第一部分　　关于本教程

欢迎使用 Rational Rose 教程 Rational Rose 是一套可视化建模工具，用于在 C/S，分布式企业环境下开发健壮的，有效的解决方案以满足真正的业务需求。本教程通过指导你一步步地进行一个复杂的业务问题的真正实现解决，教给你如何使用 Rose。

估计完成时间：完成整个教程需要花大约 10 小时。当然可以从任何部分开始和结束，剩下的部分以后再看。每一部分还有完成该部分所需时间，例如前言部分需要大约 5 分钟。

示例模型：在本教程中，你将为一个叫做 Classics 的虚构的公司开发一套 POS 系统。Classics 公司是一家快速成长的连锁店，经营经典的音乐、电影、图书。该公司的 POS 系统已经过时，现在必须作一套新的定单处理和实现系统（OPFS）。随着本教程的不断深入，你将开发 OPFS 系统以帮助 Classics 公司更好的管理其商店和存货。

该教程适合我吗？如果你有一点或没有 Rational Rose 知识，都可以使用本教程。当然你得对 Windows (NT/95/98)操作系统、面向对象分析&设计（OOAD）概念、UML 语言有一些基本了解。

本教程是如何组织的？本教程是按照在业务和应用软件建模中的行为步骤来进行的，每一部分是建立在前一部分提出的知识的基础上的。但是每一部分都是独立的单元。这就意味着你可以从任何一部分开始，而跳过其他部分。例如，如果你懂业务建模，就可以跳过第 3 部分直接进行第 4 部分。

我可以按照各部分的顺序吗？按照各部分的顺序，你可以模拟一个真实世界的开发环境。看完了第 1 和第 2 部分的介绍材料，就自然地进入第 3 部分的 OPFS 的工作。在第 3 部分中你将开始 Classic 公司的业务建模。接下来进行建模应用需求以及实现。在最后部分里将产生代码以及正逆向模型和代码

每一部分里面有什么？每个部分包含：

- （一个解释你要干什么以及为什么干的）介绍
- （一个完成该部分所需时间的）估计
- （一个完成部分开发的）样例模型
- （一个在该部分引入的特性的详细指导）‘试一试’部分
- 一个已完成的模型
- （一个在该部分所学的 Rose 的特性的）概要

使用样本模型 本教程采用手把手的方式教你如何学习 Rational Rose。在每一部分的开始，你要装载一个针对该部分的半成品模型。随着在该部分的一步步执行指令，模型被细化并进入下一个开发阶段。如果你选择跳过某个部分，你仍然可以使用后面的模型，只要装载适当的样本模型

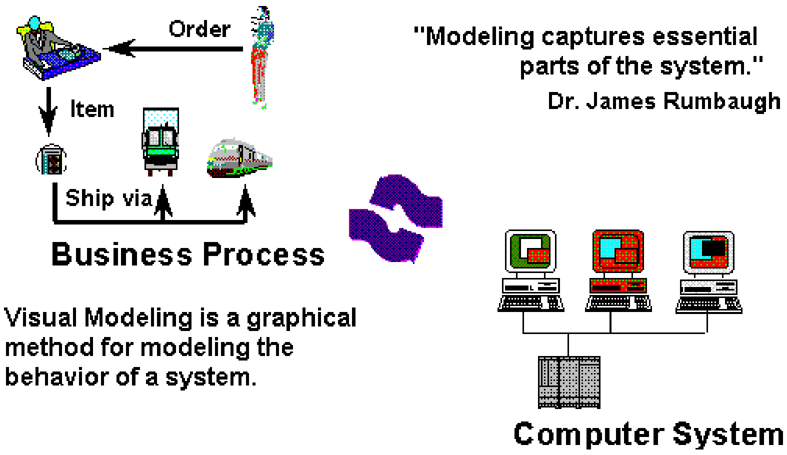
附加信息 你可以得到关于 Rational Rose 公司的其他信息，包括术语和视觉文本

下一步干什么？第 2 部分涉及建模的优点，并引入一些 Rational Rose 的主要原理

第二部分　关于 Rose

关于这部分 这部分向你介绍用 Rational Rose 建模。在这部分里，你会学到建模是如何帮助你生产更好的软件。你也可以学习使用 Rose 的一些基本知识。

估计完成时间：大约 15 分钟



"Modeling captures essential parts of the system."
Dr. James Rumbaugh

Visual Modeling is a graphical method for modeling the behavior of a system.

什么是可视化建模？可视化建模就是以图形的方式描述所开发的系统的过程。可视化建模允许你提出一个复杂问题的必要细节，过滤不必要的细节。它也提供了一种从不同的视角观察被开发系统的机制。

为什么要建模？设计一个软件的模型就好比是一幢大楼需要蓝图一样重要。好的模型能够：

- 鉴别需求和沟通信息
- 着眼于系统的组件如何相互作用，而不是陷于具体的细节
- 使你能够了解设计组件的相互关系
- 通过使用一个共同的图形语言，改进跨团队的沟通

为什么要使用 Rational Rose？ 有大量的原因都说明应使用 Rational Rose 进行你的开发工作。这里只说几个：

- 用模型驱动的开发能提高开发者的生产力
- 用例和着眼于业务的开发能改进软件质量
- 共同的标准的语言--（UML）能改进团队沟通
- 逆向工程能力允许你集成传统的 OO 系统
- 模型和代码通过开发周期保持同步

Rose 的版本 Rose 现在有三个可用版本：

- Rose Modeler –没有语言支持
- Rose Professional –只支持一种语言
- Rose Enterprise –支持多种语言包括(VC++, VB, Java, and CORBA)

Rose Modeler
Rose Professional
Rose Enterprise

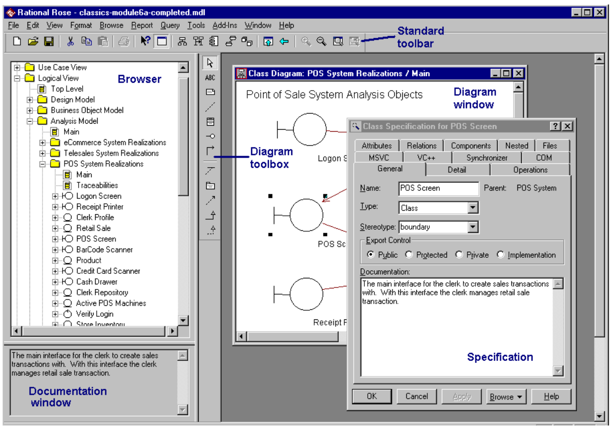
所有版本的 Rose 都可以使用本教程。但是决定于你所使用的 Rose 版本，你可能没有本教程里的所有部分的功能。

Rose GUI

你可能已经熟悉 Rose 中使用的 GUI 惯例。

- 标准工具栏
- 图表工具箱
- 图表窗口
- 文本窗口
- 规范（描述）

These items are briefly discussed in subsequent slides. How, where, and why each of these elements is used will become clearer as you progress through the tutorial.


Rational Rose - classics-module6a-completed.mdl
Standard toolbar
Browser
Diagram window
Diagram toolbox
Documentation window
Specification
Class Diagram: POS System Realizations / Main
Point of Sale System Analysis Objects
Class Specification for POS Screen

Name: POS Screen Parent: POS System

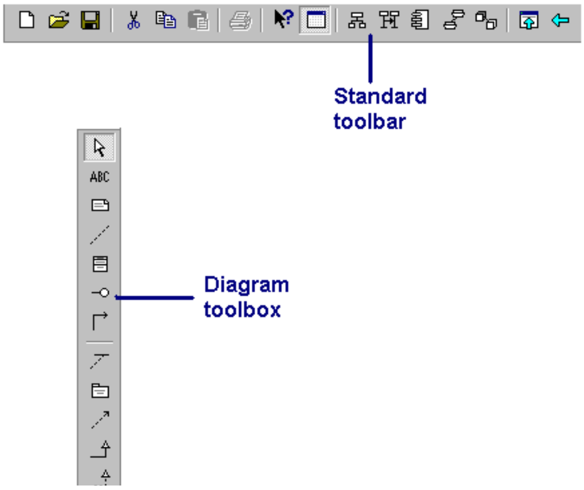
Type: Class

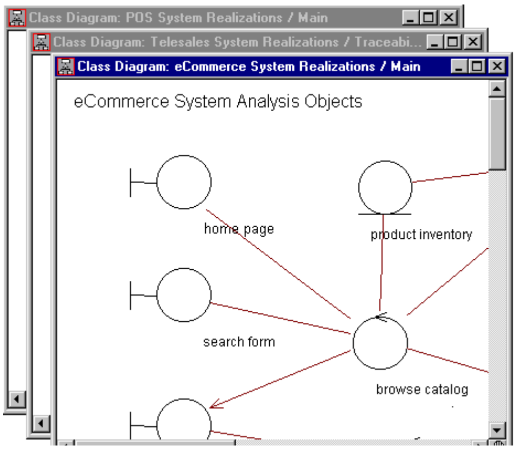
Stereotype: boundary

Documentation: The main interface for the clerk to create sales transactions with. With this interface the clerk manages retail sale transaction.

工具栏和工具箱

The Rose standard toolbar is located near the top of the window and is always displayed - independent of the current diagram type. While in Rose, place your cursor over the toolbar to display a tooltip for each icon. The Rose diagram toolbox changes based on the active diagram. (The active diagram is the one displayed with a blue title bar.) As with the standard toolbar, placing your cursor on an icon displays the tooltip for that icon.




Class Diagram: POS System Realizations / Main
Class Diagram: Telesales System Realizations / Traceabi...
Class Diagram: eCommerce System Realizations / Main
eCommerce System Analysis Objects
home page
product inventory
search form
browse catalog

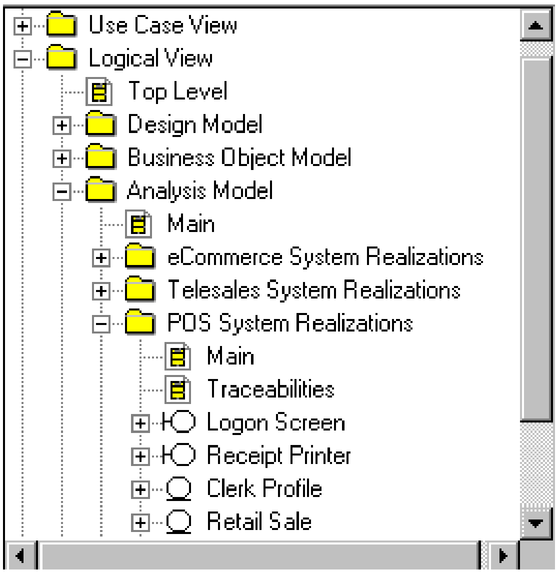
The diagram window

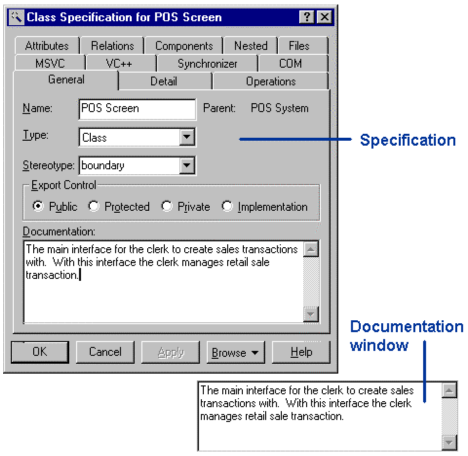
You can create, display or modify Rose diagrams in the diagram window. If you have multiple diagrams open at one time, they are either displayed in a cascading or tiled view.

A cascaded window layers multiple diagrams on top of one another with only the title bar showing for all but the top most diagram. A tiled window divides the application window into equal-sized areas and all diagrams are visible.

The browser

The Rose browser is a hierarchical navigational tool allowing you to view the names and icons representing diagrams and model elements. The plus (+) sign next to an icon indicates the item is collapsed and additional information is located under the entry. Click on the + sign and the tree is expanded. Conversely, a minus (-) sign indicates the entry is fully expanded.If the browser is not displayed, select Browser from the View menu.





Name: POS Screen　Parent: POS System

Type: Class

Stereotype: boundary

Export Control: ● Public ○ Protected ○ Private ○ Implementation

Documentation:

The main interface for the clerk to create sales transactions with. With this interface the clerk manages retail sale transaction.

The main interface for the clerk to create sales transactions with. With this interface the clerk manages retail sale transaction.

The documentation window

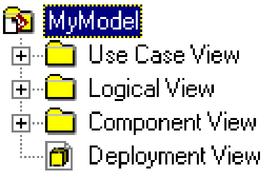
The documentation window allows you to create a self-documenting model and from that self-documenting model generate self-documenting code.

You can create, view or modify information here or in the documentation window of the specification.

If the documentation window is not visible, select **Documentation** from the **View** menu. If there is a check mark next to **Documentation** and you still can’t see the window, move your cursor to the bottom of the browser. When the pointer cursor changes to a splitter cursor, drag up on the browser window to display the documentation window. This window is updated as you select different elements on the diagram.

Views

Just as there are many views of a house under construction — the floor plan, the wiring diagram, the elevation plan, there are many views of a software project under development. Rational Rose is organized around the following views of a software project:



- Use Case
- Logical
- Component
- Deployment

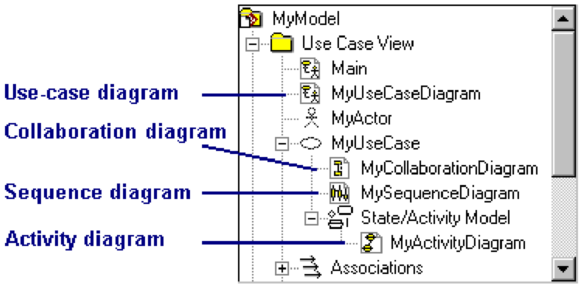
Each of these views presents a different aspect of the model and is explained in subsequent slides.

The use-case view

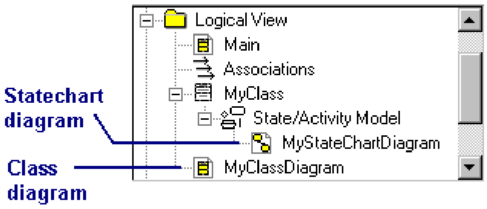
The use-case view helps you to understand and use the system. This view looks at how actors and use cases interact. The diagrams in this view are:

- Use-case diagrams
- Sequence diagrams
- Collaboration diagrams
- Activity diagrams

This view contains a Main diagram by default. Additional diagrams can be added throughout the analysis and design process.



The logical view



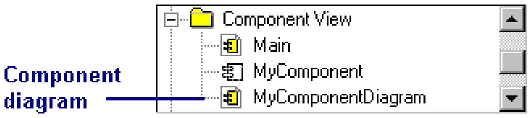
The logical view addresses the functional requirements of the system. This view looks at classes and their relationships.

The diagrams in this view are:

- Class diagrams
- Statechart diagrams

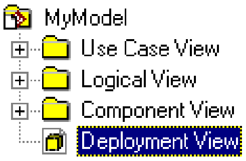
This view contains a Main diagram by default. Additional diagrams can be added throughout the analysis and design process.

The component view



The component view addresses the software organization of the system. This view contains information about the software, executable and library components for the system. This view contains only component diagrams. The component view contains a Main diagram by default. Additional diagrams can be added to this view throughout the analysis and design process.

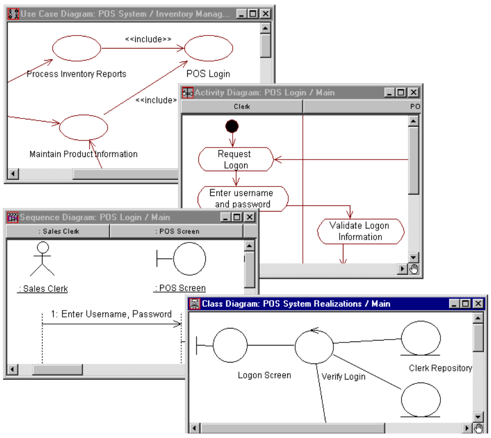
The deployment view



The deployment view shows the mapping of processes to hardware. This type of diagram is most useful in a distributed architecture environment where you might have applications and servers at different locations.

This view contains only one diagram —the deployment diagram.

Diagrams


Use Case Diagram: POS System / Inventory Manag...
<<include>>
Process Inventory Reports
POS Login
<<include>>
Maintain Product Information
Activity Diagram: POS Login / Main
Clerk
PO
Request Logon
Enter username and password
Validate Logon Information
Sequence Diagram: POS Login / Main
: Sales Clerk
: POS Screen
: Sales Clerk
: POS Screen
1: Enter Username, Password
Class Diagram: POS System Realizations / Main
Logon Screen
Verify Login
Clerk Repository

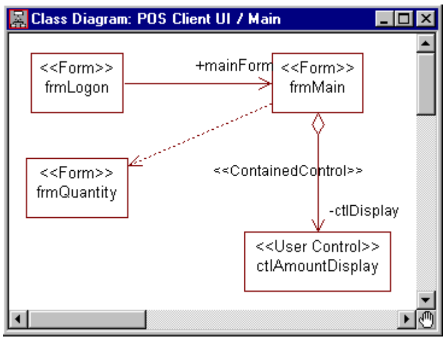
Simply put, a diagram is a graphical representation of the elements of your system. Different diagram types allow you to view your system from multiple perspectives. You can create various types of diagrams in Rational Rose. The diagram types include:

- Use-Case
- Class
- Activity
- Statechart

- Component
- Deployment

Each of these diagram types is explained in subsequent slides.

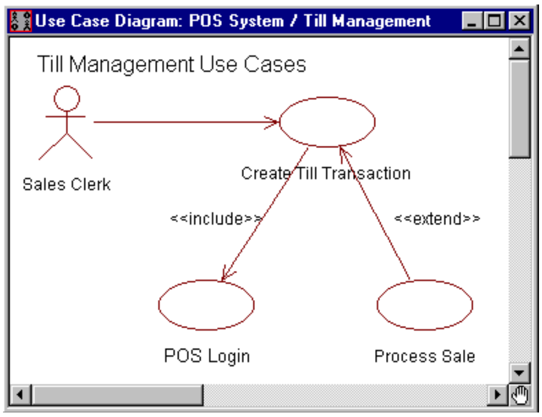
Class diagrams



A class diagram helps you visualize the structural or static view of a system and is one of the most common diagram types. Class diagrams show the relationships among and details about each class. Class diagrams are also the foundation for component and deployment diagrams.

Rose automatically creates a Main class diagram in the logical view. There are typically many class diagrams in a single model.

Use-case diagrams

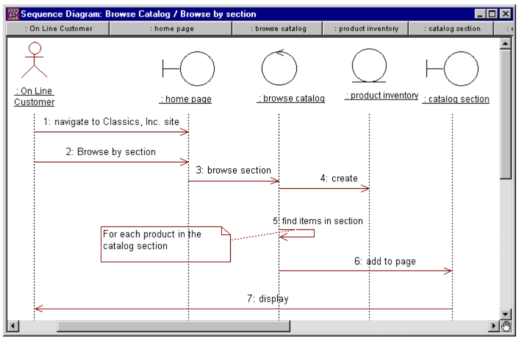


Use-case diagrams present a high-level view of system usage as viewed from an outsider's (actor's) perspective. These diagrams show the functionality of a system or a class and how the system interacts with the outside world.

Use-case diagrams can be used during analysis to capture the system requirements and to understand how the system should work. During the design phase, use-case diagrams specify the behavior of the system as implemented.

Rose automatically creates a Main use-case diagram in the use-case view. There are typically many use-case diagrams in a single model.

Sequence diagrams

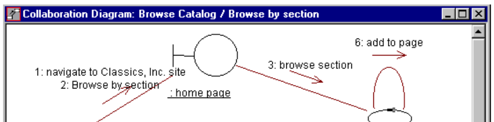


A sequence diagram illustrates object interactions arranged in a time sequence. These diagrams are typically associated with use cases. Sequence diagrams show you step-by-step what has to happen to accomplish something in the use case. This type of diagram emphasizes the sequence of events, whereas collaboration diagrams (an alternative view of the same information) emphasize the relationship.

This type of diagram is best used early in the design or analysis phase because it is simple and easy to comprehend.

Collaboration diagrams

Collaboration diagrams provide a view of the


Collaboration Diagram: Browse Catalog / Browse by section
6: add to page
3: browse section
1: navigate to Classics, Inc. site
2: Browse by section
: home page

interactions or structural relationships between objects in the current model. This type of diagram emphasizes the relationship between objects whereas sequence diagrams emphasize the sequence of events.

Collaboration diagrams contain objects, links, and messages.

Use collaboration diagrams as the primary vehicle to describe interactions that express decisions about system behavior.

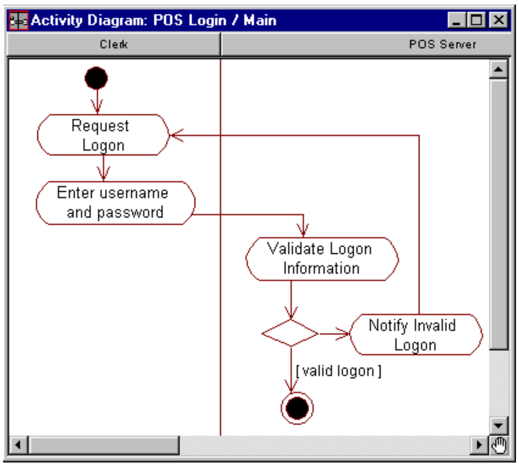
Activity diagrams

Activity diagrams model the workflow of a business process and the sequence of activities in a process.

These diagrams are very similar to a flowchart because you can model a workflow from activity to activity or from activity to state.

It is often beneficial to create an activity diagram early in the modeling of a process to help you understand the overall process.

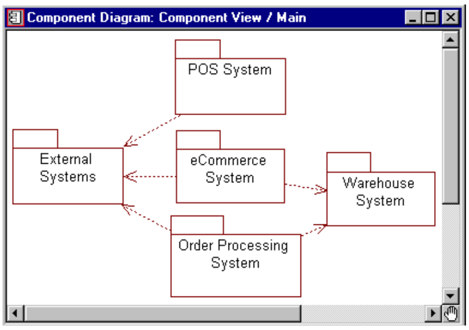
Activity diagrams are also useful when you want to describe parallel behavior or illustrate how behaviors in several use cases interact.


Activity Diagram: POS Login / Main
Clerk
POS Server
Request Logon
Enter username and password
Validate Logon Information
Notify Invalid Logon
[valid logon]

Component diagrams

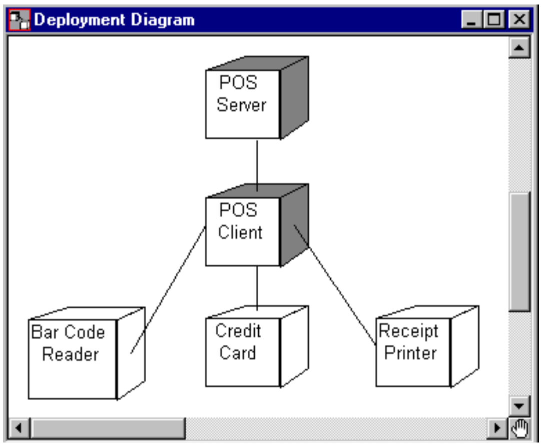
Component diagrams provide a physical view of the current model. They show the organization and dependencies among software components, including source code, binary code, and executable components.

You can create one or more component diagrams to depict components and packages or to represent the contents of each component package.



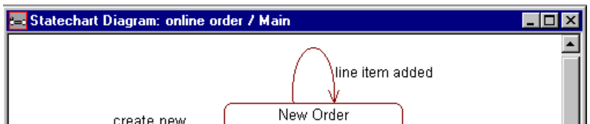
Deployment diagrams

Each model contains a single deployment diagram that shows the mapping of processes to hardware.



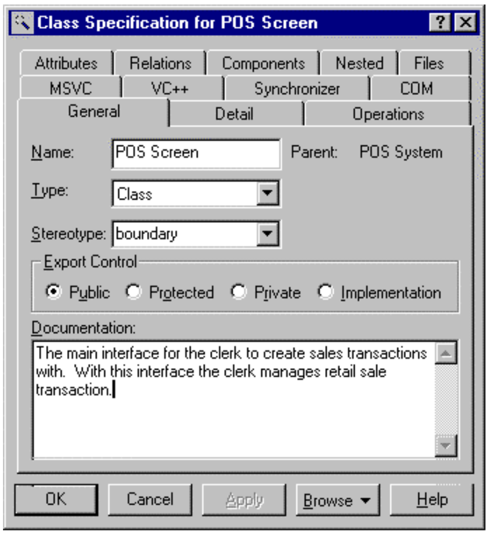
Statechart diagrams

You can use statechart diagrams to model the dynamic behavior of individual classes or objects. Statechart diagrams show the sequences of states that an object goes through, the events that cause a transition



from one state or activity to another, and the actions that result from a state or activity change.

A statechart diagram is typically used to model the discrete stages of an object’s lifetime, whereas an activity diagram is better suited to model the sequence of activities in a process.



Name: POS Screen Parent: POS System

Type: Class

Stereotype: boundary

Export Control

● Public ○ Protected ○ Private ○ Implementation

Documentation:

The main interface for the clerk to create sales transactions with. With this interface the clerk manages retail sale transaction.

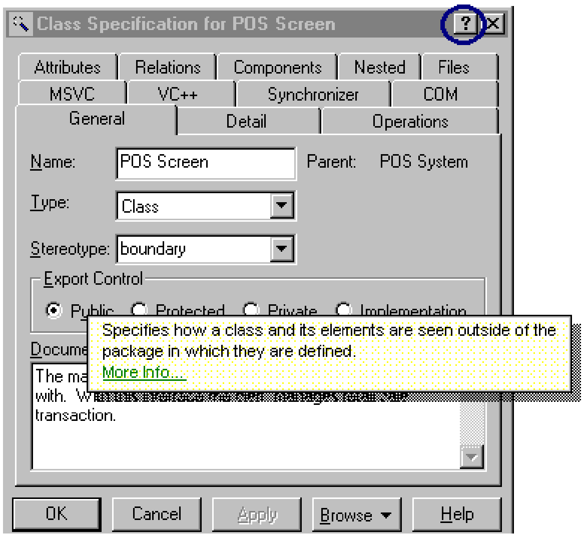
Specifications

Specifications are dialog boxes that allow you to set or change model element properties. Changes made to a model element either through the specification or directly on the icon are automatically updated throughout the model.

Getting help

As you work with Rational Rose, you can get additional help in the following ways:

- **The Help button**: Click this button to display help for a specific dialog box or specification.
- **What’s this**: Click on the **?** located on upper right corner of many dialog boxes. Then click on an area for which you want a brief description.
- **F1 or context-sensitive help**: Press F1 or the large **?** on the Rose toolbar to get information on the area of interest.



Name: POS Screen Parent: POS System

Type: Class

Stereotype: boundary

Export Control

● Public ○ Protected ○ Private ○ Implementation

Docume

The ma with. W transaction.

Specifies how a class and its elements are seen outside of the package in which they are defined.

More Info...

- **The Help menu**: click this menu to display a list of online documentation subjects.

总结

这部分介绍了可视化建模工具 Rational Rose 的一些主要特性，随着你继续进行其他部分，你会看到这个工具如何帮助你缩短开发周期，提高生产力，以及改进软件质量、团队沟通

下一步作什么 现在你已经开始建模了。下一部分将说明如何对一个业务建模，基于我们虚构的公司—Classics

第三部分　　业务建模

关于这部分

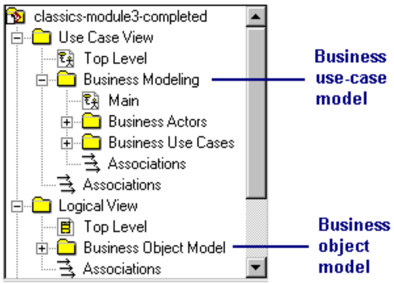
在这部分里你将学习如何用 Rational Rose 进行业务建模。教程介绍了 Classics 公司业务模型的各个部分以及利用它们来引入下列概念：

- 业务用例模型（业务角色和业务用例）
- 行为图表
- 业务对象模型（业务工作者和业务实体）

完成时间大约 30 分钟

业务模型

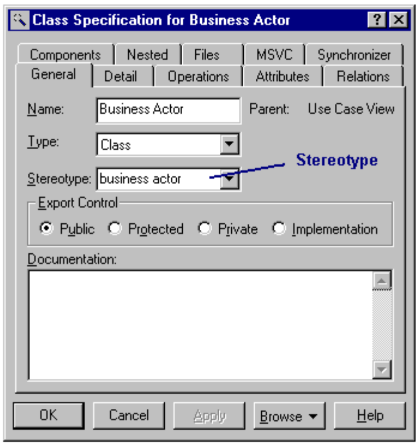
在软件开发的初期，Classics 公司管理层认识到需要纵观全部业务，而不是单个软件，以便建立一套系统来支持业务。一旦他们决定了需要检查全部业务，接下来就需要创建一个业务模型。业务模型由业务用例模型和业务对象模型构成，它是一个组织的简图，帮助你更好地理解你的软件所需要解决的问题。而且应用软件的需求也可以从业务模型中得到。Classics 的业务模型放在 Use Case View 里一个叫做“Business Modeling”的包里，业务对象模型放在 Logical View 下的一个叫做"Business Object Model"的包里。



业务模型的版式

业务用例模型包括业务角色和业务用例，业务对象模型包括业务工作者和业务实体。The following slides explain the meaning of each of those model element types.

Rational Rose provides specific stereotypes for the different model elements in the business model. A stereotype represents the subclassification of a model element. You define the stereotype of a model element in the Stereotype box in the Specification of the element. The picture shows the Specification for a business actor.A stereotype can have its own icon in Rational Rose. The icons below the Specification show how the different business model elements appear in diagrams.


Class Specification for Business Actor

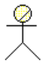
Name: Business Actor　Parent: Use Case View

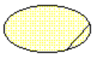
Type: Class

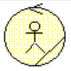
Stereotype: business actor

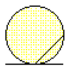
Export Control: ● Public ○ Protected ○ Private ○ Implementation

Documentation:


Business actor

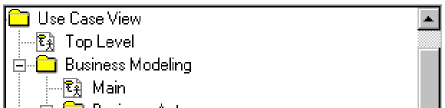

Business use case


Business worker


Business entity

The business use-case model

The business use-case model describes how the business is used by customers and partners. The model is defined in terms of business actors and processes, or as they are called in the UML, business use cases. The business use-case model is illustrated

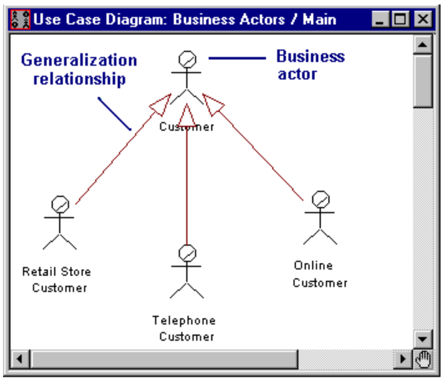


in several use-case diagrams. The use-case diagrams show how the business actors and business use cases are related. The Classics, Inc. business use cases are divided into several packages. Packages help you group similar elements together to manage complexity.

In addition to the packages for Retails Stores, Telesales, and Online Sales, there is a separate package, named Management, for supporting business use cases. The Management package defines workflows such as creating time cards for an employee.

Business actors

The business actors define the external entities and people with whom the business interacts. A business actor corresponds to a human user, but an information system that interacts with the business can also play the role of an actor. In Classics, Inc., the Customer is the most important actor. The Main diagram in the Business Actors package shows that there are three types of customers, Retail Store Customer, Telephone Customer, and Online Customer. The generalization relationships show that these three types of actors all inherit the properties of being a Customer.



试一试

1、Modeling the Classics, Inc. business

In this module's exercises you will update the Classics, Inc. business model to include an Internet store front business. You will create a new business actor, a use case package with two business use cases, an activity diagram, and an organizational unit with two business entities.

- First, start Rational Rose.
- If the **Create New Model** dialog box is displayed, click the **Existing** tab. Otherwise, click **File > Open**.
- Open the model file called **Module3\start\classics-module3-start.md**l, where " is the folder where the sample is installed.
- If you want the tutorial window to stay on top of the Rational Rose application window, right-click on the tutorial window and click **Keep Help on Top > On Top**.
- Click on the "+" to expand the Use Case View in the Rose browser.
- Click on the "+" to expand the Business Modeling package in the Use Case View. This is where the Classics, Inc. business use-case model is located.

2、Creating a business actor

The new Internet store front business attracts a new kind of customer, an Online Customer, which you are now going to create.

- Click on the "+" to expand the Business Actors package under Business Modeling in the browser.
- Open the Main diagram in the Business Actors package by double-clicking the diagram icon (Main) in the browser.
- Click on the diagram toolbox
- .Click next to the Telephone Customer in the open diagram to place the new actor.

A default name is supplied in the name box.

- With the default name still highlighted, type Online Customer (click on the default name if no longer highlighted). Then click somewhere in the diagram.
- Double-click the Online Customer actor to open its specification.
- In the **Stereotype** box, select **business actor**.

  In the **Documentation** box, type *Any customer that purchases items by using the online e-Commerce system.*

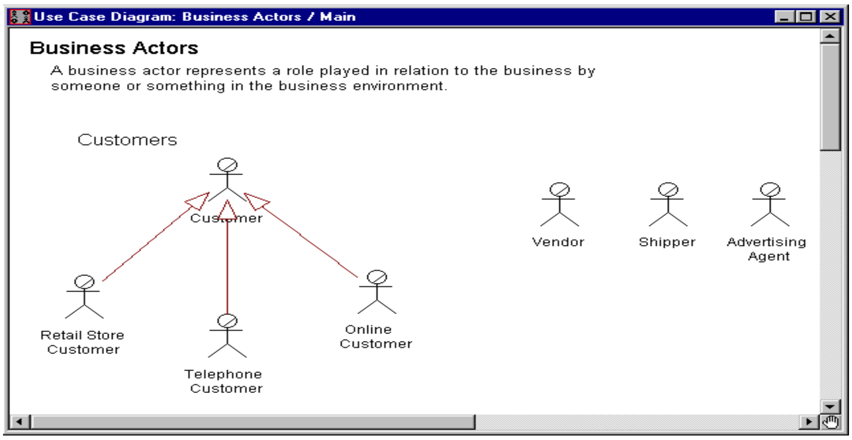
  (Each model element should have a short description to help other people to understand what the element represents.)
- Click **OK**.

3、Defining actor relationship

Like the Retail Store Customer and the Telephone Customer, the Online Customer is also a Customer. This is illustrated by a generalization relationship.

- To create a generalization relationship, click ↑ in the diagram toolbox.
- Click and drag the relationship from Online Customer to Customer.

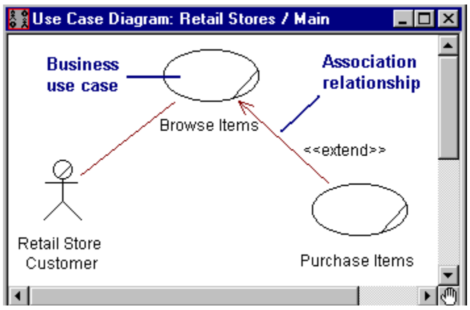
4、检查一下结果：



A business actor represents a role played in relation to the business by someone or something in the business environment.

Business use cases

The business use cases represent different workflows in the business. This diagram shows that Classics, Inc. has two business use cases related to the Retail Store business, Browse Items and Purchase Items, both of which support the Retail Store Customer. The association relationship between Purchase Items and Browse Items has the stereotype "extend". This relationship means that Purchase Items extends the Browse Items business use case. In other words, the customer can purchase an item at any time when browsing items.An association relationship between business use cases can also have the stereotype "include". Such a relationship shows that a business use case always includes the workflow of the other business use case.



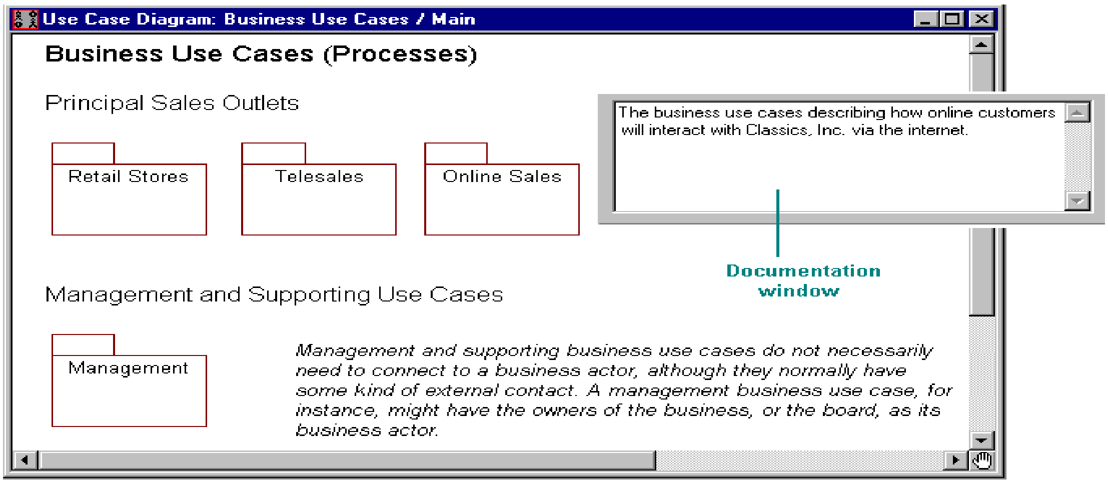
试一试：

1、Creating a business use-case package

The new Internet store front business has its own business use cases. You are now going to create a new business use case package for the new business use cases.

- Click the "+" to expand the Business Use Cases package under the Business Modeling package in the browser.
- Open the Main diagram in the Business Use Cases package by double-clicking the diagram icon in the browser.
- Click ▭ in the diagram toolbox and click next to the Telesales package in the open diagram to add the new package.
- With the name still highlighted, type Online Sales and click in the diagram.
- Instead of entering the description of the new package in its specification, as you did before, you will enter it in the documentation window, which is usually located under the browser. (If the documentation window is not visible, click here) Select the new package and type the following text in the documentation window:*Business use cases describing how online customers interact with Classics, Inc. via the Internet.*

检查一下结果：


Use Case Diagram: Business Use Cases / Main
Business Use Cases (Processes)
Principal Sales Outlets
Retail Stores
Telesales
Online Sales
Management and Supporting Use Cases
Management
Documentation window

The business use cases describing how online customers will interact with Classics, Inc. via the internet.

Management and supporting business use cases do not necessarily need to connect to a business actor, although they normally have some kind of external contact. A management business use case, for instance, might have the owners of the business, or the board, as its business actor.

2、Creating a business use case

One of the processes in the new Internet store front business is to browse an online catalog of product items. You are now going to add that business use case to the model.

- Every package in the model has a Main diagram. Open the Main diagram for the business use case package you just created, by double-clicking the package in the diagram.
- Add a new business use case, by clicking ⬭ in the diagram toolbox.
- Click somewhere in the open diagram to place the new use case there.
- Type Browse Online Catalog and click anywhere in the diagram.
- Double-click the use case and select **business use case** in the **Stereotype** box.
- Click **OK**.

3、Creating an association

The Browse Online Catalog business use case is used by the Online Customer. To show that the Online Customer interacts with the Browse Online Catalog business use case, you need to add an association between them. This exercise requires that you created a new actor, called Online Customer, in a previous exercise.

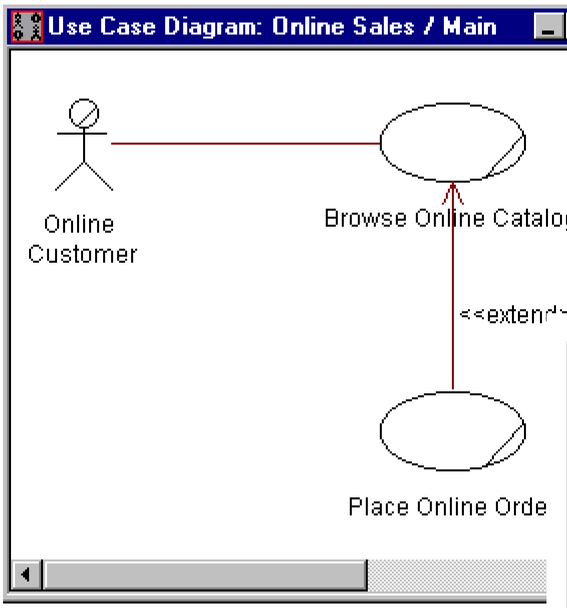
- Add the new actor to the diagram, by dragging 웃 Online Customer from the browser into the diagram.
- Click ↱ in the diagram toolbox.
- Click and drag from Online Customer, to the Online Catalog use case.
- The arrow indicates that the direction of the communication is from the actor to the business use case. Because the communication is bi-directional, you need to change the navigability of the association. To do that, right-click on the actor side (the Online Customer) of the association and click **Navigable**.
- The association becomes bi-directional, which is indicated by a straight line. This means that the direction of the communication is from the actor to the business use case and vice versa.

4、Extending a business use case

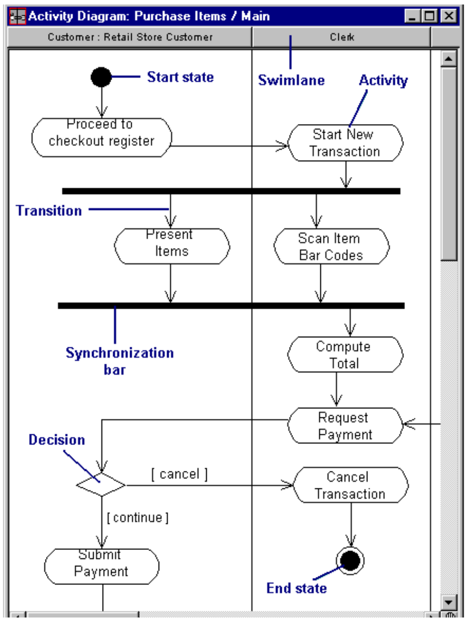
At any time when the customer browses the online catalog, the customer can purchase an item. To enable this optional feature, you will create a new use case that extends the Browse Online Catalog use case.

- Create a new business use case called Place Online Order. Do not forget to change the use case's stereotype. (If you need help, go back to the previous page where those steps were described.)
- Because the customer should be able to place an order at any time when he or she browses the catalog, the new use case needs to extend the workflow of the Browse Online Catalog use case. To define that relationship, create an association pointing from Place Online Order to Browse Online Catalog.
- Double-click the association and select **extend** in the **Stereotype** box.
- Click **OK**.

检查一下结果：



Activity diagrams



The workflows of the business use cases are described in detail using activity diagrams, which can have the following elements:

- A **start state** and an **end state**
- **Activities** represent a step in the workflow
- **Transitions** show what state follows another
- **Decisions** allow you to show alternative threads in the workflow
- **Synchronization bars** show parallel subflows
- **Swimlanes** represent business roles responsible for the contained activities

This diagram describes a part of the workflow of the Purchase Items business use case in Classics, Inc. The diagram shows that the flow starts with the customer coming to a checkout counter and it ends if the customer cancels the transaction.

试一试

1、Creating an activity diagram: Swimlanes

You are now going to describe the workflow for the Browse Online Catalog business use case by creating an activity diagram. Thus, this exercise requires that you created a new business use case, called Browse Online Catalog, in the previous exercise. In the diagram, you will model the steps a user might follow to browse items online, including what the application must do. In this scenario, the customer goes to the online store, looks at a catalog online or searches for a specific entry. Once an item is found, the customer can add it to the cart and browse for more items.

- Right-click the Browse Online Catalog use case in the Main use-case diagram and click **Select in Browser**. The use case is high-lighted in the browser.
- Right-click the high-lighted use case in the browser, and click **New > Activity Diagram.**
- Name the new diagram Main.
- Open the diagram by double-clicking it.
- Click the swimlane icon--□--from the diagram toolbox and click in the activity diagram to position the swimlane.
- Open the Swimlane Specification by double-clicking the header of the swimlane (called NewSwimlane) in the diagram.
- The activities in this swimlane will be performed by the Classics, Inc. customers. Therefore, name the new swimlane Customer.
- Because the Browse Online Catalog business use case interacts with on-line customers only, select **Online Customer** in the **Class** box.
- Click **OK**.
- Create a second swimlane, called *Online Store Front*, to the right of the first swimlane. The second swimlane represents the system you are going to develop.

2、Creating an activity diagram: Activity states

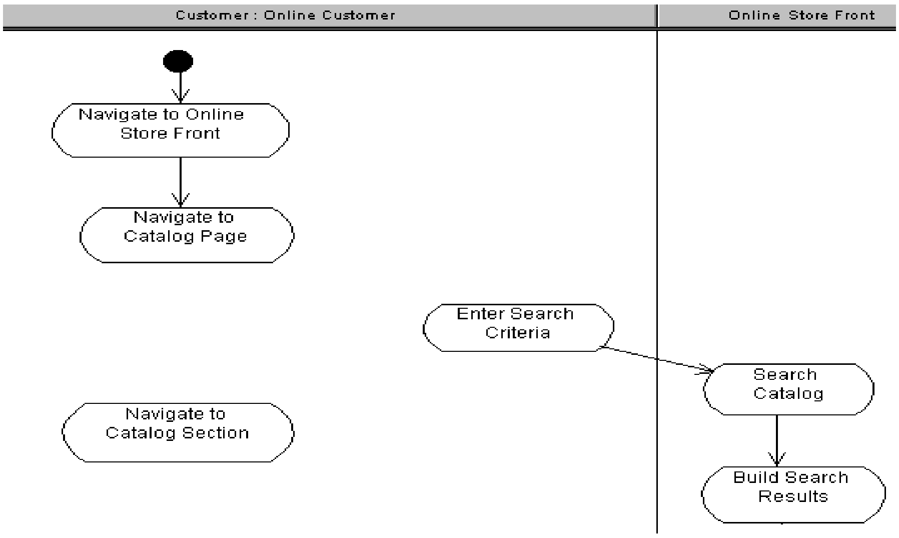
The next step is to create activities that represent the steps the customer might follow when browsing the online catalog. The starting activity that initiates the business process is when the customer navigates to the online store front.

- Click the start state icon—●—in the toolbox, and then click in the left swimlane.

The start state is deposited on the diagram and shows the beginning of the workflow.

- Click the activity icon—▭—in the toolbox, and then click in the left swimlane. The activity state is deposited on the diagram.
- Replace the default name "NewActivity" with *Navigate to Online Store Front.*
- To show that this activity follows the start state, click the state transition icon—↗—in the toolbox.
- Click and drag the transition from the start state to the activity.
- When the customer has navigated to the online store front, he or she navigates to a catalog page. Create a second activity under the first one, called *Navigate to Catalog Pag*e.
- Create a state transition from Navigate to Online Store Front to Navigate to Catalog Page.
- The customer can either navigate to a catalog section, where he or she selects an item, or search for a certain catalog item. Therefore, create a new activity under Navigate to Catalog Page, named *Navigate to Catalog Section.*
- Then create an activity called *Enter Search Criteria* in the left swimlane, and place it to the right of Navigate to Catalog Page. (In the next exercise, you will create the transitions to Navigate to Catalog Section and Enter Search Criteria using a decision point.)
- If the customer wants to search for an item, the system searches the catalog and creates a result. Therefore, create two activities in the right swimlane, named *Search Catalog and Build Search Results.*
- Create state transitions from Enter Search Criteria to Search Catalog, and from Search Catalog to Build Search Results.

检查一下结果：



3、Creating an activity diagram: Decision point

To show that the customer can either navigate to an online catalog section or search for a certain catalog item, you need to add a decision point.

- Click the decision icon—◇—in the toolbox, and then click below the Navigate to Catalog Page activity.
- Create the following state transitions:

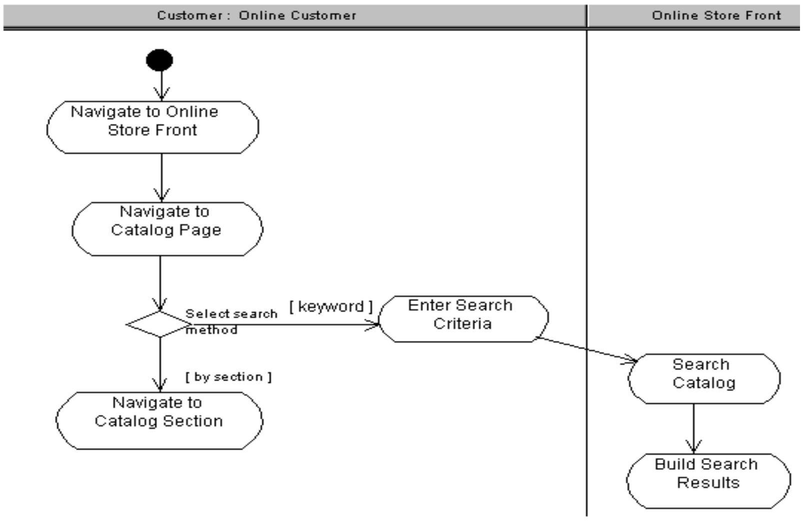
  - from Navigate to Catalog Page to the decision point
  - from the decision point to Navigate to Catalog Section
  - from the decision point to Enter Search Criteria

- Open the Decision Specification by double-clicking the decision point (or select Open Specification from the shortcut menu.)
- Enter the name of the decision point, *Select search method*, in the **Name** box.
- Select the **Transitions** tab. This tab displays the incoming and the two outgoing transitions that you just created, including the names of the destination activities.

To show that the customer can either navigate to an online catalog section or search for a certain catalog item, you need to add a decision point.

- Double-click the empty **Event** field associated with the transition to Navigate to Catalog Section to open its specification.
- On the **Detail** tab, enter a guard condition label called, *by section*, in the **Guard Condition** box. The guard condition means that this transition is triggered only when the user searches by section.
- Click **OK**.
- Open the State Transition Specification for the other outgoing transition. Give it the guard condition label keyword, which means that this transition is triggered when the user searched by keyword.
- Click **OK** twice. The guard condition labels are displayed beside the outgoing transitions on the activity diagram. (You may have to move the activities or labels to see the annotation clearly.)

检查一下：

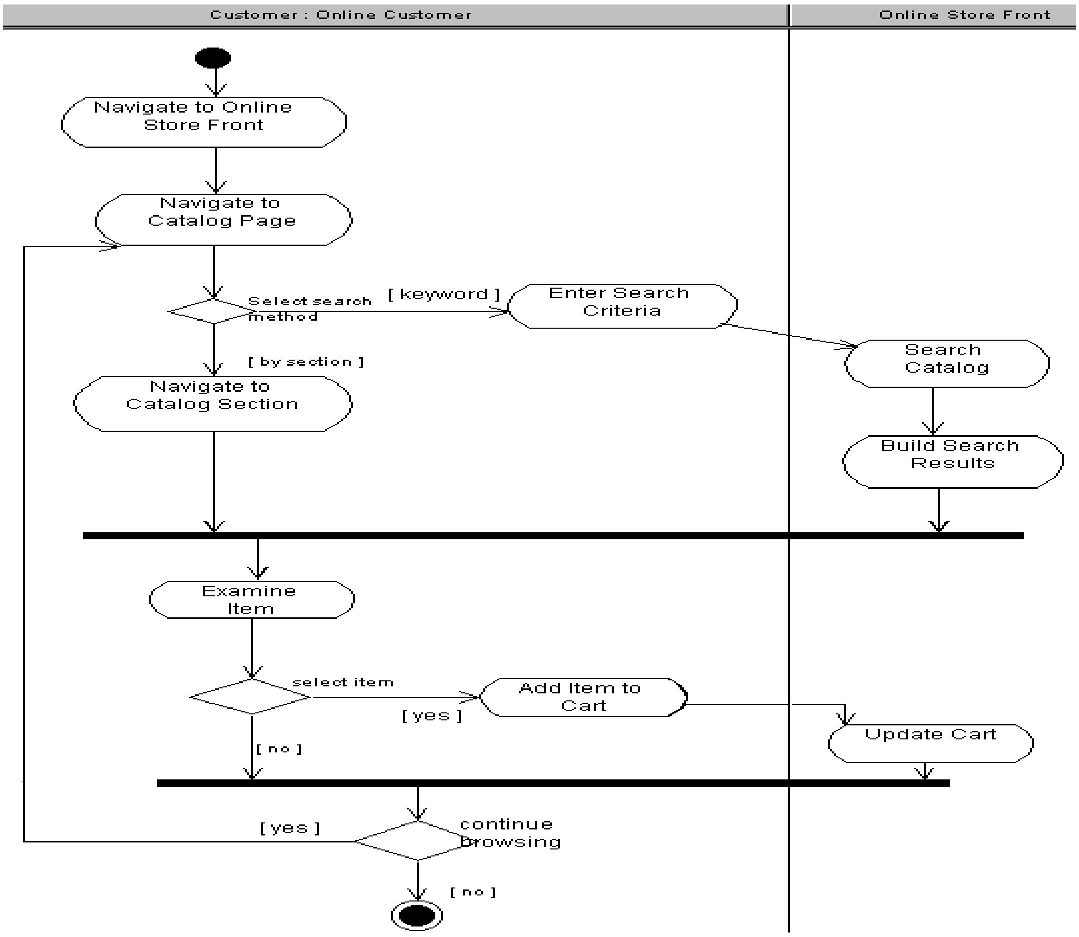


4、Completing the activity diagram

The next activity the customer performs is examining a catalog item. Before the customer can examine the catalog item, the item must be retrieved from the database. To show that the customer cannot examine the catalog item before it has been retrieved, you need to add a synchronization line to the diagram. You also need to add transitions from the Navigate to Catalog Section activity and the Build Search Results activity to the synchronization line.

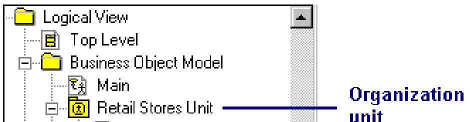
- Click the horizontal synchronization icon——▬——in the toolbox, and then click on the diagram.
- Resize the synchronization bar by dragging the edges of the bar to span the entire length of all the swimlanes.
- Create state transitions from the Navigate to Catalog Section activity and the Build Search Results activity to the synchronization line.
- You have not described the complete workflow yet, but you have created the most important types of elements in an activity diagram swimlanes, start state, activities, decision point, synchronization line. If you want some more practice, complete the activity diagram from looking at the resulting diagram.

检查一下结果：



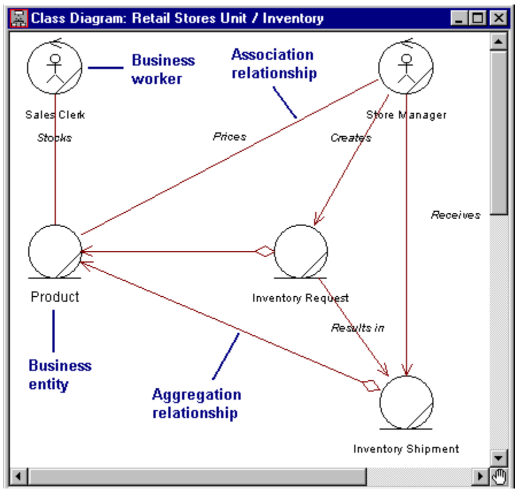
The business object model

The business object model identifies all "roles" and "things" in the business, which are represented as classes in the Logical View. There are two different types of classes in a business model: business workers and business entities. The type of a class is specified by its stereotype. A business worker is a



class with the stereotype "business worker" () and a business entity is a class with the stereotype "business entity" (.)The business object model can also identify organization units in the business. An organization unit is represented as a logical package with the stereotype "organization unit" (.) The Classics, Inc. business is divided into several organization units, as you can see in the picture.In the exercise at the end of this module you will create the Online Sales Unit for the business.

Business workers and entities



A business worker represents an abstraction of a human that acts within the business. Business entities represent "things" handled or used by the business workers as they execute a business use case. For example, a business entity represents a document or an essential part of a product.

The business object model is illustrated in class diagrams. The class diagrams show how the elements are related.

The displayed diagram shows the workers and entities that are involved during an inventory of the store. The association relationships show how the elements are related. An aggregation relationship is a special form of an association relationship, which shows that an object is composed of another object. For example, an Inventory Shipment has a Product.

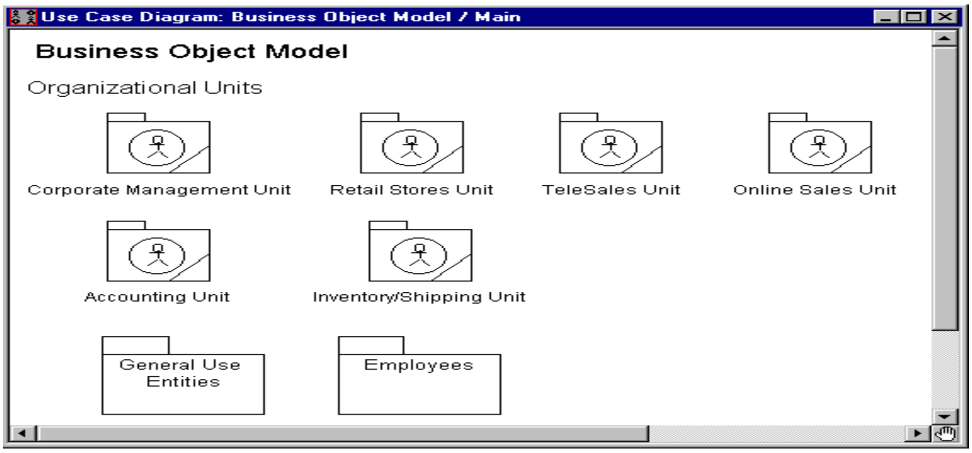
试一试：

1、Creating an organization unit

The Classics, Inc. business is structured in different departments, which are represented by organization units in the business object model. In this exercise, you are going to add a new organization unit for the Internet store business, which will contain some new business workers and business entities.

- From the Logical View in the browser, open the Business Object Model package to view the Classics, Inc. business object model.
- Open the Main diagram (Main) in the Business Object Model package.
- Click the package icon——in the diagram toolbox and place the new unit in the diagram.
- Give the new unit the name Online Sales Unit.
- Right-click the package and click **Open Specification**. (Double-clicking a package opens the main diagram for the package, not the specification.)
- Select **organization unit** in the **Stereotype** box.
- Click **OK**.

检查一下结果：



2、Creating business entities

The business model of Classics, Inc. has a business entity for an Order. However, the workers in the Online Sales Unit must use their own kind of order, which you will create as a new business entity. Also, a customer profile should be gathered for each customer as a result of an online order. Thus, a business entity for the customer's profile is needed in the Online Sales Unit.

- Double-click the Online Sales Unit package you just created to open its main diagram.
- A business entity is a class with the stereotype "business entity". Click the class icon—▤—in the diagram toolbox.
- Place the class in the diagram and name it Online Order.
- Assign **business entity** as the stereotype for the Online Order class. After changing the stereotype, the general class icon in the diagram changes into a business entity icon: ○
- Create another business entity in the same diagram. Give it the name *OnlineCustomerProfile.*

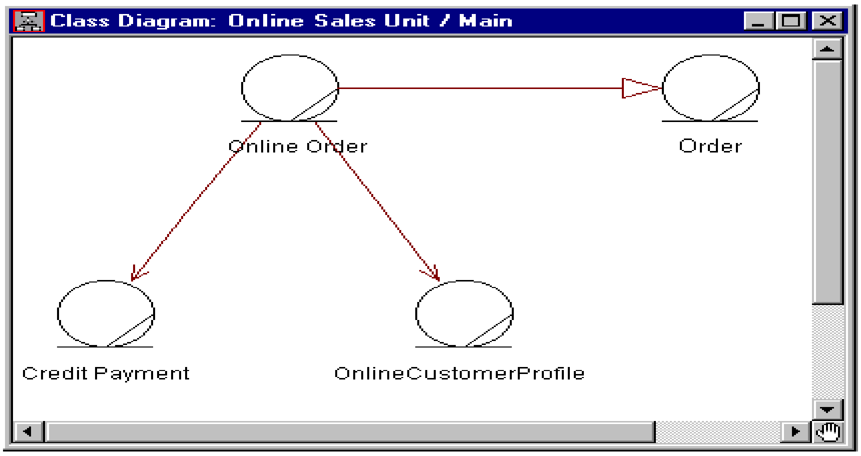
3、Defining relationships between business entities

To show that an Online Order has a customer profile and requires a credit payment, you need to add associations from Online Order to and OnlineCustomerProfile and Credit Payment. Also, because the Online Order entity is a special kind of order you need a generalization relationship between Online Order and Order.

- Click the ↱ diagram tool.
- Click and drag the association from Online Order to OnlineCustomerProfile.
- Add the Order entity to the diagram, by dragging ○ Order from the Logical View/Business Object Model/Inventory-Shipping Unit package in the browser onto the diagram.
- Create a generalization relationship between Online Order and Order, by clicking △ in the diagram toolbox. Make sure that the relationship points to Order.
- Drag the ○ Credit Payment business entity, located in the Logical View/Business Object Model/Retail Stores Unit package, from the browser onto the diagram.

- Finally, add an association between Online Order and Credit Payment.

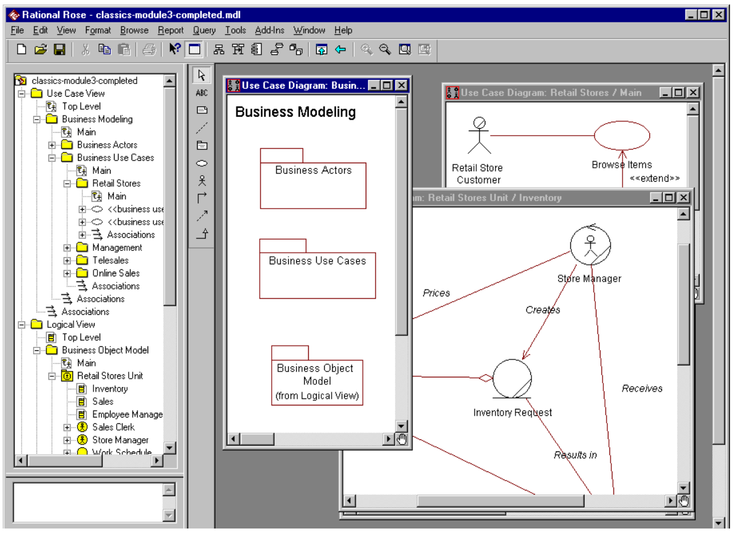
检查一下：



4、Congratulations!

You have completed the business modeling exercises.

Summary



In this module you learned how to use Rational Rose to model a business process. The purpose of a business model is to help you get a better understanding of the problem that your software must solve. In fact, the requirements for the application can be derived from the business model.

A business model consists of a **business use-case model** and a **business object model**. The business use-case model describes how the business is used by customers and partners. The workflows of the **business use cases** are described in detail using activity diagrams. The **business object model**, which contains business entities and business workers, identifies all "roles" and "things" in the business.

More information

For more information on how to create models in Rose, please refer to the "Using Rational Rose 2000" manual.

For information on how to perform business modeling, please refer to:

- The Rational Unified Process product
- "The Rational Unified Process, an Introduction" by Philippe Kruchten
- "The Object Advantage - Business Process Reengineering with Object Technology" by Ivar Jacobson, et al

What’s next?

So far, you have examined and updated a model of the Classics, Inc. business. Next step is to examine and update a model of the supporting application. In the next module, you will learn how to define the requirements on the Classics, Inc. application in a use-case model. Click Main Menu to get to the main menu where you start the Modeling Application Requirements module.

第四章 建模应用软件需求

一、关于这一章

In Module 3, you modeled a business process, defined uses for business objects, and defined how business use cases can be described in activity diagrams. You learned about business modeling.

In this module, you develop a software system use-case model that supports the business modeled in module 3. Most of the use case modeling is already complete as you start Module 4. All you need to do in this module is add the new Ecommerce system use cases and actors so customers can shop at home through the World Wide Web.

大约需要 30 分钟

二、The ecommerce system

The Classics Inc. management team decided that the functionality of the Ecommerce system would be simple for this iteration. The Ecommerce system allows internet users to do two things:

- Browse the Classics Inc. online catalog
- Purchase items online with a valid credit card

The Classics Inc. management team might include enhanced Ecommerce functionality in future iterations of the software. For example, Classics Inc. could monitor order status and allow customers to preview musical selections before a purchase.

三、The vision document

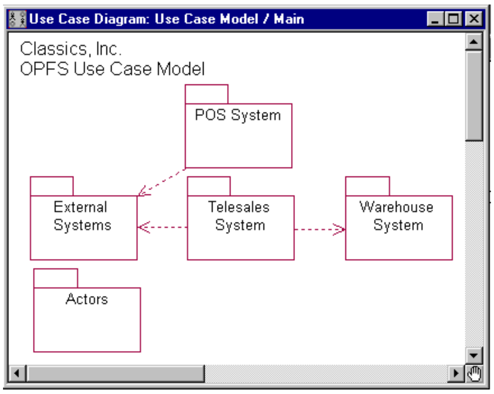
One of the **artifacts** that the management team created is the Classics Inc. Vision document. A **Vision Document** is created early in this phase and is used as input for use-case modeling. A vision document captures very high-level requirements that give the reader an understanding of the software features. **Stakeholders** may also use vision documents as the beginning of software requirements.

四、What does a vision document do?

A vision document:

- Explains user demographics, profiles, environments and requirements
- Expresses the vision in terms of its use cases
- Describes briefly the business opportunity addressed by the project
- Identifies system capabilities required to deliver benefits to the user

For more information on a vision document, refer to the Rational Unified Process (RUP). To view the Classics Inc. Vision Document, 请见附录 .



五、Let's take a look at the use-case model

You should become familiar with the following model elements on the OPFS Use Case Model:

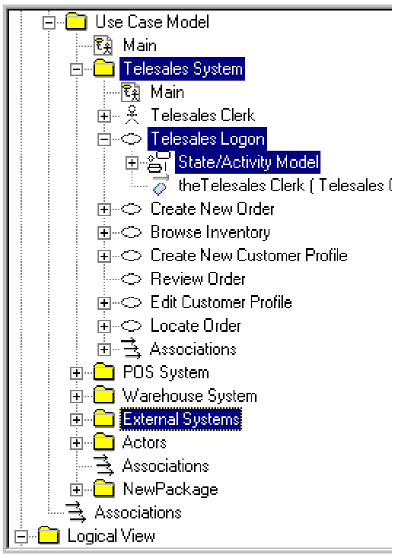
- The Point of

Sales (POS) System package and the large number of use cases it contains

- The External Systems package
- The Telesales System which contains the process phone order functionality
- The Actors package that contains all of the actors that participate in multiple use case packages
- The Warehouse system that is responsible for managing the inventory

Remember, this tutorial does not step you through every model element or diagram. Please browse through the OPFS Use Case Model packages listed above on your own.

六、Let's take a look at some specific use-case model elements



```
⊟ Use Case Model
    Main
  ⊟ Telesales System
      Main
    ⊞ Telesales Clerk
    ⊟ Telesales Logon
      ⊞ State/Activity Model
        theTelesales Clerk ( Telesales (
    ⊞ Create New Order
    ⊞ Browse Inventory
    ⊞ Create New Customer Profile
      Review Order
    ⊞ Edit Customer Profile
    ⊞ Locate Order
    ⊞ Associations
  ⊞ POS System
  ⊞ Warehouse System
  ⊞ External Systems
  ⊞ Actors
    Associations
  ⊞ NewPackage
  Associations
⊟ Logical View
```

View and examine the Classics, Inc. use-case model. The purpose of this exercise is to familiarize you with the additions that have been made since module 3. More specifically, you will examine:

- The Telesales package that includes facilities to process phone orders
- The Telesales Logon use case that allows an actor to logon to the system along with the associated activity diagram (State/Activity Model)
- The External Systems package that includes any system that the **Order Processing Fulfillment System** (OPFS) must work with.

试一试：

1、Modeling application requirements

Follow these steps to familiarize yourself with some of the changes since Module 3:

- Start Rational Rose.
- Click **File > Open**.
- Open the model file (.mdl) called classics-module4-start.mdl.
- Expand (by clicking on the + sign) the Use Case View in the browser so the Top Level diagram, Business Modeling package, and Use Case Model package are visible.
- Expand the Use Case Model package.
- Expand the Telesales System package.

The Telesales System package acts as a container that stores all of the use cases related to telesales. Notice the telesales clerk and the many use cases contained by the Telesales System package, including:

- Browse Inventory

- Create New Customer Profile
- Edit Customer Profile

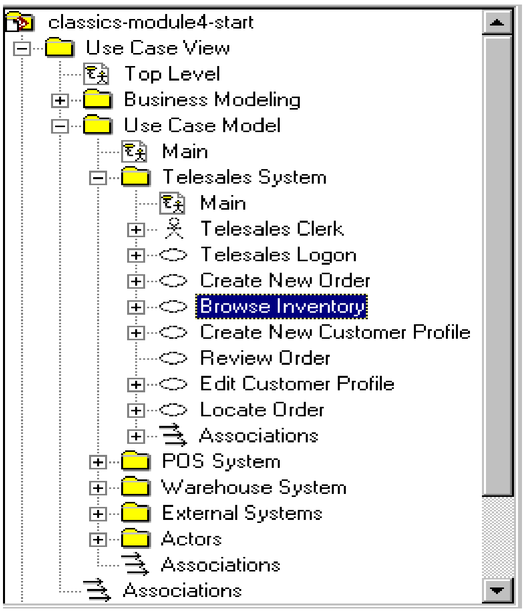
The Telesales system is important to the Classics Inc. employees because it helps them enter new orders and perform other customer-related tasks.

In many models, it is often possible to have hundreds of model elements in the browser and it may be difficult to locate a specific item. However, there is a quick and convenient way to locate a model element so you can see where you are in the model hierarchy. To see how this feature works, use the following steps:

- Double-click the Main diagram under the Telesales System package.
- Select the Browse Inventory use case on the Main diagram.
- Right-click on the Browse Inventory use case.
- Click, **Select in Browser**.

Notice how the Browse Inventory use case appears highlighted in blue. This tip works on any diagram or model element that appears in the browser.

检查一下：

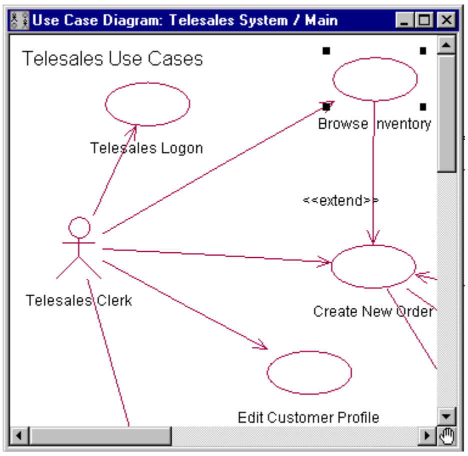


- classics-module4-start
  - Use Case View
    - Top Level
    - Business Modeling
    - Use Case Model
      - Main
      - Telesales System
        - Main
        - Telesales Clerk
        - Telesales Logon
        - Create New Order
        - Browse Inventory
        - Create New Customer Profile
        - Review Order
        - Edit Customer Profile
        - Locate Order
        - Associations
      - POS System
      - Warehouse System
      - External Systems
      - Actors
      - Associations
    - Associations

Use cases

A use case describes the behavior of the system, including the interaction between the actor and the system. In this diagram, the Telesales Clerk interacts with the Browse Inventory use case by browsing and searching for items in the inventory system.In general terms, a use case is:

- A pattern of behavior the system exhibits
- A sequence of related transactions performed by an actor and



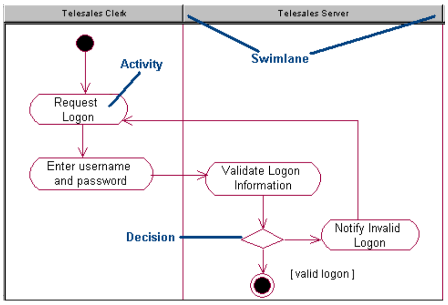
the system

- A system or thing that delivers something of value to the actor.

Sometimes you cannot cover all the needs of a system in one use case. It is usual to place and organize a collection of use in various packages.

Activity diagrams

An activity diagram is a view of a state machine that models business or object workflow. You can have only one state machine per use case, package, or class; however, you can have multiple activity diagrams per state machine.Activity diagrams are similar to statechart diagrams but they have a distinct difference. Activity diagrams model workflow whereas statechart diagrams model the various states that an object is in throughout its lifetime. Statechart diagrams are discussed in detail in module 5.Classics Inc. uses activity diagrams throughout the system when they want to expand upon complex model elements and processes.



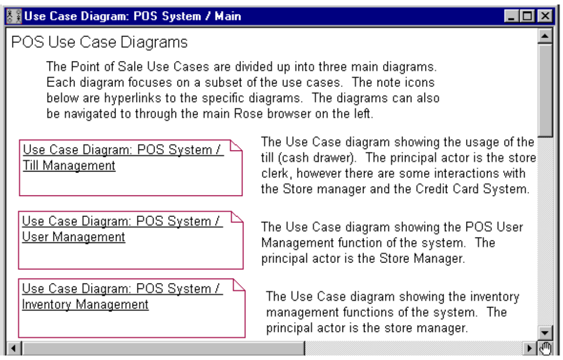
Classics Inc. used an activity diagram to show the workflow of an On Line customer interacting with the On Line Store front in a business use case.

The activity diagram on the left models the Telesales Logon found under the Telesales System package. This diagram shows the workflow of a Telesales clerk connecting with the Telesales server.

Pay particular attention to the use of swimlanes in this diagram. Swimlanes represent organizational units within activity diagrams.

For example, look at how the Request Logon and Enter username and password are located in the Telesales Clerk swimlane, while the Validate Logon Information and Notify Invalid Logon are located in the Telesales Server swimlane.

Linking diagrams to notes



Use Case Diagram: POS System / Main

POS Use Case Diagrams

The Point of Sale Use Cases are divided up into three main diagrams. Each diagram focuses on a subset of the use cases. The note icons below are hyperlinks to the specific diagrams. The diagrams can also be navigated to through the main Rose browser on the left.

Use Case Diagram: POS System / Till Management

The Use Case diagram showing the usage of the till (cash drawer). The principal actor is the store clerk, however there are some interactions with the Store manager and the Credit Card System.

Use Case Diagram: POS System / User Management

The Use Case diagram showing the POS User Management function of the system. The principal actor is the Store Manager.

Use Case Diagram: POS System / Inventory Management

The Use Case diagram showing the inventory management functions of the system. The principal actor is the store manager.

The POS System package contains a large number of use cases. In order to make the use cases more understandable, the Classics Inc. analysis team divided the use cases up into three separate diagrams: Till Management, User Management, and Inventory Management.The POS System main diagram (to the left) contains three notes that link you to each POS system use-case diagram. You can hyperlink any diagram to a note by dragging the diagram from the browser to a note that 担 been placed on a diagram. Once linked, you can double-click on a note in the POS System main diagram to jump you to the diagram.

Adding the ecommerce package

Now that you have viewed and become acquainted with the new diagrams and model elements created between module 3 and where you are now, begin the interactive section. In this exercise, you update the Classics, Inc. model to include an Ecommerce package. The Ecommerce package contains an online catalog and an order-entry system for online sales which enables customers to locate, browse, and buy items through the Classics, Inc internet web site. Begin to add the Ecommerce System Package:

试一试：

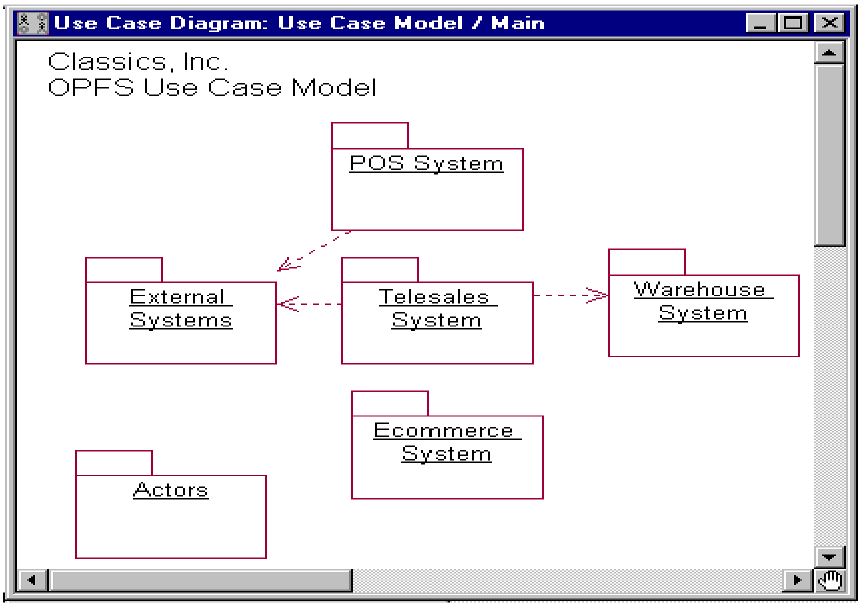
1、Adding the ecommerce package

- Expand the Use Case View in the browser so the Top Level diagram, Business Modeling package, and Use Case Model package are visible.
- Double-click the Top Level diagram.
- Double-click the Use Case Model package in the diagram window to display the OPFS Use Case Model.
- Click (Package) in the diagram toolbox.
- Click underneath the Telesales System package in the diagram to place a new package.
- Click in a blank area in the diagram.
- Right-click the package.
- Select **Open Specification.**
- Type *Ecommerce System* in the package specification name field.
- Type the following text into the Package Specification Documentation field:

The home shopping e-commerce system will initiate Classics Inc.'s presence on the World Wide Web, including an online catalog for web visitors to browse and an order-entry capability supporting online sales and order fulfillment, interfacing with the Order Processing system.

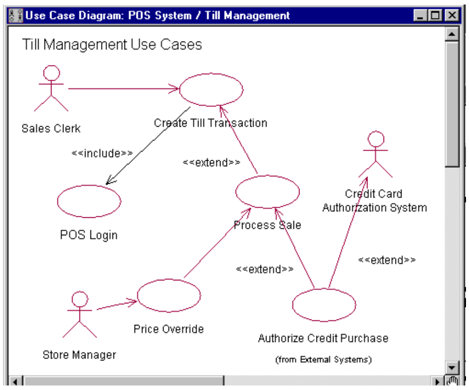
- Click **OK** to close the Package Specification.

检查一下结果：



Unidirectional associations

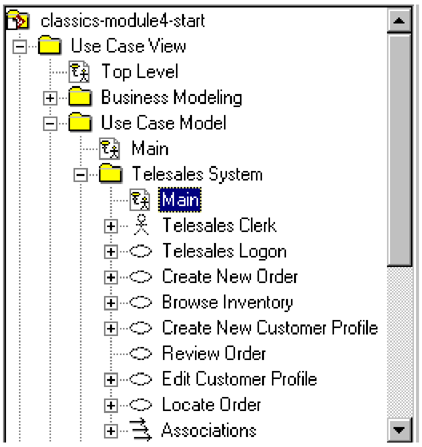
Association relationships connect similar model elements on use case diagrams. In this module we see both <<extends>> and <<includes>> relationships. Many times, the same system functionality is shared by several different use cases.



A unidirectional association provides a pathway for communication. The communication can be between use cases and actors, between two classes or between a class and an interface. Associations are the most general of all relationships and consequentially the most semantically weak. If two objects are usually considered independently, the relationship is an association.

Use-case diagrams

Use-case diagrams graphically depict system behavior. These diagrams present a high-level view of the system from an outsider 抯 perspective. A use-case diagram can depict all or some of the use cases of a system. During requirement analysis, create a use-case diagram to show system behavior. The browser on the left shows the main use case diagram for the Telesales System package.



- classics-module4-start
  - Use Case View
    - Top Level
    - Business Modeling
    - Use Case Model
      - Main
      - Telesales System
        - Main
        - Telesales Clerk
        - Telesales Logon
        - Create New Order
        - Browse Inventory
        - Create New Customer Profile
        - Review Order
        - Edit Customer Profile
        - Locate Order
        - Associations

What does a use-case diagram contain?

A use-case diagram contains:

- Actors - "things" outside the system
- Use cases
- **Interactions** or **relationships** between actors and use cases, including the **associations** and generalizations .

The OPFS has many use cases that define the requirements of the system. For example, all of the use cases for the Telesales package describe the following various requirements:

- Entering new telephone orders from customers
- Processing credit card payments
- Providing status reports

Creating a Use-Case Diagram

When you double-click on any package on a diagram, Rational Rose creates a new use-case diagram. Click on the Try it button below to create and name the Ecommerce System Use-Case diagram.

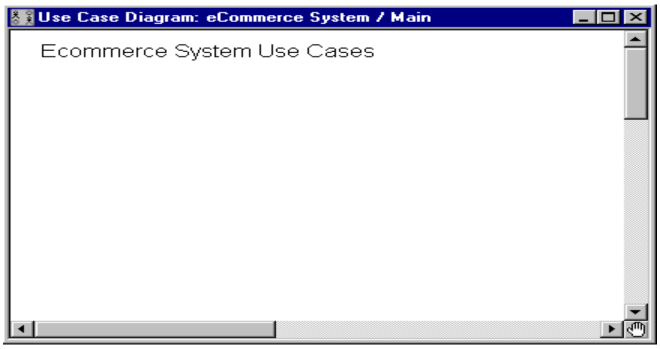
试一试：

1、Creating a use-case diagram

- Double-click the Ecommerce System package. Rose automatically creates a new use-case diagram after you double-click on any package. The use-case diagram will appears blank.

- Click ABC (Text Box) in the toolbox.
- Click the text pointer | near the top of the diagram window to create a text box.
- Type *Ecommerce System Use Cases* as for the diagram name. Labeling diagrams is important because it makes diagrams more understandable, especially to those that are looking at the diagram from outside the model.
- Click on any white space on the diagram to close the text box.

检查一下：

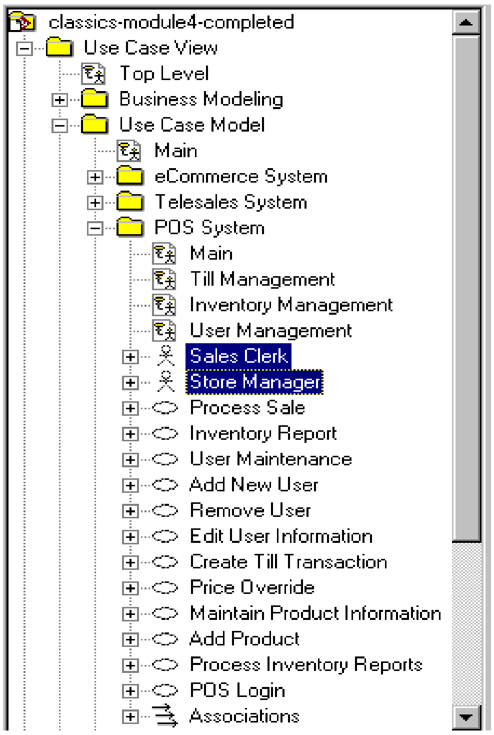


Ecommerce System Use Cases

Actors

Actors represent system users and appear as stickmen in the browser and on diagrams. They help define the system and give a clearer picture of what the system should do. For example, an actor can provide input to a system and it can receive information from the system. It is important to note that an actor interacts with, but has no control over use cases. An actor may be a human, a hardware device, or another system.

Create an actor named Online Customer that will browse or purchase items from Classics Inc. online store in the next Try it sequence.



```
classics-module4-completed
  Use Case View
    Top Level
    Business Modeling
    Use Case Model
      Main
      eCommerce System
      Telesales System
      POS System
        Main
        Till Management
        Inventory Management
        User Management
        Sales Clerk
        Store Manager
        Process Sale
        Inventory Report
        User Maintenance
        Add New User
        Remove User
        Edit User Information
        Create Till Transaction
        Price Override
        Maintain Product Information
        Add Product
        Process Inventory Reports
        POS Login
        Associations
```

Namespace

In this Try it sequence, it is important that you enter the name of the actor exactly as the instructions state. For example, if you enter the name of the actor as On Line Customer (with a space between On and Line) instead of Online Customer, you will receive a message stating that the On Line customer exists in another area of your model. Rational Rose warns you of multiple namespaces in case you want to reuse the element instead of creating a new element. When different elements have the same name, the elements are "verloaded".

试一试：

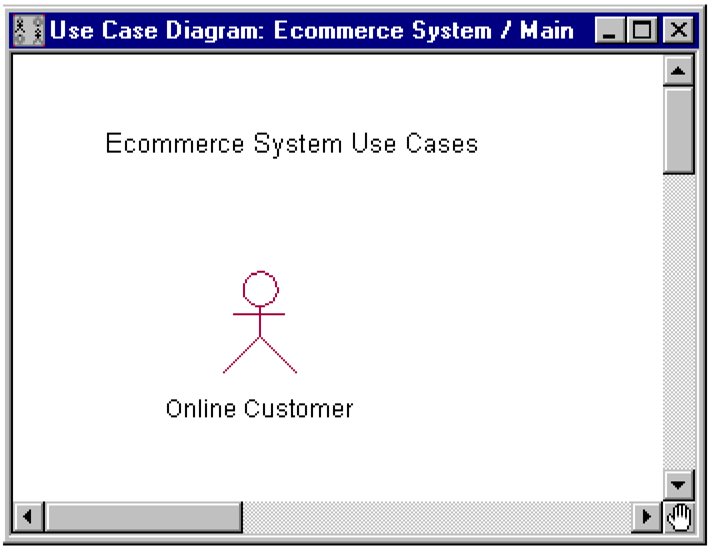
1、Creating an actor

- Click 웃 (Actor) in the diagram toolbox.
- Click on the left-hand side of the diagram window to create the new actor.
- Right-click on the new actor you just created to display a shortcut menu.
- Select, **Open Specification**.
- Name the actor by typing *Online Customer* in the Name field.
- Type the following text into the Documentation field of the Class Specification:

An Online Customer is any customer that browsers or purchases items from Classics, Inc. On Line Store. These actors access the Classics, Inc. Ecommerce system via the internet with any standard HTML 3.2 compliant web browser.

- Click OK to close the Class Specification.

检查一下结果：


Use Case Diagram: Ecommerce System / Main
Ecommerce System Use Cases
Online Customer

Creating the browse catalog use case

The Browse Catalog use case illustrates how customers can browse through items in the Classics, Inc. inventory. Customers can find items by searching for keywords in product names or product descriptions and by navigating the catalog by section

试一试：

1、Creating the browse catalog use case

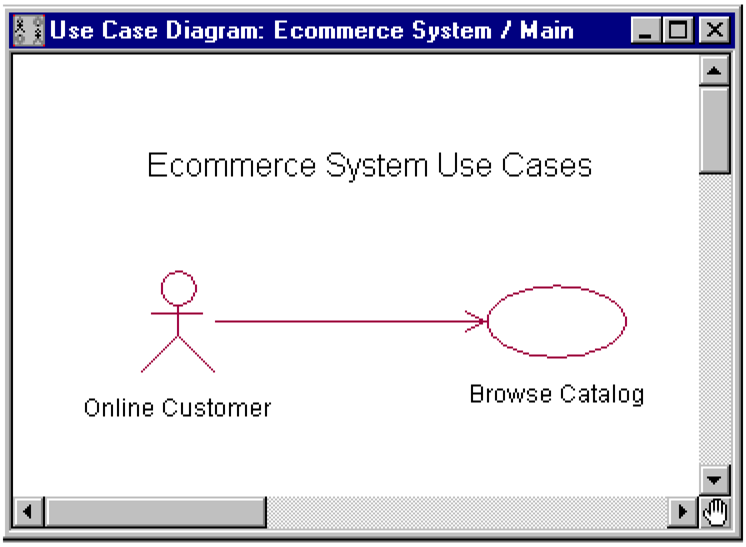
- To create a use case, click ⬭ in the diagram toolbox.
- Click in the middle of the diagram to place a new use case.
- Right-click on the new use case you just created to display a shortcut menu.
- Select, **Open Specification**.
- Name the use case by typing Browse Catalog in the Use Case specification Name field.
- Type the following text into the Documentation field of the Browse Catalog specification:

The online customer browses items in the Classics, Inc. inventory. Items can be found by either searching for keywords in the products name or descriptions, or by navigating the catalog by section. While browsing items the customer can add or remove items from a *virtual shopping cart.*

2、Creating a unidirectional association

- Click OK to close the Use Case Specification.
- Click (unidirectional association) from the diagram toolbox to connect an actor and a use case. The pointer changes to a vertical arrow ?as it 扭 moved.
- To add a unidirectional association, click on the Online Customer actor and drag a line to the Browse Catalog use case.
- Release the pointer so a unidirectional association connects the Online Customer actor and Browse Catalog use case.

检查一下结果：


Use Case Diagram: Ecommerce System / Main
Ecommerce System Use Cases
Online Customer
Browse Catalog

3、Adding the place order use case

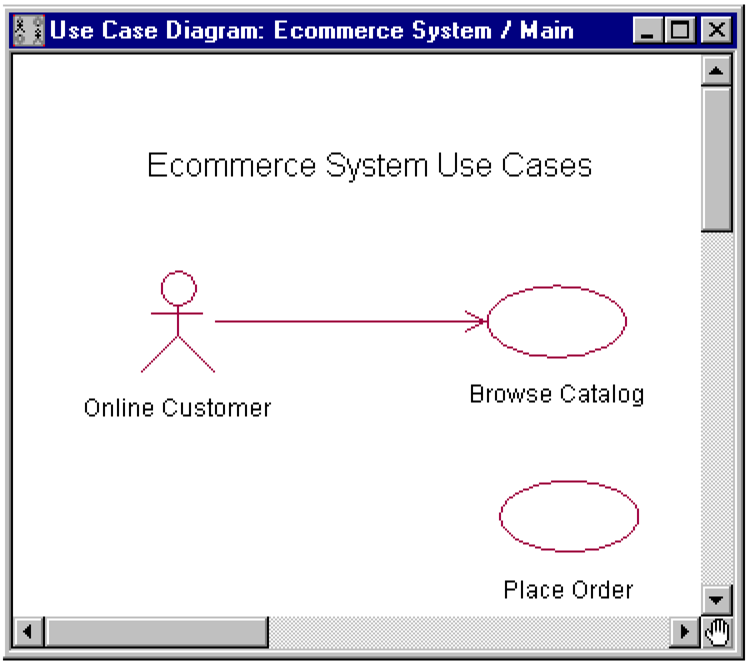
The Classics, Inc. online store must now provide a way for customers to place an order. The customer must provide contact and credit card information to the system. In return, the system will authorize the credit card purchase and create a new order in the warehouse system. Create a Place Order use case:

- Click (Use Case) in the diagram toolbox.
- Place a use case below the browse catalog use case.
- Name the use case by typing Place Order.
- Type the following text into the Documentation field of the Browse Catalog specification:

The customer, having decided to purchase the items in the current shopping cart, places an order with Classics, Inc. The customer provides customer contact and payment information to the system. The system in turn authorizes the credit card purchase, and creates a new order in the warehouse system to be filled.

- Click OK to close the Place Order Specification.
- Click OK to remove the namespace dialog box.

检查一下结果：

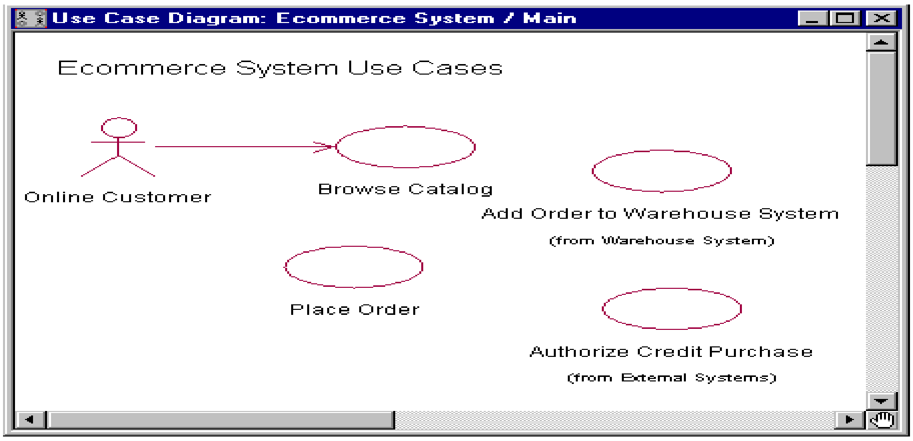

Use Case Diagram: Ecommerce System / Main
Ecommerce System Use Cases
Online Customer
Browse Catalog
Place Order

4、Adding existing use cases to the diagram

Add two use cases that already appear in other locations in the model so the Ecommerce system can notify the warehouse about the order and the credit card company can verify the credit of the customer. Drag and drop the Add Order to Warehouse System use case and the Authorize Credit Purchase use case to the Ecommerce System use case.

- Expand the Use Case Model package in the browser.
- Expand the Warehouse System and External System packages that reside under the Use Case model in the browser.
- Drag and drop the Add Order to Warehouse System use case on the diagram.
- Drag and drop the Authorize Credit Purchase use case on the diagram.Notice that each use case you added contains (from External Systems) under the Authorize Credit Purchase use case and (from Warehouse System) under the Add Order to Warehouse System use case. The text that appears within parentheses tells you where the use case resides within the model hierarchy.

检查一下结果：


Use Case Diagram: Ecommerce System / Main
Ecommerce System Use Cases
Online Customer
Browse Catalog
Add Order to Warehouse System
(from Warehouse System)
Place Order
Authorize Credit Purchase
(from External Systems)

Extend and include relationships

Association relationships connect similar model elements on use case diagrams. In this module we see both <<extends>> and <<includes>> relationships. Many times, the same system functionality is shared by several different use cases. An extends relationship between use cases means one use case optionally adds the functionality of the other use case when certain conditions are met. An includes relationship means one use case completely encompasses all the functionality of another use case.

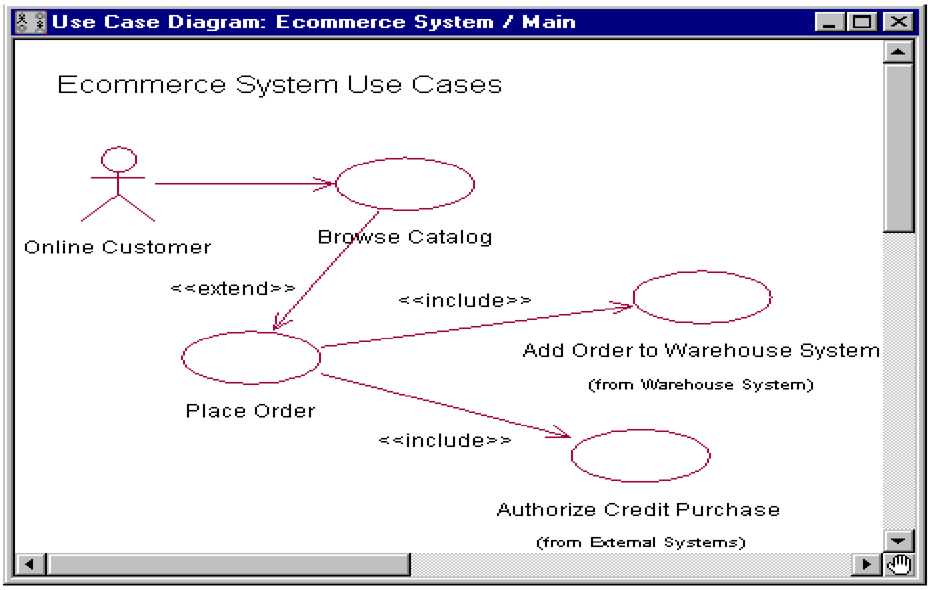
试一下：

1、Adding extend and include relationships

To show how the Browse Catalog use case and the Place Order use case interact, we must create an extend relationship between both use cases. The extend relationship is significant because it implies that both use cases are connected, but the Place Order use case is conditional based upon what happens in the Browse Catalog use case. If the customer does not buy anything, the use case is not "extended" to the Place Order use case. However, if the customer decides to purchase something, the Browse Catalog use case is extended to the Place Order use case.

- Click ↱ (Unidirectional Association) from the diagram toolbox.
- Click on the Browse Catalog use case and drag a line to the Place Order use case.
- Double click on the Unidirectional Association between the Browse Catalog use case and the Place Order use case to open the Association specification.
- Select **extend** from the Stereotype drop-down menu.
- Click **OK** to close the Association Specification.
- Click ↱ (Unidirectional Association) from the diagram toolbox.
- Click on the Place Order use case and drag a line to the Add Order to Warehouse System use case.
- Double-click on the Unidirectional Association to display the specification and select **Include** from the Stereotype drop-down menu.
- Click **OK** to Association specification.
- Repeat the same steps to create an association with an include relationship between the Place Order use case and Authorize Credit Purchase use case.
- Collapse the all of the elements in the browser so the Top Level diagram, Business Modeling package, and Use Case Model package are visible.

检查一下结果：


Use Case Diagram: Ecommerce System / Main
Ecommerce System Use Cases
Online Customer
Browse Catalog
<<extend>>
<<include>>
Add Order to Warehouse System
(from Warehouse System)
Place Order
<<include>>
Authorize Credit Purchase
(from External Systems)

Summary

In module 4, you have established the system requirements between the customers and users on how and what the Ecommerce system should do. Further, you have learned how to develop a use-case model to represent system functionality. Some of the model elements you learned about in developing Classics, Inc. Ecommerce system are:

- actors
- use cases
- packages
- use-case diagrams

You can check the diagrams you created and modified with theclassics-4-completed.mdl file.

What's next?

In the next module, you are going to help create the analysis model. The analysis model is an object model that describes the realization of use cases, and serves as a simplification of the design model

Some of the features that you will be working with are object classes, class diagrams, sequence diagrams, and statechart diagrams.

第五章　　创建分析模型

关于本章

Module 5 examines the analysis model and associated analysis objects. The tutorial explains the different parts of the analysis model and introduces you to the following concepts:

- Analysis class stereotypes
  - Boundary
  - Control
  - Entity
- Use-case realizations
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams

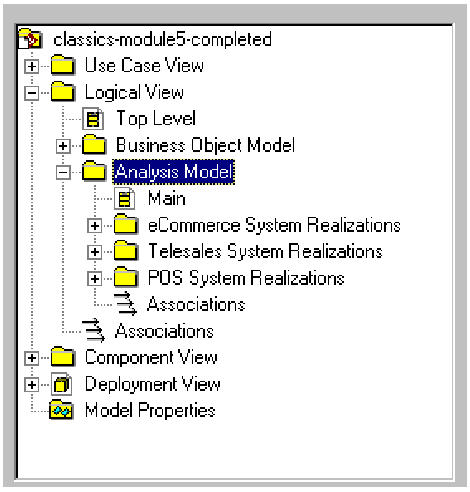
大约需要 30 分钟。

分析模型

In Rational Rose all diagrams in a model represent the different views of the same system. Each view is a different level of abstraction.

The analysis model represents a high-level overview of the Ecommerce, Telesales, and POS systems, including many analysis level classes stereotyped as boundary, controller and entity. Module 5 teaches you more about the analysis level classes later.

The goal of an analysis model is to create a preliminary mapping of required behavior onto modeling elements in the system. In most cases, it omits the detail of a design model in order to provide an overview of the system functionality.

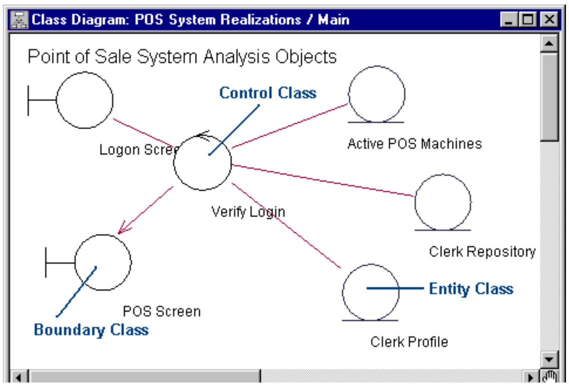


- classics-module5-completed
  - Use Case View
  - Logical View
    - Top Level
    - Business Object Model
    - Analysis Model
      - Main
      - eCommerce System Realizations
      - Telesales System Realizations
      - POS System Realizations
      - Associations
    - Associations
  - Component View
  - Deployment View
  - Model Properties

The analysis model eventually transitions into the design model, and the analysis classes directly evolve into design model elements.

Analysis classes

One of the most common groups of model elements found in the analysis model are the analysis classes, or sometimes called analysis objects. The analysis classes are stereotyped classes that represent an early conceptual model for elements in the system that have responsibility and behavior. There are three types of analysis classes and they are used throughout the analysis model:

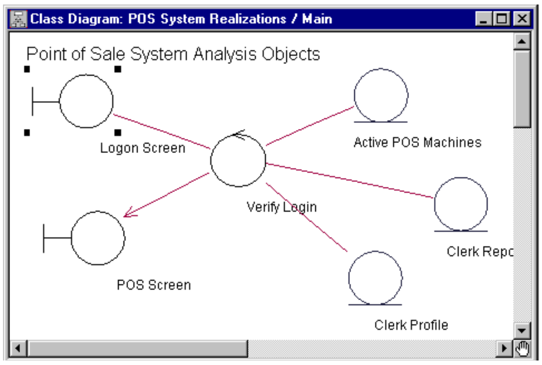


- Boundary
- Control

- Entity

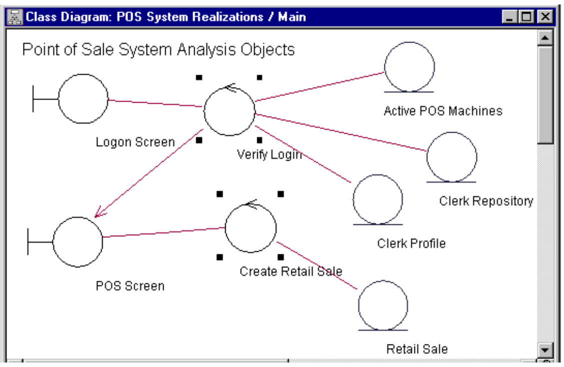
Before you examine the model, it is important that you understand these analysis classes. The POS System analysis diagram contains all three types of analysis classes.

Boundary classes


Class Diagram: POS System Realizations / Main
Point of Sale System Analysis Objects
Logon Screen
Active POS Machines
Verify Login
Clerk Repo
POS Screen
Clerk Profile

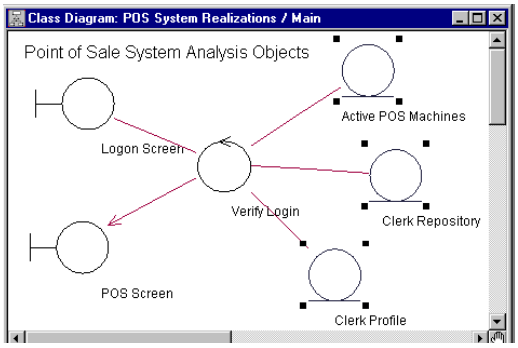
A boundary class is a stereotyped class that models the interaction between one or more actors and the system. You can use boundary classes to capture the requirements of a user interface. Boundary classes can be windows, printer interfaces, sensors, and terminals. Take a look at the Logon Screen boundary class to the left. The Logon Screen boundary class represents an interface that clerks must use to gain access to the system. It requests a user id number and password. The Logon Screen connects with the Verify Logon control class through an association because the Verify Logon control class must confirm that that the user id number and password are correct.

Control classes



A control class models behavior specific to one or a few use cases. Control classes often control other objects and encapsulate use-case specific behavior. Control classes coordinate system behavior and they represent the dynamics of a system, handling the main tasks and control flows. Take a look at the verify login and create retail sale control classes on the left. The Verify Logon class represents the dynamics of the system when it validates the logon information and registers the machine as an active POS machine. The Create Retail Sale class coordinates product scanning and accepting the payment as cash or credit.

Entity classes


Class Diagram: POS System Realizations / Main
Point of Sale System Analysis Objects
Active POS Machines
Logon Screen
Verify Login
Clerk Repository
POS Screen
Clerk Profile

An entity class models information stored by the system and its associated behavior. An entity class has persistent characteristics that are frequently reused in other system use cases. Entity classes show the logical data structure of the system. Look at the Clerk Profile and Active POS Machines entity classes on the left. The Clerk Profile class contains the clerk 扭 personal work file while the active POS machines class stores all information from the machines that have been logged on to the system.

Add a boundary class and an entity class to the Ecommerce System realizations.

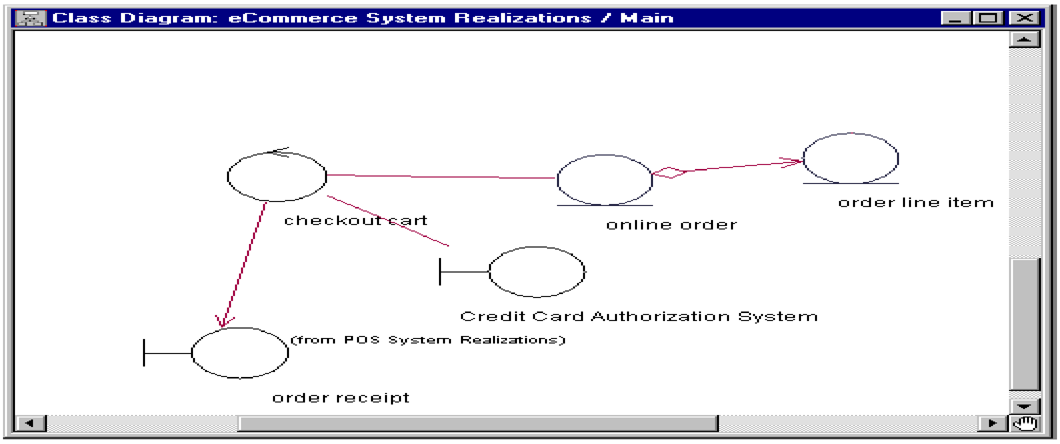
试一试：

1、Creating analysis classes

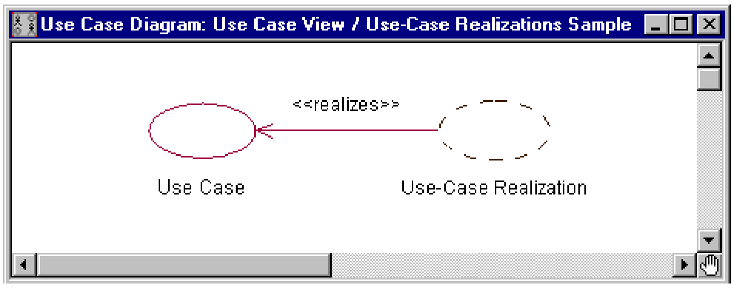
In this exercise, you will add a boundary class called Order Receipt, and an entity class called Online Order to the main diagram. The main diagram represents the analysis level classes. Use the following steps to create the analysis objects:

- Start Rational Rose.
- Click **File** > **Open**.
- Open the model file (.mdl) calledclassics-module5-start.mdl.
- Expand (by clicking on the + sign) the Logical view.
- Expand the Analysis model and then the Ecommerce System Realizations package.
- Double-click the Main diagram.
- Place a ▤ (class) at the bottom of the diagram between the Order Confirmation and Credit Card Authorization System.
- Double-click on the class to display the specification.
- Name the class Order Receipt.
- Set the Stereotype to **boundary.**
- Click **OK** to close the specification.
- Place another class on the diagram directly above the Credit Card Authorization System object.
- Double-click on the class to display the specification.
- Name the class online order.
- Set the Stereotype to **entity**.
- Click **OK** to close the specification.
- Click ↱ and drag an association from Online Order to Order Line Item.
- Right click on the association near the Online Order to display the shortcut menu.
- Click **Aggregate** to make the association an aggregation association.An aggregation is a special form of association that models a whole-part relationship between an aggregate (Online Order) and its parts (Order Line Item).
- Collapse all model elements in the browser so only the Use Case, Logical, Component, and Deployment Views are visible.

检查一下结果：


Class Diagram: eCommerce System Realizations / Main
checkout cart
online order
order line item
Credit Card Authorization System
(from POS System Realizations)
order receipt

Use-case realizations


Use Case Diagram: Use Case View / Use-Case Realizations Sample
<<realizes>>
Use Case
Use-Case Realization

A use-case realization represents the design perspective of a use case. It is an organization model element used to group a number of artifacts related to the use case design. Use cases are separate from use-case realizations so you can manage each individually and so you can change the design of the use case without affecting the baseline use case.For each use case in the use-case model, there is a use-case realization in the design model with a dependency (stereotyped 玶 ealize? to the use case.You can see a use case and a use-case realization connected with a stereotyped dependency to the left. Add a use-case realization to the model.

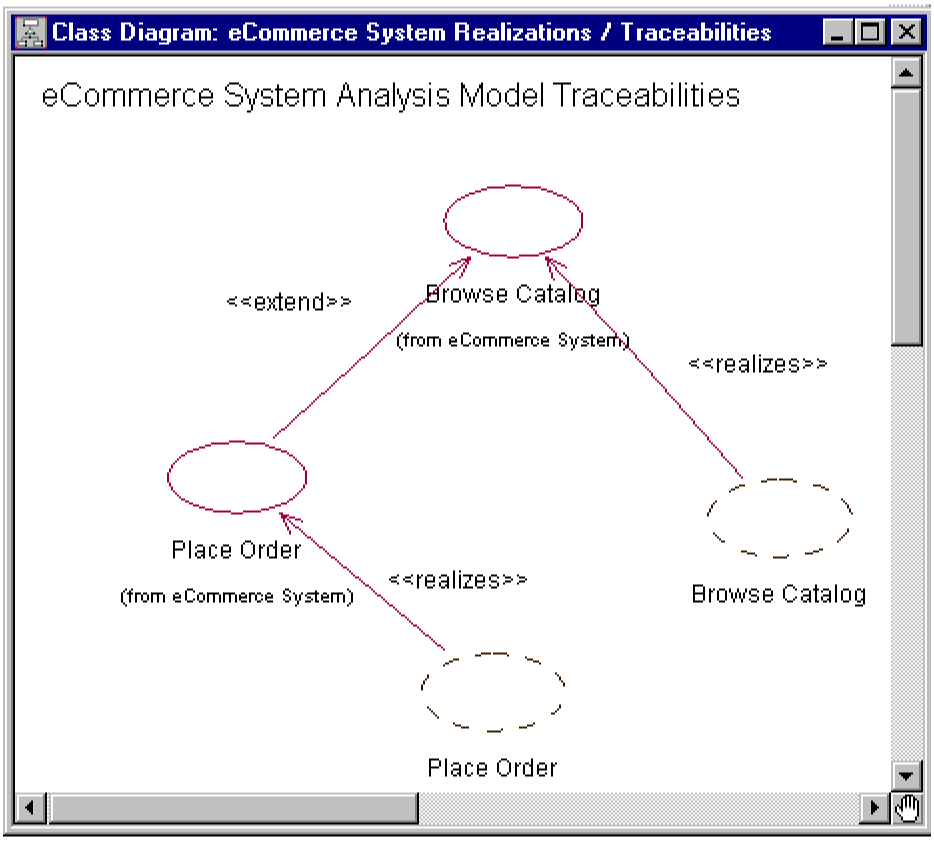
试一试：

Adding a use-case realization

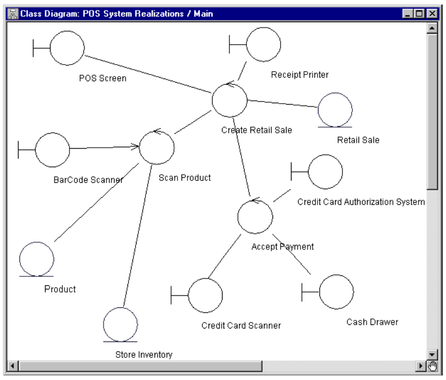
In this exercise, add the place order use-realization to the Traceabilities diagram. The traceability diagram displays use cases and use-case realizations and how they are implemented in the system.

- Expand the Logical View package, the Analysis Model package, and the eCommerce System Realizations package in the browser.
- Double-click the Traceabilities diagram.
- Click **Tools** > **Create** > **Use Case**.
- Place the + sign below the place order to create a new use case.
- Display the specification for the new use case you just created.
- Name the use case Place Order and set the stereotype to **use-case realization**.
- Click **OK**. Notice the namespace warning. Rose is letting you know that you already have another use case named place order.
- Click **OK** to dismiss the warning dialog. You want to create a new use case with the same name because it is stereotyped as a realization.
- Click (Association) from the toolbox.
- Draw an association from the Place Order use-case realization to the Place Order use case.
- Double-click on the association to display the specification.
- Set the stereotype to **realizes**.
- Click **OK** to close the specification.

检查一下：



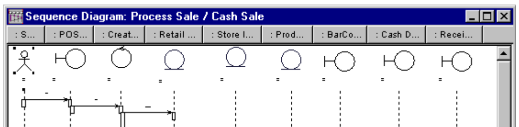
Use-case realizations and interaction diagrams



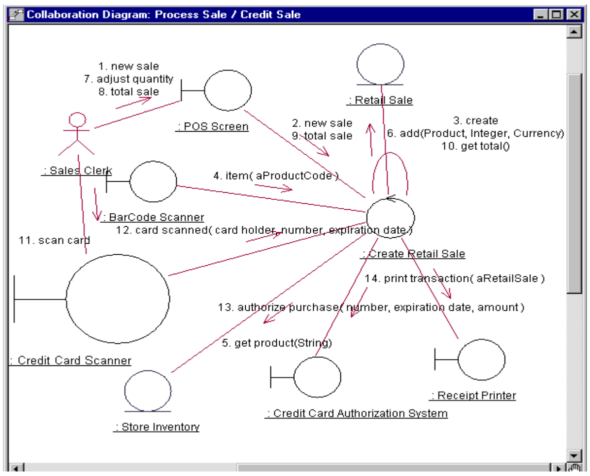
For each use-case realization there may be one or more interaction diagrams depicting objects and their interactions. Interaction diagrams show how each boundary, control, and entity class correspond and relate to each other. Both types of interaction diagrams, sequence and collaboration, display similar information in different ways. Sequence diagrams show how objects interact to perform the behavior of a use case chronologically. Collaboration diagrams show the relationships between objects. Look at the Main class diagram to the left. The Main class diagram is located under the Process Sale use-case realization in the Classics Inc. model.

Use-case realization with a sequence diagram

The analysis team has created two separate sequence diagrams under the Process Sale use-case realization because each diagram represents a separate flow in the system. One diagram represents a cash sale flow and the other diagram represents a credit sale flow. Both diagrams use most of the same analysis classes, but the



chronological order between classes is different. Rational Rose displays all of the classes across the top of the diagram and the messages that show the chronological flow of events between classes in a sequence diagram. The Cash Sale sequence diagram appears to the left. The diagram shows the flow of events as the actor (Store Clerk) interacts with the various classes during a cash sale.



Use-case realization with a collaboration diagram

On any interaction diagram, press the F5 key to display the opposite diagram. For example, if you press F5 when you have a collaboration diagram displayed, a corresponding sequence diagram appears. On collaboration diagrams, reorganize the objects and messages to better view the interaction between the objects. Note the message numbering on the Credit Sale collaboration diagram to the left. Message numbering identifies the time-based order of events as messages flow from object to object.In the next Try it sequence, create a sequence diagram that shows how and when the various classes of the Place Order realization interact.

试一试:

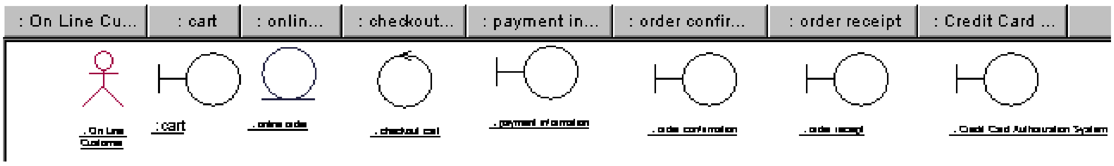
1、Creating a sequence diagram

- In the browser, right-click the Place Order use-case realization in the Ecommerce System Realizations Package.
- Select **New** > **Sequence Diagram**.
- Name the diagram Main.
- Double-click on the **Main** sequence diagram you just created.

  Notice how the diagram does not contain any model elements.　In the next few steps, you populate the diagram with an online actor and various analysis classes.
- Select the **On line custome**r from under the **Ecommerce System** package in the use-case model.　This package is located near the top of the browser.
- Drag and drop the **On line customer** actor on the Main sequence diagram.
- From the browser, drag and drop the following analysis classes located under the Ecommerce System Realizations to the Main sequence diagram:
  - Cart
  - Online order
  - Checkout cart
  - Payment information
  - Order confirmation
  - Order receipt

- Drag and drop the Credit Card Authorization System from the POS System Realizations package to the main sequence diagram.
- Place the Credit Card Authorization System between the Order Confirmation object and the Order Receipt object.Click the Check It button to view the results.

检查一下：



Adding messages on a sequence diagram

Using the following steps, set the sequence numbering so you can see the interaction order and add messages between objects. Each message represents the communication between objects indicating that some sort of action will follow.

- Select **Tools > Options** and then select the **Diagram** tab.
- Check the **Sequence numbering** check box and close the dialog box.

To create a message on a sequence diagram, select the —>(message) on the toolbar. Next, drag the icon from the **lifeline** of one object to the lifeline of another object.

- Create messages between the following classes:

  From On Line Customer to Cart

  From Cart to Checkout Cart

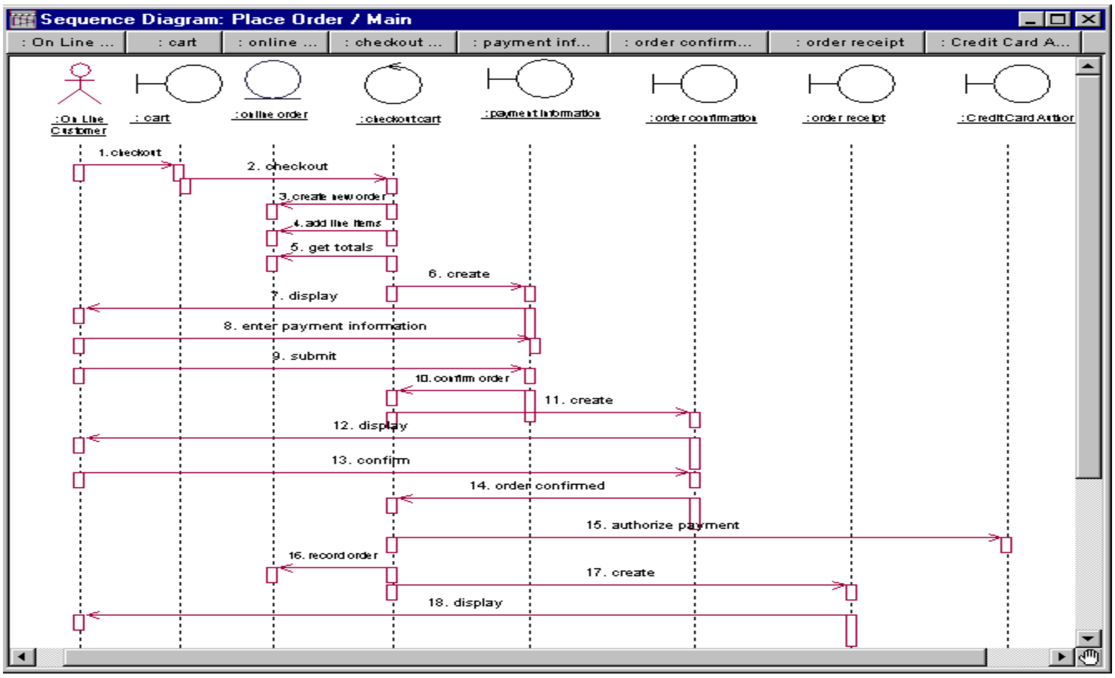
- Double-click to display the message specification for both messages you just created.

  Name the first message checkout

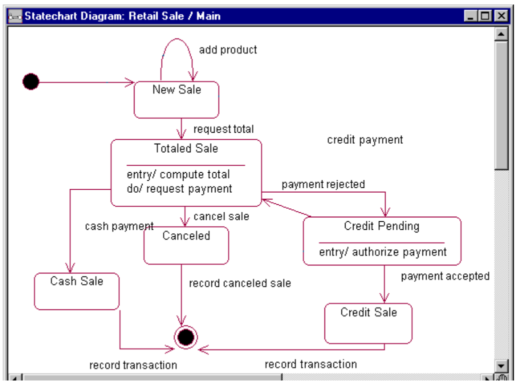
  Name the second message checkout

Both messages are named the same because both are responsible in the checkout process. Click the Check It button to view the results or load classics-module5-completed.mdl. Notice there are many more messages displayed in the results diagram. You may continue to add them if you wish.

检查一下：



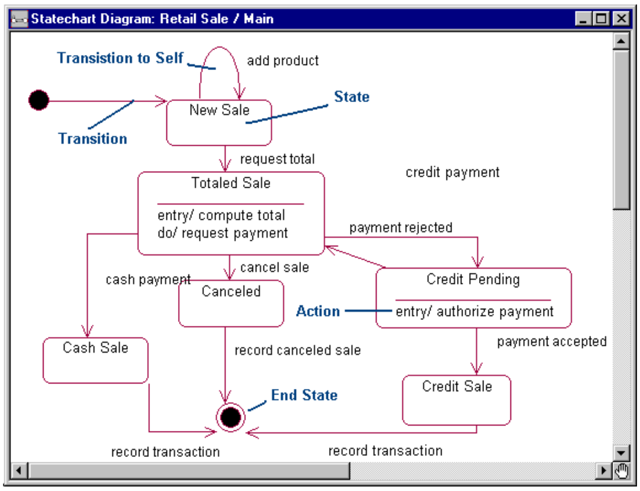
Statechart diagrams



A statechart diagram is a view of a state machine that models the changing behavior of a state. Statechart diagrams show the various states that an object goes through, as well as the events that cause a transition from one state to another.

To gain a better understanding of a statechart diagram, think about the following analogy. Pretend you have ordered a new computer from an online computer store. A few days after you placed the order, you call the customer service department and ask about the status of your order. The service representative responds by saying "The payment is pending"or the order has been "shipped."payent pending"or Shipped?are states?of the order. A statechart diagram represents the states of an object's lifetime.

Statechart diagram model elements



Here are some of the common model elements that statechart diagrams contain:

- States
- Start and end states
- Transitions
- Entry, do, and exit actions

A state represents a condition during the life of an object during which it satisfies some condition or waits for some event. Start and end states represent the beginning or ending of a process.

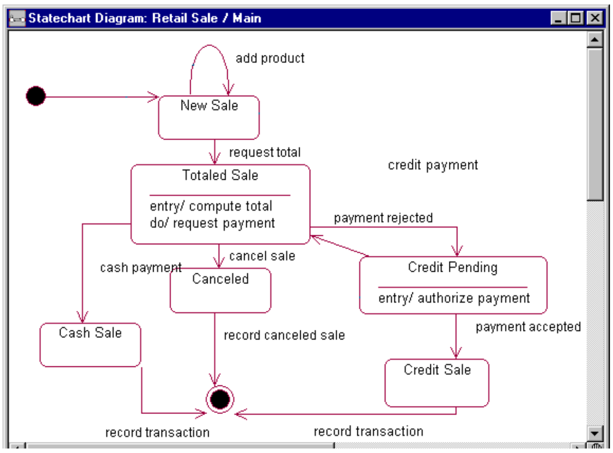
A state transition is a relationship between two states that indicates when an object can move the focus of control on to another state once certain conditions are met. In a statechart diagram, a transition to self element is similar to a state transition, however, it does not move the focus of control. A state transition contains the same source and target state.

Statechart diagrams and actions

Each state on a statechart diagram can contain multiple internal actions. An action is best described as a task that takes place within a state. There are four possible actions within a state:

- On entry
- On exit
- Do
- On event

The retail statechart diagram to the left contains the main statechart diagram model elements. The online order class that you created earlier is complex and needs a statechart diagram to show the states of the class. In this try it sequence, create a statechart diagram for the online order class.



试一试：

1、Creating a statechart diagram

Create a statechart diagram on the online order entity class:

- In the browser, navigate to the Online Order entity class underneath the Ecommerce System Realizations package.
- Right-click on top of the Online Order entity class in the browser.
- Select New > Statechart Diagram.
- Place one start state, one end state, and six states on the diagram from the diagram toolbox.
- Name each state the following names:
  - New Order
  - Customer Order
  - Canceled
  - Completed Request
  - Confirmed
  - Shipped

Select the Check It button to view the statechart diagram arrangement.

2、Creating transitions

Create a transition to self and some transitions between the states you just created using the following steps:

You need to place a transition to self on the New order state because the transition to self simply returns the state of the order back to new after a customer places an order.

- To create a transition to self, click the ↻ icon from the statechart diagram toolbox.
- Click on the New Order state. Create transitions between the states.
- Click the |(transition) icon on the diagram toolbox.
- Place and name transitions between the states as shown in the results window.

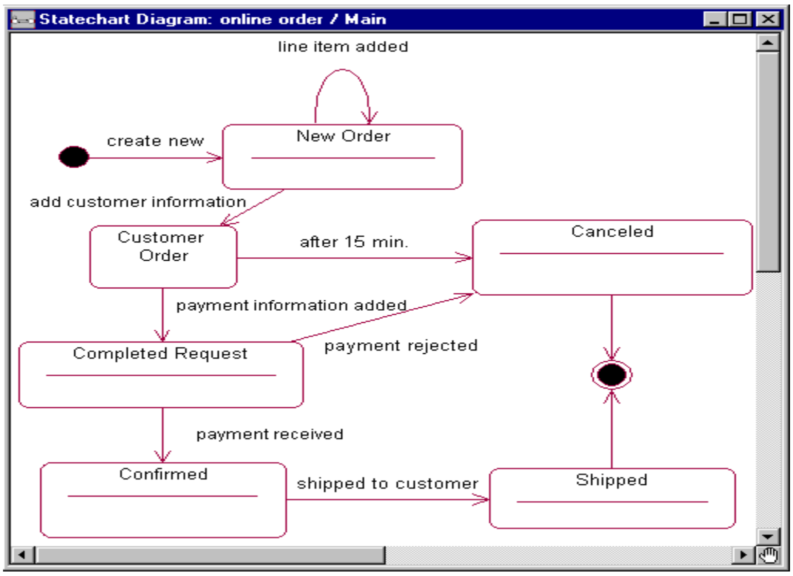
3、Creating actions

Now create some actions on the states. At the end of the steps, use the Check It button to see how the online order statechart diagram should appear.

Add the following actions to the online order statechart diagram using the steps below:

- New Order: **Do**
- Cancelled: **Entry**
- Completed Request: **Entry**
- Confirmed: **Entry**
- Shipped: **Entry**
    - Open a State specification and go to the actions tab.
    - While on the Actions tab of a State Specification, move the pointer to the Type and Action Expression fields and right-click to display the shortcut menu.
    - Click **Insert** to add an entry item.
    - Double-click the entry to display the Action Specification.
    - Type the action description in the **Name** field. If this field is not active, click **Action** on the **Type** field.
    - Select the type action you want to create from the **When** field on the Action specification.

检查一下：



Summary

In this module you have examined the Classics Inc. analysis model. The Classics Inc. analysis model represents a high-level overview of the Ecommerce, Telesales, and POS systems, including many analysis level objects stereotyped as boundary, controller and entity.You have created analysis classes, use-case realizations, sequence diagrams, and statechart diagrams. You can open up the Classics-module-5-completed.mdl file to view the completed module.

What's Next?

Module 6, Creating the Implementation, explains the different parts of the Classics, Inc. design model and how it is implemented in Visual Basic and Visual C++. Module 6 also introduces you to the Visual Studio integration features in Rational Rose. At the end of the module you get the chance to practice code generation and implementation.

第六章　Creating the Implementation

一、关于这一章

This module explains the different parts of the Classics, Inc. design model and how it is implemented in Visual Basic and Visual C++. This module also introduces you to the Visual Studio integration features in Rational Rose.

At the end of this module you get the chance to practice some code generation and implementation. There, you learn:

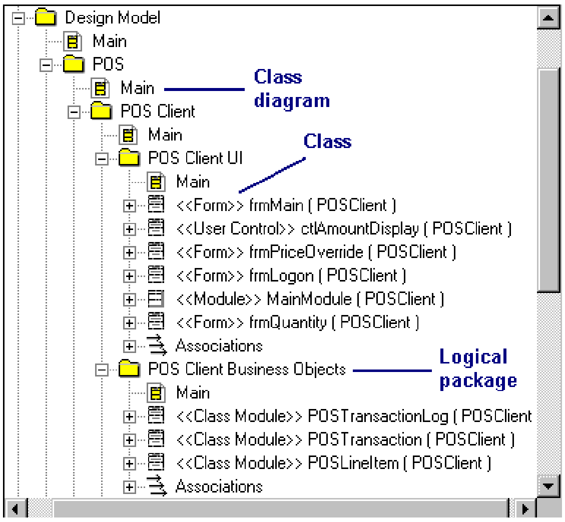
- How to use Rational Rose to generate Visual Basic and Visual C++ code
- How to update the model with code changes
- How to import and use the type libraries of COM components in the model

The exercises are divided into two parts: Visual C++ and Visual Basic round-trip engineering.

Approximate completion time60 minutes

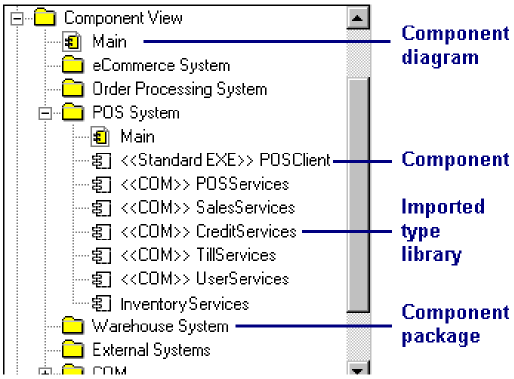
二、The design model

The analysis model represents the system in terms of objects that domain experts know. The design model represents the same system, but at a level of abstraction closer to the source code. For example, the classes in the design model have a language and model properties assigned, which define the mapping to code.The picture shows a part of the design model for the client application of the POS system implemented in Visual Basic. The user interface is separated from the business objects into a package of its own, POS Client UI. This separation helps manage the two conceptually, but there are several ways to organize a model. The stereotype of a design class indicates to what kind of source code item the class corresponds. For example, the frmMain class in the picture is implemented as a Visual Basic form.



```
Design Model
  Main
  POS
    Main
    POS Client
      Main
      POS Client UI
        Main
        <<Form>> frmMain ( POSClient )
        <<User Control>> ctlAmountDisplay ( POSClient )
        <<Form>> frmPriceOverride ( POSClient )
        <<Form>> frmLogon ( POSClient )
        <<Module>> MainModule ( POSClient )
        <<Form>> frmQuantity ( POSClient )
        Associations
      POS Client Business Objects
        Main
        <<Class Module>> POSTransactionLog ( POSClient
        <<Class Module>> POSTransaction ( POSClient )
        <<Class Module>> POSLineItem ( POSClient )
        Associations
```

三、The component view

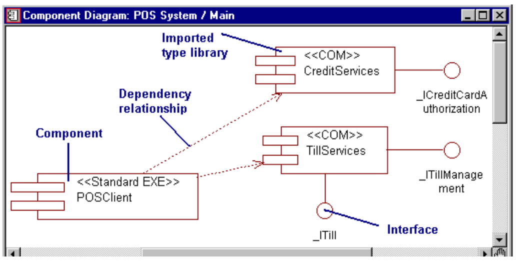


```
Component View
  Main
  eCommerce System
  Order Processing System
  POS System
    Main
    <<Standard EXE>> POSClient
    <<COM>> POSServices
    <<COM>> SalesServices
    <<COM>> CreditServices
    <<COM>> TillServices
    <<COM>> UserServices
    InventoryServices
  Warehouse System
  External Systems
  COM
```

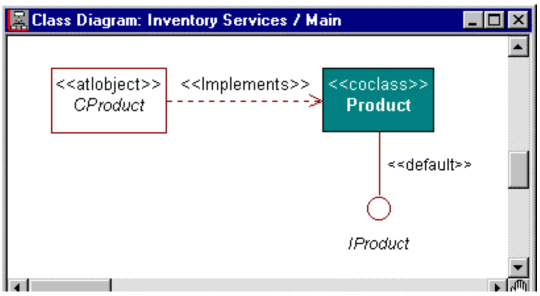
The component view represents the software modules that together realize the system. The components are needed to map each class to the appropriate implementation language and source code project. For example, in order to generate code for a class in the logical view, the class must be assigned to one or several components. Also, to update a model from a source code project, a component corresponding to that project must exist in the model. A model can contain several components of different languages but a class can only be assigned to components with the same implementation language.The picture shows that the POS System is implemented as a Visual Basic Standard EXE project, called POSClient. You can also see that the type libraries of several COM components have been imported into the model.

四、Component diagrams

The component view is illustrated in component diagrams. A component diagram shows how the components are related using dependency relationships.A component diagram also shows the interface of imported COM components in terms of classes with the stereotype "interface".This diagram illustrates the components involved in the POS System component package of the Classics, Inc. model. This part of the diagram shows that the POSClient component references the CreditServices and



TillServices COM components. The type libraries of these two COM components have been imported into the model. In the component diagram you can see that the CreditServices component provides one interface called IcreditCardAuthorization. That interface can be realized by classes in the modeled POSClient component.



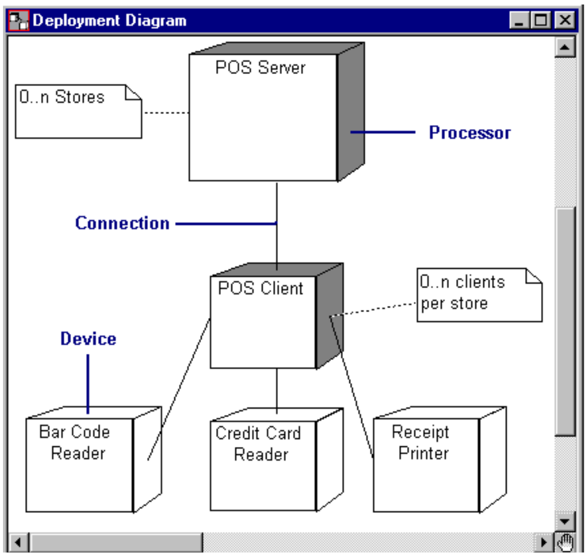
五、 Why should I import a type library?

By importing the type libraries of COM components into the model, you can show how classes in the model use and depend upon classes in other components, regardless of their implementation language. You can:

- Reuse COM components by showing how the classes in the model instantiate, use, and communicate with the items in a COM component
- Show how classes in the model realize the interface of a COM component
- Use the data types defined by a COM component when specifying attributes and operations

The diagram shows how the **ATL** object CProduct implements the default interface of the Product coclass. You can draw a relationship either to a coclass or to its default interface. The generated code gives the same result when compiled.

六、The deployment view

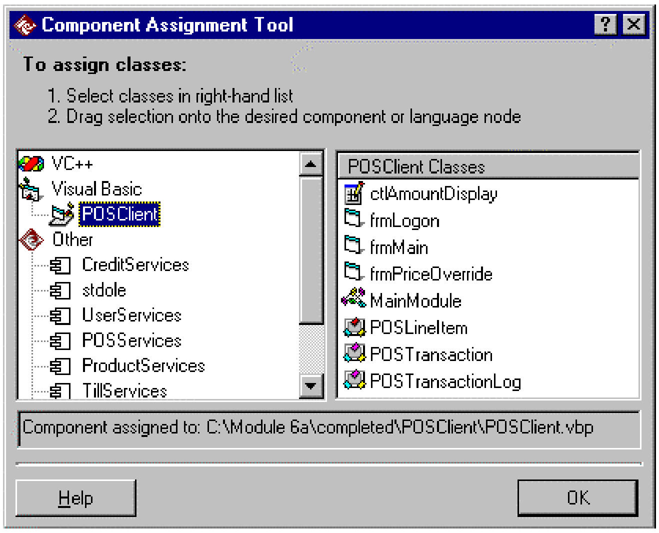


The deployment view shows the different processes of the running system and how they are related.The deployment view is available as a separate node in the Rose browser. The deployment view is illustrated in one single diagram, which you open by double-clicking the Deployment View node in the browser.The picture shows a part of the deployment diagram for the Classics, Inc. system. As you can see a Bar Code Reader, a Credit Card Reader, a Receipt Printer attach to the POS Client. You can also see that the POS Client has a server connection.Rational Rose does not use the deployment view when generating Visual Basic or Visual C++ code.

七、Visual Studio integration features

Rational Rose is tightly integrated with the Microsoft Visual Studio environment. This enables you to generate skeleton code, change the source code, update the model with the code changes, and update the code with model changes. The process of alternating between the model and the code is called round-trip engineering. Rational Rose provides the following features for round-trip engineering Microsoft Visual Studio applications:

- Component Assignment Tool
- Code Update Tool
- Model Update Tool
- Model Assistant
- Type Library Importer



Component Assignment Tool

To assign classes:

1. Select classes in right-hand list
2. Drag selection onto the desired component or language node

VC++
Visual Basic
POSClient
Other
CreditServices
stdole
UserServices
POSServices
ProductServices
TillServices

POSClient Classes: ctlAmountDisplay, frmLogon, frmMain, frmPriceOverride, MainModule, POSLineItem, POSTransaction, POSTransactionLog

Component assigned to: C:\Module 6a\completed\POSClient\POSClient.vbp

八、Component Assignment Tool

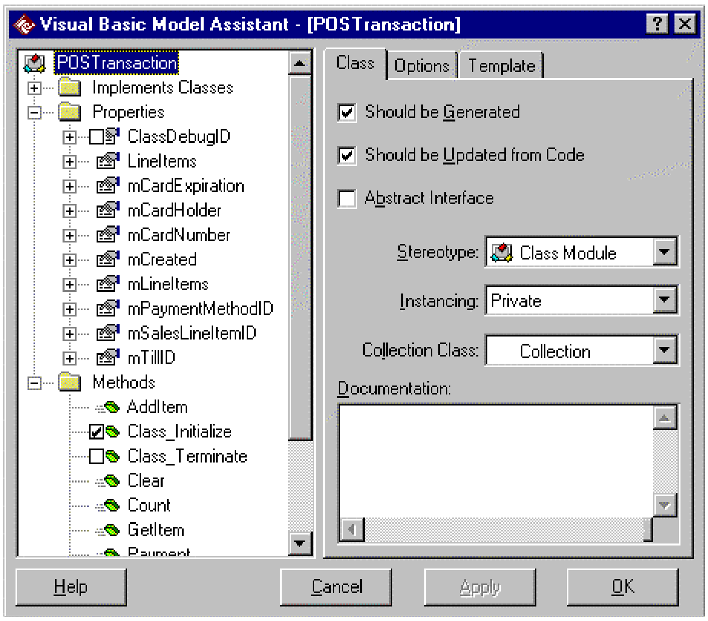
The Component Assignment Tool provides you with an easy-to-use interface to:

- Create new components in the model
- Assign classes to components
- Associate a component with a Microsoft Visual Studio project

The picture shows the contents of the Component Assignment Tool for the Classics, Inc. model. As you can see, there are eight model elements assigned to the POSClient component. The Component Assignment Tool displays the model elements as they are implemented in the source code. For example, the three forms, frmLogon, frmMain, and frmPriceOverride, have Visual Basic form icons.

You assign a class to a component by dragging the class from the right pane and dropping it onto the component in the left pane.

九、Model Assistant



Visual Basic Model Assistant - [POSTransaction]

POSTransaction
Implements Classes
Properties
ClassDebugID
LineItems
mCardExpiration
mCardHolder
mCardNumber
mCreated
mLineItems
mPaymentMethodID
mSalesLineItemID
mTillID
Methods
AddItem
☑ Class_Initialize
□ Class_Terminate
Clear
Count
GetItem
Payment

Class | Options | Template

☑ Should be Generated

☑ Should be Updated from Code

□ Abstract Interface

Stereotype: Class Module

Instancing: Private

Collection Class: Collection

Documentation:

The Model Assistantprovides an alternate way to create attributes and operations on Visual Basic and Visual C++ classes in the model. It also lets you customize the code to generate from a class. With the Model Assistant you can:

- Create attributes and operations on a class
- Preview the code to be generated for a class
- Specify implementation details for a class

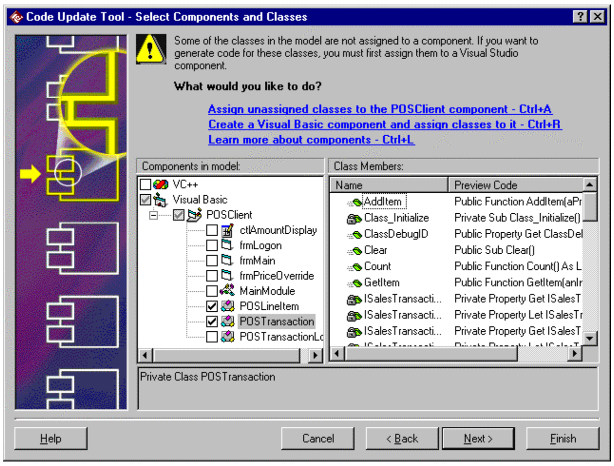
The Model Assistant shows the information about a class as it is implemented in the source code. It maps each UML element found in the Class Specification into the corresponding source code element.

The Model Assistant in the picture shows the POSTransaction class in the Classics, Inc. model.

十、Code Update Tool

Using the Code Update Tool, you can produce Microsoft Visual Studio source code from the information contained in the model. The Code Update Tool generates code from the components in your model into the corresponding Visual Studio projects. With the Code Update Tool you can:

- Generate and update several projects of different implementation languages at the same time
- Preview the code to be generated for each class
- Further specify the mapping between the classes in the model and the code, by opening the Model Assistant
- Keep the model and code synchronized, as it detects any project items that have been renamed or deleted from the model



Code Update Tool - Select Components and Classes

Some of the classes in the model are not assigned to a component. If you want to generate code for these classes, you must first assign them to a Visual Studio component.

What would you like to do?

Assign unassigned classes to the POSClient component - Ctrl+A
Create a Visual Basic component and assign classes to it - Ctrl+R
Learn more about components - Ctrl+L

Components in model:

□ VC++
☑ Visual Basic
☑ POSClient
□ ctlAmountDisplay
□ frmLogon
□ frmMain
□ frmPriceOverride
□ MainModule
☑ POSLineItem
☑ POSTransaction
□ POSTransactionLo

Class Members:

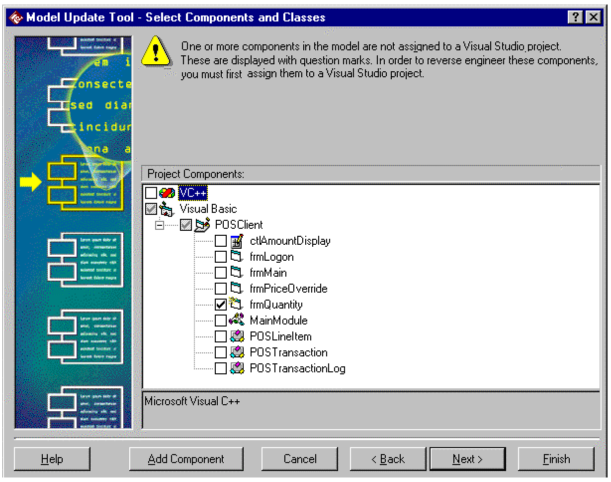
Name	Preview Code
AddItem	Public Function AddItem(aPr
Class_Initialize	Private Sub Class_Initialize()
ClassDebugID	Public Property Get ClassDel
Clear	Public Sub Clear()
Count	Public Function Count() As L
GetItem	Public Function GetItem(anIr
ISalesTransacti...	Private Property Get ISalesT
ISalesTransacti...	Private Property Let ISalesTr
ISalesTransacti...	Private Property Get ISalesT

Private Class POSTransaction

十一、Model Update Tool

Using the Model Update Tool you can reverse engineer a Microsoft Visual Studio project to create a new model from a project, or update an existing model with changes made to the code. With the Model Update Tool you can:

- Update several components of different implementation languages at the same time
- Keep the model and source code synchronized as it detects any model elements that may have been deleted from the code
- Add new components to the model



Model Update Tool - Select Components and Classes

One or more components in the model are not assigned to a Visual Studio project. These are displayed with question marks. In order to reverse engineer these components, you must first assign them to a Visual Studio project.

Project Components:

□ VC++
☑ Visual Basic
☑ POSClient
□ ctlAmountDisplay
□ frmLogon
□ frmMain
□ frmPriceOverride
☑ frmQuantity
□ MainModule
□ POSLineItem
□ POSTransaction
□ POSTransactionLog

Microsoft Visual C++

十二、Model Ids

Each generated code item is tagged in the code with a model identifier (model

POSClient - POSLineItem (Code)

(General) SetSellingPrice

```
Option Explicit
```

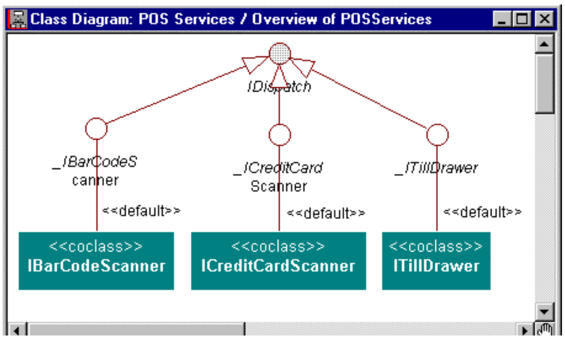
ID). This allows the code item to be matched with a model element independent of the actual names. For example, the mQuantity attribute in the picture has the model ID ?#ModelId=37C347220154.

The Code Update Tool and the Model Update Tool use the model IDs to synchronize the model and code. The model IDs are required to round-trip engineer the code properly. You should not edit the inserted model IDs.

If the ability to synchronize the model and code is not important 椇--for example, if you are going to generate code from a model only once you can suppress the generation of model IDs. However, Rose synchronizes the model and the code using the model IDs. Deleting the IDs disables the synchronization feature.

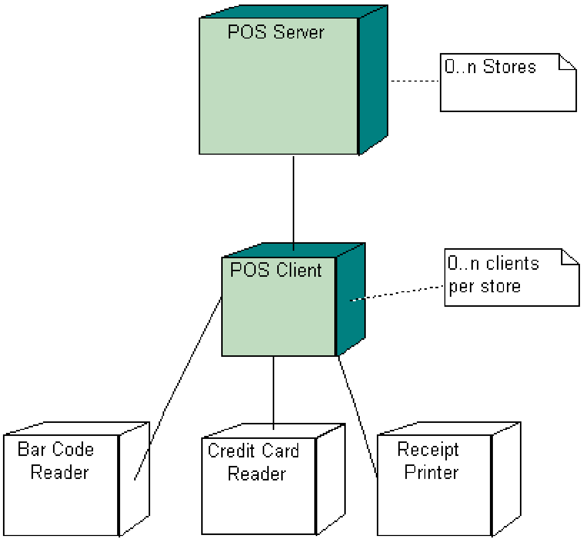
十三、Type Library Importer

The Type Library Importer allows you to import a type library of a COM component into the model. You can either drag the COM component from the Windows Explorer and drop it in Rational Rose, or you can use the **Tools > COM > Import Type Library** command.



A type library contains a description of a COM component as viewed from the outside. The description includes the coclasses, interface items, dispinterfaces, properties (called attributes in UML), methods (called operations in UML), data types, etc. of the component. Type library information is needed to provide a language-neutral description of the interfaces and data types that a COM component exposes.This tutorial does not explain the different kinds of items in a type library. You can find references to useful information sources at the end of this module.

十四、Try it yourself: Introduction



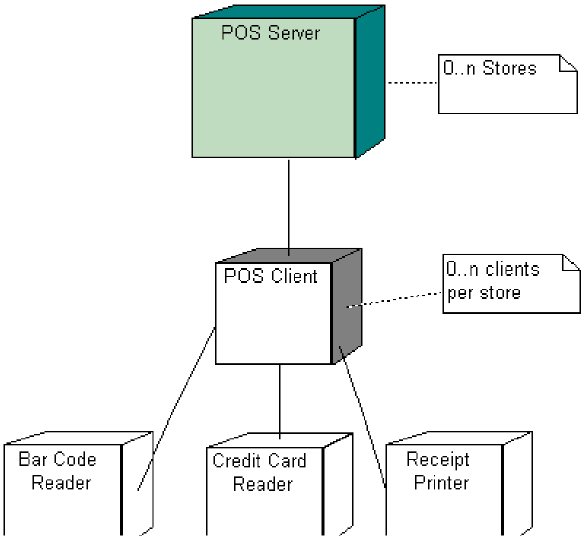
In this section, you can try some round-trip engineering. The exercises in this module are divided into two parts: Visual C++ (ATL) development and Visual Basic development. You can skip either of the parts if you want to perform only one of them.

In the first part, which starts on the next slide, you are a member of the POS Server project team, which is responsible for developing the software that runs on the server.

In the other part, you are a member

of the POS Client project team, which uses Visual Basic to implement the software that runs on the client machines. The ATL Inventory component that you develop in the first part of the exercise is used in the next part of the exercise.

十五、Try it yourself: Visual C++ development



When you start this exercise, you are in an early iteration of the POS Server development effort. The POS Server provides common services to the POS Client machines. The services include credit card authorizations, product lookups, sales transaction recording, and basic till management functions. Most of these services are provided by COM objects running on the server, which also manages the store database. The client applications use DCOM to access the server components, so no special coding considerations are needed to make the server components visible to the client machines.This exercise leads you through the process of importing a type library and creating an ATL based implementation of an interface in the model. During this process you create new objects from inside the Visual C++ IDE and reverse engineer them back into the model. To start the exercise, click **Try it!**. If you don't want to perform this exercise, click **Next**.

试一试:

Registering DLLs

The Classics, Inc. application is dependent on common components (DLLs) that must be registered before you can run the application.

- Locate the regsvr32.exe program in the Windows Explorer. It is usually located under **C:\Windows\System**.
- Drop the program onto the desktop to create a shortcut.
- Register the four DLLs that are located in the **“Module6a\dlls\POSServer** folder, by dragging them onto the regsvr32.exe shortcut. ("is the folder where the sample is installed.)
- From the “**Module6a\dlls\POSServices** folder, register the POSServices.dll.

Defining virtual path maps

The Classics, Inc. model contains references to the Visual C++ projects that implement the modeled system. Rational Rose stores these references using virtual file paths. This makes the references independent of where you installed the sample on your computer. You must now define the virtual path maps that map the file references to the actual paths where the source code is located.

- Start Rational Rose and open the “**Module 6a\start\classics-module6a-start.mdl** model.
- Click **File > Edit Path Map.**
- In the **Symbol** box, type CLASSICS6AS.
- In the **Actual Path** box, click **Browse**.
- Browse for and double-click the 匼 **Module6a\start folder.**
- Click **OK**.
- Click **Add** in the **Virtual Path Map dialog** box.
- To create a virtual path map also for the completed model, type CLASSICS6AC in the **Symbol** box.
- In the **Actual Path** box, click **Browse**.
- Browse for and double-click on the 匼 **Module6a\completed** folder.
- Click **OK**.
- Click **Add** in the **Virtual Path Map** dialog box.
- Click **Close**.
- Finally, make sure that the Visual C++ add-in is installed and enabled in Rational Rose, by clicking **Add-Ins > Add-In Manager**.

Importing a DLL

The Classics, Inc. application is dependent on some existing components (DLLs). To show how the application uses those DLLs in the model, the DLLs must be imported into the model. All these DLLs, except SalesServices.dll, have already been imported. Follow these steps to import SalesServices.dll:

- Click **Tools > COM > Properties** to open the **COM Properties** dialog box. This is where you define where in the logical and component view imported COM components are placed.
- Enter the following values in these fields:

Logical view - Default Package

Design Model/COM/$library

Logical view - Overview diagram

Design Model/COM/$library/Overview of $library

Component view - Default Package

COM

Component view - Overview diagram

COM/Overview of type libraries

- Click **OK**.
- Locate “**Module6a\dlls\POSServer\SalesServices.dll** in the Windows Explorer.
- Drag **SalesServices.dll** from the Windows Explorer and drop it in the browser or in a diagram in Rational Rose. (If the Rational Rose application window is hidden or minimized, point to the Rose icon in the Windows task bar before dropping the file, which brings the application to the front.)

- Click **Quick Import**, which means that Rose does not import the attributes and operations of the DLL's COM objects.
- When Rose is finished, an overview diagram of the imported DLL's type library is displayed. The different icons mean:
  - Coclass —

    <<coclass>>
ISalesLineItem
  - Interface —○.

Examining the imported DLLs

In this exercise, review the imported DLL and organize its contents in the model.

- Expand the **Component View \ COM** package. Note that Rational Rose has created a component for the imported DLL. The stereotype of the component is "COM".
- To organize the component view better, move the new component, SalesServices, by dragging it onto the **Component View \ POS System package**.
- Expand the package **Logical View Design Model \ COM**. Note that Rose has created a logical package for the imported DLL. That package contains the items that are defined in the DLL's type library.
- Expand the **Logical View \ Design Model \ COM \ SalesServices** package.
- Note how the interfaces, -○, are separated from the coclasses, □.
  A coclass defines both the public view of a type of object and its private implementation. The interface is just a public declaration of functionality, which can be implemented by a number of coclasses.
- There are no attributes and operations specified on the items. To import the attributes and operations of an item, right-click on the item, for example the _ISalesLineItem interface, and click **Full Import**.
- To organize the logical view better, drag the SalesServices logical package and drop it on the **Logical View \ Design Model \ POS \ POS Server package**.

Implementing the interfaces

Implementing an interface in Rose Visual C++ is an interactive process between Rose Visual 爕++ and Visual Studio Visual 爕++.

- Implementing an interface requires four steps:
  - Create an ATL component and project to contain the interface source code
  - Modify the new project 抯 IDL file to forward-reference the interfaces
  - Convert the interfaces to simple ATL objects
  - Generate code for the interfaces
- Open the Class diagram for **Logical View \ Design Model \ POS \ POS Server \ Inventory Services \ Main**. This diagram contains two interfaces (IProduct and IProductInventory) that you will implement.

Creating the ATL component and project

The following steps use the Model Update Tool to create a new ATL component for the IProduct and IProductInventory interfaces (this keeps the model in-synch with the new ATL component):

- Right-click the IProduct interface, and then click Update Model. If the Welcome page displays, click Next.
- Click Add Component.
- Double-click on the Project icon to open Microsoft Visual Studio at the New Projects tab.
- Select ATL COM Appwizard, browse the Location path to 匼 Module6a\start and click OK.
- Enter the Project Name as InventoryServices. Make certain InventoryServices appears only once in the path.
- Finish the creation process, in both Visual Studio and Rose, accepting the default settings, until you are back in the Model Update Tool. The VC++ language icon and the InventoryServices component should be checked (and should be the only items checked).
- Click Finish, and when the process completes, click Close. You should see no errors or warnings on the Log tab.

Modifying the IDL file

Because Rose Visual C++ does not generate forward declarations in the IDL file, the file must be manually modified.

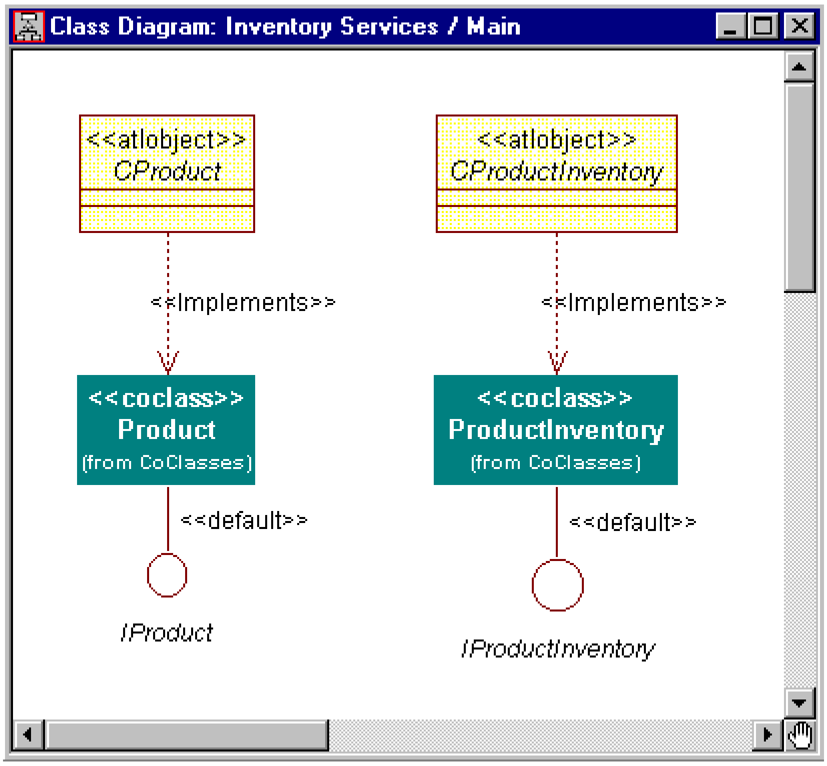
- In Visual Studio (Visual C++), navigate to the InventoryServices IDL file.
- Immediately below the last import entry, enter:
  interface IProduct;
  interface IProductInventory;
- Save the IDL file.

Converting the interfaces to simple ATL objects

The interfaces now need to be converted to simple ATL objects.

?From Rose Visual C++, right-click on the IProduct interface and click COM > New ATL Object. ?In the Component drop-down box, select InventoryServices (this should be the only option available). ?Click OK. ?Repeat this process for the IProductInventory interface.?In the Rose browser component view, drag and drop the InventoryServices component into the Component View \ POS System folder. ?Save the model.

检查一下：



Generating code for the interfaces

With the interfaces properly configured and assigned, you can generate code for them.

- Right-click on the InventoryServices component in the browser and click Update Code.
- Confirm that the VC++ icon and the InventoryServices components are checked (and should be the only items checked), then click Finish. When the process completes, you may see some informational messages on the Log tab, this is normal.
- Click Close.
- Switch back to Visual Studio.
- A dialog displays indicating that the file was modified outside the source editor. This is true, Rose modified the file. Click Yes to reload the file.
- Compile the source code for the InventoryServices project. There should be no warnings or errors.

Working from the IDE

From the IDE, create new classes and reverse engineer them back into the model.

- Add a new ATL object to the project. Make it a Data Access and a Consumer object, and then click Next.
- Click Select Datasource.
- On the Provider tab, select Jet 4.0, and then click Next.
- On the Connection tab, in the Select box, browse to the 叵 Module6a\database\ClassicsData.mdb database, and then click Open.
- Test the connection and click OK when it succeeds.

- Click OK.
- On the Select Database Table dialog, select the PRODUCTS table and click OK.
- Change the short name to ProductRecord.
- Check support for Change, Insert, and Delete, accept the Type default of Command, and click OK.

  In this exercise, you'll add a member variable and a member function to CProductRecord.

- Right-click on the CProductRecord class to add a new member variable: public long m_sku;
- Click OK.
- In the ProductRecord.h file, add a new public member function: HRESULT OpenProduct(long sku){ m_sku = sku; return Open(); }You can copy this code from the help window and paste it into the class in Visual Studio.
- Save all files in the workspace.

  Continue to work on the ProductRecord class.

- Add a #include <comdef.h> statement to the ProductRecord.h file (this is needed to access the _bstr_t class).
- Still in the ProductRecord.h file, change the OpenRowset() method to read (by copying the following code):

```
HRESULT OpenRowset()
{
    _bstr_t sql;
    char idbuf[10];
        // Set properties for open
        CDBPropSet  propset(DBPROPSET_ROWSET);
    propset.AddProperty(DBPROP_IRowsetChange, true);
    propset.AddProperty(DBPROP_UPDATABILITY,
DBPROPVAL_UP_CHANGE | DBPROPVAL_UP_INSERT | DBPROPVAL_UP_DELETE);   sql = "SELECT * FROM PRODUCTS WHERE SKU=";   ltoa(m_sku,idbuf,10);   sql += idbuf;   return CCommand<CAccessor<CProductRecordAccessor> >::Open(m_session, sql, &propset);}
```

Working from the IDE (continued)

Finish up the ProductRecord class.

- Confirm that the COLUMN MAP in the ProductRecord.h file reads: BEGIN_COLUMN_MAP(CProductRecordAccessor) COLUMN_ENTRY(1, m_SKU) COLUMN_ENTRY(2, m_COMPOSER) COLUMN_ENTRY(3, m_COMPOSITION) COLUMN_ENTRY(4, m_CONDUCTOR) COLUMN_ENTRY(5, m_PERFORMER) COLUMN_ENTRY(6, m_DESCRIPTION) COLUMN_ENTRY(7, m_IMAGE) COLUMN_ENTRY(8, m_PRICE) COLUMN_ENTRY(9, m_ONHAND) COLUMN_ENTRY(10, m_REORDERLEVEL)END_COLUMN_MAP()

- Save the ProductRecord.h file.

Now, let 抯 modify the CProduct class.

- In the CProduct.h file, add the following member variables: private: CURRENCY m_price; BSTR m_performer; BSTR m_image; BSTR m_description; BSTR m_conductor; BSTR m_composition; long m_productSKU; BSTR m_composer;
- Save the CProduct.h file.

The CProduct.cpp file requires quite a few additions. The additions are listed on this page and the following two pages.

- In the CProduct.cpp file, add a #include statement for <comdef.h>.
- Then add the following implementation for the get and put operations:
- For get_ProductSKU:
  *sku = m_productSKU;For put_
  ProductSKU: m_productSKU = sku;
  For get_Composer:
  CComBSTR composer((char*) m_composer);
  *aComposer = composer.Copy();
  For put_Composer: SysReAllocString(&m_composer, aComposer);
  For get_Composition: CComBSTR composition((char*) m_composition);
  *aComposition = composition.Copy();
  For put_Composition: SysReAllocString(&m_composition, aComposition);

For get_Conductor: CComBSTR conductor((char*) m_conductor); *aConductor = conductor.Copy();For put_Conductor: SysReAllocString(&m_conductor, aConductor);For get_Performer:CComBSTR performer((char*) m_performer); *aPerformer = performer.Copy();For put_Performer: SysReAllocString(&m_performer, aPerformer); For get_Description: CComBSTR description((char*) m_description); *aDescription = description.Copy();

Finish up CProduct.cpp in the CProduct class:

For put_Description: SysReAllocString(&m_description, aDescription);
For get_Image:CComBSTR image((char*) m_image);
*anImage = image.Copy();
For put_Image: SysReAllocString(&m_image, anImage);
For get_Price: aPrice->Hi = m_price.Hi; aPrice->Lo = m_price.Lo;
aPrice->int64 = m_price.int64;
For put_Price: m_price.Hi = aPrice.Hi;

```
m_price.Lo = aPrice.Lo;
m_price.int64 = aPrice.int64;
```

- Save the CProduct.cpp file.

In this exercise, you are modifying the CProductInventory.cpp file.

- In the CProductInventory.cpp file, add #include statements for ProductRecord.h and CProduct.h.
- Then add the implementation for the GetProduct operation:

```
{       CProductRecord products;
CComObject<CProduct>* pProduct;
CURRENCY price;
HRESULT hr;
*aProduct = NULL;
hr = products.OpenProduct(sku);    if ( SUCCEEDED(hr) ) {        hr                    =
products.MoveFirst();            if ( SUCCEEDED(hr) ) {            //pProduct        =        new
CComObject<CProduct>();                hr                                             =
CComObject<CProduct>::CreateInstance(&pProduct);               //HRESULT hr = pProduct-
>FinalConstruct();               //HRESULT hr = pProduct->              if
(SUCCEEDED(hr)){                     //pProduct->AddRef();                     pProduct-
>put_ProductSKU( products.m_SKU );                      pProduct-
>put_Composer( SysAllocStringLen( (OLECHAR FAR*)products.m_COMPOSER, 21) );
                pProduct->put_Composition(          SysAllocString(          (OLECHAR
FAR*)products.m_COMPOSITION) );                        pProduct-
>put_Conductor( SysAllocString( (OLECHAR FAR*)products.m_CONDUCTOR) );
        pProduct->put_Description(              SysAllocString(               (OLECHAR
FAR*)products.m_DESCRIPTION) );                        pProduct-
>put_Image( SysAllocString( (OLECHAR FAR*)products.m_IMAGE) );
    pProduct->put_Performer(                SysAllocString(                   (OLECHAR
FAR*)products.m_PERFORMER) );                   VarCyFromR4(products.m_PRICE,
&price);                pProduct->put_Price( price );              hr     =     pProduct-
>QueryInterface(IID_IProduct, reinterpret_cast<void**>(aProduct));           }         }
}       return hr;}
```

With the code written, it 抯 time to complete the process.

- Save all files.
- Compile the project. The compilation process should register the new ATL component. There should be no warnings or errors.

Finishing the model for InventoryServices

With modifications to the IDL complete, you need to bring those changes into the Rose model and clean up the Rose browser.

- From Rose, right-click on the InventoryServices component and update the model. There should be no warnings or errors.
- Move the new classes and associations from the Logical View \ Reverse Engineereed \ Inventory Services package to the Logical View \ Design Model \ POS \ POS Server \ Inventory Services package.
- Save the model.

Congratulations!

You have completed the Visual C++ exercises.

Try it yourself: Visual Basic development

In this exercise, you are going to further develop the POS Client application, which is implemented in Visual Basic. The machine that the client is running on is a specialized computer (cash drawer) with additional devices as you can see in the picture. The manufacturer supplies an interface component and device drivers in the form of a Windows DLL, POSServices.dll, which the POS Client application uses.Each POS Client is connected to a POS Server. The functionality of the server is accessible through DLLs, which have been created by the POS Server development team and imported into the model. These DLLs also define common interfaces that the entire application uses.This exercise leads you through the process of importing a type library into the model, generating code for a new class module, as well as reverse engineering a Visual Basic form into the model. To start the exercise, click Try it!. If you don't want to perform this exercise, click Next.

试一试：

Configuring your system

The following exercises require Microsoft Visual Basic 6 and the Microsoft ActiveX Data Objects 2.0 Library, MSADO15.DLL. The exercises also require some additional configuration of your computer. Follow these steps to configure your computer:

- The sample uses a Microsoft Access 97 database called ClassicsData.mdb. You must create a DSN for that database. To do that, open the Control Panel in Windows.
- Double-click the ODBC icon, which opens the **ODBC Data Source Administrator**.
- Click Add and select the **Microsoft Access Driver** in the displayed dialog box.
- Click **Finish.**
- Type ClassicsPOSServer in the **Data Source Name** box.
- Click Select and browse for the database file, “Module6a\database\ClassicsData.mdb, where "? is the folder where the sample is installed.

- Click OK in each dialog box.

If you performed the Visual C++ (ATL) exercises, you have already performed the rest of the configuration steps. If so, click here to jump to the first Visual Basic exercise.

If you did not perform the Visual C++ exercises, click Next for further configuration instructions.

Registering DLLs

The part of the Classics, Inc. application that you are going to extend, the POS Client, is dependent on common components. Before you can run the application, you must register those DLLs.

- Locate the regsvr32.exe program in the Windows Explorer. It is usually located under **C:\Windows\System**.
- Drop the program onto the desktop to create a shortcut.
- Register the four DLLs that are located in the 📁 **Module6a\dlls\POSServer** folder, by dragging the files onto the regsvr32.exe shortcut.
- In the same way, register also the 📁 Module6a\dlls\POSServices\POSServices.dll and the 📁 Module6a\completed\InventoryServices\ Debug\InventoryServices.dll.

Defining virtual path maps

The Classics, Inc. model contains a reference to the Visual Basic project that implements the modeled system. Rational Rose stores the reference using a virtual file path, to make the reference independent of where you installed the sample on your computer. Rose provides a mechanism called virtual path maps that makes this possible. You must now define virtual path maps that map the file references in the model file to the actual paths where the files located.

- Start Rose and open the 📁 Module6a\start\\classics-module6a-start.mdl model.
- Click File > Edit Path Map.
- In the Symbol box, type CLASSICS6AS.
- In the Actual Path box, click Browse.
- Browse for and double-click the 📁 Module6a\start folder.
- Click OK.
- Click Add in the Virtual Path Map dialog box.
- To create a virtual path map also for the completed model, type CLASSICS6AC in the Symbol box.
- In the Actual Path box, click Browse.
- Browse for and double-click the 📁 Module6a\completed folder.
- Click OK.
- Click Add in the Virtual Path Map dialog box.
- Click Close.
- Finally, make sure that the Visual Basic add-in is installed and enabled in Rational Rose, by clicking Add-Ins > Add-In Manager.

Importing DLLs

The part of the Classics, Inc. application that you are going to extend, the POS Client, is dependent on existing components (DLLs). To show how the POS Client application uses those DLLs in the model, the DLLs must be imported into the model. Follow these steps to import the DLLs:

- Click Tools > COM > Properties to open the COM Properties dialog box. This is where you define where in the logical and component view imported COM components are placed.
- Enter the following values in these fields:
- Logical view - Default Package
  Design Model/COM/$library
  Logical view - Overview diagram
  Design Model/COM/$library/Overview of $library
  Component view - Default Package
  COM
  Component view - Overview diagram
  COM/Overview of type libraries
- Click OK.
- Locate 匼 Module6a\dlls\POSServer\SalesServices.dll in the Windows Explorer.
- Drag SalesServices.dll from the Windows Explorer and drop it in the browser or in a diagram in Rational Rose. (If the Rational Rose application window is hidden or minimized, point to the Rose icon in the Windows task bar before dropping the file, which brings the application to the front.)
- Click Quick Import, which means that Rose does not import the attributes and operations of the DLL's COM objects.
- When Rose is finished, note that an overview diagram of the imported DLL's type library is displayed. The different icons mean:
- Coclass
- Interface
- In the same way, import also the 匼 **Module6a\completed\InventoryServices\Debug\InventoryServices.dll** into the model.

Examining the imported DLLs

Review the imported DLLs and organize their contents in the model.

- Expand the Component View \ COM package. Note that Rational Rose has created components for the imported DLLs. The stereotype of those components is <<COM>>.
- To organize the component view better, move the new components, SalesServices and INVENTORYSERVICESLib, by dragging each one onto the Component View \ POS System package.
- Expand the package Logical View \ Design Model \ COM. Note that Rose has created a logical package for each of the imported DLLs. Those packages contain the items that are defined in the DLLs' type libraries.
- Expand the Logical View \ Design Model \ COM \ SalesServices package.

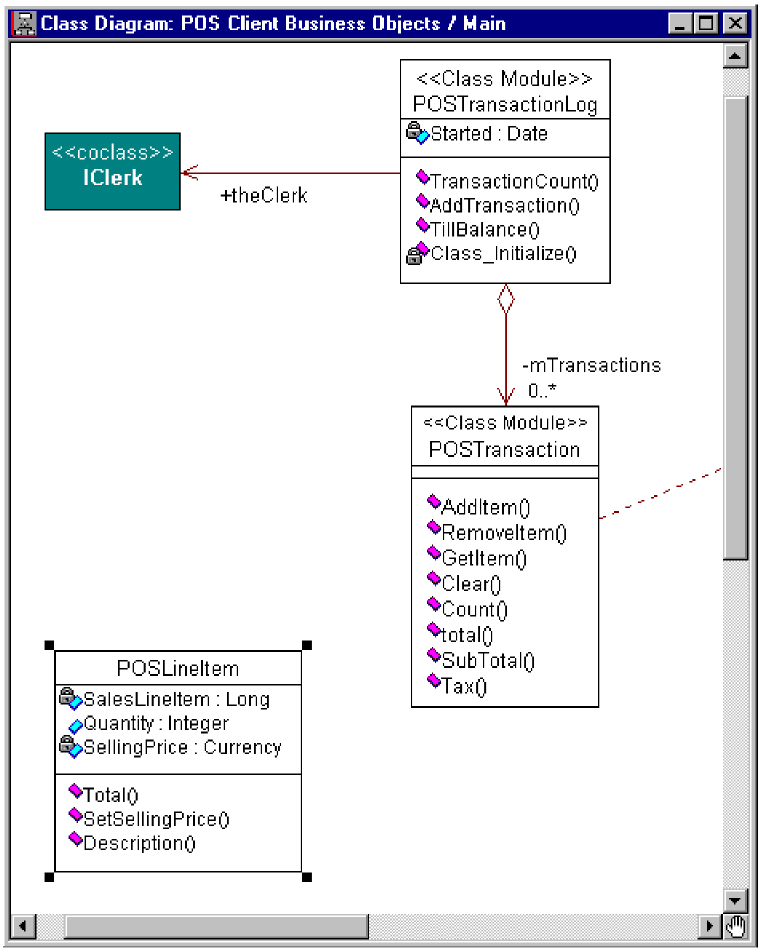
- Note how the interfaces, , are separated from the coclasses, . A coclass defines both the public view of a type of object and its private implementation. The interface is just a public declaration of functionality. This functionality can be implemented by a number of coclasses.
- There are no attributes and operations specified on the items. To import the attributes and operations of an item, right-click on the item, for example the _ISalesLineItem interface, and click Full Import.
- To organize the logical view better, drag the SalesServices logical package and drop it on the Logical View \ Design Model \ POS \ POS Server package.
- Drag also the INVENTORYSERVICESLib logical package onto the Logical View \ Design Model \ POS \ POS Server package.

Adding a new business object

In this exercise you add a new business class to the model of the POS Client. The new class is an order line item class called POSLineItem.

- If you haven't done it already, start Rose and open the
  匼 Module6a\start\classics-module6a-start.mdl model.
- Open the Main diagram in the **Logical View \ Design Model \ POS \ POS Client \ POS Client Business Objects package**.
- Add a new class called **POSLineItem**.
- Add the following three attributes (use any approach you like):
  SalesLineItem : Long (private)
  Quantity : Integer (public)
  SellingPrice : Currency (private)
- Add the following three operations (use any approach you like):
  Total() : Currency
  SetSellingPrice(aPrice :
  Currency, overriddenBy : IClerk)
  Description() : String

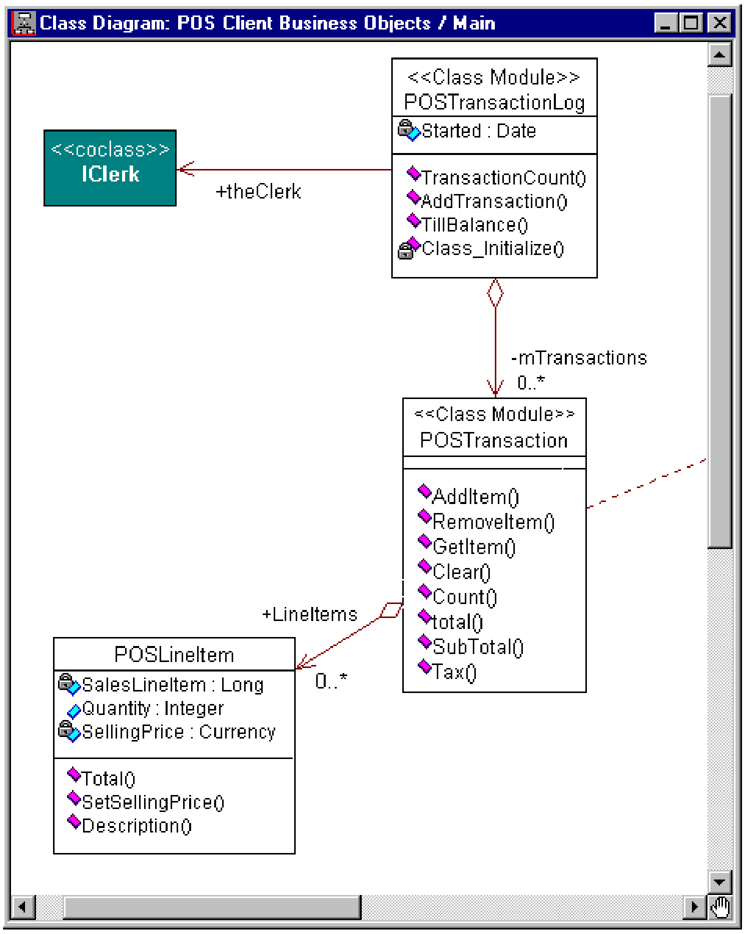
检查一下：



A transaction consists of line items, which is indicated by an aggregation relationship between the POSTransaction class and the POSLineItem class.

- Add an association relationship from the POSTransaction class to the POSLineItem class.
- Set the multiplicity to 0...* on the POSLineItem side, which indicates that the transaction contains a variable number of line items. (Use the context menu or specification for the association.)
- Set the role name on the POSLineItem side to LineItems.
- Make sure that only the POSLineItem side of the relationship is navigable. It is navigable if there is an arrow pointing to the POSLineItem side.
- Define the POSTransaction side as an aggregate. One way to do that is to right-click on the POSTransaction side of the relationship and select **Aggregate** on the displayed men

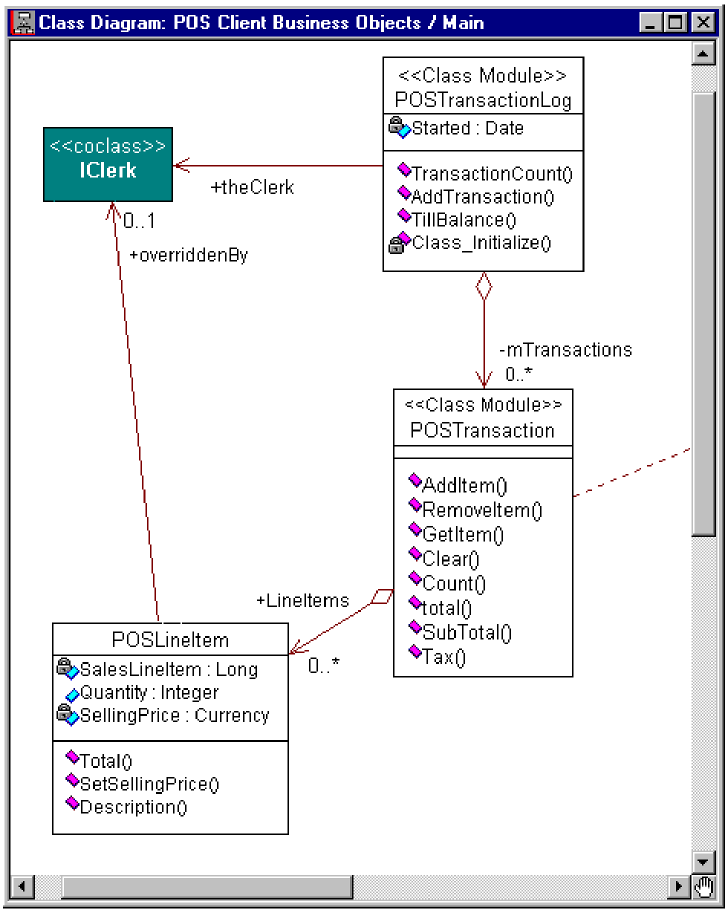
检查一下：



The coclass IClerk represents an instance of an IClerk interface supporting object. The actual object providing the implementation can be another class, but for the purposes of the POSLineItem instance the coclass object supports the IClerk interface. Now add an association between the POSLineItem and the IClerk coclass.

- Add an association relationship from the POSLineItem class to the coclass IClerk.
- Set the multiplicity to 0...1 on the IClerk side, since not every price is overridden by a manager.
- Set the role name on the IClerk side to overriddenBy.
- Make sure that only the IClerk side of the relationship is navigable.

检查一下：



In order for our line item class to be used elsewhere in the system, it must implement the public _ISalesLineItem interface, which is indicated by a realize relationship. The POSTransaction class also implements an interface, _ISalesTransaction, which is defined in the Sales Services type library. You can draw relationships either to a coclass or its default interface. The generated code gives the same result when compiled. Here we use the coclasses.

- Drag the **Logical View \ Design Model \ POS \ POS Server \ Sales Services \ ISalesLineItem** coclass from the Rose browser and drop it on the diagram.
- Add a realize relationship— —from POSLineItem to the ISalesLineItem coclass.
- Drag the **Logical View \ Design Model \ POS \ POS Server \ Sales Services \ ISalesTransaction** coclass from the Rose browser and drop it on the diagram.
- Add a realize relationship from POSTransaction to the ISalesTransaction coclass.

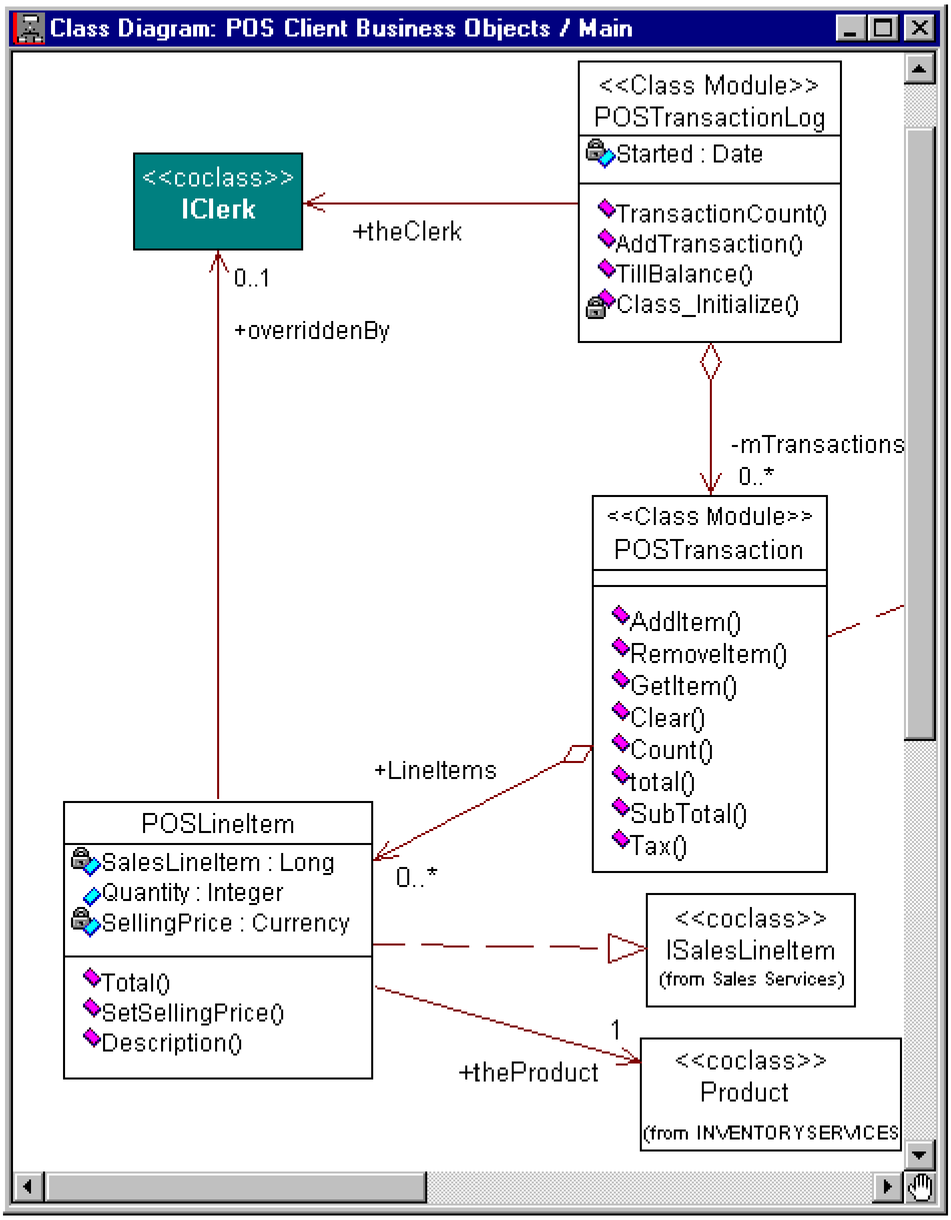
The POSLineItem class uses the IProduct interface, which is indicated by an association relationship.

- Drag the Product coclass from the **Logical View \ Design Model \ POS \ POS Server \ INVENTORYSERVICESLib package** in the browser and drop it on the diagram. (The INVENTORYSERVICESLib package is called Inventory Services if you created the DLL yourself, and the coclass is located in a package called

CoClasses.)

- Add an association relationship from the POSLineItem class to the Product coclass.
- Set the multiplicity to 1 and the role name to theProduct on the Product side.
- Make sure that only the Product side of the relationship is navigable.

检查一下：



Now that the POSLineItem class has been initially created, you need to assign it to the POSClient component. You must also configure the Visual Basic language integration.

- Click Tools > Visual Basic > Component Assignment Tool.
- Select the Unassigned node in the left pane.
- Drag the POSLineItem class from the right pane and drop it on the POSClient

component in the left pane.

- Click Yes.
- Click OK.
- Click Tools > Visual Basic > Properties and clear the check box next to the following options: Save model before code and model updateGenerate debug codeGenerate error-handling code
- Click OK.

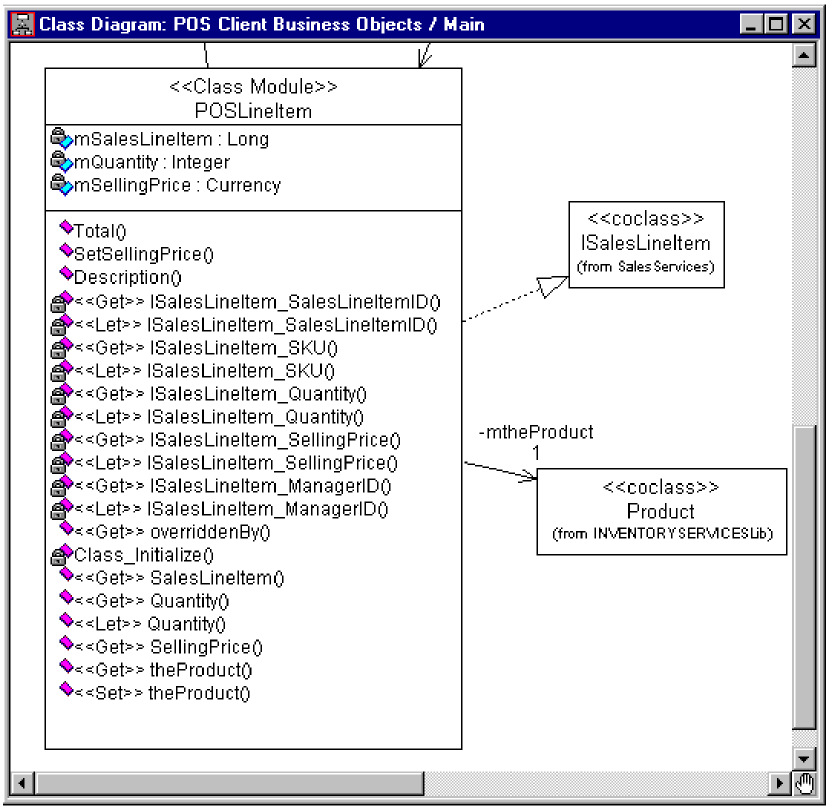
Now that the POSLineItem class has been assigned to a Visual Basic component, you can specify implementation details for the class. Create Property Get and Property Set procedures for the class' attributes by using the Model Assistant.

- Right-click on the POSLineItem class and click Model Assistant.
- Under the Properties folder, expand the overriddenBy property (attribute). The displayed overriddenBy item represents the data member that Rose generates from the property.
- Click the check box next to Get. Note that the name of the data member changes to "moverriddenBy" to avoid a name collision with the property procedure. This data member is modified by other operations on the class, so you don't need a Property Set procedure.
- In the same way, create a Property Get procedure for the SellingPrice and SalesLineItem data members.
- Create both Property Get and Property Set procedures for theProduct and Quantity data members.

Because the class needs to be initialized when created, you should also add a Class_Initialize method to the class.

- Under the Methods folder, click the check box next to Class_Initialize to add an initialization method to the class.
- Click OK.

检查一下：


Class Diagram: POS Client Business Objects / Main
<<Class Module>>
POSLineItem
mSalesLineItem : Long
mQuantity : Integer
mSellingPrice : Currency
Total()
SetSellingPrice()
Description()
<<Get>> ISalesLineItem_SalesLineItemID()
<<Let>> ISalesLineItem_SalesLineItemID()
<<Get>> ISalesLineItem_SKU()
<<Let>> ISalesLineItem_SKU()
<<Get>> ISalesLineItem_Quantity()
<<Let>> ISalesLineItem_Quantity()
<<Get>> ISalesLineItem_SellingPrice()
<<Let>> ISalesLineItem_SellingPrice()
<<Get>> ISalesLineItem_ManagerID()
<<Let>> ISalesLineItem_ManagerID()
<<Get>> overriddenBy()
Class_Initialize()
<<Get>> SalesLineItem()
<<Get>> Quantity()
<<Let>> Quantity()
<<Get>> SellingPrice()
<<Get>> theProduct()
<<Set>> theProduct()
<<coclass>>
ISalesLineItem
(from Sales Services)
-mtheProduct
1
<<coclass>>
Product
(from INVENTORYSERVICESLib)

In this exercise, you specify how to implement the aggregation relationship between POSTransaction and POSLineItem.

- Right-click on the POSTransaction class and click **Model Assistant**.
- Under the Properties folder, expand the LineItems property (attribute).
- Click the check box next to Get. In the **Preview** box for the data member, note that the data member is implemented as a collection because the multiplicity for the relationship is unbounded.
- Click **OK**

Generating code for the new business object

Now that you have specified all implementation details about the new class, you are ready to generate Visual Basic code. You will also update the source code for the POSTransaction class with the new aggregation relationship.

- Start Microsoft Visual Basic 6 and open **Module6a\start\POSClient\POSClient.vbp**.
- Right-click on the POSLineItem class in Rational Rose and click **Update Code**.
- Click **Next** if the **Welcome** page is shown.
- Expand the POSClient component and make sure that POSLineItem is selected, which it should be by default.
- Select the POSTransaction class by clicking the check box next to its name.

- Click **Next** and verify that the two selected classes are listed on the **Finish** page.
- Click **Finish**.
- When Rational Rose is finished, take a look at the result on the **Summary** page.
- Click **Close**.
- Right-click on POSTransaction in the model and click **Browse Source**. Note that a line for the mLineItems private property has been added.
- Also note that a new class module, POSLineItem, has been added to the project. Take a look at the generated source code for the new class module.

Completing the implementation of the new business object

The next step is to complete the source code for the new class module, POSLineItem. Complete the following methods by adding the specified lines of code. You can copy the code from the help window and paste it into the POSLineItem class module in Visual Basic.

```
Public Function total?
    Total = SellingPrice * Quantity

End Function

Public Sub SetSellingPrice?
    If overriddenBy Is Nothing Then
        mSellingPrice = theProduct.Price
    Else
        mSellingPrice = aPrice
        Set moverriddenBy = overriddenBy
    End If

End Sub
Public Function Description?
    Dim descr As String
    descr = "[" & Quantity & "] " & theProduct.ProductSKU & " " & theProduct.Composer
& " " & theProduct.Composition & " (" & FormatCurrency(SellingPrice) & ")"

    Description = descr

End Function

Private Property Get ISalesLineItem_SKU?
    If Not theProduct Is Nothing Then
        ISalesLineItem_SKU = theProduct.ProductSKU
    End If
End Property
Private Property Get ISalesLineItem_ManagerID?
```

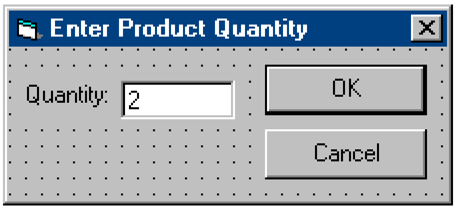
```
        If Not overriddenBy Is Nothing Then
            ISalesLineItem_ManagerID = overriddenBy.ClerkID
        End If
    End Property


    Public Property Set theProduct?
        Set mtheProduct = vNewValue

      mSellingPrice = mtheProduct.Price

    End Property

    Private Sub Class_Initialize?      Quantity = 1
        Set mOverriddenBy = Nothing
        Set mtheProduct = Nothing
    End Sub
```

Adding a new form

The only thing that is missing from the POSClient is a form for displaying quantities. User interface elements are often easier to add to the model by first creating them in the IDE and then reverse engineering them back into the model.

- In Microsoft Visual Basic, add a form of the type dialog.
- Make sure that the form have an OK and a Cancel button (which it should have by default if you created a dialog).
- Give the form the name frmQuantity.
- Save the form as 🗁 **Module6a\start\POSClient\Quantity.frm**.
- Give the form the caption Enter Product Quantity.
- Add a text box control called txtQuantity.
- Add a label with the caption Quantity: in front of the text box.

检查一下：



The next step is to complete the source code for the new form, frmQuantity. Complete the following methods by adding the specified lines of code.

```
    Public Quantity As Long

    Private Sub CancelButton_Click()
        Quantity = -1
        Unload Me
        End Sub

        Private Sub Form_Load()
                TxtQuantity = Quantity
End Sub
        Private Sub OKButton_Click()
                If txtQuantity.Text <> "" Then
                        On Error GoTo badamount
                        Quantity = CLng(txtQuantity.Text)
                        On Error GoTo 0
                Else
                        Quantity = 1
                End If
                Unload Me
                Exit Sub
        badamount:
                MsgBox "The value entered in for the quantity is invalid."
        End Sub

        Private Sub txtQuantity_GotFocus()
                TxtQuantity.SelStart = 0
                TxtQuantity.SelLength = Len(txtQuantity.Text)
End Sub
```

Testing the application

After generating and completing the code, and before updating the model with your changes, you need to verify the code by compiling the project. Because you have completed the implementation of the application, you should be able to run it.

- If you created the Inventory Services component in the Visual C++ exercise, it is marked as MISSING in the **Project > References** dialog. If so, add a reference to the InventoryServices.dll that you previously created and remove the missing reference.
- In Microsoft Visual Basic, click **Run > Start With Full Compile**.
- Correct any errors and recompile until the code is free from errors.

Updating the model from code changes

The next step is to update the model with the new form.

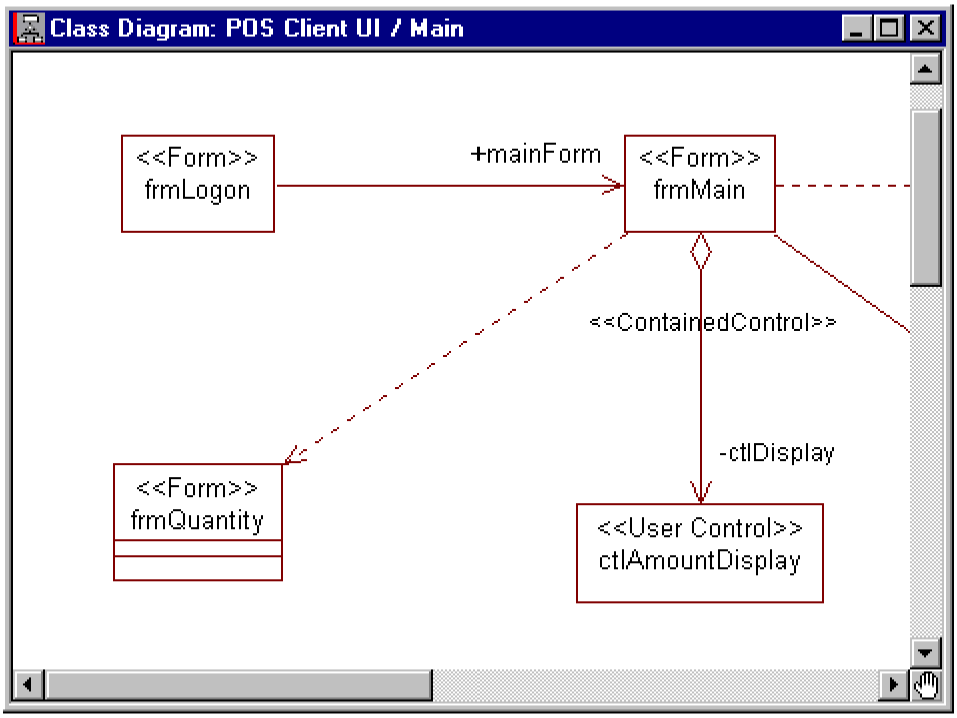
- In Rational Rose, click **Tools > Visual Basic > Update Model from Code**.
- Click **Next** if the **Welcome** page is shown.
- Clear all check boxes except the check box next to the Visual Basic node.
- Expand the POSClient component and select only the frmQuantity form.
- Click **Next** and verify that frmQuantity is listed on the Finish page.
- Click Finish.

- When Rational Rose is finished, take a look at the result on the Summary page.
- Click Close.

Because the main form creates and uses an instance of the Quantity form but isn't associated with a specific instance of it, you need to add a dependency relationship between the main form and the new form in the model.

- Rational Rose puts new reverse engineered model elements in a logical package called **Logical View \ Reverse Engineered**. Drag the new form from that package and drop it onto the **Logical View \ Design Model \ POS \ POS Client \ POS Client UI package**.
- Open the Main diagram in the **Logical View \ Design Model \ POS \ POS Client \ POS Client UI pack**age.
- Drag frmQuantity from the browser and drop it on the diagram.
- Add a dependency relationship from frmMain to frmQuantity.

检查一下：



Congratulations!

You have completed the Visual Basic exercises and the POS Client project.

Summary

In this module you learned how to use Rational Rose to round-trip engineer Visual C++ and Visual Basic applications.

The round-trip engineering tools for Visual Studio projects in Rational Rose are:

- Component Assignment Tool
- Model Assistant

- Code Update Tool
- Model Update Tool

You also saw how to import the type libraries of COM components into the model by using the Type Library Importer in Rational Rose.

More information

For more information on how to round-trip engineer a Visual Basic and Visual C++ application in Rational Rose, please refer to:

- Using Rational Rose Visual C++ manual
- Using Rational Rose Visual Basic manual
- Quick Start with Rose Visual Basic manual

For more information about the Type Library Importer in Rational Rose, please refer to the corresponding chapter in the "Using Rational Rose 2000" manual.

For information about COM components and type libraries, please refer to:

- Don Box, "Essential COM", Addison-Wesley Pub Co, ISBN 0201634465
- Ted Pattison, "Programming Distributed Applications with COM and Microsoft Visual Basic 6.0", Microsoft Press, ISBN 1-57231-961-5
- http://msdn.microsoft.com/library

What's next?

The Classics, Inc. model in this part of the tutorial was designed for a single user updating the information. In real world development, a model is often shared by a team. The next module describes how to set up a model for team development and what to consider when working in a team.

第七章　　**Working in a Team**

一、关于这一章

This module describes how Rational Rose supports teams of analysts, architects, and software developers working in parallel. The module begins by describing overall strategies for team development, then introduces the Rose tools and features that support those strategies, including:

- Controlled units
- Virtual path maps
- Model Integrator

Approximate completion time30 minutes.

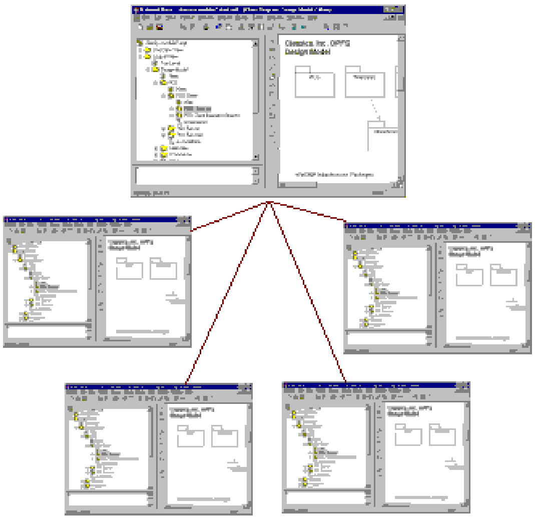
二、**Planning for team development**

Developing complex systems requires that groups of analysts, architects, and developers be able to see and access the “big picture”while working on their own portion of that picture ---simultaneously.

Successfully managing an environment where multiple team members have different kinds of access to the same model requires:

- Formulating a working strategy for managing team activity
- Having the tools that can support that strategy

A single model needs to be shared



by teams of designers and developers

三、**Formulating a strategy**

When developing a strategy for working in teams, remember that there are two facets to consider:

- Developing a strategy that supports current development
- ?Developing a strategy for maintaining and retrieving the reusable modeling artifacts that result

Developing a strategy for current projects

When developing current projects, the tools a team uses must be able to:

- Provide all team members with simultaneous access to the entire model
- Control which team members can update different model elements

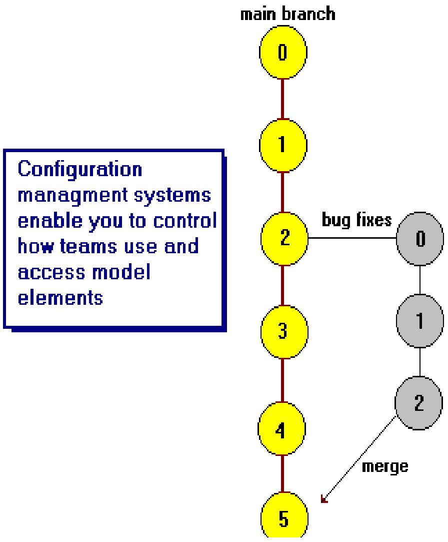
- Introduce change in a controlled manner
- Maintain multiple versions of a model

The role a configuration management system plays

Implementing a configuration management system is essential for complex projects. A configuration management system can effectively support team development as long as it:

- Protects developers from unapproved model changes
- Supports comparing and merging all changes made by multiple contributors
- Supports distributed (geographically dispersed) development

Consider using the same configuration management tool to maintain your models that you use for your other project artifacts, such as source code, and dlls.



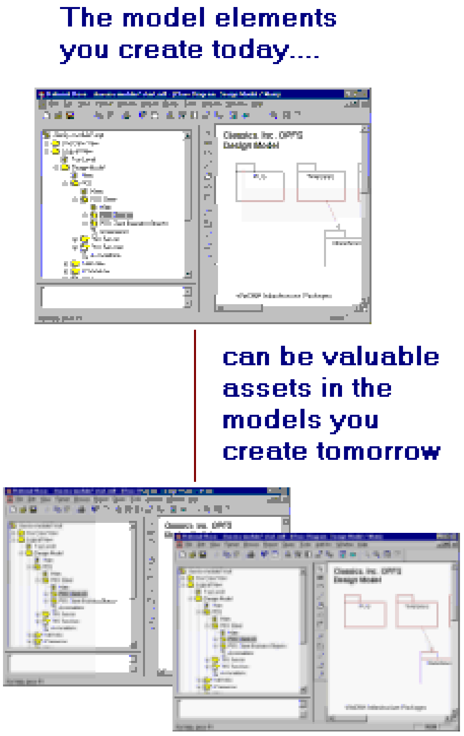
Configuration managment systems enable you to control how teams use and access model elements

Integrating a configuration management system in Rose

Since managing parallel development is so crucial, Rose provides integrations with Rational ClearCase and with SCC -compliant version control systems such as Microsoft Visual Source Safe.By integrating configuration management systems, Rose makes the most frequently used version control commands directly accessible from Rose menus, including the typical check in and check out functions that are used every day.

Developing a strategy for reuse

When you develop a system, you are developing valuable project artifacts that can be reused. Artifacts are typically maintained in some type of repository. To support reuse:?Model artifacts should be architecturally significant units, such as patterns, frameworks, and components (not usually individual classes)?All members of a team, no matter where they are located, should have access to reusable artifacts?It should be easy to catalog, find, and then apply these artifacts in a modelA reuse repository can differ from your project 抯 configuration management system as long as it supports versioning. The repository should also support



The model elements you create today....

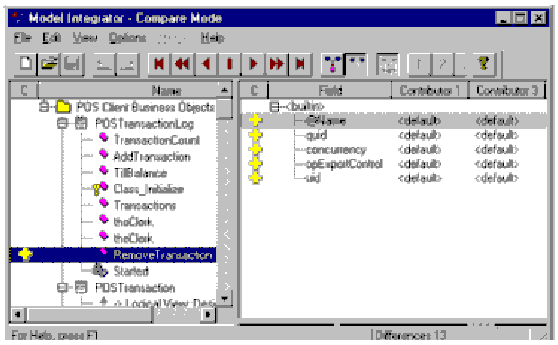
can be valuable assets in the models you create tomorrow

cataloging artifacts at an appropriate level of granularity (for example, at the component level).

Sample tools for managing reuse

Two tools for managing reusable artifacts are:

- Rose Framework Wizard. You can create your own frameworks that enable you to capture and reapply entire generic models. (Rose uses this feature to supply frameworks such as those for Oracle 8 and Java?)
- Microsoft Visual Component Manager (VCM). VCM is built on top of Microsoft's Repository. Rose integrates with VCM, enabling you to use it to publish entire models. Plus, you can drag and drop packages and/or classes directly from VCM into a Rose model.



C	Field	Contributor 1	Contributor 3
	<builtin>		
	@Name	<default>	<default>
	quid	<default>	<default>
	concurrency	<default>	<default>
	opExportControl	<default>	<default>
	uid	<default>	<default>

And now about controlled units, virtual path maps, and Model Integrator

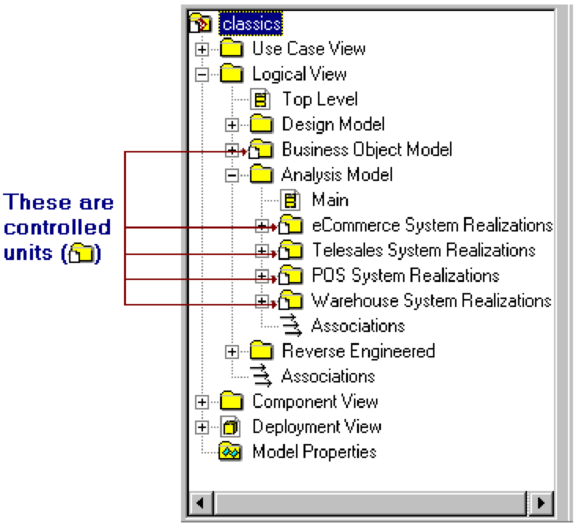
In addition to integrating with configuration management systems, Rose provides tools and features that enable you to support parallel development by teams, large and small. They are:

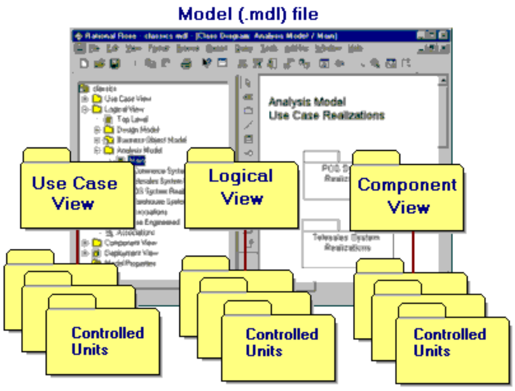
- Controlled units
- Virtual path maps
- Model Integrator

Each of these tools plays a crucial role in successful team development.

What is a controlled unit?

In other modules in this tutorial, you worked with a version of a single model file, Classics.mdl. In a typical development environment, a complex system like Classics would involve many team members working on portions of the model. Rose supports this by letting you partition a model into separate files called controlled units. When using controlled units each team or each team member is responsible for maintaining or updating a specific unit.The lowest level of granularity for a controlled unit is a package, since packages are considered the smallest architecturally significant elements in a model (classes are not).Controlled units are the basic building blocks that you put under version control.

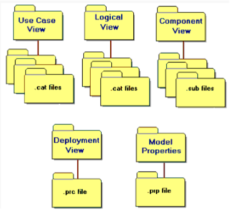

These are controlled units ( )



Creating a hierarchy of controlled units

You can create a hierarchy of controlled units where top level controlled units can consist of references to other controlled units.For example, you could make all packages controlled units with top-level packages that are pointers to nested packages. When you do this, you enable two developers to check out packages that belong to the same higher level package.How you partition a model and the type of hierarchy you implement will depend on how team members will operate, both physically (who works on which packages) as well as logically (how best to partition the model and preserve its design).Later in this tutorial there are guidelines that should help you plan how to best divide a model into controlled units.

What can be a controlled unit



You can create controlled units for packages, deployment diagrams, and model properties. When you create controlled units, you name the new file but you use one of these four extensions for the particular type of controlled unit you're creating:

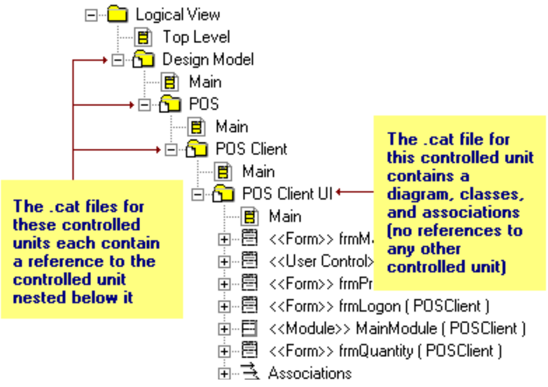
- Logical packages and use-case packages are stored in .cat files
- Component packages are stored in .sub files
- Deployment diagrams are stored in .prc files
- Model properties are stored in a .prp file

You can have an unlimited number of .cat and .sub files but since a Rose model supports one deployment diagram, there is only one .prc file. Similarly, there is a single set of model properties and only one .prp file.

How controlled units are referenced

When you create a controlled unit from a package, you physically move the contents of the package to a new file and you leave behind a reference to the new file.

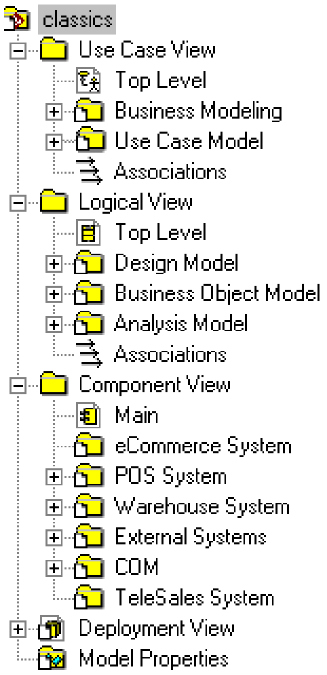
The "contents"of this new file can vary. When you create a controlled unit from another controlled unit, the enclosing controlled unit will contain a reference to the other unit. You could conceivably have cascading levels of controlled unit files that



The .cat files for these controlled units each contain a reference to the controlled unit nested below it

The .cat file for this controlled unit contains a diagram, classes, and associations (no references to any other controlled unit) contain nothing but references to other units.

Also, only the enclosing controlled unit has a reference to its subunits--the references don 抰 propagate through your hierarchy (i.e., the top level doesn’t know how far it is to the bottom).



Guidelines for partitioning a model into controlled units

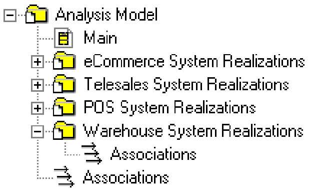
Use these guidelines when deciding how to partition a model into controlled units. First and foremost:

- Make it your goal to partition a model so that it can evolve in a controlled manner, but…
- Try to reduce dependencies between different portions of a model in order to support future reuse

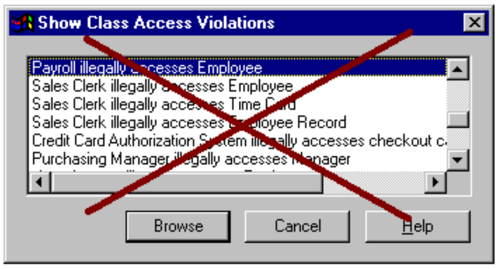
More specifically:

- The model should be a shell with nothing but controlled units under the use case, logical, and component views
- Create design model, analysis model, and business model controlled units under the logical view
- Create an implementation model controlled unit under component view
- Consider separating actors and use case controlled units
- Also consider separate controlled units for each use case
- Prevent your use case controlled units from including any diagrams that describe internal system operations or structure, such as class or interaction diagrams
- Under the design model and analysis model packages, provide a use case realizations controlled unit

For example...


Analysis Model
Main
eCommerce System Realizations
Telesales System Realizations
POS System Realizations
Warehouse System Realizations
Associations
Associations

- Provide a separate controlled unit for each realization
- Class and interaction diagrams that describe system internals should go with the use case realizations
- Describe the system structure using a series of nested packages that become controlled units
- Layers and global packages should be at the top level of nesting
- Maintain interfaces in separate controlled units
- Describe each significant mechanism in its own controlled unit



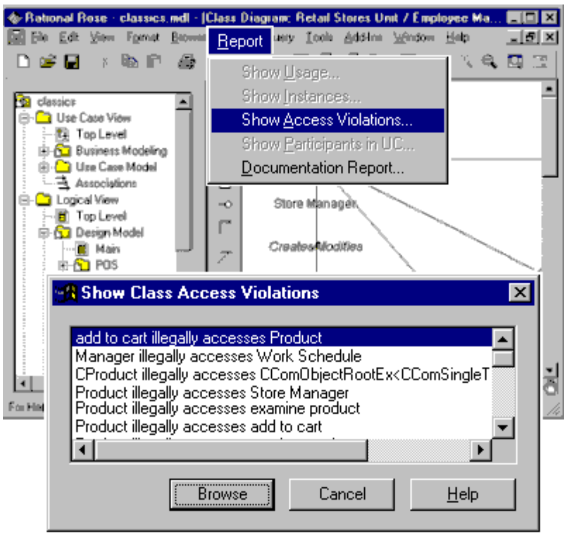
Payroll illegally accesses Employee
Sales Clerk illegally accesses Employee
Sales Clerk illegally accesses Time Card
Sales Clerk illegally accesses Employee Record
Credit Card Authorization System illegally accesses checkout c
Purchasing Manager illegally accesses Manager

And remember…

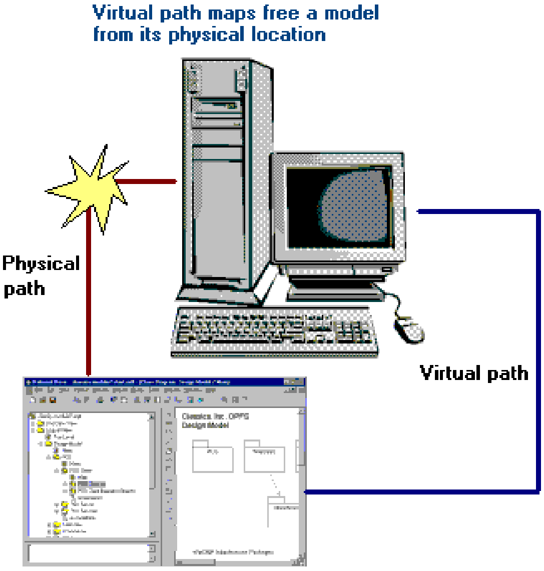
- A controlled unit strategy is

compromised if the contents of the controlled units are too intertwined

- Control dependencies: create UML subsystems by using packages that provide discrete, well-defined services
- Subsystems should expose services only via UML interfaces — they provide strong separation between major portions of the model
- Subsystem internals should depend only on the interfaces that are offered by other subsystems
- Developers sometimes define class-level relations that violate dependencies between packages and subsystems. To detect this in a model, use **Report >Show Access Violations.**

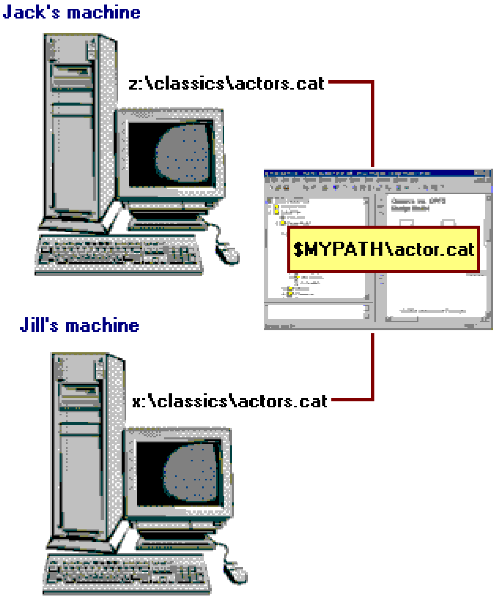

Rational Rose - classics.mdl - [Class Diagram: Retail Stores Unit / Employee Ma...
File Edit View Format Browse Report Query Tools Add-Ins Window Help
Show Usage...
Show Instances...
Show Access Violations...
Show Participants in UC...
Documentation Report...
classics
Use Case View
Top Level
Business Modeling
Use Case Model
Associations
Logical View
Top Level
Design Model
Main
POS
Store Manager
CreatesModifies
Show Class Access Violations
Browse
Cancel
Help

add to cart illegally accesses Product
Manager illegally accesses Work Schedule
CProduct illegally accesses CComObjectRootEx<CComSingleT
Product illegally accesses Store Manager
Product illegally accesses examine product
Product illegally accesses add to cart



Virtual path maps—making your model file and controlled units portable

Virtual path maps enable Rose to use relative file paths instead of physical file paths when referencing controlled units. This feature enables you to move a model between different systems or directories and to update a model from different workspaces.When you save a model or you create a controlled unit, you save it to a physical location. However, your model file and any parent controlled units rely on that file path to locate the controlled units that belong to it. By creating and using virtual path maps, you enable Rose to substitute the physical file path with a relative file path, freeing your model from its ties to a physical location.



Virtual path map example

For example, if you define the virtual path:

$MYPATH=z:\classics

and save a package as:

z:\classics\actors.cat

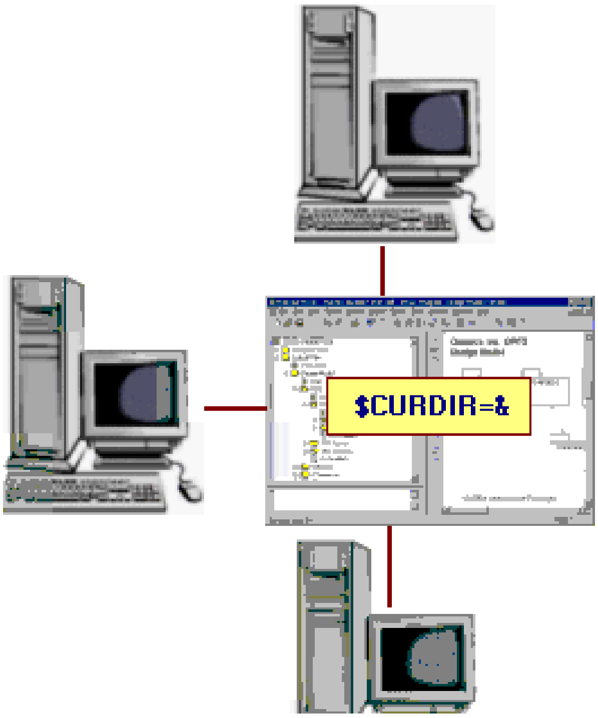
the model file will refer to the package as:
$MYPATH\actor.cat
When another user who has defined $MYPATH as
$MYPATH=x:\classics
opens the same model from his or her X drive, Rose resolves the internal reference to the controlled unit and loads the file
x:\classics\actor.cat

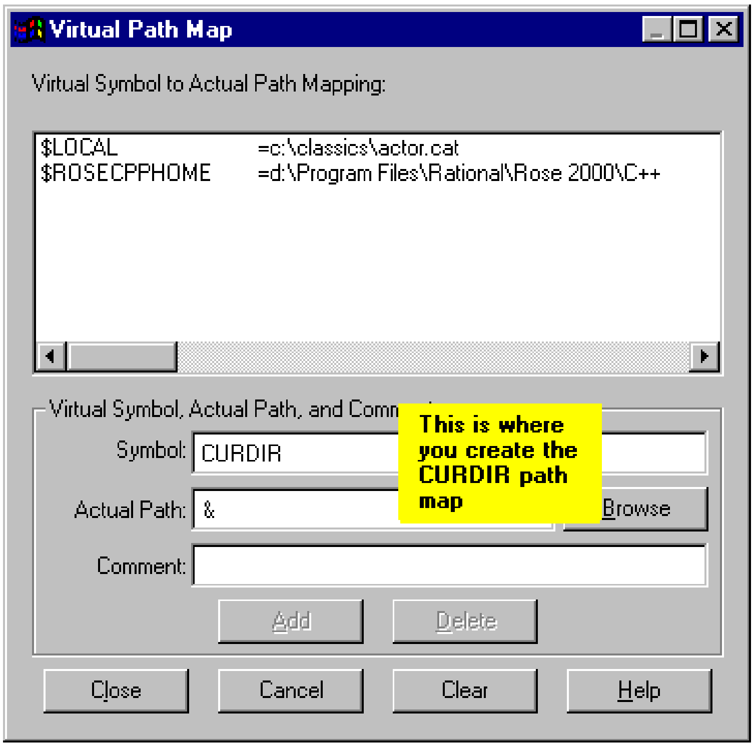
Implementing virtual path maps for a team

A leading ampersand (&) in a virtual path map indicates that the path is relative to the model file or the enclosing (parent) controlled unit.

A common way to implement path maps is to have all team members define $CURDIR=&. This enables you to save a model and controlled units relative to the surrounding context letting different users open the model and load the unit in different workspaces. For example, if each user created the path map:
$CURDIR=&
When the model is saved, the model file 扭 reference to the actor package is stored as:
$CURDIR\units\actor.cat

When another user in another workspace opens the model, $CURDIR is expanded to the physical path to the model in that specific workspace. For example:z:\classics




Virtual Path Map

Virtual Symbol to Actual Path Mapping:

```
$LOCAL          =c:\classics\actor.cat
$ROSECPPHOME    =d:\Program Files\Rational\Rose 2000\C++
```

Virtual Symbol, Actual Path, and Comm

Symbol: CURDIR

Actual Path: &

Comment:

This is where you create the CURDIR path map

Try it yourself

This module's sample model uses controlled units. To open the model, create your own virtual path map.

试一试：

Creating a virtual path map

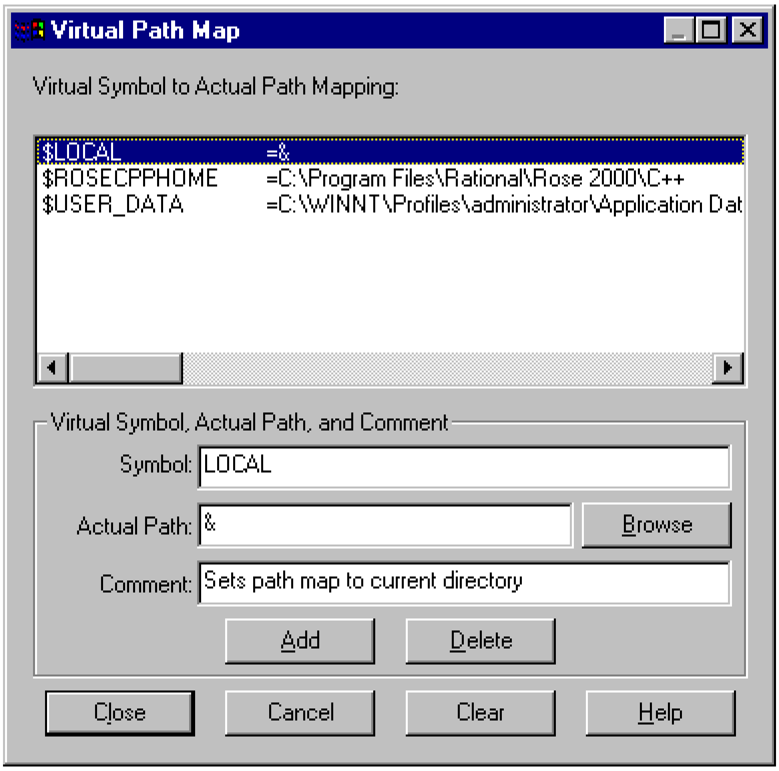
The developer who created

the model you are working on in Module 7 stored its subunits using a path map symbol called $LOCAL. In this exercise, you play the role of another developer on the team that is working on this model.

For Rose to locate and load the model's subunits on your system, regardless of where you install the model, you must define the same path map symbol ($LOCAL) and set it to the location of the model's subunits.

- Start Rational Rose.
- If the Create New Model dialog box is displayed, click Cancel.
- Click File > Edit Path Map…
- Add a pathmap symbol called $LOCAL and set its actual path to &.

检查一下：



Virtual Symbol to Actual Path Mapping:

```
$LOCAL          =&
$ROSECPPHOME    =C:\Program Files\Rational\Rose 2000\C++
$USER_DATA      =C:\WINNT\Profiles\administrator\Application Dat
```

Virtual Symbol, Actual Path, and Comment

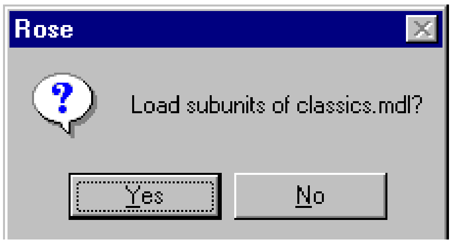
Symbol: LOCAL

Actual Path: &

Comment: Sets path map to current directory

Loading and unloading controlled units

When you open a model, Rose prompts you whether to load all referenced controlled units. If the model is large, or you are planning to work on a few specific units, you can click No, then load individual controlled units only when you need to view or modify their contents.



Load subunits of classics.mdl?

To view or update a unit that has been modified by another developer since you loaded it, you must reload the unit.

Controlled units are protected, either in the file system (individual files can be write-protected) or by version control (a file is checked out or checked in).

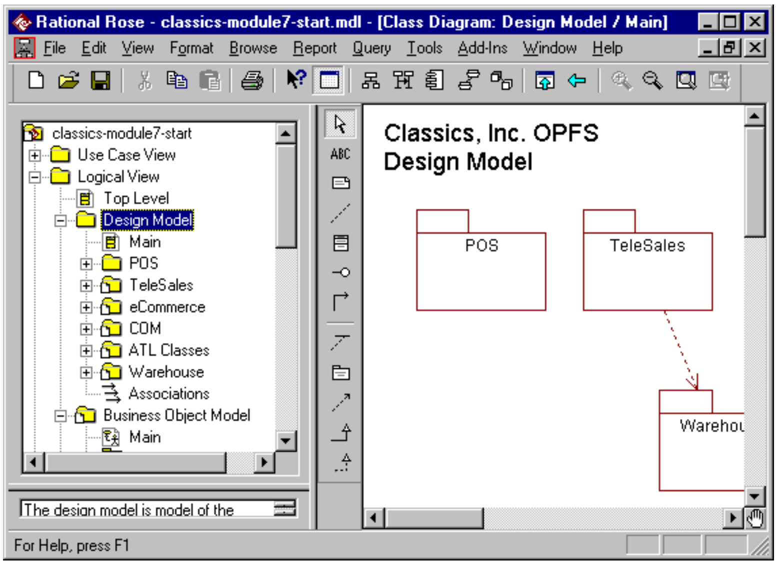
试一试：

Loading and unloading controlled units

In this exercise, you open a model, load its subunits, and also unload a unit from the model.

- If you did not complete the Path Map exercise, which precedes this exercise, do so now. If you do not, you will not have a valid path map, and you will be unable to load the subunits of the model.
- Create a folder called teamdevmodel on your system.
- Find the tutorial's .../Module7/start/ folder, and copy its entire contents to the teamdevmodel folder you just created.
- In Rose, click File > Open, browse to find the **teamdevmodel** folder, and select **classics-module7-start.mdl**.
- When you are prompted to load the model's subunits, click Yes.
- In the Rose browser, expand some of the packages and notice the icon, which indicates that a package is a unit and that it is currently loaded.
- Right-click one of the loaded controlled units, click Units > Unload and notice the change in the icon of the unloaded unit.
- Right-click the unloaded unit and click Units > Load to reload the unit.

检查一下：



The design model is model of the

Creating controlled units

There are two ways to create a controlled unit:

- If you're not using a configuration management system, in the browser select the package,

right-click and choose Units > Control from the shortcut menu

- If you're using ClearCase or an SCC-compliant configuration management system, add the package to version control by selecting the package and selecting Add to Version Control from the shortcut menu

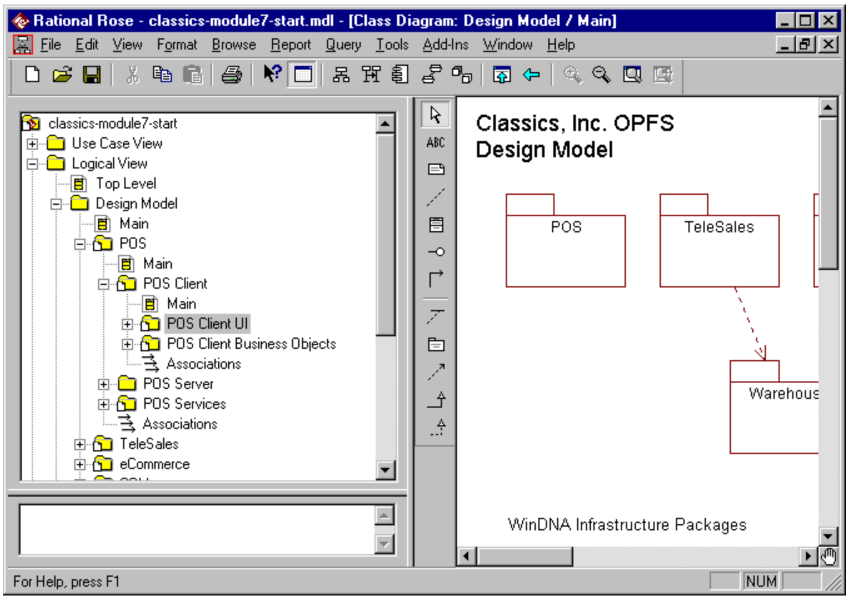
试一试：

Controlling model packages

Much of classics-module7-start.mdl is already divided into controlled units. In this exercise, you control two additional logical packages: POS Client UI and POS Business Objects. Both of these packages are nested in the Design Model in the Logical View of the model.

- Control the packages by right-clicking each package and clicking Units > Control.
  - Save the POS Client UI unit as lv-POSClientUI.cat
  - Save the POS Client Business Objects unit as lv-POSClientBusinessObjects.cat.
- Save the model and its subunits.If you have access to a live ClearCase environment, you can use the ClearCase add-in to control the packages in your ClearCase Versioned Object Base (VOB). To do this, right-click the package and click **Add to Version Control**. However, before using this method, you must ensure that ClearCase is set up properly, your VOB is created, your ClearCase view has been added to version control, and the ClearCase add-in is activated in Rational Rose.

检查一下：



Controlled units in the browser

In the browser, you can see whether a controlled unit is:

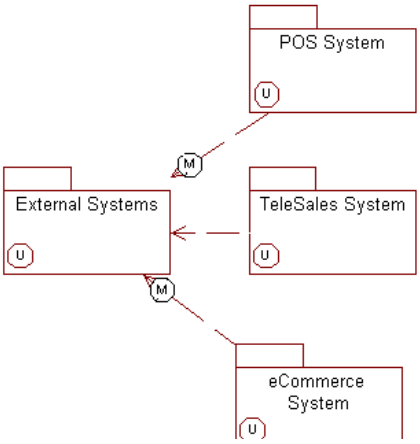
- Loaded
- Not loaded

If you are using ClearCase or Rose's Version

Control add-in, the browser also indicates whether a controlled unit is:

- Checked-in
- Checked-out

For a controlled unit that is selected in the browser, the status bar displays information about the unit’s current state (write enabled, write protected, or modified) together with the unit 抯 file path.



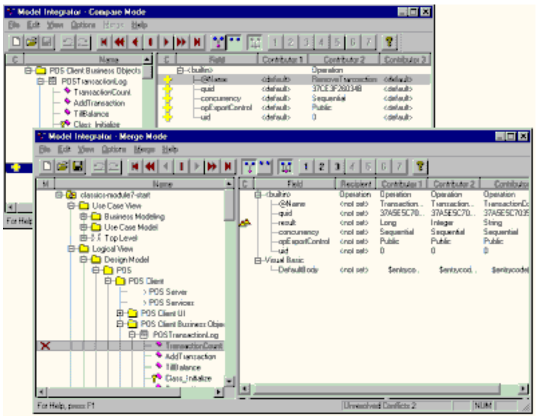
Controlled units in diagrams

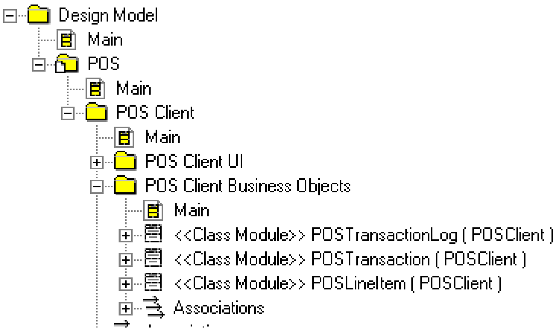
Roses uses the following icons to identify controlled units and their status:

- U on a package means that the package is a controlled unit
- M on a model element means that the package where the model element is contained is not loaded

Using Model Integrator to compare and merge

Rose Model Integrator is a powerful tool that lets you compare model elements from up to seven contributor files, discover their differences, and merge the elements into a receiving model. The files you compare can be .mdl, .cat, .sub, .pty, .prc, or .prp files, although you can only compare and merge like items (.mdl files to other .mdl files, .cat files to other .cat files).


Model Integrator - Compare Mode
Model Integrator - Merge Mode



Design Model
Main
POS
Main
POS Client
Main
POS Client UI
POS Client Business Objects
Main
<<Class Module>> POSTransactionLog (POSClient)
<<Class Module>> POSTransaction (POSClient)
<<Class Module>> POSLineItem (POSClient)
Associations

Allowing multiple team members to change the same controlled unit

For example in the Classics, Inc project, there is a baseline model and two different teams making changes in private workspaces.First, the Point of Sale team updates a controlled unit in the Classics, Inc. model.Next, the Corporate team changes the same controlled unit.Using Model Integrator, compare and merge the changes made by the different teams.

试一试：

Setting up the environment

In this exercise, you set up two additional folders on your system and copy the classics-module7-start.mdl, along with its controlled units, into each folder. The folder you already have holds the baseline version of the model. The two new folders provide the private workspaces for two team members who independently make changes in their views of the model.

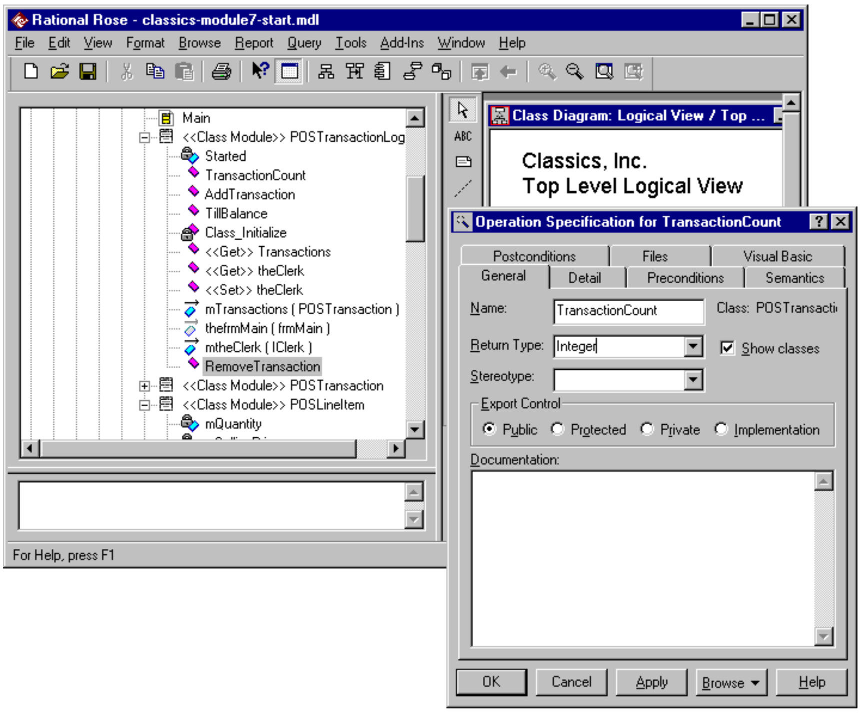
Together, these three folders allow you to simulate a ClearCase (or similar configuration management system) environment.

- Create the following additional folders on your system:
  - teamdevmodel-posteam (a view to be used by a developer on the Point of Sale team)
  - teamdevmodel-corp (a view to be used by a developer at Classics corporate headquarters)
- Copy the entire contents of the teamdevmodel folder into the teamdevmodel-posteam folder and again into the teamdevmodel-corp folder.

Point of Sale team updates the model

- Open classics-module7-start.mdl from the teamdevmodel-posteam folder.
- When prompted to load all units, answer No. Notice that no controlled units are loaded.
- Right-click the POS package in the Design Model and click Units > Load to load the POS package and its subunits.
- For the POSTransactionLog class in the POS Client Business Objects subunit, change the return type of the TransactionCount operation and add a new operation:
- Double-click TransactionCount to open its specification and change the return type to Integer.
- Right-click POSTransactionLog, click New > Operation and type RemoveTransaction over the opname placeholder.

检查一下：



Name: TransactionCount

Class: POSTransacti

Return Type: Integer

☑ Show classes

Stereotype:

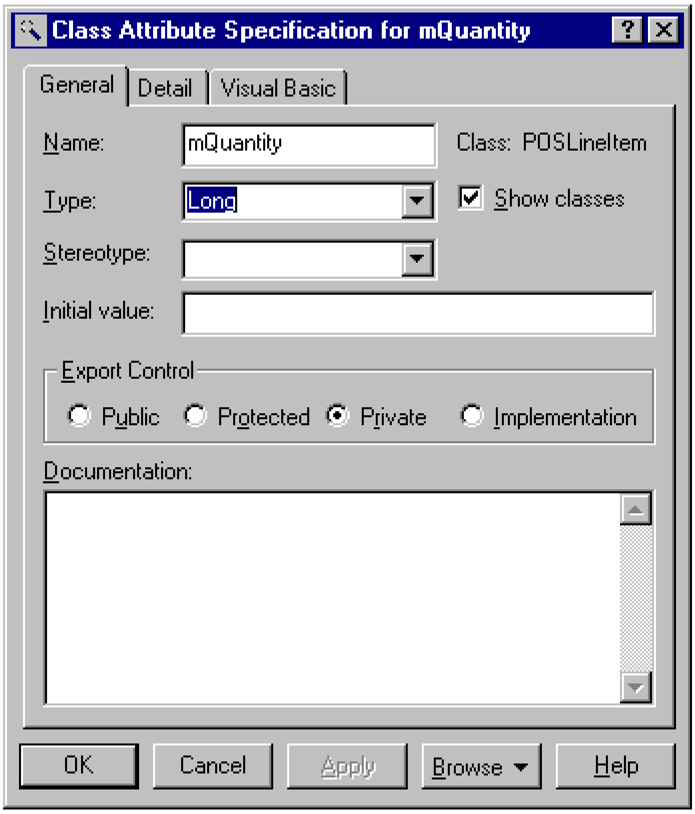
Export Control

● Public ○ Protected ○ Private ○ Implementation

More Point of Sale changes

- For the POSLineItem class in the POS Client Business Objects subunit, change the data type of the mQuantity attribute to Long.
  - Double-click mQuantity to open its specification.
  - Change the type to Long.
- Save the model file and its subunits.

检查一下：



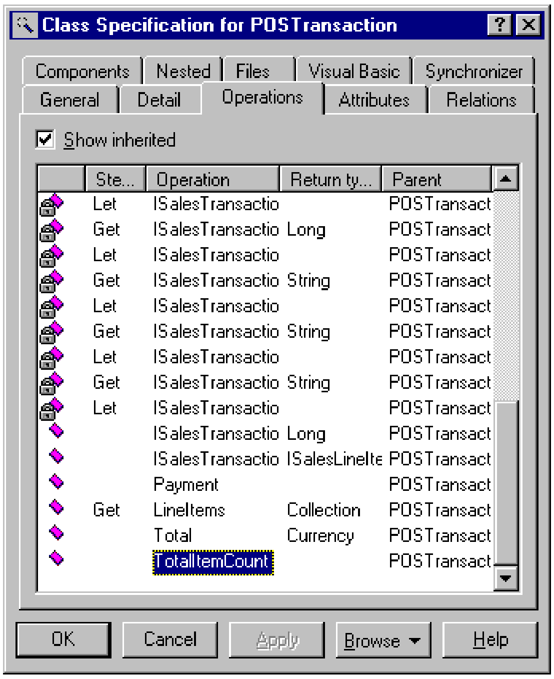
The corporate team changes the model

- Open classics-module7-start.mdl from teamdevmodel-corp
- This time, when prompted to load all units, answer Yes.

  Notice that all of the model's units are loaded. In this case, there is no need to explicitly load the units you want to change. However, opening the model takes longer and the loaded model uses more memory resources on your system.
- To help the corporate office track inventory, add a new operation to the POSTransaction class in the POS Client Business Objects subunit.
  - Right-click POSTransaction.
  - Click New > Operation.
  - Type TotalItemCount over the opname placeholder.

检查一下：结果

OSTransaction has a new operation called TotalItemCount.



☑ Show inherited

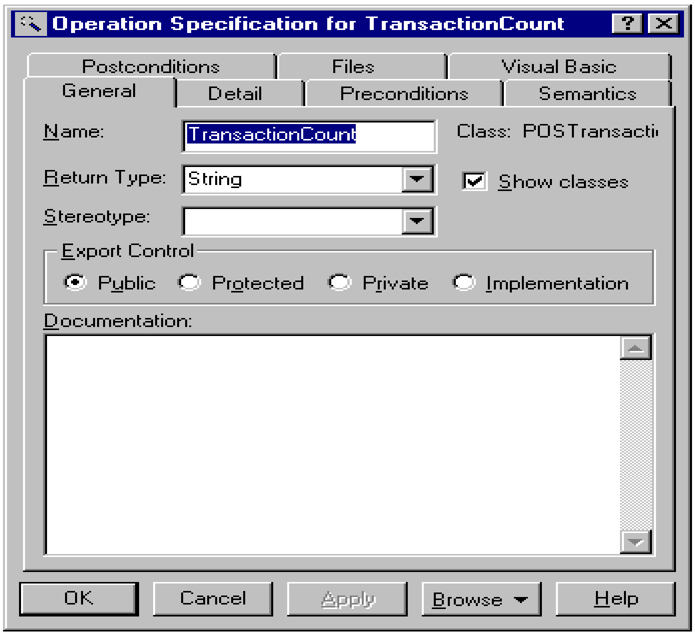
Ste...	Operation	Return ty...	Parent
Let	ISalesTransactio		POSTransact
Get	ISalesTransactio	Long	POSTransact
Let	ISalesTransactio		POSTransact
Get	ISalesTransactio	String	POSTransact
Let	ISalesTransactio		POSTransact
Get	ISalesTransactio	String	POSTransact
Let	ISalesTransactio		POSTransact
Get	ISalesTransactio	String	POSTransact
Let	ISalesTransactio		POSTransact
	ISalesTransactio	Long	POSTransact
	ISalesTransactio	ISalesLineIte	POSTransact
	Payment		POSTransact
Get	LineItems	Collection	POSTransact
	Total	Currency	POSTransact
	TotalItemCount		POSTransact

More corporate team changes

- The corporate team decides that the TransactionCount operation of the POSTransactionLog class will be more useful with a different return type.
  - Double-click TransactionCount to open its specification, and change its return type to String.
  - Save the model file and its subunits.

检查一下：结果

The TransactionCount specification shows the new return type.



Name: TransactionCount

Class: POSTransacti

Return Type: String

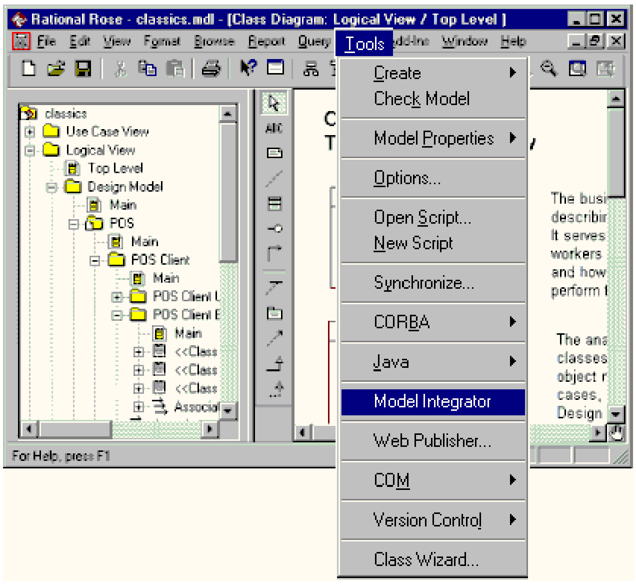
☑ Show classes

Stereotype:

Export Control

● Public ○ Protected ○ Private ○ Implementation

Documentation:



Comparing and merging the updated controlled units

In this part of the exercise, you compare and merge the changes the two teams made to the model. Since the POS team and the Corporate team made all their changes to the same controlled unit (POS Client Business Objects), you compare the .cat files.

试一试：

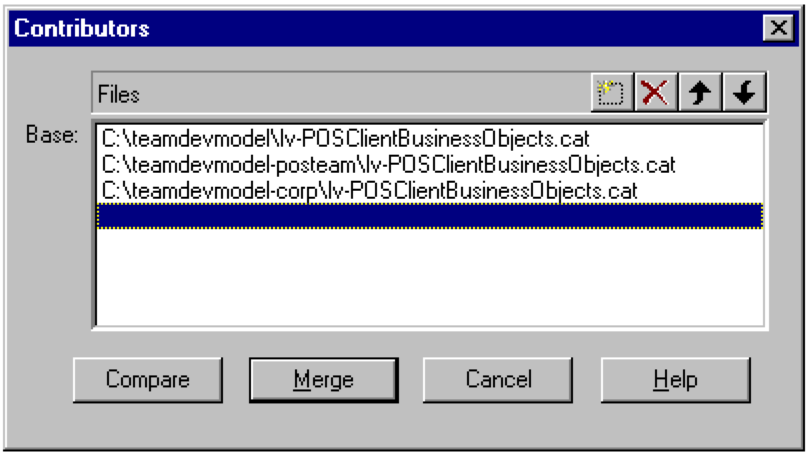
In this exercise, you compare the changes made to the .cat files in the various views (folders) and merge them into the baseline model.

If you did not complete the previous exercises, look in the tutorial folder **Module7**/ to find sample folders and files that match the ones used in this exercise.

- In Rose, click **Tools > Model Integrator**.
- Click **File > Contributors** to open the Contributors dialog box.
- Click the Browse button ...
- Change **Files of type** to **Rose Subunits**.
- Select lv-POSClientBusinessObjects.cat in the teamdevmodel folder (the baseline view).
- Browse again, this time to find and select lv-POSBusinessObjects.cat in the teamdevmodel-posteam folder and, one more time, in the teamdevmodel-corp folders.
  - Need Detailed Instructions? Click the Help button in the Contributors dialog box.

检查一下：结果

The baseline .cat file is the Base file, and the two teams' updated .cat files are the contributor files.


Contributors

Files

Base: C:\teamdevmodel\lv-POSClientBusinessObjects.cat
C:\teamdevmodel-posteam\lv-POSClientBusinessObjects.cat
C:\teamdevmodel-corp\lv-POSClientBusinessObjects.cat

Performing the compare

- Click the **Compare** button in the Contributors dialog box.

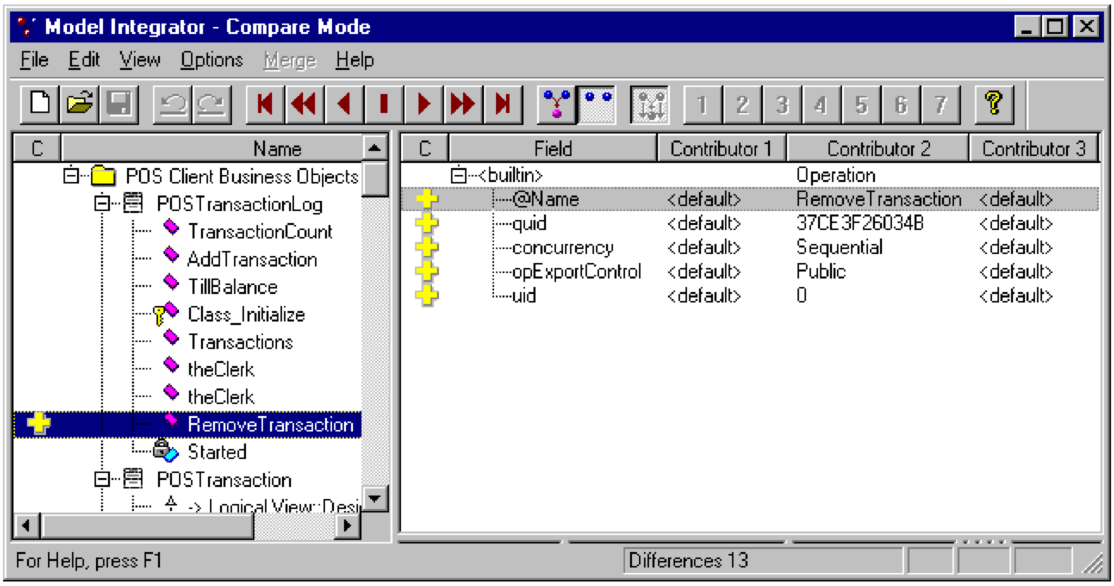
Doing a Compare rather than a Merge allows you to see all the differences between the files. When you do a Merge, Model Integrator only shows the conflicts -- different changes that have been made to the same file.

- When Rose prompts you to load the unit's subunits, click **OK**.

检查一下：结果

In Compare mode, differences between the .cat files appear in the Result window. Here, you can see the RemoveTransaction operation that was added by the Point of Sale Team.

- Use the toolbar buttons to move forward and backward through the results of the Compare.
- Use the Model Integrator online help if you want to know about all the symbols and their meanings.


Model Integrator - Compare Mode
File Edit View Options Merge Help
1 2 3 4 5 6 7

C	Name
	POS Client Business Objects
	POSTransactionLog
	TransactionCount
	AddTransaction
	TillBalance
	Class_Initialize
	Transactions
	theClerk
	theClerk
✚	RemoveTransaction
	Started
	POSTransaction
	-> Logical View::Desi

C	Field	Contributor 1	Contributor 2	Contributor 3
	<builtin>		Operation	
✚	@Name	<default>	RemoveTransaction	<default>
✚	quid	<default>	37CE3F26034B	<default>
✚	concurrency	<default>	Sequential	<default>
✚	opExportControl	<default>	Public	<default>
✚	uid	<default>	0	<default>

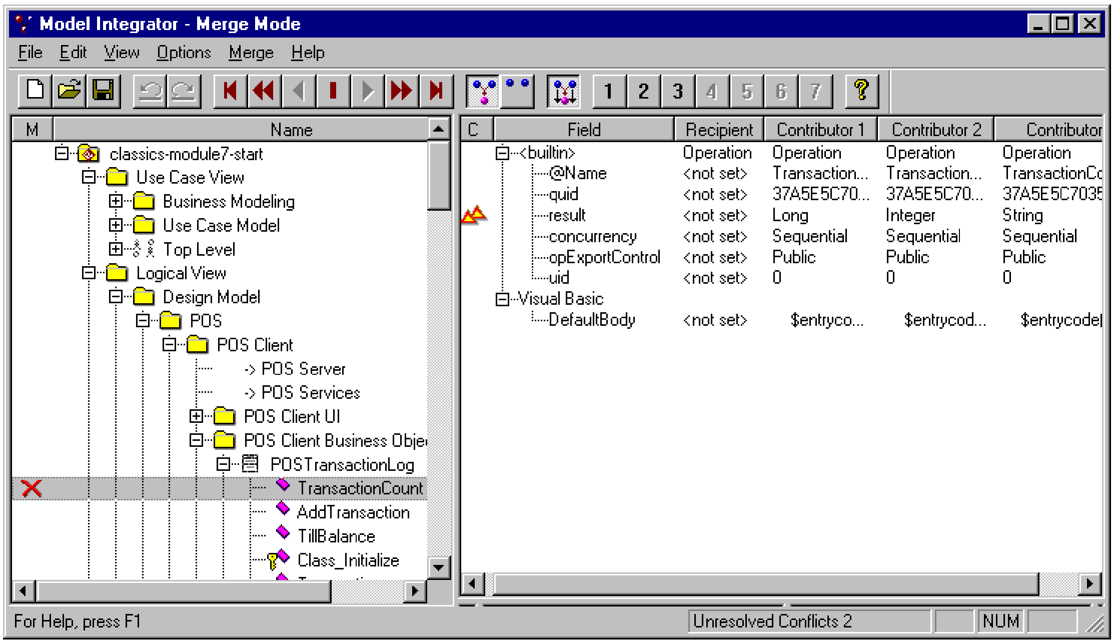
For Help, press F1　　Differences 13

Merging the differences into the baseline file

- Click on the toolbar to switch from Compare mode to Merge mode.
  - A Recipient column appears in the results window. This column contains the merged information that Model Integrator has created from any non-conflicting changes in the contributor files.
  - The left pane of the results window shows the conflicting changes that require a decision on your part.

检查一下：结果

In Merge mode, conflicts between the .cat files appear in the Result window. Notice the conflicting changes to the return type of TransactionCount.

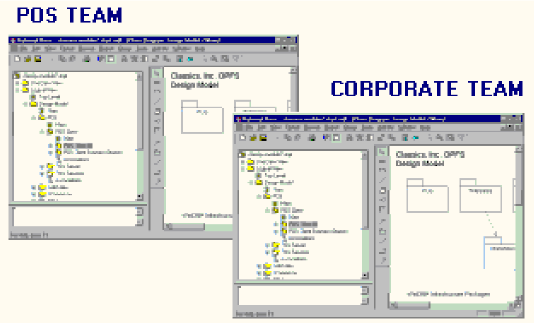

Model Integrator - Merge Mode
File Edit View Options Merge Help
1 2 3 4 5 6 7

```
M   Name
    classics-module7-start
      Use Case View
        Business Modeling
        Use Case Model
        Top Level
      Logical View
        Design Model
          POS
            POS Client
              -> POS Server
              -> POS Services
              POS Client UI
              POS Client Business Obje
                POSTransactionLog
✕                 TransactionCount
                  AddTransaction
                  TillBalance
                  Class_Initialize
```

C	Field	Recipient	Contributor 1	Contributor 2	Contributor
	<builtin>	Operation	Operation	Operation	Operation
	@Name	<not set>	Transaction...	Transaction...	TransactionCo
	quid	<not set>	37A5E5C70...	37A5E5C70...	37A5E5C703
⚠	result	<not set>	Long	Integer	String
	concurrency	<not set>	Sequential	Sequential	Sequential
	opExportControl	<not set>	Public	Public	Public
	uid	<not set>	0	0	0
	Visual Basic				
	DefaultBody	<not set>	$entryco...	$entrycod...	$entrycode

For Help, press F1 | Unresolved Conflicts 2 | NUM

Accepting changes from conflicting contributors



Accepting changes from conflicting contributors requires you to make a judgment about which changes are integrated and which changes are dropped. In this exercise, the Point of Sale team wanted TransactionCount to return an Integer, but the Corporate team wanted it to return a String. You decide which change prevails using reasoning that supports other features or components in the project or system.

试一试:

Accepting changes from contributors

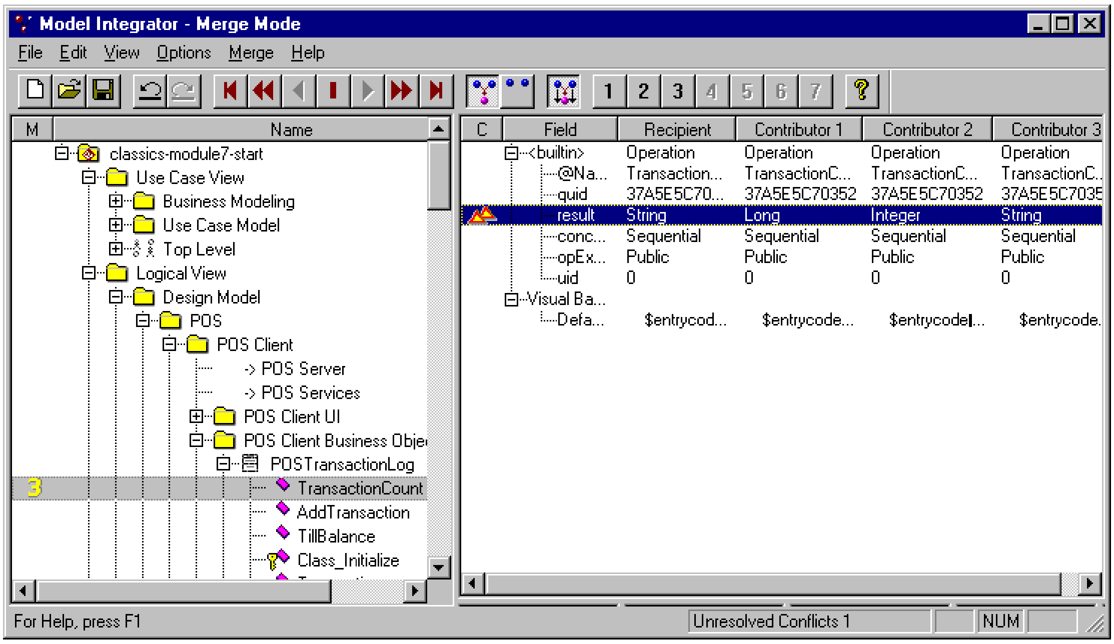
- Select the conflict and evaluate the choices.
- Click **Merge > Resolve Selected Using>Contributor n** (choose 1, 2, or 3).The Recipient column is updated with the value from the contributor file you chose.
- Before you exit the Model Integrator, click File > Save As and save your merged output into the baseline .cat file (the one that resides in the teamdevmodel folder).
- If there are other conflicts between your .cat files, you will not be able to save the results until you resolve those conflicts. For simplicity, click **Merge > Resolve All Conflicts Using Contributor n** (choose 1, 2, or 3). This resolves any remaining conflicts so that you can save your results.
- You can compare your updated baseline model with the model provided in **Module7/completed**.
- If you are using ClearCase, check in the baseline file so that the latest version is available

for further development.

检查一下结果：

The Corporate Team's change is accepted. You know this because:

- The value in the Recipient field now matches the value specified in Contributor 3, which is the corporate model.
- The 3 appears in the Merged column next to TransactionCount


Model Integrator - Merge Mode
File Edit View Options Merge Help
1 2 3 4 5 6 7
?
M
Name

- classics-module7-start
  - Use Case View
    - Business Modeling
    - Use Case Model
    - Top Level
  - Logical View
    - Design Model
      - POS
        - POS Client
          - -> POS Server
          - -> POS Services
          - POS Client UI
          - POS Client Business Obje
            - POSTransactionLog
              - 3 TransactionCount
              - AddTransaction
              - TillBalance
              - Class_Initialize

C	Field	Recipient	Contributor 1	Contributor 2	Contributor 3
	<builtin>	Operation	Operation	Operation	Operation
	@Na...	Transaction...	TransactionC...	TransactionC...	TransactionC..
	quid	37A5E5C70...	37A5E5C70352	37A5E5C70352	37A5E5C7035
	result	String	Long	Integer	String
	conc...	Sequential	Sequential	Sequential	Sequential
	opEx...	Public	Public	Public	Public
	uid	0	0	0	0
	Visual Ba...				
	Defa...	$entrycod...	$entrycode...	$entrycodel...	$entrycode.

Summary

In this module you learned about the Rose tools and features that support team development, including:

- Controlled units
- Virtual path maps ?
- Model Integrator

In addition, the guidelines included with the module described how to develop a strategy for effectively managing current projects and future artifact reuse.

What's next?

This is the last of the teaching modules in the tutorial. Module 8, the next module in the series, summarizes what the tutorial has covered and points you to additional sources of information.

附录 A

Vision Document - Classics Inc.

1. Introduction

1.1. Purpose of the VRD

The purpose of this document is to collect, analyze and define high-level business requirements, user needs and features of the Classics Inc. Order Processing and Fulfillment System (OPFS). This system is an application that would be available in all Classics Inc. retail stores. Classics Inc. is a rapidly growing chain of retail stores that sells items considered 揷lassics??classical music, classic movies, classic books, etc. After founding a single store selling classical music in 1993, Classics Inc. quickly expanded its regional business. It is now poised to open a nationwide chain of 127 stores by 2005.

Rapid growth has taken its effect on Classics Inc.抯 infrastructure. We quickly grew out of the packaged Point of Sale (POS) application we bought in 1993, and in 1996 took on the task of creating a custom POS application. Although fielded to all our stores, the project was a failure. When delivered, some 20% of the functionally originally planned had not yet been implemented or was still undergoing testing. This included the module that connected all the retail stores to the corporate order processing system.

The Classics Inc. management staff looked at what they would need to do to expand and meet their growth targets. Everyone quickly realized that not only was a new POS system needed 梩hey could not afford the same results as their previous attempts.

Classics Inc. realized that in order to ensure the success of the new system, they would need to adopt industry best practices, tools, and processes. They also realized that they would need to look at the entire business as a whole and not just a single application.

1.2. Product Overview

Classics Inc. wants to integrate its retail stores with the corporation 抯 order processing and fulfillment system. We envision a smaller scale supply-chain management system which ties all the stores, suppliers, and warehouses together. This includes:

?A point of sale (POS) system (telesales)

?An Order Processing system (store front)

?A Warehouse System (a legacy system this application will interact with but not modify)

?A Home Shopping system that will initiate Classics Inc.抯 presence on the World Wide Web (e-commerce)

2. User Description

2.1. User/Market Demographics

By 2005 we will have 127 Classics Inc. Stores. Average size stores typically have 2-3 sales people working at any given time. The OPFS will significantly improve productivity and efficiency of managing sales and inventory.

2.2 User Profiles

There are two primary users of this system: the store clerk and the store manager. The store clerk is anyone who is qualified to work in a retail store including teenagers and senior citizens. The store clerk may be a person with physical disabilities. This person is responsible for making sales and checking on orders for customers. This system should make it easier for the store clerk to enter orders, check orders and reconcile their cash drawer at the end of the day.

The manager of the system may be either the store owner or clerk. The manager of the store should not require any type of college or computer education. The manager may be a person with physical disabilities. The manager is responsible for the store employees, the stores inventory as well as reconciling all sales at the end of each day. This point of sale system should make the manager 抯 job easier by automating some of their tedious and error prone tasks.

2.3. User environment

The OPFS systems will be placed in all Classics Inc. retail stores. There will be multiple cash registers in each store, so the system should support simultaneous users.

This system will also include a Home Shopping e-commerce system to enable customers to order Classics Inc. products from their homes or work.

2.4. Key User Needs

Store managers are spending a significant amount of time in managing their sales and inventories manually. It is also difficult to get the status of current orders for customers. Classics Inc. needs a more accurate and efficient way to track sales by store and by product, and to maintain adequate levels of inventory in each store.

2.5. Alternatives and Competition

The alternative to developing this system in house would be to buy a commercially available product. The products on the market to day are not customizable or scalable enough to support the needs of Classics Inc.

3. Product Overview

3.1. Product Perspective

The Order Processing and Fulfillment System will be integrated with all Classics Inc. retail stores. This system will tie all the stores and suppliers together.

3.2. Product Position Statement

For Classics Inc. stores to better manage their sales and inventory, the Order Processing and Fulfillment System will make it more accurate and efficient for store Classics Inc. to manage sales and inventory. The OPFS will make it easy for information to be processed automatically when a sale is made rather than requiring manual data entry.

3.3. Summary of Capabilities

ORDER PROCESSING AND FULFILLMENT SYSTEM

Customer Benefit Supporting Features

Inventory is kept automatically up to date. The order processing system will be capable of maintaining the store 抯 inventory.

Sales clerks can easily track orders for customers. The point of sale system will be integrated with the order processing system.

If a store is out of a particular product a sales clerk can easily obtain products from another store. All retail stores will be connected together allowing the sales clerk to initiating orders to replenish stock when necessary.

Customers can purchase products themselves via the web, lowering cost of sales. A Home Shopping e-commerce system will be developed to initiate Classics Inc.抯 presence on the World Wide Web.

3.4. Assumptions and Dependencies

Classics POS will run on Windows 95, Windows 98, and Windows NT 4.0 and Windows 2000.

4. Product Features

The order processing and fulfillment system will include:

A point of sale (POS) system

An Order Processing system

A Warehouse System

A Home Shopping e-commerce system that will initiate Classics Inc. 扭 presence on the World Wide Web

4.1 Point of Sale System?

The point of sale system includes:

Cash register functions?

Maintaining the store 扭 inventory?

Supporting multiple cash registers per store?

Initiating orders to replenish stock when necessary?

4.2 Order Processing System?

The Order Processing system includes facilities to:

Provide for both automated and human-assisted order entry?

4.3 Home Shopping e-commerce system?

The home shopping e-commerce system will initiate Classics Inc. 扭 presence on the World Wide Web, including:

An online catalog for web visitors to browse?

A customer account maintenance facility for opening and maintaining accounts for individual consumers?

An order-entry capability supporting online sales and order fulfillment, interfacing with the Order Processing system?

An order status inquiry system that allows customers to check on existing orders and send messages to the Classics Inc. Customer Service staff?

5. Other Product Requirements

5.1. Applicable Standards

Classics Inc. applications must comply with all Microsoft?Windows?user interface standards.

5.2. System Requirements

The minimum system requirements for Classics POS are:

s 64 Mbytes memory

s 40 Mbytes disk space

s 166 Pentium Processor

s Windows 95, Windows NT 4.0 or 5.0, or Windows 98

5.3. Performance Requirements

The Classics POS system shall be able to support a minimum of 10 concurrent users for stores with multiple systems connected to a single database.

5.4. Environmental Requirements

The Classics POS software will be installed on PCs in Classics Inc. retail stores. Administration of the system will be performed by the store manager with potentially little to no computer experience.

There will be no users manual required for the Classics Inc. store manager. A manual should be

Docudatabase.

Documentation Requirements

7.1 User Manual

There will be no users manual required for the Classic Inc. store manager. A manual should be developed for the administrator, covering topics such as configuring tables and administering the database.

7.2. Online Help

The Classics POS system will include an extensive online help system to assist the user.

7.3. Installation Guides, Configuration, Read Me File

A document that includes installation instructions and configuration guidelines is important to a full solution offering. Also, a Read Me file will be included as a standard component. The Readme will include a "What's New With This Release Section," and a discussion of compatibility issues with earlier releases. We will also include documentation defining any known bugs and workarounds in the Read Me file.

7.4. Labeling and Packaging

The Classics POS application will ship on a CD which will be packaged in the back cover of the Administrators Manual.

Installation billboards will be developed and displayed during the installation program. The installation program will include the corporate license agreement.

A splash screen will be displayed upon execution of the Classics POS application. The Classics POS copyright information will appear on the splash screen.

附录 **B** 术语表

Abstraction

The creation of a view or model that suppresses unnecessary details to focus on a specific set of details of interest.

Accounting system

The accounting system is an "off the shelf" software application for managing the company's general ledger, accounts payable, accounts receivable, payroll, and for preparing the companies taxes.

Active class

A class representing a thread of control in the system.

Active Template Library (ATL)

A set of C++ classes that simplify the usage and creation of Component Object Model (COM) objects.**Activity**

A unit of work a worker may be asked to perform.

Activity diagram

An activity diagram illustrates the workflow of a use case.

Actor (instance)

Someone or something outside the system or business that interacts with the system or business.

Actor class

Defines a set of actor instances, in which each actor instance plays the same role in relation to the system or business.

Actor-generalization

An actor-generalization from an actor class (descendant) to another actor class (ancestor) indicates that the descendant inherits the role the ancestor can play in a use case.

Advertising agent

An organization that provides advertising services (i.e. newspaper, radio, television, web site).

Aggregation

An association that models a whole-part relationship between an aggregate (the whole) and its parts. analysis The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses on what to do, design focuses on how to do it.

Analysis and design

A process component in the Unified Process, whose purpose is to show how the system's use cases will be realized in implementation; (general) activities during which strategic and tactical decisions are made to meet the required functional and quality requirements of a system. For the result of analysis and design activities, see "Design model."

Analysis model

An object model describing the realization of use cases; serves as an abstraction of the design model.

Architectural view

A view of the system architecture from a given perspective; focuses primarily on structure, modularity, essential components, and the main control flows.

Architecture

The highest level concept of a system in its environment [IEEE]. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces.

Artifact

A piece of information that (1) is produced, modified, or used by a process, (2) defines an area of responsibility, and (3) is subject to version control. An artifact can be a model, a model element, or a document. A document can enclose other documents.

Association

A relationship that models a bi-directional semantic connection among instances.

Attribute

An attribute defined by a class represents a named property of the class or its objects. An attribute has a type that defines the type of its instances.

Baseline

A reviewed and approved release of artifacts that constitutes an agreed basis for further evolution or development and that can be changed only through a formal procedure, such as change management and configuration control.

Boundary class

A class used to model communication between the system's environments and its inner workings.

Build

An operational version of a system or part of a system that demonstrates a subset of the capabilities to be provided in the final product.

Business actor

A business actor represents a role played in relation to the business by someone or something in the business environment.

Business entity

A business entity represents a "thing" handled or used by business workers.

Business object model

The business object model describes the realization of business use cases.

Business use case

A business use case instance is a sequence of actions a business performs that yields an observable result of value to a particular business actor.

Business use-case model

A business use-case model describes the processes of a business and the interactions with external parties like customers and partners.

Business worker

A business worker represents a role or set of roles in the business. A business worker interacts with other workers and manipulates business entities while participating in business use-case realizations.

Class

A class is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.

Class diagram

A class diagram shows a collection of declarative (static) model elements, such as classes, packages, their contents, and relationships.

ClearCase

ClearCase is part of an integrated change management solution from Rational that supports both Windows and Unix platforms. The ClearCase product family provides a comprehensive solution for software configuration management (ClearCase), distributed development (ClearCase MultiSite), and defect tracking (ClearQuest).

Collaboration

Interaction among objects.

Collaboration diagram

A collaboration diagram describes a pattern of interaction among objects; it shows the objects participating in the interaction by their links to each other and the messages they send to each other.

Component Object Model (COM)

A binary standard for building and using software components.

Communicates-association

An association between an actor class and a use case class, indicating that their instances interact. The direction of the association indicates the initiator of the communication.

Component

A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

Component-based development (CBD)

The creation and deployment of software-intensive systems assembled from components as well as the development and harvesting of such components.

Component subsystem

A stereotyped subsystem (i.e. «component») representing the logical abstraction in design of a component. It realizes one or more interfaces, and may be dependent on one or more interfaces. It may enclose zero or more classes, packages or other component subsystems, none of which are visible externally (only interfaces are visible). It may also enclose zero or more diagrams which illustrate internal behavior (e.g. state, sequence or collaboration diagrams).

Concrete

An entity in a configuration that satisfies an end-use function and can be uniquely identified at a given reference point (ISO)

Controlled unit

A file that contains part of a Rose model. Controlled units are a Rose feature that enables you to divide a model into several files that teams of developers can work on in isolation.

Configuration

(1) general: The arrangement of a system or network as defined by the nature, number, and chief characteristics of its functional units; applies to both hardware or software configuration.

(2) The requirements, design, and implementation that define a particular version of a system or system component.

Customer

Any individual or organization that purchases classical music items from Classics, Inc.

Defect

A product anomaly. Examples include such things as omissions and imperfections found during early lifecycle phases and symptoms of faults contained in software sufficiently mature for test or operation. A defect can be any kind of issue you want tracked and resolved.

Deliverable

An output from a process that has a value, material or otherwise, to a customer.

Deployment view

An architectural view that describes one or several system configurations; the mapping of software components (tasks, modules) to the computing nodes in these configurations.

Design

The part of the software development process whose primary purpose is to decide how the system will be implemented. During design, strategic and tactical decisions are made to meet the required functional and quality requirements of a system. See Analysis.

Design model

An object model describing the realization of use cases; serves as an abstraction of the implementation model and its source code.

Design package

A collection of classes, relationships, use-case realizations, diagrams, and other packages; it is used to structure the design model by dividing it into smaller parts.

Design subsystem

A design package that contains a collection of design packages and classes, and used to structure the design model by dividing it into smaller parts. See Design Package.

Developer

A person responsible for developing the required functionality in accordance with project-adopted standards and procedures. This can include performing activities in any of the requirements, analysis & design, implementation, and test workflows.

Device

A type of node which provides supporting capabilities to a processor. Although it may be capable of running embedded programs (device drivers), it cannot execute general-purpose applications, but instead exists only to serve a processor running general-purpose applications.

Diagram

A graphical depiction of all or part of a model.

Document

A document is a collection of information that is intended to be represented on paper, or in a medium using a paper metaphor. The paper metaphor includes the concept of pages, and it has either an implicit or explicit sequence of contents. The information is in text or two-dimensional pictures. Examples of paper metaphors are word processor documents, spreadsheets, schedules, Gantt charts, web pages, or overhead slide presentations.

Domain

An area of knowledge or activity characterized by a family of related systems.

Ecommerce system

See: Online sales

Employee schedule

See: Work schedule.

Evolution

The life of the software after its initial development cycle; any subsequent cycle, during which the product evolves.

Extend-relationship

An extend-relationship from a use-case class A to a use-case class B indicates that an instance of B may include (subject to specific conditions specified in the extension) the behavior specified by A. Behavior specified by several extenders of a single target use case can occur within a single use-case instance.

Feature

An externally observable service provided by the system which directly fulfills a stakeholder need.

Framework

A micro-architecture that provides an incomplete template for applications within a specific domain.

Generalization

A generalization is a relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element can be used where the more general element is allowed.

Implementation model

The implementation model is a collection of components, and the implementation subsystems that contain them. implementation subsystem. A collection of components and other implementation subsystems, and is used to structure the implementation model by dividing it into smaller parts.

Implementation view

An architectural view that describes the organization of the static software elements (code, data, and other accompanying artifacts) on the development environment, in terms of both packaging, layering, and configuration management (ownership, release strategy, and so on). In the Unified Process it is a view on the implementation model.

Import-dependency

A stereotyped dependency in the design whose source is a design package, and whose target is a different design package. The import dependency causes the public contents of the target package to be referenceable in the source package.

Include-relationship

An include-relationship is a relationship from a base use case to an inclusion use case, specifying how the behavior defined for the inclusion use case is explicitly inserted into the behavior defined for the base use case.

Increment

The difference (delta) between two releases at the end of subsequent iterations.

Incremental

Qualifies an iterative development strategy in which the system is built by adding more and more functionality at each iteration.

Inheritance

The mechanism that makes generalization possible; a mechanism for creating full class descriptions out of individual class segments

Input

An artifact used by a process. See “Static artifact.”

Instance

An individual entity satisfying the description of a class or type.

Integration

The software development activity in which separate software components are combined into an executable whole. integration build plan Defines the order in which components are to be implemented and integrated in a specific iteration.

Interaction

Important sequence of events between objects.

Interface

A collection of operations that are used to specify a service of a class or a component iteration. A distinct sequence of activities with a base-lined plan and valuation criteria resulting in a release (internal or external).

Inventory

The generic term for the collection of products/items that Classics, Inc. sells to its customers, that is contained in either a retail store or the corporate warehouse.

Inventory management system

The inventory management system manages the inventory at the Classic's Inc. corporate offices. This includes telephone sales, on-line sales and inventory for retail stores.

Inventory request

A request from a retail store for replacement inventory. Store managers make a request for items that have sold.

Item

The generic term for any product or purchasable item that Classics, Inc. sells to its customers.

Layer

A specific way of grouping packages in a model at the same level of abstraction.

Lifeline

Each object appearing on a sequence diagram contains a dashed, vertical line, called a lifeline, which represents the location of an object at a particular point in time.

Logical view

An architectural view that describes the main classes in the design of the system: major business-related classes, and the classes that define key behavioral and structural mechanisms (persistency, communications, fault-tolerance, user interface). In the Unified Process, the logical view is a view of the design model.

Management

A core supporting workflow in the software-engineering process, whose purpose is to plan and manage the development project.

Method

(1) A regular and systematic way of accomplishing something; the detailed, logically ordered plans or procedures followed to accomplish a task or attain a goal. (2) UML 1.1: The implementation of an operation, the algorithm or procedure that effects the results of an operation.

Milestone

The point at which an iteration formally ends; corresponds to a release point.

Model

A semantically closed abstraction of a system. In the Unified Process, a complete description of a system from perspective ('complete' meaning you don't need any additional information to understand the system from that perspective); a set of model elements. Two models cannot overlap.

Model element

An element that is an abstraction drawn from the system being modeled.

Namespace

The location or area for one model element name in an entire model.

Node

A run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. Run-time objects and components may reside on nodes.

Object

An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations and methods. An object is an instance of a class.

Object model

An abstraction of a system's implementation.

Online customer

Any customer that purchases items by using the on line e-Commerce system.

Online sales

The internet store front for Classics. Internet users can browse Classics catalog and place orders. The only form of payment accepted is credit cards. All orders are filled and shipped from the corporate office.

Operation

A service that can be requested from an object to effect behavior.

Order processing and fulfillment system (OPFS)

A system that Classics Inc. clerks can use to order and ship products to customers. Customers walk in the store, or place orders over the phone or via the web. The only form of payment acceptable via telesales or the web is credit card.

Output

Any artifact that is the result of a process step. See Deliverable.

Package

A grouping of modeling elements.

Phase

The time between two major project milestones, during which a well-defined set of objectives is met, artifacts are completed, and decisions are made to move or not move into the next phase.

Point of sale (POS)

The point of sale system is a combination of hardware and software that performs the role of a checkout register. The POS device is capable of scanning bar codes and credit cards. POS systems are used in all Classic Inc. retail stores.

Postcondition

A textual description defining a constraint on the system when a use case has terminated.

precondition A textual description defining a constraint on the system when a use case may start.

process (1) A thread of control that can logically execute concurrently with other processes, specifically an operating system process. See also thread. (2) A set of partially ordered steps intended to reach a goal; in software engineering the goal is to build a software product or to enhance an existing one; in process engineering, the goal is to develop or enhance a process model; corresponds to a business use case in business engineering.

Process view

An architectural view that describes the concurrent aspect of the system: tasks (processes) and their interactions.

Processor

A type of node which can run one or more processes. Generally this requires a computational capability, memory, input-output devices, etc. See also node, process, and device.

Product

(1) A generic term for any item that can be bought at a Classics, Inc. retail store. Products include all items except services.(2) Software that is the result of development, and some of the

associated artifacts (documentation, release medium, training).

Product requirements document (PRD)

A high level description of the product (system), its intended use, and the set of features it provides.

Project manager

The worker with overall responsibility for the project. The Project Manager needs to ensure tasks are scheduled, allocated and completed in accordance with project schedules, budgets and quality requirements.

Prototype

A release that is not necessarily subject to change management and configuration control.

Purchasing manager

Manages the purchase of items from Classic's vendors

Quality assurance (QA)

QA reports to the Project Manager and are responsible for ensuring that project standards are correctly and verifiably followed by all project staff.

Rationale

A statement, or explanation of the reasons for a choice

Release

A subset of the end product that is the object of evaluation at a major milestone. See "Prototype."

Report

An automatically generated description, describing one or several artifacts. A report is not an artifact in itself. A report is in most cases a transitory product of the development process, and a vehicle to communicate certain aspects of the evolving system; it is a snapshot description of artifacts that are not documents themselves.

Requirement

A requirement describes a condition or capability of a system; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document.

Requirement attribute

Information associated with a particular requirement providing a link between the requirement and other project elements - priorities, schedules, status, design elements, resources, costs, hazards.

Requirements

A core process workflow in the software-engineering process, whose purpose is to define what the system should do. The most significant activity is to develop a use-case model.

Requirements capture

A process component in the software-engineering process, whose purpose is to define what the system should do. The most significant activity is to develop a use-case model.

Requirements management

A systematic approach to eliciting, organizing and documenting the requirements of the system, and establishing and maintaining agreement between the customer and the project team on the changing requirements of the system.

Retail price

The unit cost of an item that Classics, Inc. charges its customers.

Retail store customer

Any customer that purchases items from a retail store.

Reuse

Further use or repeated use of an artifact

Risk

An ongoing or upcoming concern that has a significant probability of adversely affecting the success of major milestones.

Round-trip engineering

The process of alternating between the model and the code.

SCC

SCC (Source Code Control) is the Microsoft standard API for version control systems.

Sales clerk

A Classics, Inc. employee responsible for processing sales in retail stores. The employee is minimally trained to operate the POS system.

Scenario

A described use-case instance, a subset of a use case.

Schedule

See: Work schedule.

Sequence diagram

A diagram that describes a pattern of interaction among objects, arranged in a chronological order; it shows the objects participating in the interaction by their "lifelines" and the messages that they send to each other.

Shipper

The shipping company delivers items from the Classics Inc. warehouse and delivers them to the customers. The Shipping company can be the postal service, next day air company, or a preferred ground transportation company.

Software architecture

Software architecture encompasses: the significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements, the composition of the structural and behavioral elements into progressively larger subsystems, the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition.

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and tradeoffs, and aesthetic concerns.

Software requirement

A specification of an externally observable behavior of the system, (e.g., inputs to the system, outputs from the system, functions of the system, attributes of the system, or attributes of the system environment).

Software requirements specifications (SRS)

A set of requirements which completely defines the external behavior of the system to be built. (sometimes called a functional specification)

Stakeholder

An individual affected by the outcome of the system.

State diagram (or chart)

A state diagram shows a state machine, that is, a behavior that specifies the sequences of states that an object goes through during its life in response to events, together with its responses and actions.

State machine

Behavior that specifies the valid sequences of activities that an object or interaction goes through during its life. One state machine can hold multiple activity diagrams and statechart diagrams.

Static artifact

An artifact that is used, but not changed, by a process.

Stereotype

A meta-classification of an element. Stereotypes have semantic implications which can be specified for every specific stereotype value.

Store

A physical location where retail customers can come in and browse inventory. Customers can select items and purchase them at the store (through a checkout counter). Acceptable payment options include cash and credit card.

Store owner

An individual that owns one or more Classics, Inc. retail stores. The owner is responsible for all the aspects of store management. Store owners are free to manage and operate their stores any way they like, but must maintain compliance with the Classics, Inc. franchise agreement.

Stub

A component containing functionality for testing purposes. A stub is either a pure "dummy", just returning some predefined values, or it is "simulating" a more complex behavior.

Subsystem

A model element which has the semantics of a package, such that it can contain other model elements, and a class, such that it has behavior. (The behavior of the subsystem is provided by classes or other subsystems it contains.) A subsystem realizes one or more interfaces, which define the behavior it can perform.

System

As an instance, an executable configuration of a software application or software application family; the execution is done on a hardware platform. As a class, a particular software application or software application family that can be configured and installed on a hardware platform. In a general sense, an arbitrary system instance.

Task

See operating system process, process and thread.

Telephone customer

Any customer that purchases items by placing an order over the phone

Telesales

Sales outlet that is based on telephone call ins. Customer call Classics, toll free number to place orders for products. The only form of payment accepted is credit cards. Orders are shipped from the corporate office.

Telesales order

An order created through the telesales system. A customer called up, placed and paid for a selection of products.

Telesales system

The Telesales system is used by telesales workers to process telephone sales. All sales capture by this system are automatically sent to the inventory management system.

Test

A core process workflow in the software-engineering process whose purpose is to integrate and test the system.

Test case

A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test procedure

A test procedure is a set of detailed instructions for the set-up, execution, and evaluation of results for a given test case.

Thread

An independent computation executing within an execution environment and address space defined by an enclosing operating system process. Also sometimes called a 'lightweight process.’

Till

The cash drawer that is part of the point of sale device.

Time card

A document indicating the hours worked by an employee. Time cards must be verified by each employee's manager before they can be submitted to the payroll system.

Traceability

The ability to trace a project element to other related project elements.

Type

Description of a set of entities which share common characteristics, relations, attributes, and semantics.

Type library

A type library contains a description of a COM (component object model) component as viewed from the outside. The description includes the coclasses, interface items, dispinterfaces, properties (called attributes in UML), methods (called operations in UML), data types, etc. of the type library. The type library of a COM component can be imported into a Rational Rose model.

Use case (class)

A use case defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor. A use-case class contains all main, alternate flows of events related to producing the 'observable result of value'. Technically, a use-case is a class whose instances are scenarios.

Use-case diagram

A use-case diagram shows actors, use cases, use-case packages, and their relationships.

Use-case instance

A sequence of actions performed by a system that yields an observable result of value to a particular actor.

Use-case model

A model of what the system is supposed to do and the system environment.

Use-case package

A use-case package is a collection of use cases, actors, relationships, diagrams, and other packages; it is used to structure the use-case model by dividing it into smaller parts.

Use-case realization

A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects.

Use-case view

An architectural view that describes how critical use cases are performed in the system, focusing mostly on architecturally significant components (objects, tasks, nodes). In the Unified Process, it is a view of the use-case model.

Vendor

Provides wholesale items for Classics, Inc. for resale.

Version

A variant of some artifact; later versions of an artifact typically expand on earlier versions.

Virtual path maps

A way to map a physical file location to a logical path. You can use this feature to make Rose models more portable.

View

A simplified description (an abstraction) of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective. See also architectural view. vision The user's or customer's view of the product to be developed.

Vision document

A vision document outlines a user’s or customer’s view of the product to be produced.

Wholesale price

The unit cost of an item that Classics, Inc. must pay it vendors.

Worker

Role figure to be played by individual members in the business.

Workflow

The sequence of activities performed in a business that produces a result of observable value to an individual actor of the business.

Work schedule

The times an employee must show up at the office to work. Schedules are created by managers for each store on a periodic basis (typically each week).