. Your diagram should resemble Figure 24.

Figure 24: Completed Class Diagram



Saving the Class Diagram

To save your diagram, choose **Save** from the File menu. Your diagram should be error-free, so you should not receive error messages. If you do, correct the error and save the diagram again.

Continuing the Tutorial

In this chapter, you:

- Modeled generalization, aggregation, and composition relationships
- Modeled abstract classes, association classes, and interfaces
- Extended the semantics of the model using stereotypes
- Completed the class diagram for the Email application

In Chapter 5, you use the Class Table Editor to expand the definitions of the attributes and operations for the *Message* class.

5 Using the Class Table Editor

In Chapters 3 and 4, you created a class diagram for the Email application using the Class Editor. In this chapter, you navigate to the Class Table Editor (CTE), where you expand the definitions of the *Message* class's attributes and operations. You also construct a class table for the *TextInput* class.

About the Class Table Editor

The CTE allows you to capture the full definitions of a class's attributes and operations. In this chapter, you create class tables for two classes. In a complete UML model, you would complete a class table for each class in the model.

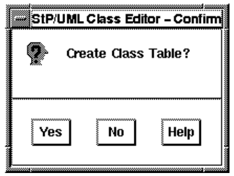
Navigating to the *Message* Class Table

To start the CTE, you navigate from the *Message* class in the *Application_1* class diagram.

To navigate to the CTE:

1. Select the *Message* class symbol on your class diagram.
   Selection handles appear on the symbol.
2. Choose **Class Table** from the Navigation menu.
   Since the *Message* class does not have a class table, a dialog box appears prompting you to create one.

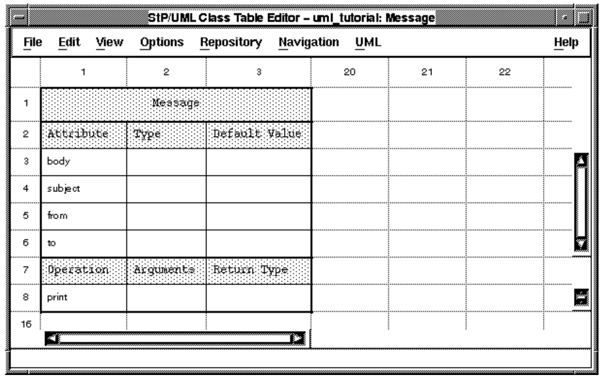
Figure 1: Create Class Table Dialog



Create Class Table?

3. Click **Yes**.
   The CTE appears with a class table for *Message*.

Figure 2: The Class Table Editor (CTE)



	1	2	3	20	21	22
1	Message					
2	Attribute	Type	Default Value			
3	body					
4	subject					
5	from					
6	to					
7	Operation	Arguments	Return Type			
8	print					
16						

The table contains the attributes and operations that you added to the *Message* class in your class diagram. However, not all of the operations are visible. Later in the tutorial, you enlarge the Operations section of the table to make these operations visible.

Hiding and Showing Class Table Sections

The class table is divided into sections that you can hide from view or show. When you first create a class table, all but the following sections are hidden by default:

- Class—This section displays the name of the class.
- Attributes—This section contains the name, type, and default value of the class's attributes.
- Operations—This section contains the name, arguments, and return type of the class's operations.

To provide complete definitions for the attributes and operations of the *Message* class, you need to display these hidden sections:

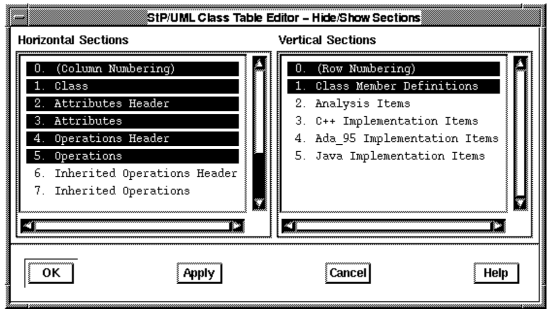
- Analysis Items—This section contains information related to UML modeling.
- C++ Implementation Items—This section contains information for generating C++ code.

To display hidden sections:

1. Choose **Hide/Show** from the CTE View menu.

   The Hide/Show dialog box appears. The sections that are hidden from view are not highlighted.

Figure 3: The Hide/Show Dialog Box



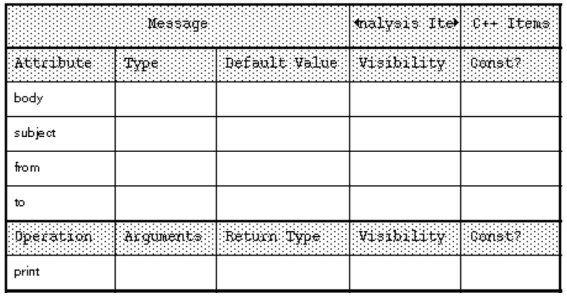
---

2. Click on Analysis Items and C++ Implementation Items in the Vertical Sections scrolling list.
3. Click **OK**.

   The Analysis Items and C++ Items sections appear in the class table.

   You may need to expand the editor window to see all of the columns. Your class table should resemble the one shown in Figure 4.

Figure 4: Class Table with Additional Columns



Message			◂nalysis Ite▸	C++ Items
Attribute	Type	Default Value	Visibility	Const?
body				
subject				
from				
to				
Operation	Arguments	Return Type	Visibility	Const?
print				

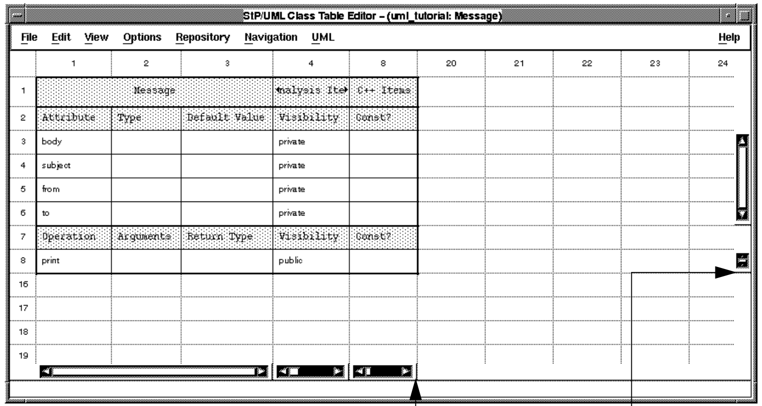
Enlarging Class Table Sections

To view all the rows and columns in your table, you need to enlarge two table sections:

- C++ Items
- Operations

Before you get started, stretch the size of your CTE window to display at least three additional rows and four additional columns. Your CTE window should resemble the one shown in Figure 5.

Figure 5: Enlarged CTE Window



	1	2	3	4	8	20	21	22	23	24
1	Message			◂nalysis Ite▸	C++ Items					
2	Attribute	Type	Default Value	Visibility	Const?					
3	body			private						
4	subject			private						
5	from			private						
6	to			private						
7	Operation	Arguments	Return Type	Visibility	Const?					
8	print			public						
16										
17										
18										
19										

Place the pointer here to enlarge the C++ Items section.

Place the pointer here to enlarge the Operations section.

Enlarging the Analysis Items Section

The C++ section of the class table contains five columns. Currently, only one column (Const) is visible.

To make the remaining columns visible:

1. Place your pointer on the right side of the scroll bar for the C++ Items section of the table, as shown in Figure 5.
2. When the pointer changes to the shape shown in Figure 6, drag the pointer four columns to the right.

   Figure 6: Pointer Shape

3. Release the mouse button.

All five C++ columns should now be visible, as shown in Figure 8.

Enlarging the Operations Section

Currently, the Operations section of the class table displays only one row. You need to enlarge this section of the class table to make the remaining operation visible.

To enlarge the Operations section:

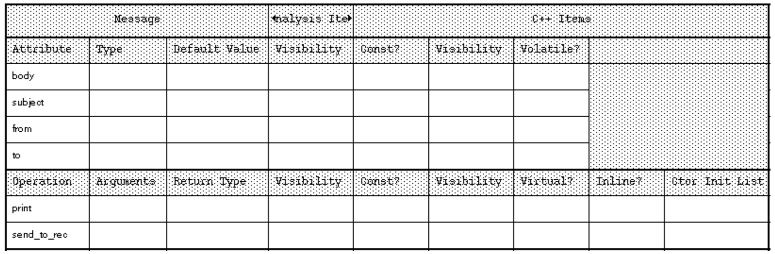
1. Place your pointer at the bottom of the scroll bar for the Operations section of the table, as shown in Figure 5.
2. When the pointer changes to the shape shown in Figure 7, drag the pointer down one row.

   Figure 7: Pointer Shape

3. Release the mouse button.
   The remaining operation should now be visible, as shown in Figure 8.

Figure 8: Expanded Class Table



Message			◂nalysis Ite▸	C++ Items				
Attribute	Type	Default Value	Visibility	Const?	Visibility	Volatile?		
body								
subject								
from								
to								
Operation	Arguments	Return Type	Visibility	Const?	Visibility	Virtual?	Inline?	Ctor Init List
print								
send_to_rec								

Defining Attributes

Since you navigated to the class table from the Email class diagram, the table already contains the *Message* class's attribute and operation names. In this section, you specify each attribute's type and default values. StP/UML uses this information when you generate C++ code later in the tutorial.

Entering the Data Type

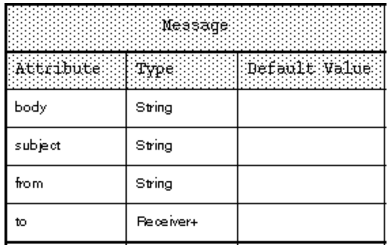
You enter each attribute's data type in the Type column. Attribute data types for the *Message* class attributes are:

- body — String
- subject — String
- from — String
- to — Receiver*

To specify type:

1. Click on the first empty cell in the Type column.
   The cell is highlighted, indicating that it is active.
2. Label this cell **`String`**.
3. Press the down arrow key to activate the next empty cell in the column.
4. Enter the data types for the remaining attributes of the *Message* class.
   The Type column should resemble the one shown in Figure 9.

Figure 9: **Entering Data Types**



Message		
Attribute	Type	Default Value
body	String	
subject	String	
from	String	
to	Receiver*	

Entering Default Values

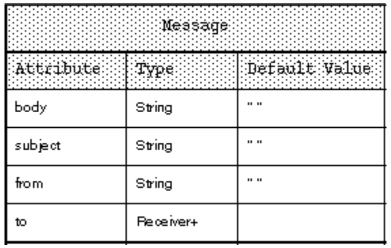
You enter the default value for each attribute in the Default Value column. Default values allow you to provide a value for the initialization list of a constructor in cases where the value of an attribute is not static. These default values are required for the *Message* class attributes:

- body — " "
- subject — " "
- from — " "

To specify default values:

1. Click on the first empty cell in the Default Value column.
   The cell is highlighted.
2. Type **`" "`** (quotation mark, space, quotation mark).
3. Press the down arrow key to activate the next empty cell in the column.
4. Enter the default values for the remaining attributes of the *Message* class.
   The Default Value column should resemble the one shown in Figure 10.

Figure 10: **Entering Default Values**



Message		
Attribute	Type	Default Value
body	String	" "
subject	String	" "
from	String	" "
to	Receiver+	

Defining Operations

In this section, you specify valid arguments and return types for the *Message* class's operations. This information is used when you generate C++ code later in the tutorial.

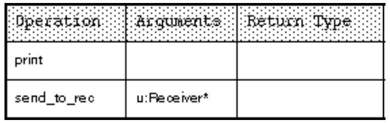
Entering Arguments

You enter valid arguments for each operation in the Arguments column. In this example, only the *send_to_rec* operation takes an argument. Since the other operations do not have arguments, leave them blank. A blank field indicates that no argument is provided when calling the operation.

To specify arguments for the *send_to_rec* operation:

1. In the Arguments column, click on the empty cell aligned with the *send_to_rec* operation.
   The cell is highlighted.
2. Label this cell `u:Receiver*`
   The Arguments column should resemble the one shown in Figure 11.

Figure 11: **Entering Arguments**



Operation	Arguments	Return Type
print		
send_to_rec	u:Receiver*	

Entering the Return Type

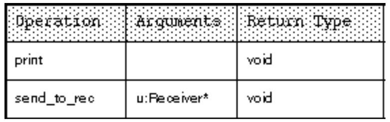
You enter a valid return type for an operation in the Return Type column. These return types are required by the *Message* class operations:

- print — void
- send_to_rec — void

To specify return types:

1. Click on the first empty cell in the Return Type column.
   The cell is highlighted.
2. Type `void`.
3. Press the down arrow key to activate the next empty cell in the column.
4. Type `void`.
   The Return Type column should resemble the one shown in Figure 12.

Figure 12: **Entering Return Types**



Operation	Arguments	Return Type
print		void
send_to_rec	u:Receiver*	void

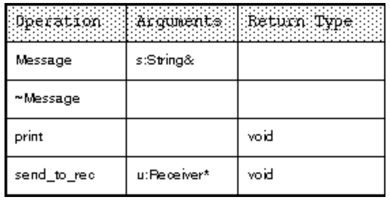
Adding Constructor and Destructor Functions

A constructor function lets you build (construct) an object. A destructor function lets you discard an object. In this section, you insert the constructor *Message* and the destructor *~Message* into the Message class table. (The ~ symbol indicates that the function is a destructor.)

To insert constructors and destructors:

1. Select the row containing the *print* operation.
   To select a row, click on the number beside the row.
2. Choose **2** from the Insert Rows Before submenu of the Edit menu.
   Two blank rows appear in the class table.
3. Label the first empty cell in the Operation column `Message`.
4. Label the first empty cell in the Arguments column `s:String&`.
5. Label the second empty cell in the Operation column `~Message`.
   The Operation and Arguments columns should resemble Figure 13.

Figure 13: Inserting Constructor and Destructor Functions



Operation	Arguments	Return Type
Message	s:String&	
~Message		
print		void
send_to_rec	u:Receiver*	void

Entering Analysis Items and C++ Items

The following sections describe the analysis and C++ information you can specify in your class table.

Analysis Items

The Analysis Items section of the class table allows you to capture information related to UML modeling.

The following Analysis Items are for attributes:

- Visibility—Indicates the access to the attribute: public, private, or protected.
- Classes Attr?—A value of True indicates that the attribute is a static data member of the class.
- Derived?—A value of True indicates that the attribute is derived.

The following Analysis Items are for operations:

- Visibility—Indicates the access to the operation: public, private, or protected.
- Class Op?—A value of True indicates that the operation is a static member function of the class.
- Abstract?—A value of True indicates that the operation is a pure virtual function type.
- Throws—Specifies a comma-separated list of exception names that the operation might throw.

By default, the Visibility item is set to private for attributes and public for operations. You do not specify additional Analysis Item values in this tutorial.

C++ Items

The C++ Items section of the class table allows you to capture information that relates specifically to C++ constructs. This information is used when you generate C++ code later in the tutorial.

C++ Items for Attributes

The following C++ Items are for attributes:

- Const?—A value of True indicates that the attribute is a C++ constant.
- Visibility—Indicates the access to the attribute: public, private, or protected. A value in this column overrides a value for the Visibility item in the Analysis Items section.
- Volatile?—A value of True indicates that the attribute is a "volatile" attribute in C++.

Table 1 lists the C++ values required for the attributes in your class table.

Table 1: C++ Values for Attributes of the Message Class

Attribute	Const	Visibility	Volatile
body	False	private	False
from	False	private	False
to	False	private	False
subject	False	private	False

C++ Items for Operations

The following Analysis Items are for operations:

- Const?—A value of True indicates that the operation is a C++ constant.
- Visibility—Indicates the access to the operation: public, private, or protected. A value in this column overrides a value for the Visibility item in the Analysis Items section.
- Virtual?—A value of True indicates that the operation is virtual (a subclass can implement the same operation) for C++ code.
- Inline?—A value of True indicates that the operation is an inline operation for C++ code.
- Ctor Init List—Specifies an initialization list if the operation is a constructor.

Table 2 lists the C++ values required for the operations in your class table.

Table 2: C++ Values for Operations of the Message Class

Operation	Const	Visibility	Virtual	Inline	Ctor Init List
Message	False	public	False	True	body(""), subject(""), from("")
~Message	False	public	False	False	
print	False	public	False	False	
send_to_rec	False	public	False	False	

Using the Set Label Menu

The Set Label submenu, available on the Edit menu, provides lists of valid values for Analysis Items and C++ Items. (When there are no discrete values for a particular column and cell combination, the menu is blank.)

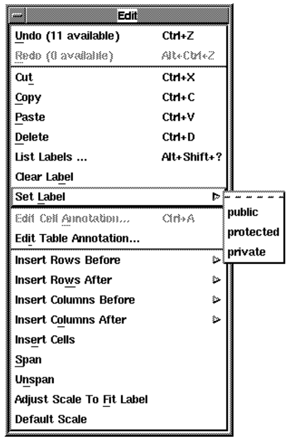
To use the Set Label submenu:

1. In the Visibility column of the C++ Items section, click on the empty cell aligned with the *body* attribute.

   The cell is highlighted.

2. Display the Set Label submenu of the Edit menu, as shown in Figure 14.

Figure 14: The Set Label Submenu for the Visibility Item

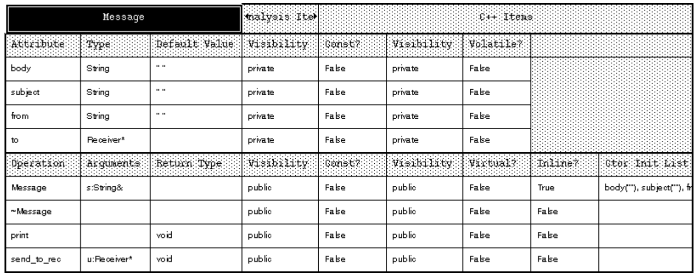


3. Choose **private.**
4. The value appears in the attribute's Visibility cell.
5. Add the remaining C++ values for Message's attributes and operations, either manually or using the Set Label submenu.

Checking a Class Table

You have now completed the class table for the *Message* class, which should resemble the one shown in Figure 15. When you are satisfied with your class table, choose **Save** from the File menu.

Figure 15: Completed Class Table for the Message Class



Message			nalysis Ite	C++ Items				
Attribute	Type	Default Value	Visibility	Const?	Visibility	Volatile?		
body	String	" "	private	False	private	False		
subject	String	" "	private	False	private	False		
from	String	" "	private	False	private	False		
to	Receiver*		private	False	private	False		
Operation	Arguments	Return Type	Visibility	Const?	Visibility	Virtual?	Inline?	Ctor Init List
Message	s:String&		public	False	public	False	True	body(""), subject(""), fr
~Message			public	False	public	False	False	
print		void	public	False	public	False	False	
send_to_rec	u:Receiver*	void	public	False	public	False	False	

StP/UML checks the correctness and consistency of your class table's syntax whenever you choose **Save** from the File menu. However, you can perform a syntax check at any time by choosing **Check Table Syntax** from the Repository menu.

After the **Check Table Syntax** command has completed the check, the message "Check Table Syntax succeeded" appears in the message area of the CTE window. If there are errors in the table's syntax, the Message Log appears listing the errors. For more information about the Message Log, see *Fundamentals of StP*.

Constructing a Class Table

You can construct a class table (or update an existing table) from information defined elsewhere in your model. The following commands are available from the UML menu:

- **Construct Class from Diagram Definitions command**—Constructs or updates a class table with information from the class diagram
- **The Construct Class from All Definitions command**—Constructs or updates a class table with information from the repository

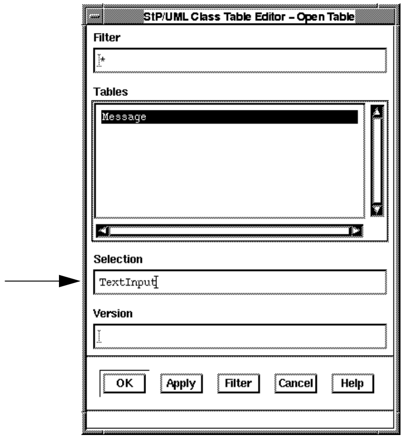
In this section, you construct a class table for the *TextInput* class based on its definition in the *Application_1* class diagram.

Opening a New Class Table

To open a new class table from the CTE:

1. Choose **Open** from the File menu.
   The Open Table dialog box appears.
2. Type `TextInput` in the Selection text field, as shown in Figure 16.

Figure 16: Open Table Dialog



StP/UML Class Table Editor – Open Table

Filter

*

Tables

Message

Selection

TextInput

Version

3. Click **OK**.
   The dialog is dismissed and a new class table named *TextInput* appears in the editor window. The empty cell at the top of the class table is highlighted.
4. Type the name of the class, `TextInput`, in the highlighted cell.

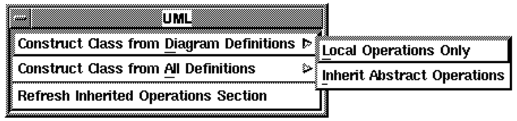
5. Use the Hide/Show dialog to display the Analysis Items section of the table.
   If necessary, refer to "Hiding and Showing Class Table Sections" on page 5-3.
6. Choose **Save** from the File menu.

Constructing a Table from the Repository

To construct a class table from information in a class diagram:

1. Display the **Construct Class from Diagram Definitions** submenu of the UML menu.

Figure 17: Construct Class from Diagram Definitions Submenu



StP gives you the option to include the class's inherited operations in addition to its local operations. Since the *TextInput* class does not inherit operations, you do not need this option.

2. Choose **Local Operations Only**.
3. The *enter_text* operation appears in the Operation column.
4. Use the column scroll bar below the Analysis Items section of the table to display the **Abstract?** analysis item.
5. Type `True` in the empty cell below the Abstract? item.
6. Choose **Save** from the File menu.

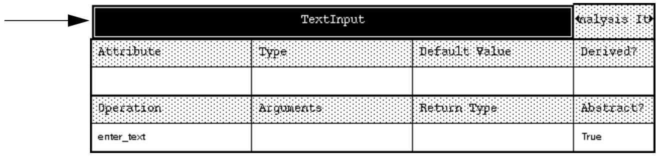
Navigating to the Class Editor

Now that you have completed the *Message* and *TextInput* class tables, return to the Class Editor.

To navigate to the Class Editor from the CTE:

1. Select the *TextInput* cell at the top of the class table.

Figure 18: Selected TextInput Cell



TextInput			nalysis It
Attribute	Type	Default Value	Derived?
Operation	Arguments	Return Type	Abstract?
enter_text			True

2. Choose **Class Diagram** from the Navigation menu.

   The *Application_1* class diagram appears with the *TextInput* symbol selected.

   In the class diagram, a rectangular display mark appears above classes with corresponding class tables.

Figure 19: Class Table Display Mark

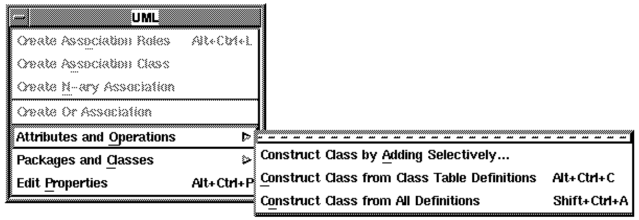
Updating a Class Diagram

Since you added information about attributes and operations using the CTE, you may wish to update your class diagram to reflect that new information.

To update a class on a diagram:

1. Select the *Message* class.
2. Display the Attributes and Operations submenu of the UML menu.

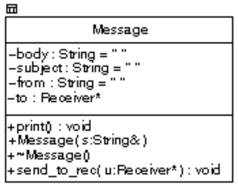
Figure 20: The Attributes and Operations Submenu



3. Choose **Construct Class from Class Table Definitions**.

   The class symbol is updated to reflect the information in the class table. The *Message* class in your diagram should resemble Figure 21.

Figure 21: Updating the Class Diagram



4. Choose **Save** from the File menu.

Exiting the CTE

You do not use the CTE again in this tutorial, so exit it now.

Keep your class diagram open. In the next section you check the diagram's syntax and semantics. In the chapters that follow, you use the class diagram to navigate to other StP editors.

Checking the Class Diagram

You have now completed the class diagram for this tutorial. Before moving to the next chapter, check the diagram's syntax and semantics.

Checking Syntax

Checking the diagram syntax validates that the drawing is correct. StP/UML checks a diagram's syntax whenever you choose **Save** from the File menu. However, you can perform a syntax check at any time by choosing **Check Diagram Syntax** from the Repository menu. Syntax checks include:

- Are labels missing from the diagram?
- Are the links legal for the symbol types?
- Are required links present?
- Does any package have the same name as a class?
- Do all operation symbols have valid labels?

When the syntax check is completed, the message "No errors found in the diagram" appears in the message area of the Class Editor window.

Checking Semantics

Checking the diagram's semantics validates that the underlying model, as reflected in repository data from all editors and diagrams, is correct. A semantics check ensures consistency for all modeling constructs across the entire model.

Semantics checks include:

- Is each class or package a member of no more than one package?
- Are there any duplicate attribute names in the same scope?
- Are any aggregation parent classes incorrectly stereotyped as "type", "utility" or "metaclass"?
- Do only parameterized classes have class parameters?

- Do the actual parameters on each refines link match the formal parameters of the parameterized class in type and number?
- Does each refines link have the stereotype "bind"?

To check the semantics of the diagram, choose **Check Diagram Semantics** from the Repository menu.

When the semantics check is completed, the message "Check Diagram Semantics completed" appears in the message area of the Class Editor window.

Errors in the Diagram

If there are errors in the syntax of the diagram, the Message Log appears listing the errors. Correct the errors and check the diagram again.

For more information about the Message Log, see *Fundamentals of StP*.

Continuing the Tutorial

In this chapter, you:

- Navigated to the Class Table Editor from the Class Editor, and back
- Created class tables for the *Message* and *TextInput* classes
- Updated the *Application_1* class diagram based on the information added to the *Message* class table
- Checked the syntax and semantics of the class diagram

In the next chapter, you create a state diagram.

6 Creating a State Diagram

In this chapter you create a state diagram. State diagrams show the behavior of a single class over time.

About the State Editor

You use the State Editor to create state diagrams. In a complete model, each class in the class diagram would have a corresponding state diagram. For this tutorial, you create a state diagram for the *Message* class.

For a complete description of the State Editor, see *Creating UML Models*.

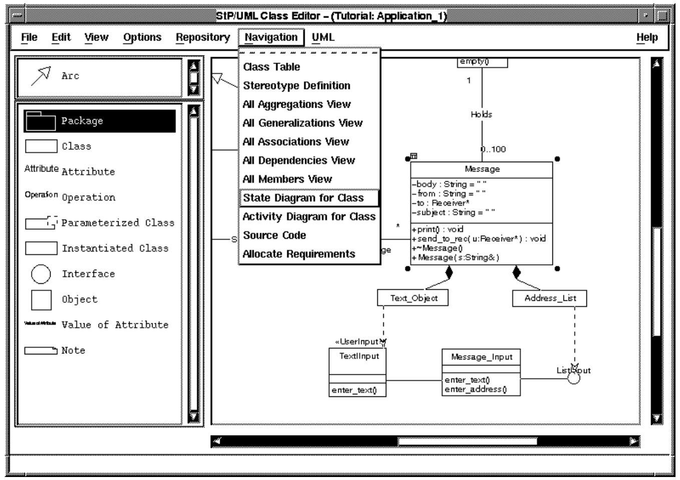
Navigating to the State Editor

To start the State Editor, you navigate from the *Message* class in the *Application_1* class diagram.

To navigate to the State Editor:

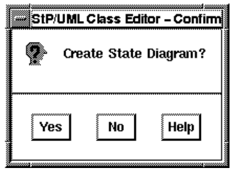
1. Select the *Message* class symbol on your class diagram.
   Selection handles appear on the symbol.
2. Choose **State Diagram for Class** from the Navigation menu, as shown in Figure 1.

Figure 1: Navigating to the State Editor



Because no state diagram for *Message* exists, a dialog box appears prompting you to create one.

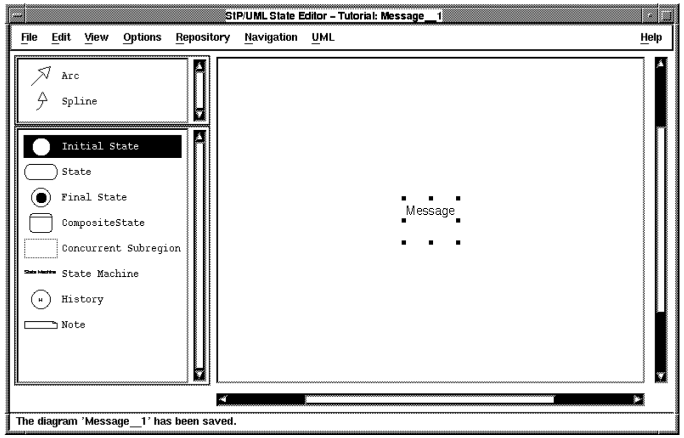
Figure 2: Create State Diagram Dialog



Create State Diagram?

3. Click **Yes**.
   The State Editor appears with a flashing context symbol. The context symbol (called a state machine) indicates that the diagram represents a state machine for the *Message* class.

Figure 3: The State Editor



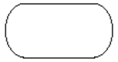
The diagram 'Message__1' has been saved.

4. Move the state machine symbol to the top left corner of the drawing area to make room for your diagram.

Modeling States

The fundamental symbol in a state diagram is the state, which is the condition of an object as determined by its attribute values and links. A state can also be thought of as an object's response to an event.

Figure 4: State Symbol



About Initial States and Final States

An initial state indicates the creation of an object. A final state indicates the ultimate condition of the object. Initial state and final state symbols are used in the state diagram for the *Message* class to indicate that any object of this class has a beginning and an end.

Figure 5: Initial State and Final State Symbols

State Labels

A state is usually labeled with a short descriptive name. Initial states and final states do not require labels.

The states for the *Message* class are:

- Initial state (unlabeled)
- *Sender known*
- *Receiver known*
- *Body complete*
- *Message in receiver mailbox*
- Final state (unlabeled)

Inserting States

Begin your state diagram by inserting and labeling the states of the *Message* class.

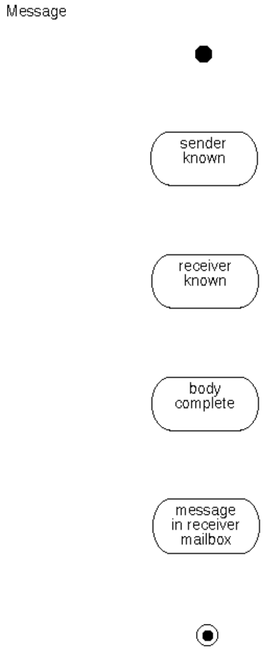
To insert and label states:

1. Drag an initial state symbol from the Symbol List to the top of the diagram.
2. Insert a state symbol below the initial state symbol.

3. Label the state **`sender known`**.
4. Insert and label the remaining state symbols, placing each symbol below the previous.
5. Insert a final state symbol below the *message in receiver mailbox* state.

Your diagram should resemble Figure 6.

Figure 6: Inserting and Labeling States



Modeling State Transitions

An occurrence that triggers a state transition is called an event. An event usually corresponds to:

- A signal
- A call to an operation
- A condition becoming true

Typically, an action is performed during the state transition. An action corresponds to a class operation.

The transition link between two states is labeled with the event that causes the transition and the action performed during the transition, with the two names separated by a slash (/). A transition label may also include guard conditions and send clauses.

Table 1 lists the transition links for the *Message* state diagram.

Table 1: State Transition Links for Message

State Transition Link	Description
Request message creation/Message	The request to create a message is the event that causes the transition between the initial state and the *sender known* state. The action performed is the creation of the message (*Message* constructor).
Request address/enter_address	The request for a message address is the event that causes the transition between the *sender known* state and the *receiver known* state. The action performed is the collection of the message address (*enter_address*).
Request text/ enter_text	The request for the text of the message is the event that causes the transition between the *receiver known* state and the *body complete* state. The action performed is the collection of the text of the message (*enter_text*).

Table 1: State Transition Links for Message (Continued)

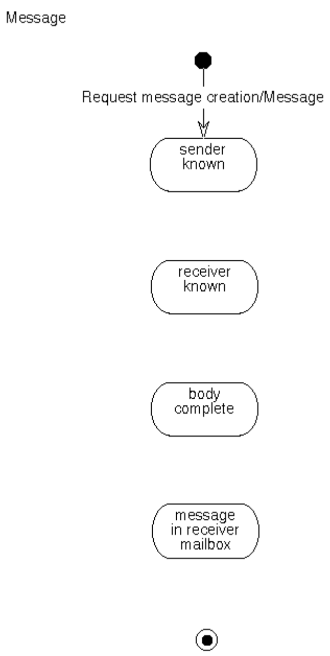
State Transition Link	Description
Request delivery/send_to_rec	The request to deliver the message is the event that causes the transition between the *body complete* state and the *message in receiver mailbox* state. The action performed is the delivery of the message (*send_to_rec*).
Delete message/~Message	The request to delete the message is the event that causes the transition between the *message in receiver mailbox* state and the final state. The action performed is to delete the message (*~Message* destructor).

Drawing and Labeling State Transition Links

To draw and label state transition links:

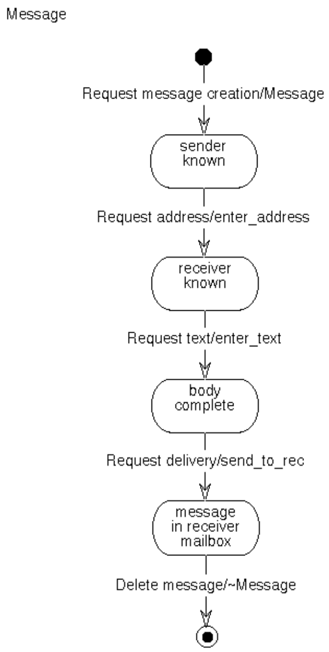
1. Insert an arc from the initial state to the *sender known* state.
   Selection handles appear on the arc.
2. Type **`Request message/Message`**.
   Your diagram should resemble Figure 7.

Figure 7: Labeling State Transition Links



3. Continue to insert and label the state transition links described in Table 1.
   When you are finished, your diagram should resemble Figure 8.

Figure 8: Completed State Transition Links



4. Choose **Save** from the File menu.

Decomposing a State

Decomposing a state creates a lower-level state diagram where you can show more detail about the state. You can decompose a state in a state diagram to several levels, thereby creating a hierarchy of decompositions.

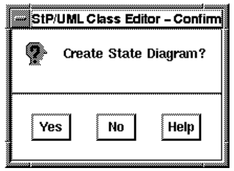
The *message in receiver mailbox* state has two substates: *message unread* and *message read*. To model these substates, you navigate from the *message in receiver mailbox* state to a lower-level diagram.

Navigating to a Lower-Level Diagram

To navigate to a lower-level diagram:

1. Select the *message in receiver mailbox* state symbol.
   Selection handles appear on the symbol.
2. Choose **Refine** from the Navigation menu.
   A dialog box appears prompting you to create a state diagram.

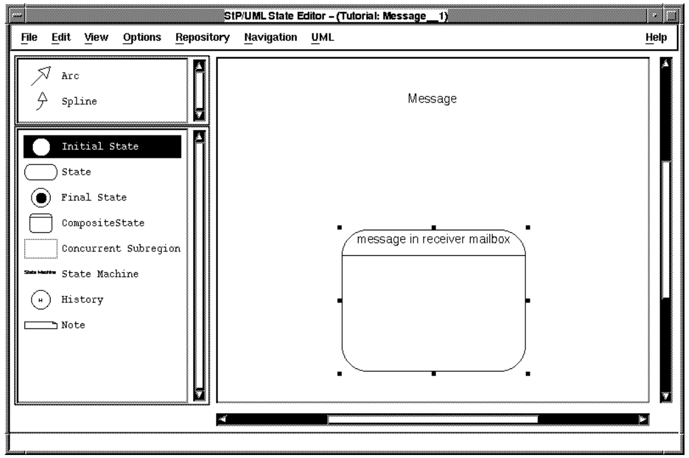
   Figure 9: Create State Diagram Dialog

   Create State Diagram?

3. Click **Yes**.
   A new diagram named *Message_1* appears in the State Editor window. The diagram contains a composite state symbol representing the state from which you navigated. The state machine symbol from the top-level diagram also appears in the decomposed diagram.

Figure 10: The State Editor with a Decomposition Diagram



4. Resize the composite state so that it can accommodate the *message in receiver mailbox* substates.

Modeling Substates

The decomposition diagram for *message in receiver mailbox* includes an initial state, a final state, and the substates *message unread* and *message read*.

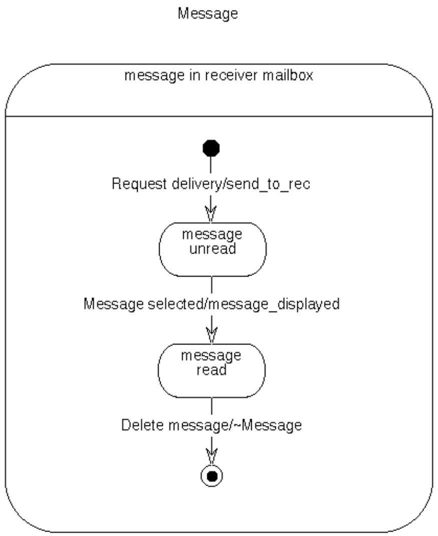
Table 2 lists the transition links for the diagram.

Table 2: State Transition Links for Message_1

State Transition Link	Description
Request delivery/send_to_rec	The request to deliver the message is the event that causes the transition between the initial state and the *message unread* state. The action performed is the delivery of the message (*send_to_rec*).
Message selected/message_displayed	The selection of a message in the mailbox is the event that causes the transition between the *message unread* state and the *message read* state. The action performed is the display of the message (*message_displayed*).
Delete message/~Message	The request to delete the message is the event that causes the transition between the *message read* state and the final state. The action performed is to delete the message (~*Message* destructor).

Draw the necessary states and transition links inside the composite state symbol. When you are finished, your diagram should resemble Figure 11.

Figure 11: State Diagram for Message In Receiver Mailbox



You have now completed the state diagrams for the *Message* class. Save your diagram and exit the State Editor.

Continuing the Tutorial

In this chapter you:

- Navigated to the State Editor from the Class Editor
- Created a state diagram
- Decomposed a state

In the next chapter, you create an activity diagram.

7 Creating an Activity Diagram

In this chapter, you create an activity diagram. An activity diagram shows how a task is performed by breaking it down into a series of actions and decisions.

You can use activity diagrams to model an operation, a use case, or a behavior spanning a set of use cases. In this tutorial, you create an activity diagram for the *send_msg* operation.

About the Activity Editor

You use the Activity Editor to create activity diagrams. For a complete description of the Activity Editor, see *Creating UML Models*.

Navigating to the Activity Editor

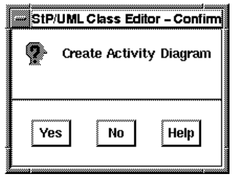
To start the Activity Editor, you navigate from *send_msg* operation in the *Application_1* class diagram.

To navigate to the Activity Editor:

1. Select the *send_msg* operation in the *User* class on your class diagram.
2. Choose **Activity Diagram for Operation** from the Navigation menu.

   Because no activity diagram for *send_msg* exists, a dialog box appears prompting you to create one.

Figure 1: Create Activity Diagram Dialog

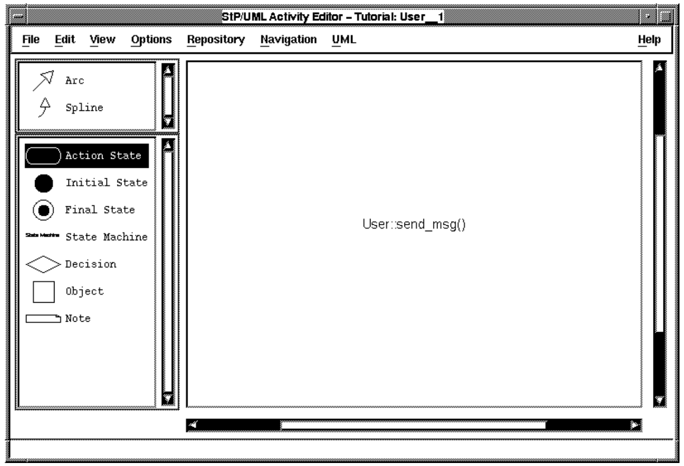


Create Activity Diagram

3. Click **Yes**.

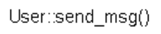
   The Activity Editor appears with a state machine context symbol in the center of the drawing area. The new diagram is named *User_1*, as shown in the editor's title bar.

Figure 2: The Activity Editor



The state machine symbol shows the name of the operation's class followed by the name of the operation.

Figure 3: **State Machine for Operation**



Move the state machine symbol to the upper left corner of the drawing area.

Modeling Action States

The fundamental symbol in an activity diagram is the action state. An action state represents an activity which, when completed, triggers a transition to another action state. Action states correspond to steps in a procedure.

Consider the steps required to send a message using the Email system. The user makes a request to send a message, then supplies an email address. The system checks the address to make sure that it is valid. If the address is invalid, the system displays an error message; if the address is valid, the user can supply—in any order—a subject line and the text of the message. Once the subject line and text have been entered, the user can either send the message or cancel the operation.

Inserting Action States

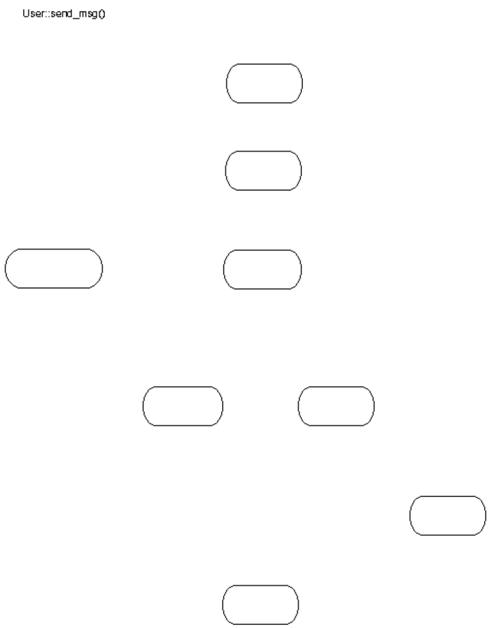
You begin your activity diagram by inserting action state symbols for the steps in the *send_msg* operation.

To model action states:

1. Select the action state symbol in the Symbol List.
   The symbol is highlighted
2. Move the pointer into the drawing area.
3. Use the Alt+Ctrl accelerator, described on page 3-8, to insert eight action state symbols.
4. Reposition the symbols so that your diagram resembles Figure 4.

You may need to use the Zoom features to enlarge the drawing area. If necessary, refer to “Using Zoom” on page 3-9.

Figure 4: **Inserting Action States**



Labeling Action States

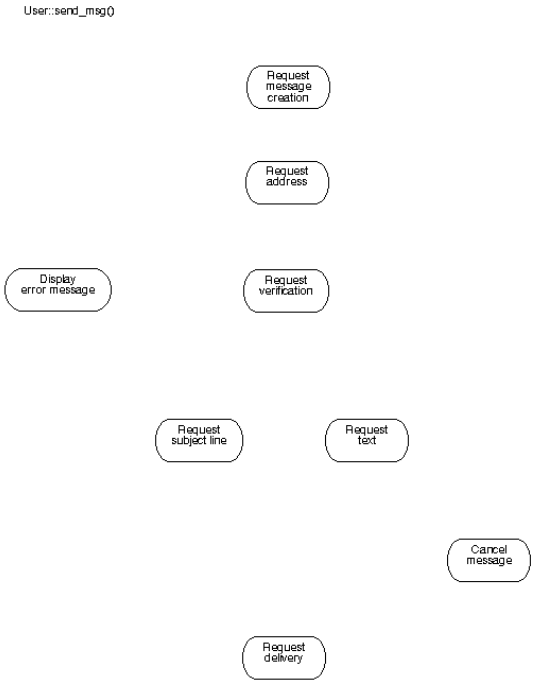
Action state labels are called action-expressions. Action expressions can consist of code, pseudocode, or natural language. This tutorial uses the following labels for the *send_msg* action states:

- Request message creation
- Request address
- Request verification
- Display error message
- Request subject line
- Request text
- Cancel message
- Request delivery

To label action state symbols:

1. Select the top-most action state symbol on your diagram and type **`Request message creation`**.
2. Label the remaining action state symbols so that your diagram resembles Figure 5.

Figure 5: **Labeled Action States**



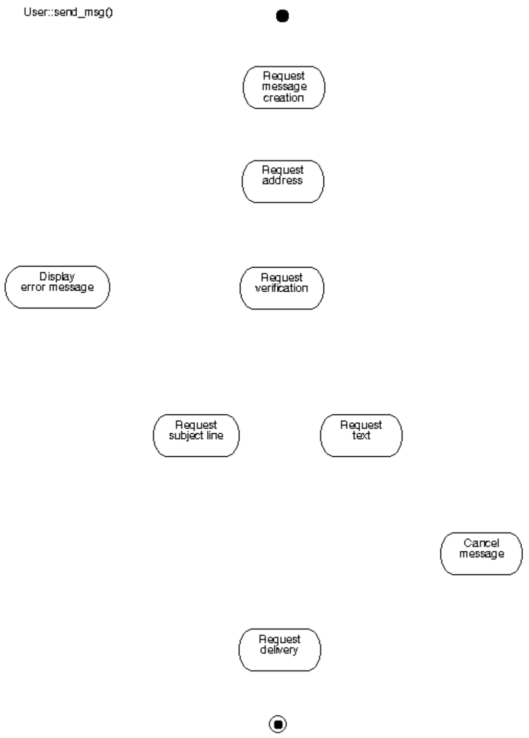
Modeling Initial States and Final States

Initial state and final state symbols are used in the activity diagram to indicate the start and completion of the task.

To insert initial and final states:

1. Drag an initial state symbol from the Symbol List and place it above the *Request message creation* action state.
2. Insert a final state symbol below the *Request delivery* action state.
   Your diagram should resemble Figure 6.

Figure 6: Inserting Initial and Final States



Modeling Transitions

Completion of the activity represented by an action state triggers a transition. The transition can be a simple, sequential transition to another action state, or it can be a more complex transition involving decisions or concurrent activities.

You learn about the more complex transitions later in the chapter. For now, review the *send_msg* description on page 7-3 and try to identify the sequential transitions in the *send_msg* operation. If you need help, refer to Table 1.

Table 1: Transition Links for the User_1 Activity Diagram

From Action State	To Action State
Initial State	Request message creation
Request message creation	Request address
Request address	Request verification
Request delivery	Final state

Drawing Transition Links

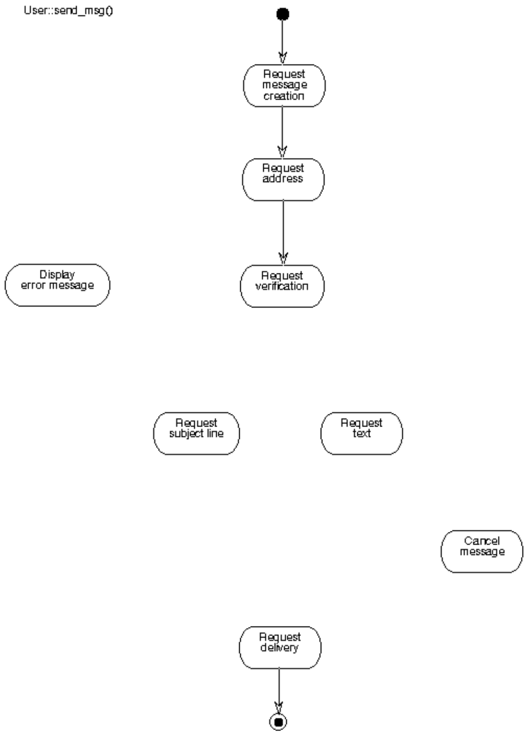
You model the transition from one action state to another with a transition link.

To draw transition links:

1. Drag an arc into the initial state symbol and release the mouse button.
   This action attaches one end of the transition link to the initial state symbol.
2. Move the pointer into the *Request message creation* action state and click the left mouse button.
   This action attaches the opposite end of the arc to the action state.
3. Continue to insert the transition links described in Table 1.

When you are finished, your diagram should resemble Figure 7.

Figure 7: Transition Links



Modeling Non-Sequential Activities

In modeling the execution of a task, you may need to show behavior that is not sequential. Some activities may occur simultaneously; others may have no specified order. For example, a user can enter a subject line and message text in any order.

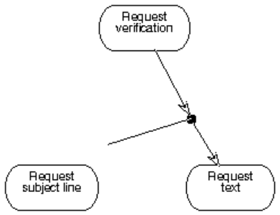
You use a split flow to model concurrent activities and activities that can be performed in any order.

Splitting a Flow

To draw a split flow:

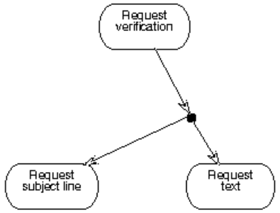
1. Insert a transition link from the *Request verification* action state to the *Request text* action state.
   Selection handles appear on the arc.
2. Choose **Split Control** from the UML menu.
   A synchronization vertex appears with an outgoing flow, which is attached to the pointer.

Figure 8: Synchronization Vertex with Outgoing Flow



3. Drag the flow into the *Request subject line* action state and click the mouse button.
   Your diagram should resemble Figure 9.

Figure 9: Split Flow



You can reposition a synchronization vertex by selecting and dragging it to the desired position. Reposition the vertex now, centering it beneath the *Request verification* state.

To delete a synchronization vertex, select it and choose **Delete** from the Edit menu.

Merging a Flow

Once the user has supplied the subject line and text, the message can be sent. The message cannot be sent until both of these activities have occurred.

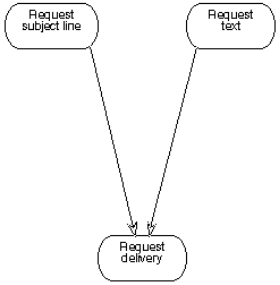
You use a merged flow to show that activities must be synchronized before the next step of a task can be taken.

To draw a merged flow:

1. Insert transition links from the *Request subject line* and *Request text* action states to the *Request delivery* action state.

   Your diagram should resemble Figure 10.

Figure 10: Transition Arcs



2. Select both transition arcs.

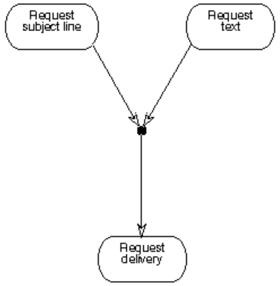
   You use your Shift key to make multiple selections. For information, refer to "Selecting a Group of Symbols" on page 3-14.

3. Choose **Merge Control** from the UML menu.

   The two arcs are merged at a synchronization vertex, with an outgoing flow pointing to the *Request delivery* action state.

   Your diagram should resemble Figure 11.

Figure 11: Merged Flow



Modeling Decisions

The execution of a task can involve a conditional construct, or decision, in which one of a set of actions is taken based on a rule or test.

There are two ways to model decisions in an activity diagram. You can draw a separate transition link to each potential action state, or you can insert a diamond-shaped decision symbol and draw links from the symbol to the potential states.

In either case, you label the transition arcs with the condition that must be satisfied in order for the transition to occur. These labels are called guard conditions.

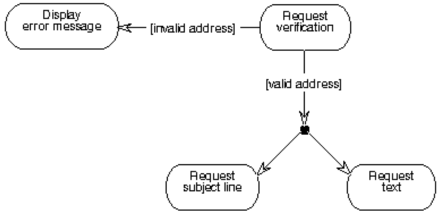
Modeling a Decision with Separate Links

When the user enters an email address, the Email system checks to make sure the address is valid. If the address is valid, the message can be sent. If the address is invalid, the system displays an error message. You model this decision using separate transition links.

To model a decision:

1. Insert a transition link from the *Request verification* action state to the *Display error message* action state.
   Selection handles appear on the arc.
2. Type `[invalid address]`.
3. Select the transition link between *Request verification* and the synchronization vertex.
   Selection handles appear on the arc.
4. Type `[valid address]`.
   Your diagram should resemble Figure 12.

Figure 12: Decision with Guard Conditions



Using a Decision Symbol

A user can choose to cancel a message rather than send it. You use a decision symbol to model this choice.

You will need to reposition the transition link leading from the synchronization vertex to the *Request delivery* action state. The procedure below works the same way in all diagram editors.

Repositioning an Arc

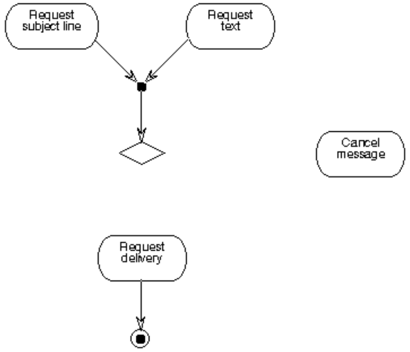
To reposition an arc:

1. Drag a decision symbol from the Symbol List to the drawing area and place it on the transition link between the *Request delivery* action state and the synchronization vertex.
2. Select the transition link.
   Selection handles appear on the transition link.
3. Position the pointer over the selection handle nearest the *Request delivery* action state.
   The pointer changes to a crosshair.
4. Press the left mouse button and drag the handle into the decision symbol.

5. Release the mouse button.
6. The arc now leads from the synchronization vertex to the decision symbol.

   Your diagram should resemble Figure 13.

Figure 13: Repositioned Arc



Modeling the Decision

To model a decision using a decision symbol:

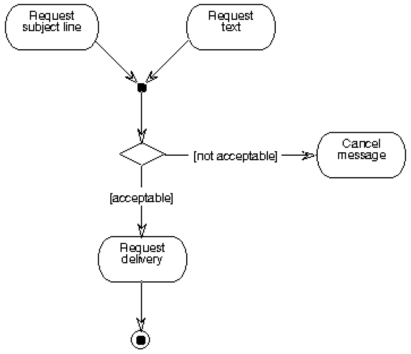
1. Insert a transition link from the decision symbol to the *Request delivery* action state.

   Selection handles appear on the arc.
2. Type **`[acceptable]`**.
3. Insert a transition link from the decision symbol to the *Cancel message* action state.

   Selection handles appear on the arc.
4. Type **`[not acceptable]`**.

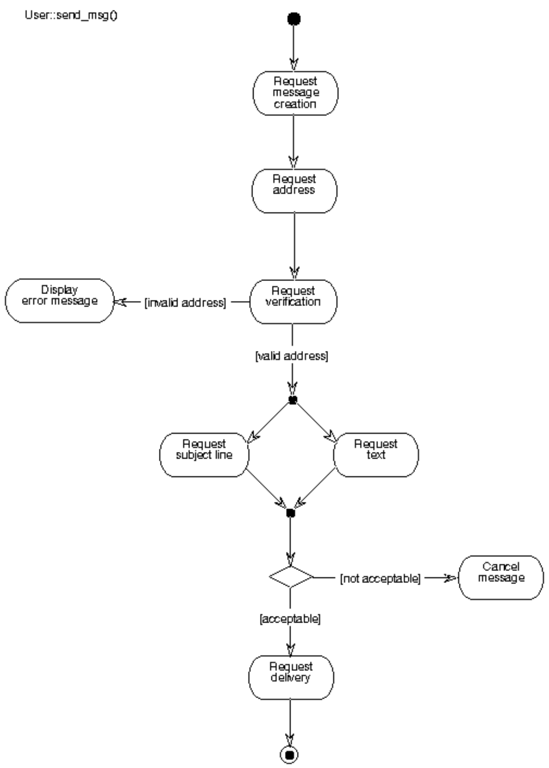
   Your diagram should resemble Figure 14.

Figure 14: Decision Symbol and Guard Conditions



You have now completed the activity diagram for the *send_msg* operation. Your diagram should resemble Figure 15.

Figure 15: **Completed Activity Diagram**



When you are satisfied with your diagram, choose **Save** from the File menu.

Exiting the Editors

Choose **Exit** from the File menu to exit the Activity Editor.

You can also exit the Class Editor at this time.

Continuing the Tutorial

In this chapter, you:

- Navigated to the Activity Editor from the Class Editor
- Created an activity diagram for the *send_msg* operation

In the next chapter, you create sequence and collaboration diagrams.

8

Creating Interaction Diagrams

Interaction diagrams show how objects interact through message passing to accomplish a task. The two types of interaction diagrams—sequence and collaboration—show different views of object interaction. Sequence diagrams emphasize the order of message passing, while collaboration diagrams emphasize the structure of the interaction.

As an analysis tool, interaction diagrams allow you to capture the different ways in which your system might implement a particular use case. For example, the Email system implements the *Send_Message* use case differently in each of these situations, or “scenarios”:

- A message is successfully sent to a valid user
- A message is sent to an invalid user
- A message is sent to a full mailbox
- A message is cancelled

In this tutorial, you create a sequence and collaboration diagram to model the scenario of sending a message to a valid user.

About the Sequence and Collaboration Editors

You use the Sequence Editor to create sequence diagrams and the Collaboration Editor to create collaboration diagrams. You can create:

- A sequence diagram, and then have StP/UML create an equivalent collaboration diagram
- A collaboration diagram, and then have StP/UML create an equivalent sequence diagram

- A sequence diagram that does not have an equivalent collaboration diagram
- A collaboration diagram that does not have an equivalent sequence diagram

For this tutorial, you create a sequence diagram, and then you have StP/UML create an equivalent collaboration diagram.

Navigating to the Sequence Editor

You begin the sequence diagram by navigating to the Sequence Editor from the Use Case Editor. During the navigation, StP/UML sets the sequence diagram’s context, and you set the scenario name.

Setting the Diagram Context

The context of a sequence diagram is the use case it describes. When you navigate to a sequence diagram from the Use Case Editor, StP/UML automatically sets the context of the diagram to the use case you navigated from. In this example, you navigate from the *Send_Message* use case to the sequence diagram, so the diagram’s context is the *Send_Message* use case.

Setting the Scenario Name

A sequence diagram represents a single scenario, which describes the interactions between objects in a particular situation. In this example, the diagram’s scenario name is *user valid*.

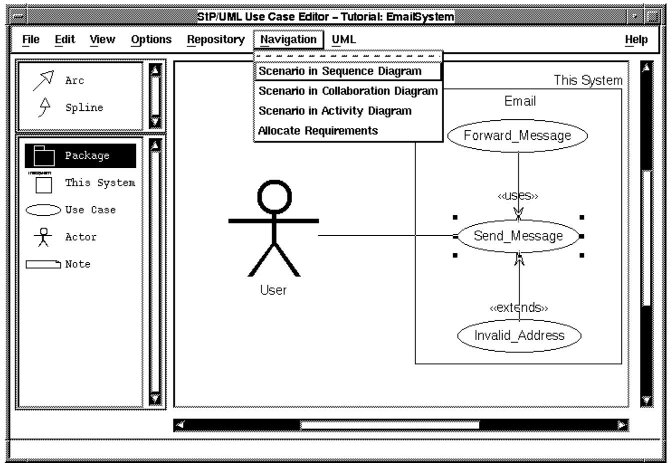
Navigating to the Sequence Editor

To start the Sequence Editor, you navigate from *EmailSystem*, the use case diagram you completed in Chapter 2.

To navigate to the Sequence Editor from the Use Case Editor:

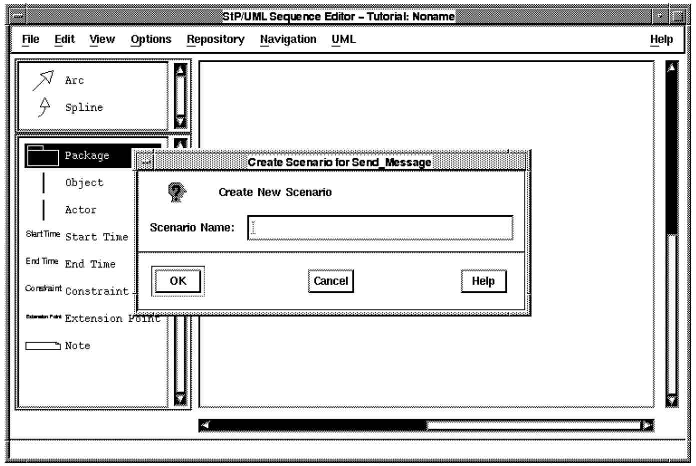
1. Select Use Case Diagrams from the Model Category on the StP Desktop.
2. Select *EmailSystem* from the Use Case Diagrams list and choose **Edit Diagram** from the Commands menu.
   The Use Case Editor appears with the *EmailSystem* diagram opened.
3. Select the *Send_Message* use case symbol.
4. Choose **Scenario in Sequence Diagram** from the Navigation menu, as shown in Figure 1.

Figure 1: Navigating to a Sequence Diagram



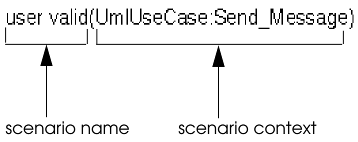
The Sequence Editor and the Create New Scenario dialog appear.

Figure 2: The Create New Scenario Dialog Box



5. Move the cursor to the Scenario Name text field on the Create New Scenario dialog and type **`user valid`**.
6. Click **OK**.
   The dialog disappears, and a context symbol appears in the editor window.

Figure 3: Sequence Diagram Context Symbol



The sequence diagram is named *Send_Message_1*, as shown in the diagram's title bar.

Setting Display Mark Options

In the remainder of this chapter, you add information to your sequence diagram that has associated display marks.

To set display mark options:

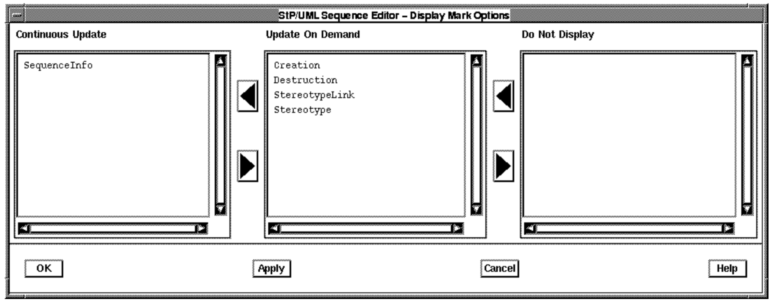
1. Choose **Display Marks** from the Options menu.

   The Display Mark Options dialog appears.

2. Ensure that the SequenceInfo display mark is in the Continuous Update window.
3. Move the remaining display marks into the Update On Demand window.

   The Display Mark Options dialog should resemble the one shown in Figure 4.

Figure 4: The Display Mark Options Dialog



StP/UML Sequence Editor – Display Mark Options

Continuous Update
SequenceInfo

Update On Demand
Creation
Destruction
StereotypeLink
Stereotype

Do Not Display

4. Click **OK** to apply the change and dismiss the dialog.

Creating a Sequence Diagram

Sequence diagrams show the messages that pass between objects as they perform a task. A sequence diagram consists of vertically-oriented object symbols connected by horizontal message arcs.

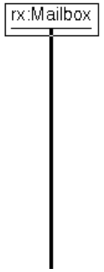
Modeling Objects

An object is an individual instance of a class. Your sequence diagram requires the objects from the following five classes:

- *User*, to send the message
- *Receiver*, to receive the message
- *Mailbox*, to store the message
- *Message*, to hold the address and body of a message
- *MailSystem*, to store the user and conference information

In a sequence diagram, an object is represented as a rectangle with a vertical line beneath it. The vertical line is called a “lifeline,” and it represents the object’s existence in time.

Figure 5: Object Symbol

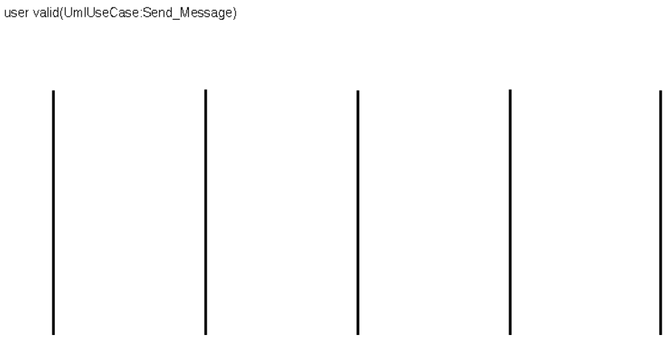


Inserting Object Symbols

Begin the sequence diagram by inserting object symbols:

1. Select the object symbol in the Symbol List.
   The symbol is highlighted.
2. Move the pointer into the drawing area.
3. Use the Alt+Ctrl accelerator, described on page 3-8, to insert five object symbols.
   If necessary, use the Zoom feature to provide more space for inserting objects. For instructions on using the Zoom feature, refer to "Using Zoom" on page 3-9.
4. Choose **Align All Objects** from the UML menu.
   The object symbols are automatically aligned. Your diagram should resemble Figure 6

Figure 6: Inserting Object Symbols



About Object Labels

An object label includes both the name of the object and the name of its class. The syntax for an object label is:

[Object Name][:Class Name]

Object labels are always underlined.

Objects can have a full or partial label. For example, they can:

- Have a fully qualified name, as in *TheSystem:MailSystem*
- Be an anonymous object of a class, as in *:User*
- Be an object of an anonymous class, as in *CollIterator*

In this tutorial, three objects have fully qualified names, and the remainder are anonymous objects of a class.

Labeling Object Symbols

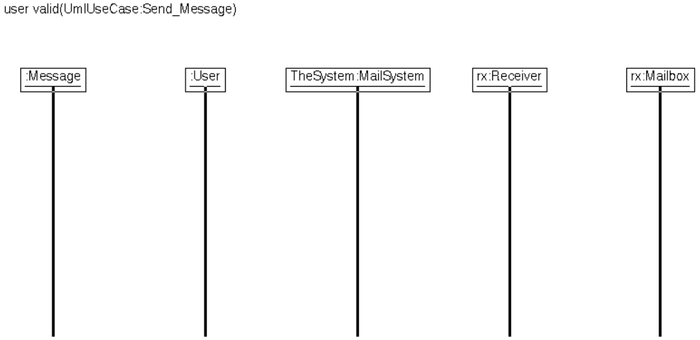
The names of the five objects in your sequence diagram are, from left to right:

- *:Message*
- *:User*
- *TheSystem:MailSystem*
- *rx:Receiver*
- *rx:Mailbox*

To label object symbols:

1. Select the object symbol on the left.
2. Type the label `:Message`.
3. Click the left mouse button to terminate the label-editing mode.
   A box appears around the object label, and the label is underlined.
4. Label the remaining objects on your diagram.
   When you are finished, your diagram should resemble Figure 7.

Figure 7: Labeling Object Symbols



Expanding Object Lifelines

You will need more vertical space in your diagram to fit all of the messages that are passed in the *user valid* scenario. Using one of the following procedures, lengthen the lifelines of the objects in your diagram.

To expand object lifelines by half, choose **Expand All Objects 50%** from the UML menu.

To expand object lifelines to a specified length:

1. Select an object and resize its lifeline to the desired length.
2. With the object still selected, choose **Expand All Objects To Selected Size** from the UML menu.

   All of the objects in the diagram resize to the length of the selected object.

Modeling Messages

Objects interact with each other through messages. One object, called the client, sends a message requesting a service to another object, called the supplier.

A message is represented by an arc drawn from the client object to the supplier object. In a sequence diagram, messages are shown in the order in which they occur, with the first message at the top of the diagram and the last message at the bottom.

Table 1 lists the messages that pass between the objects in your sequence diagram. The messages are listed in the order in which they occur.

Table 1: Messages in the Send_Message_1 Sequence Diagram

Client Object	Supplier Object	Message
:User	:User	send_msg
:User	:Message	Message
:User	:Message	enter_address
:User	TheSystem:MailSystem	find_user
:User	:Message	enter_text
:User	:Message	send_to_rec
:Message	rx:Receiver	receive_msg
rx:Receiver	rx:Mailbox	add_msg

Drawing a Message from an Object to Itself

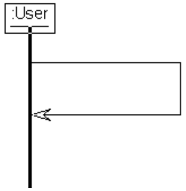
In your sequence diagram, the *:User* object sends a message to itself. To draw this message, you use two vertexes. A vertex is an angle in an arc.

To insert a message from *:User* to itself:

1. Drag an arc symbol into the drawing area and drop it on the *:User* object, directly under the object's label.

2. Drag the arc to the right and click the left mouse button.
   A vertex is created.
3. Drag the arc down and click the mouse button.
   A second vertex is created.
4. Drag the arc back to the *:User* object and click the mouse button.
   The completed arc should resemble Figure 8.

Figure 8: Message Arc with Vertexes



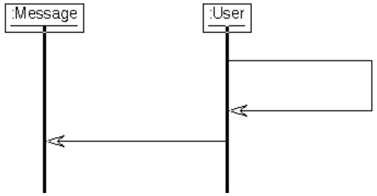
You can reposition a vertex by selecting and dragging it to the desired position. To delete a vertex, select it and choose **Delete** from the Edit menu.

Drawing Messages Between Objects

To draw messages between objects:

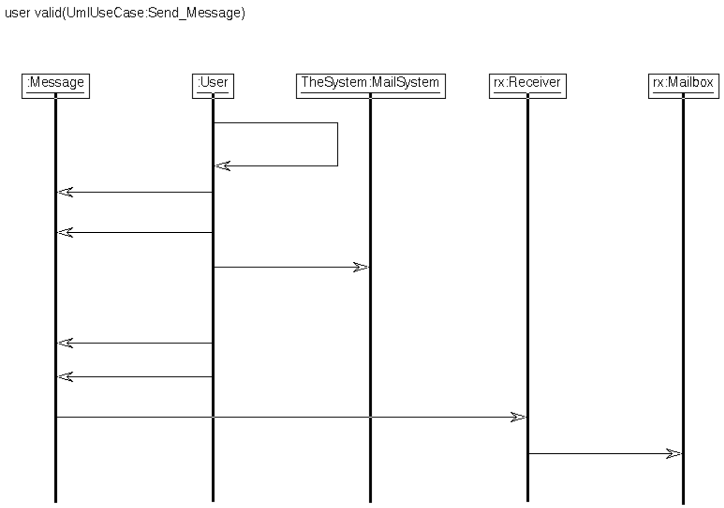
1. Drag an arc symbol into the drawing area, and drop it on the *:User* object below the arc you just inserted.
2. Drag the arc to the *:Message* object and click the left mouse button.
   Your diagram should resemble Figure 9.

Figure 9: Message between Objects



3. Insert the remaining message arcs, using Table 1 as a guide.
   When you are done, your diagram should resemble Figure 10.

Figure 10: Inserting Message Arcs



Labeling Messages

A message is labeled with the name of an operation or signal. A message label may also include:

- Argument values
- The message return type
- A message sequence number
- A predecessor guard condition
- Message recurrence information

In this tutorial you label the messages with operation names and sequence numbers. For information about other message label components, see *Creating UML Models*.

You can add a label directly to an arc, or you can add it using the Properties dialog box.

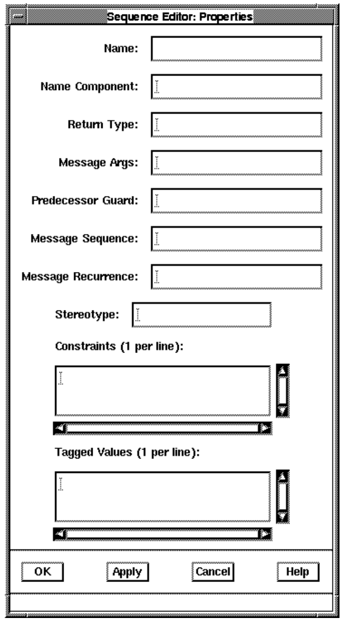
To label messages:

1. Select the bottom segment of the message arc leading from *:User* back to itself.

   Selection handles appear on the arc.
2. Type `send_msg`.
3. Select the topmost message arc between *:User* and *:Message*.
4. Choose **Edit Properties** from the UML menu.

   The Properties dialog box appears.

Figure 11: The Properties Dialog for Messages

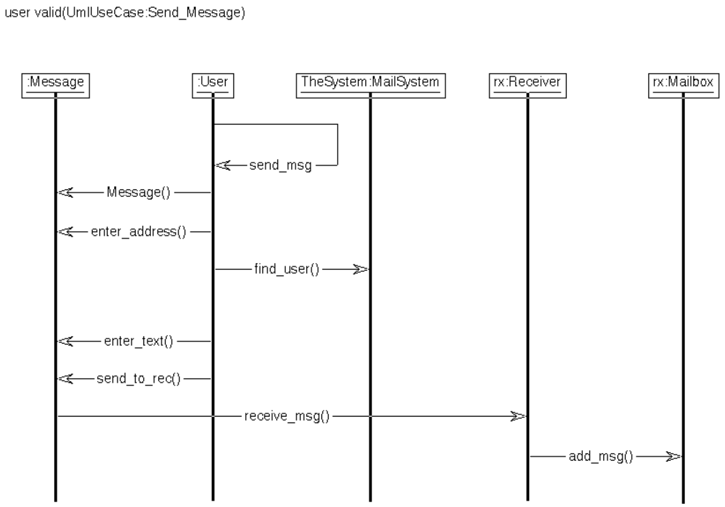


5. Type `Message` in the Name Component text field.
6. Click **Apply**.

   The name appears on the message arc in the diagram.
7. Add the remaining message labels, using Table 1 as a guide.

   When you are finished, your diagram should resemble Figure 12.

Figure 12: Labeling Messages



Numbering Messages

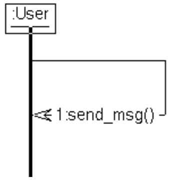
Message sequence numbers are optional on a sequence diagram, since the top-down position of the messages indicates their order. However, numbering your messages can be helpful if your diagram is complex or if you plan to generate a collaboration diagram from your sequence diagram.

In this tutorial, you number messages so that information about message sequence carries over into your collaboration diagram.

To add sequence numbers to messages:

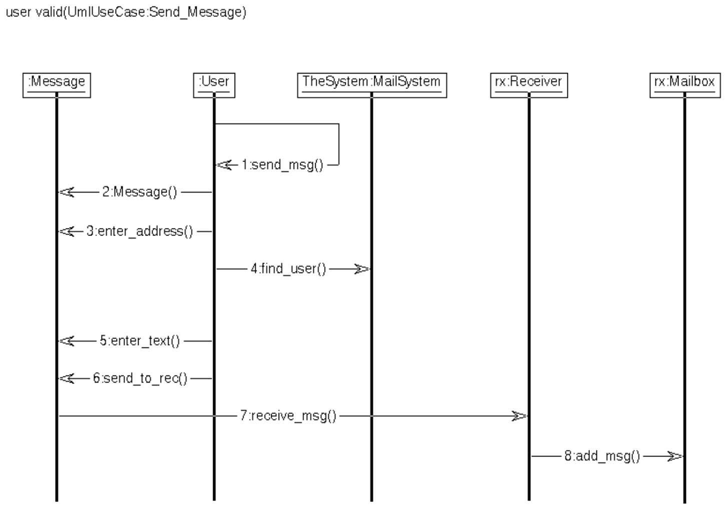
1. Select the *send_msg* message.
   Selection handles appear on the message.
2. Choose **Edit Properties** from the UML menu.
   The Properties dialog box appears.
3. Type **1** in the Message Sequence text field.
4. Click **Apply**.
   The sequence number is added to the message label.

Figure 13: Message Label with Sequence Number



5. Add sequence numbers to the remaining messages.
   When you are finished, your diagram should resemble Figure 14.

Figure 14: Adding Sequence Numbers



Annotating Messages

A message can have various effects on its target or source object. For example, a message can:

- Activate its target object
- Deactivate its source object
- Create its target object
- Destroy its source object

This information is attached to messages in the form of annotations. In some cases, StP adds the annotations automatically. In other cases, you add the annotations using the Object Annotation Editor (OAE).

Showing Activation and Return

An activation is the period of time in which a supplier object performs an action. Activations are initiated by incoming messages and end with a return message to the client object. All messages on a sequence diagram either initiate or end activations.

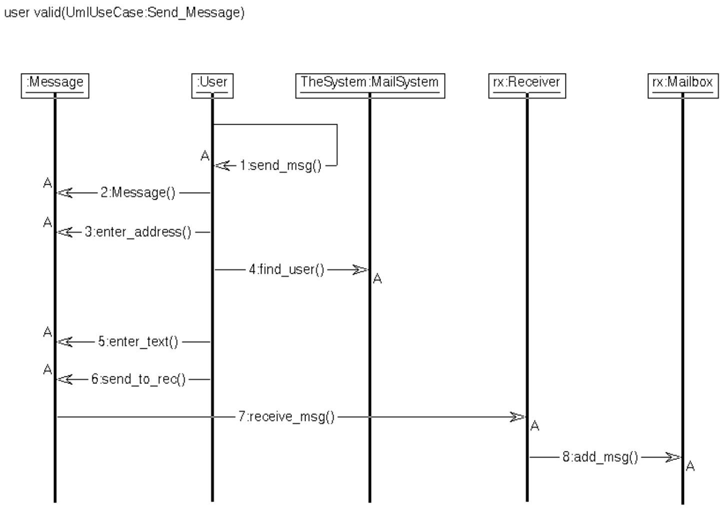
Showing Activation

StP/UML automatically assigns an Activation annotation to any message that is not designated as a return.

The activation display mark (A) appears on the diagram where the activating message meets the supplier object's lifeline.

To view the display mark, choose **Update Display Marks** from the Edit menu. Activation display marks appear next to all of the messages in your diagram, as shown in Figure 15.

Figure 15: Activation Display Marks



Modeling Returns

On a simple sequence diagram, returns are implicit and do not need to be shown. However, when a scenario includes nested or concurrent activities, it is useful to draw returns on the diagram.

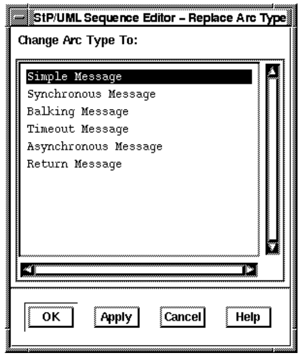
In this example, you model a return to show the completion of the *find_user* operation (although you could also allow this return to be implicit.)

Labels are optional on returns.

To model a return message:

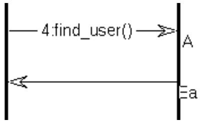
1. Insert a message arc from *MailSystem* to *User*, placing it just below the *find_user* message.
   The arc is selected.
2. Choose **Replace** from the Edit menu.
   The Replace Arc Type dialog appears.

Figure 16: Replace Arc Type Dialog



3. Choose **Return Message** from the scrolling list.
   The arc type is highlighted.
4. Click **OK**.
   The dialog is dismissed.
5. Choose **Update Display Marks** from the Edit menu.
   A display mark (Ea) appears next to the return arc, as shown in Figure 17. The display mark indicates that the return message ends the activation of its source object.

Figure 17: **Return Arc with Display Mark**



Indicating Object Creation

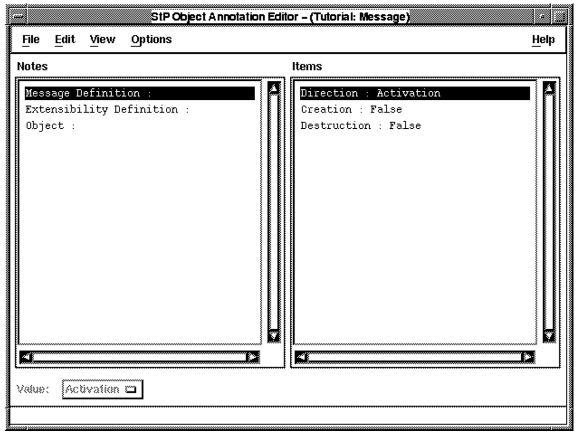
In the *user valid* scenario, a message sent by the *User* object causes the creation of a *Message* object. You use the Creation annotation item to indicate that a message creates an object.

To indicate object creation:

1. Select the *Message* arc.
   Selection handles appear on the arc.
2. Do one of the following:
   - Choose **Edit Object Annotation** from the Edit menu
   - Use the accelerator Ctrl+A

   The Object Annotation Editor appears.

Figure 18: **The Object Annotation Editor (OAE)**



Message Definition :
Extensibility Definition :
Object :

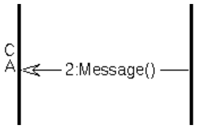
Direction : Activation
Creation : False
Destruction : False

Value: Activation

The Message Definition note is highlighted in the Notes scrolling list. The items associated with the note (Direction, Creation, and Destruction) are displayed in the Items scrolling list.

3. In the Items scrolling list, select **Creation**.
   The item is highlighted.
4. Choose **True** from the Value pop-up menu.
   The value appears in the Items scrolling list.
5. Choose **Save** from the OAE File menu.
6. Choose **Update Display Marks** from the Edit menu in the Sequence Editor.
   The creation display mark appears in the diagram, as shown in Figure 19.

Figure 19: Creation Display Mark



Indicating Message Synchronization Type

Messages are passed using one of five types of synchronization. Table 2 describes the different synchronization types.

Table 2: Message Synchronization Types

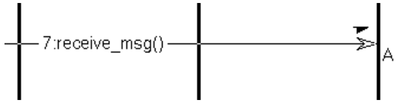
Type	Description
Simple	There is a single thread of control.
Synchronous	The client object waits for the supplier object to accept the message.
Balking	If the supplier object cannot immediately service the message, the client object discards the message.
Timeout	If the supplier object cannot service the message within a given amount of time, the client object discards the message.
Asynchronous	The client object does not wait after sending the message; the supplier object queues the message for processing.

The default synchronization type is Simple. All of the messages you have drawn so far are simple messages. In this section you assign a new message type to the *receive_message* message arc.

To set the message synchronization type:

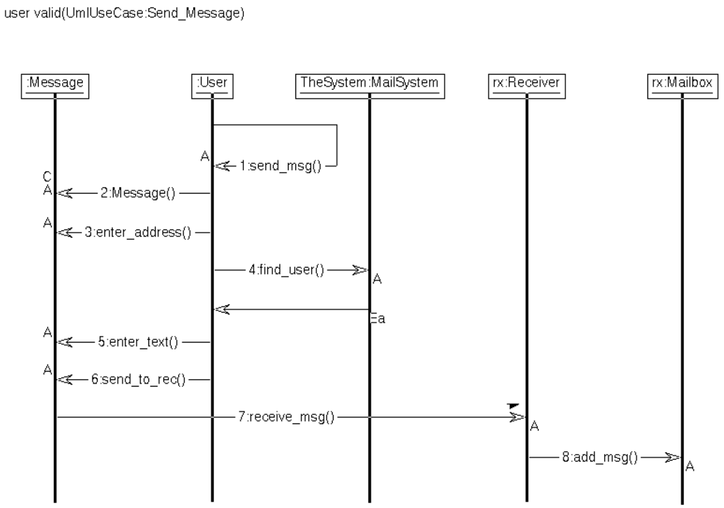
1. Select the *receive_message* message arc.
   Selection handles appear on the arc.
2. Choose **Replace** from the Edit menu.
   The Replace Arc Type dialog appears.
3. Choose **Asynchronous Message** from the scrolling list.
   The arc type is highlighted.
4. Click **OK**.
   The dialog is dismissed, and a display mark appears on the message in the diagram, as shown in Figure 20.

Figure 20: Asynchronous Message Display Mark



You have now completed your sequence diagram, which should resemble the one shown in Figure 21.

Figure 21: Completed Sequence Diagram



Generating a Collaboration Diagram

A collaboration diagram contains largely the same information as a sequence diagram, but it emphasizes the structure and relationship of the objects in the interaction.

You can create a collaboration diagram by drawing one or by having StP/UML create one from an existing sequence diagram. In this part of the tutorial, you have StP/UML create a collaboration diagram from the sequence diagram you just completed.

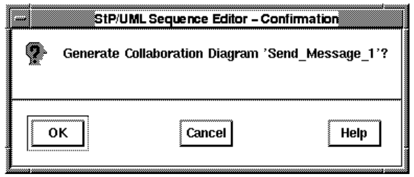
Navigating to the Collaboration Editor

To create a collaboration diagram based on the *Send_Message_1* sequence diagram:

1. Choose **Collaboration Diagram** from the Navigation menu.

   Since *Send_Message_1* does not have an associated collaboration diagram, a dialog box appears prompting you to generate one.

   Figure 22: Generate Collaboration Diagram Dialog

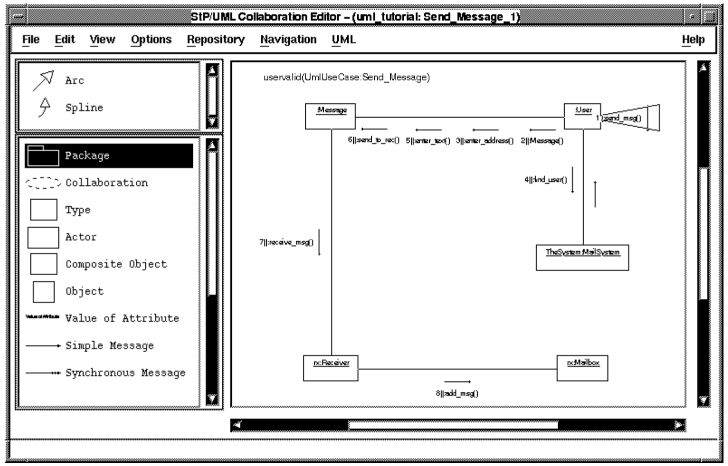
   StP/UML Sequence Editor – Confirmation

   Generate Collaboration Diagram 'Send_Message_1'?

2. Click **OK**.

   The Collaboration Editor appears displaying an automatically-generated collaboration diagram with the same name as the sequence diagram.

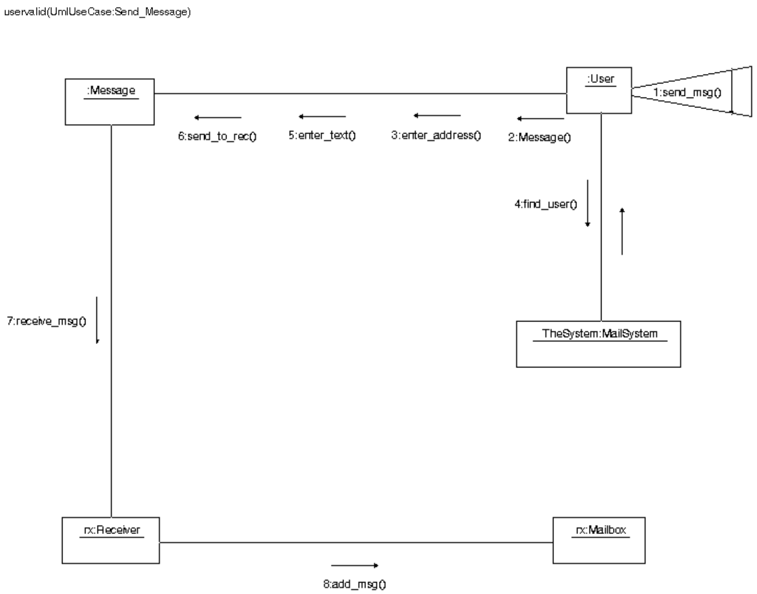
Figure 23: The Collaboration Editor Window



The position of the symbols in your collaboration diagram may not be identical to the diagram shown here. It is not necessary to adjust your diagram to match the tutorial example, however, you may want to reposition symbols to improve readability or make room for additional symbols.

The symbols and messages in your diagram should be the same as those shown in Figure 24.

Figure 24: Collaboration Diagram



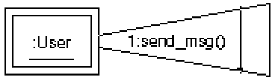
Creating Active Objects

An active object is an object that can initiate control activity. In the *user valid* scenario, the *User* object initiates all of the control activity. In this example, you indicate that *User* is an active object.

To create an active object:

1. Select the *User* object.
   Selection handles appear on the object symbol.
2. Choose **Make Object Active** from the UML menu.
   User changes from an object symbol to an active object symbol, as shown in Figure 25. An active object is represented by a rectangle with a double border.

Figure 25: Active Object in a Collaboration Diagram



You can change an active object back into a passive object by selecting the object and choosing **Make Object Passive** from the UML menu.

Modeling Composite Objects

A composite object is made up of other objects. The composite object and its components represent an instance of a composition relationship on a class diagram.

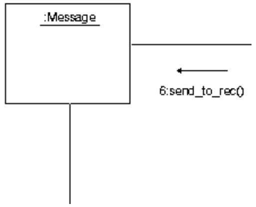
In this example, you create a composite object that corresponds to the composition relationship you modeled in "Modeling Compositions" on page 4-6.

To create a composite object:

1. Select the *Message* object.
   Selection handles appear on the object symbol.
2. In the Symbol List, select the composite object symbol.
   The symbol is highlighted.
3. Choose **Replace** from the Edit menu.

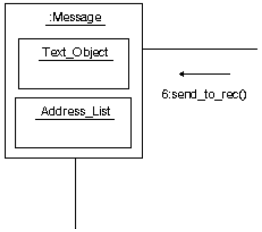
Message changes from an object symbol to a composite object symbol. A composite object is represented by a large square.

Figure 26: Composite Object Symbol



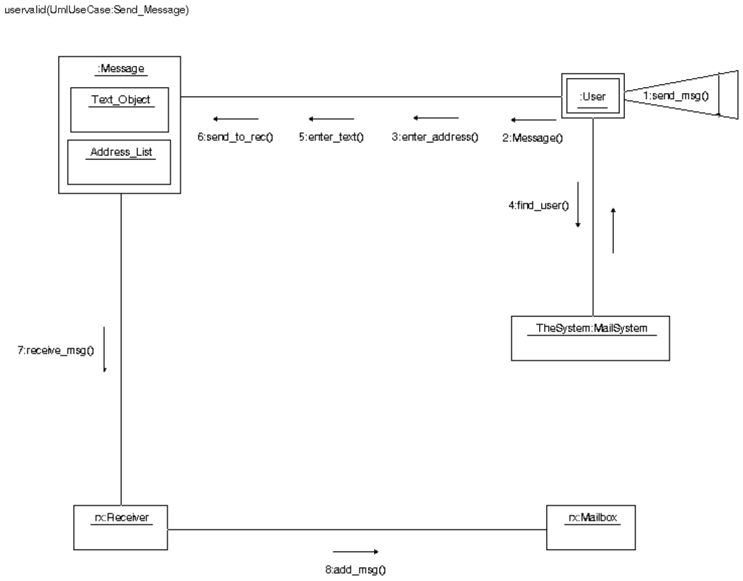
4. Scale the composite object so that it can accommodate two component objects.
   You can use the **Align Links** command on the Drawing Area menu to realign your message links after scaling the object.
5. Insert two object symbols inside the composite object.
6. Label the objects **`Text_Object`** and **`Address_List`**.
   Your diagram should resemble Figure 27.

Figure 27: Composite Object with Components



You have now completed your collaboration diagram, which should resemble the one shown in Figure 28.

Figure 28: Completed Collaboration Diagram



Exiting the Editors

Choose **Exit** from the File menu to exit the Collaboration and Sequence Editors. You can also exit the Use Case Editor and the Object Annotation Editor at this time.

Continuing the Tutorial

In this chapter, you:

- Navigated to the Sequence Editor from the Use Case Editor.
- Created a sequence diagram
- Automatically generated a collaboration diagram

In the next chapter, you create component and deployment diagrams.

9 Creating Implementation Diagrams

Implementation diagrams show the physical design of the system. There are two types of implementation diagram: component and deployment. Component diagrams show the structure of the software, and deployment diagrams show the hardware configuration.

In this chapter, you create a component and deployment diagram for the Email system. You start by creating a component diagram, then you navigate to the Deployment Editor. In the process, portions of your deployment diagram are automatically generated.

About the Component Editor

The component diagram shows the software components present at compile time, link time, run time, or a combination of stages. In this chapter, you create a component diagram showing the software components in the *Email* application at compile time.

You use the Component Editor to create component diagrams. For a complete description of the Component Editor, see *Creating UML Models*.

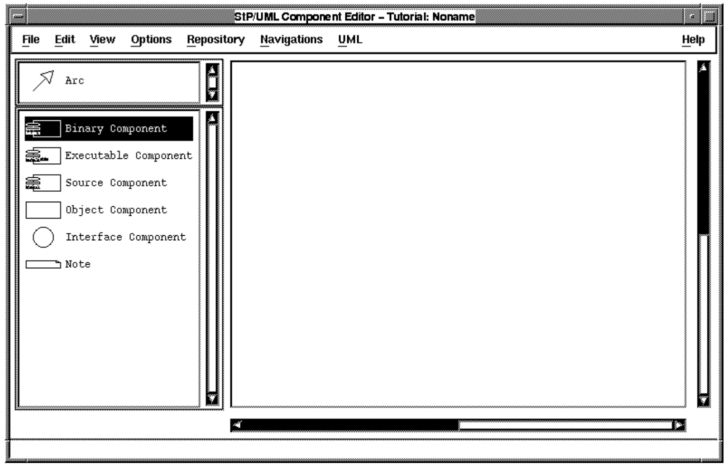
Starting the Component Editor

To start the Component Editor:

1. Select the Model category on the StP Desktop.
   A list of tools appears.
2. Select Component Diagrams.

3. Choose **Start New Editor** from the Commands menu.
   The Component Editor appears.

Figure 1: The Component Editor



Naming the Diagram

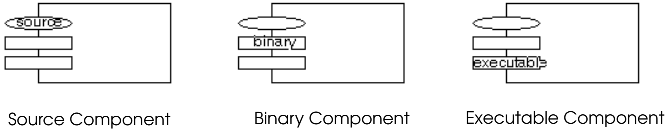
Use the Save Diagram dialog to name the diagram *Email*. If necessary, refer to "Naming the Diagram" on page 2-9.

Creating a Component Diagram

The component diagram shows the software components and dependencies of your system. Software components include:

- Source code
- Binary code
- Executables

Figure 2: Software Component Symbols



Modeling Software Components

The component diagram for the Email system shows the software components that interact to build the system. To build the Email system:

- An executable starts a C++ compiler.
- The compiler gets source code from the *Application* and *ReuseLibrary* subsystems.
- The compiler generates a binary.
- An executable starts and runs the Email system.

Table 1 shows the software components in the Email component diagram.

Table 1: Email Software Components

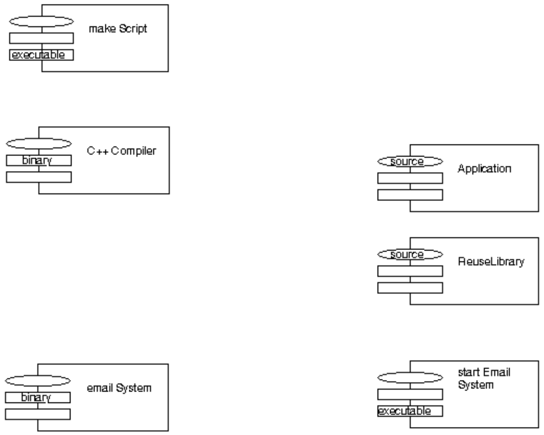
Component Type	Component Label
Executable	*make Script*
	start Email System
Binary code	*C++ Compiler*
	email System
Source code	*Application*
	ReuseLibrary

Inserting Components

To insert software components:

1. Drag two executable component symbols from the Symbol List into the drawing area.
2. Label the symbols `make Script` and `start Email System`.
3. Drag two binary component symbols into the drawing area.
4. Label the symbols `C++ Compiler` and `email System`.
5. Drag two source component symbols into the drawing area.
6. Label the symbols `Application` and `ReuseLibrary`.
7. Reposition the symbols to match Figure 3.

Figure 3: Email System Software Components



Modeling Interfaces

In addition to source code, binary code, and executables, you can model the interfaces to software components on your component diagram.

There are two interfaces on the Email diagram:

- *start Compile* processes input to the *C++ Compiler* binary
- *start Email* processes input to the *email System* binary

To insert an interface:

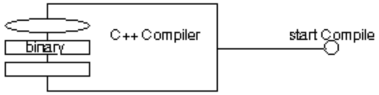
1. Drag an interface component symbol into the drawing area, and place it to the right of the *C++ Compiler* binary.
2. Label the symbol `start Compile`.

3. Insert an arc from the *start Compile* interface symbol to the *C++ Compiler* binary symbol.

   When you draw an arc from an interface to a component, StP automatically creates an interface link. The link is a solid line.

   Your diagram should resemble Figure 4.

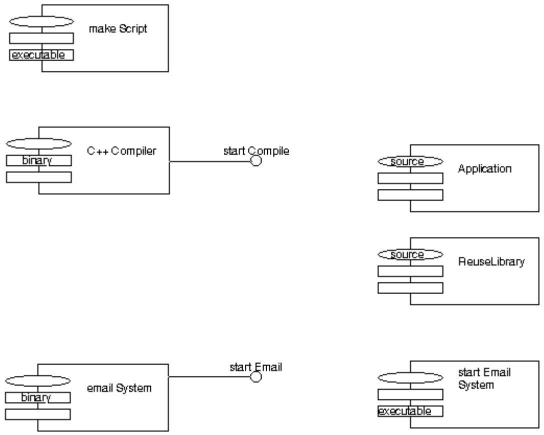
   Figure 4: Modeling an Interface



4. Drag a second interface component into the drawing area, and place it to the right of the *email System* binary.
5. Label the symbol `start Email`.
6. Insert an arc from the *start Email* interface symbol to the *email System* binary symbol.

   Your diagram should resemble Figure 5.

Figure 5: Email System Interfaces



Modeling Software Dependencies

In this section, you draw the dependencies among the software components in the Email system. A dependency is represented by a directed arc pointing from the dependent component to the depended-on component.

The software dependencies for the Email system are shown in Table 2.

Table 2: Software Dependencies in the Email System

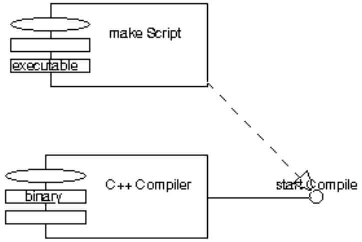
Dependent Component (from component)	Depended-on Component (to component)
make Script executable	*start Compile* interface
C++ Compiler binary	*Application* source code
	ReuseLibrary source code
email System binary	*C++ Compiler* binary
start Email System executable	*start Email* interface

To model a software dependency:

1. Insert an arc from the *make Script* executable symbol to the *start Compile* interface symbol.

   When you draw an arc from a component to an interface, StP automatically creates a dependency link. The link is a dashed line with an arrow pointing toward the depended-on component.

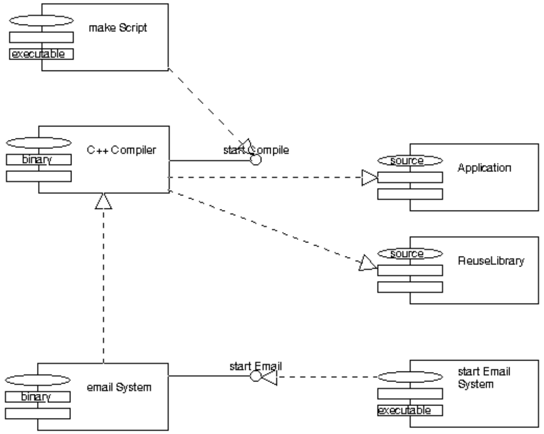
   Figure 6: Dependency Arc

2. Using Table 2 as a guide, model the remaining software dependencies.

   Your diagram should resemble Figure 7.

Figure 7: Email System Software Dependencies



Modeling Object Components

There may be components relevant to your diagram that do not fall into the categories of binary, executable, or interface. You can model these elements using the Object Component symbol.

An object component lives on a host binary or executable component. The host component calls the object to perform a particular function.

There are two ways to model the relationship between an object component and its host: you can draw a link from the host to the object component, or you can place the object component inside the host.

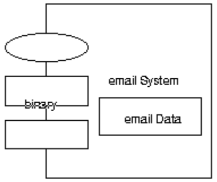
In this tutorial, you use the latter method (called containment) to model a database used by the *email System* binary.

To insert an object component:

1. Resize the *email System* binary so that it can accommodate an extra symbol.
2. Drag an Object Component symbol into the drawing area and place it inside the *email System* binary.
3. Label the component `email Data`.

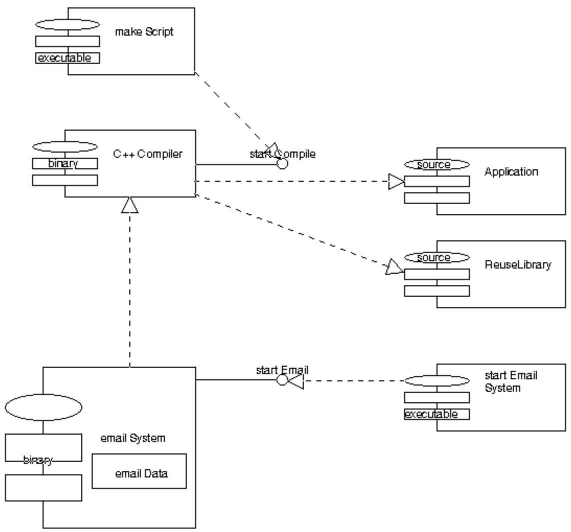
   The binary and its object component should resemble Figure 8.

Figure 8: Adding Object Components



You have now completed the component diagram for the Email system. Your diagram should resemble Figure 9.

Figure 9: Adding Object Components



Creating a Deployment Diagram

In this section, you create a deployment diagram. The deployment diagram shows the allocation of software to hardware and the interconnections between hardware components.

Navigating to the Deployment Editor

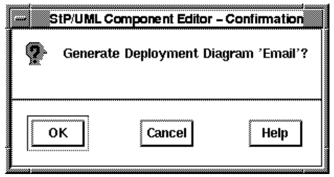
StP/UML can automatically generate parts of a deployment diagram from an existing component diagram. To do this, you navigate to the Deployment Editor from the component diagram you completed in the last section.

To create a deployment diagram through navigation:

1. In the Component Editor, choose **Deployment Diagram** from the Navigation menu.

   Because no deployment diagram exists for the *Email* component diagram, a dialog box appears prompting you to create one.

   Figure 10: Generate Deployment Diagram Dialog

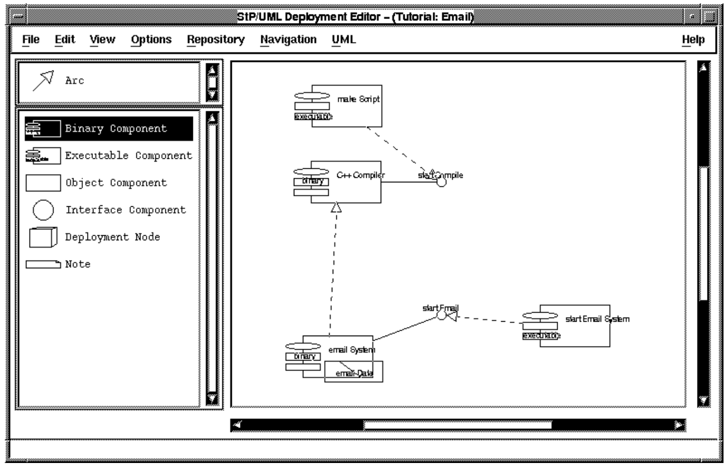
   StP/UML Component Editor – Confirmation

   Generate Deployment Diagram 'Email'?

2. Click **OK**.

   The Deployment Editor appears, as shown in Figure 11.

Figure 11: Navigating to the Deployment Editor



In navigating from the Component Editor to the Deployment Editor, StP/UML automatically creates a deployment diagram of the same name and copies into it the relevant parts of the component diagram. This diagram is provided as a starting point to which you add additional symbols to describe the hardware configuration of your system.

Notice that none of the source components on the component diagram have been carried over to the deployment diagram. This is because the deployment diagram represents the system at run time, after the source components have been compiled.

Modeling Hardware Components

A deployment diagram shows the processors and devices in a system. A processor is hardware capable of executing an application, such as a PC or a workstation. A device is hardware that is not capable of executing an application, such as a disk or a terminal.

You use a deployment node symbol to represent processors and devices. The label for a deployment node includes both the node name and the processor or device type in the format:

<name>:<type>

The Email system software runs on three processors:

- *Build:sun*, which contains the *make Script* executable and the *C++ compiler* binary
- *Any_user:sun*, which contains the *email System* binary and the *start Email System* executable
- *Email_server:sun*, which contains the *email Data* database

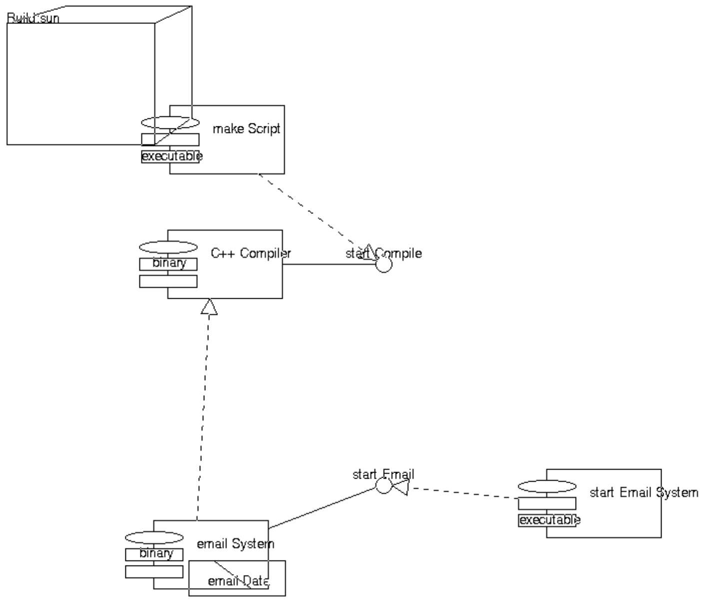
Drawing Components

To draw hardware components for the Email system:

1. Drag a Deployment Node symbol from the Symbol List into the upper left corner of the drawing area.
2. Label the deployment node `Build:sun`.

   Your diagram should resemble Figure 12.

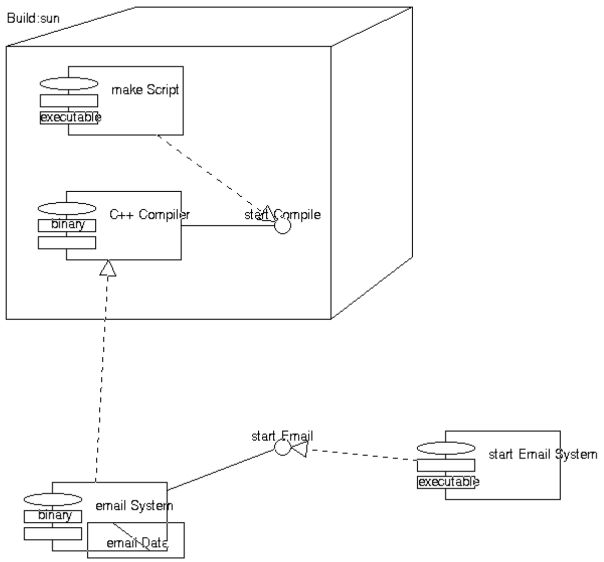
Figure 12: Labeled Deployment Node Symbol



3. Scale the symbol so that it encloses the *make Script* executable and the *C++ Compiler* binary.

   Your diagram should resemble Figure 13.

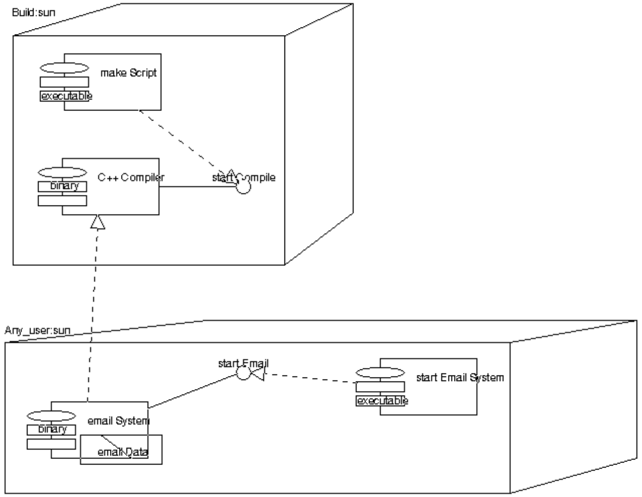
Figure 13: Scaled Deployment Node Symbol



4. Drag another Deployment Node symbol into the drawing area and place it above and to the left of the *email System* executable.
5. Label the node symbol `Any_user:sun`.
6. Scale the symbol so that it encloses the *email System* binary and the *start Email System* executable.

   Your diagram should resemble Figure 14.

Figure 14: Deployment Nodes



Deploying Object Components

On a component diagram, object components are contained within or linked to host software components. The relationship between an object component and its host persists when you generate your deployment diagram. However, an object component does not have to be deployed on the same processor as its host.

The deployment of the Email system calls for a separate node for the *email Data* database. To model this:

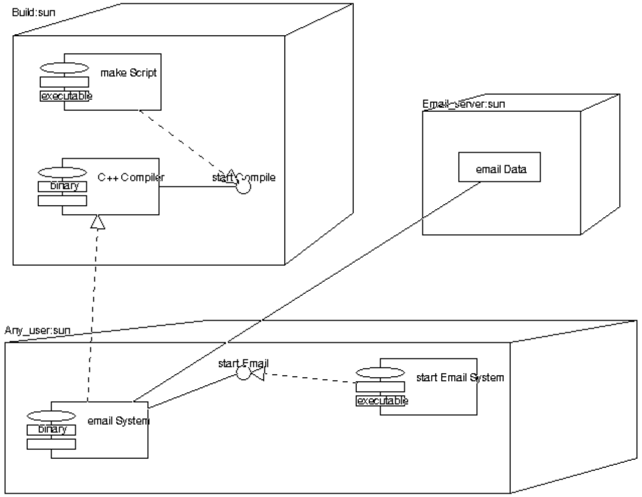
1. Select and drag the *email Data* object component to the right of the *Build:sun* node component.

An “Is Component Of” link, which was inserted when your deployment diagram was generated, preserves the connection between the object component and the *email System* binary.

2. Drag a deployment node symbol into the drawing area and place it over the *Email data* object component.
3. Label the node symbol **`Email_server:sun`**.

   Your diagram should resemble Figure 15.

Figure 15: Deploying an Object Component



You have now completed the deployment diagram for the Email system. When you are satisfied with your diagram, choose **Save** from the File menu.

Exiting the Implementation Editors

To exit the Component and Deployment editors, choose **Exit** from the File menus.

Continuing the Tutorial

In this chapter you:

- Created a component diagram
- Automatically generated parts of a deployment diagram by navigating from the Component Editor to the Deployment Editor
- Added deployment node symbols to complete the deployment diagram

In the next chapter, you generate code for a class in the Email system.

10

Generating C++ Code

StP/UML supports C++ code generation based on definitions of classes in a model. For this tutorial, you generate C++ code for a single class in the model. Typically, you would generate code for all the classes in your model.

What is Generated?

For each class in a class diagram, StP/UML generates these C++ code files:

- Interface file (“header” or *.h* file)
- Implementation file (source or *.C* file)

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

Interface Files

Interface files (*.h* files) contain definition and interface information for the class. Each file includes class declarations, definitions, and, optionally, inline member functions.

Implementation Files

Implementation files (*.C* files) contain member function headers and bodies and any C++ code specified in the C++ code annotation. The member functions are derived from operations in the class diagram.

How is C++ Generated?

StP/UML generates C++ from classes in the repository using Query and Reporting Language (QRL) scripts. You can generate code for:

- All the classes in a model
- All the classes in a diagram
- One or more selected classes or class categories

Deriving C++ from the UML Model

You can generate C++ at any time during development; StP/UML does not require a completed model for code generation. However, to generate viable C++, your model must contain certain table information and annotations. Additionally, the names of all objects must conform to C++ conventions.

Only attributes and operations defined in class tables are used to generate code. Attributes and operations that appear only in class diagrams are not used because their types are not defined.

To prepare classes for C++ generation, you annotate the classes represented in diagrams and tables with notes and items. For code generation, some class table cells are considered annotations, and these are the annotations that are used for this tutorial. For information about fully annotating your model, see *Generating and Reengineering Code*.

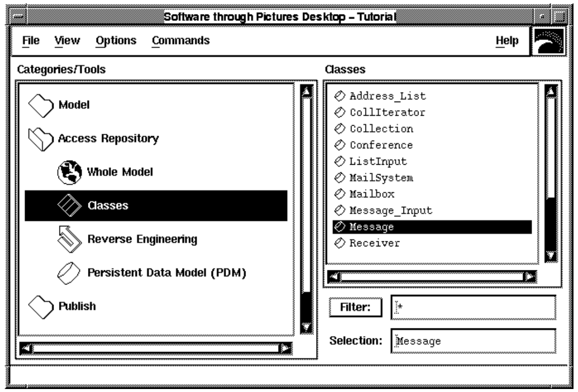
Generating C++ Code

To generate code for the *Message* class:

1. Select the Access Repository category on the StP Desktop.
   A list of tools in the Access Repository category appears.
2. Select Classes.
   The classes for the *Email* system appear in the Classes scrolling list.
3. Select the *Message* class from the scrolling list.

The *Message* class is highlighted.

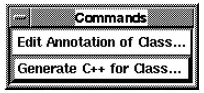
Figure 1: Selected Message Class



Selection: Message

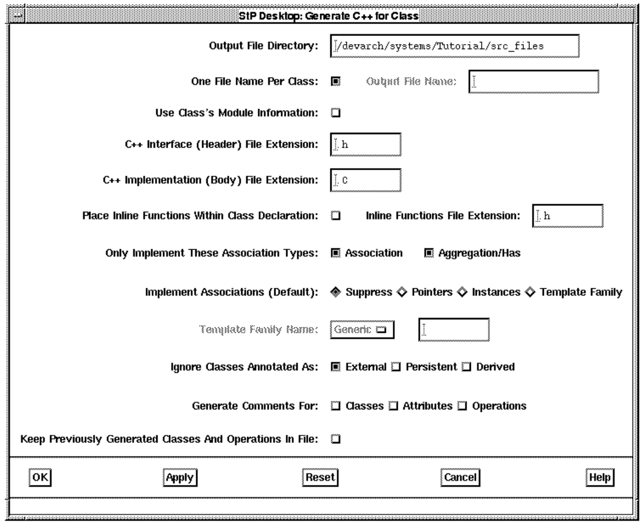
4. Choose **Generate C++ for Class** from the Commands menu.

Figure 2: Commands Menu



The Generate C++ for Class dialog box appears.

Figure 3: The Generate C++ for Class Dialog Box



Output File Directory: /devarch/systems/Tutorial/src_files

One File Name Per Class: ☒ Output File Name:

Use Class's Module Information: ☐

C++ Interface (Header) File Extension: h

C++ Implementation (Body) File Extension: C

Place Inline Functions Within Class Declaration: ☐ Inline Functions File Extension: h

Only Implement These Association Types: ☒ Association ☒ Aggregation/Has

Implement Associations (Default): ◆ Suppress ◇ Pointers ◇ Instances ◇ Template Family

Template Family Name: Generic

Ignore Classes Annotated As: ☒ External ☐ Persistent ☐ Derived

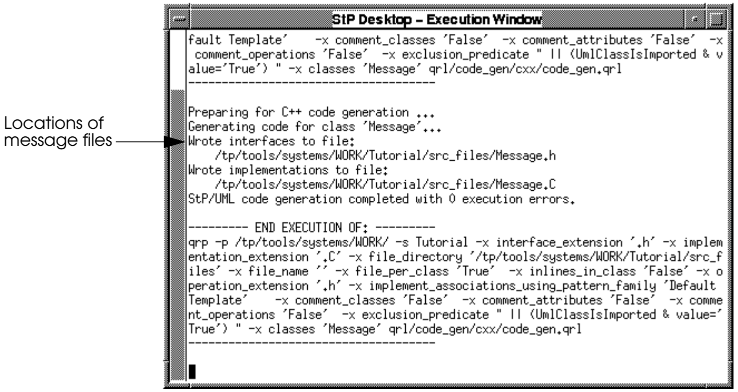
Generate Comments For: ☐ Classes ☐ Attributes ☐ Operations

Keep Previously Generated Classes And Operations In File: ☐

5. Click **OK**.

   An Execution window appears with a list of the locations of the *Message.h* and *Message.C* files, as shown in Figure 4.

Figure 4: Execution Window



```
fault Template'    -x comment_classes 'False'  -x comment_attributes 'False'  -x
 comment_operations 'False'  -x exclusion_predicate " || (UmlClassIsImported & v
alue='True') " -x classes 'Message' qrl/code_gen/cxx/code_gen.qrl
-------------------------------------


Preparing for C++ code generation ...
Generating code for class 'Message'...
Wrote interfaces to file:
    /tp/tools/systems/WORK/Tutorial/src_files/Message.h
Wrote implementations to file:
    /tp/tools/systems/WORK/Tutorial/src_files/Message.C
StP/UML code generation completed with 0 execution errors.

--------- END EXECUTION OF: ---------
qrp -p /tp/tools/systems/WORK/ -s Tutorial -x interface_extension '.h' -x implem
entation_extension '.C' -x file_directory '/tp/tools/systems/WORK/Tutorial/src_f
iles' -x file_name '' -x file_per_class 'True'  -x inlines_in_class 'False' -x o
peration_extension '.h' -x implement_associations_using_pattern_family 'Default
Template'    -x comment_classes 'False'  -x comment_attributes 'False'  -x comme
nt_operations 'False'  -x exclusion_predicate " || (UmlClassIsImported & value='
True') " -x classes 'Message' qrl/code_gen/cxx/code_gen.qrl
-------------------------------------
```

Using the text editor of your choice, examine the *Message.h* and *Message.C* files in the designated output file directory *.../src_files*. Your files should resemble the following examples.

Message.h Output

```
// StP
// Created on Tue Aug 26 13:49:56 1997 for barb@roo
//   from system /tp/tools/systems/WORK/Tutorial

#ifndef _Message_h_
#define _Message_h_

// stp class declarations
class Message;
// end stp class declarations

// stp class definition 1
class Message
{
// stp class members
public:
    void print(void);
    void send_to_rec(Receiver* u);
    ~Message(void);
    inline Message(String& s);
protected:
private:
    String body;
    String from;
    Receiver* to;
    String subject;
```

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

// stp operation 1::6
inline
Message::Message(String& s) : body(""), subject(""),
from("")
{

// stp code

// end stp code

}
// end stp operation

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

Message.C Output

```
#include "Message.h"
// stp operation 1::5
void
Message::print(void)
{

// stp code
// end stp code

}
// end stp operation

// stp operation 1::8
void
Message::send_to_rec(Receiver* u)
{

// stp code
// end stp code

}
// end stp operation

// stp operation 1::7
Message::~Message(void)
{
```

```
// stp code
// end stp code

}
// end stp operation
```

Completing the Tutorial

You have now completed the StP/UML tutorial. Exit from all open editors by choosing **Exit** from the File menus.

There is a fully specified version of the Email sample system available on your StP/UML installation media.

For further reading suggestions, refer to Table 2 in the Preface of this manual.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Index for Getting Started with StP/UML

Click on any Index entry page number to jump to the corresponding page.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z