



Introduction to IBM Rational
Rhapsody

QQ001
ERC 1.0
Student Workbook

IBM Corporation
Rational software

U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this documentation in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*Intellectual Property Dept.
IBM Corporation
20 Maguire Road
Lexington, Massachusetts 02421-3112
U.S.A.*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "[Copyright and trademark information](#)" at www.ibm.com/legal/copytrade.html

- Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.
- IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce
- Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
[Intel trademark information](#)
- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
[Microsoft trademark guidelines](#)
- ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
- Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.



Lab 1: Starting a new IBM Rational Rhapsody project

Objectives

After completing this lab, you will be able to:

- ▶ Start a new Rational Rhapsody project that is a vending machine
- ▶ Create an Object Model (Requirement) diagram
- ▶ Add nested requirements to the diagram
- ▶ Resize elements on the diagram

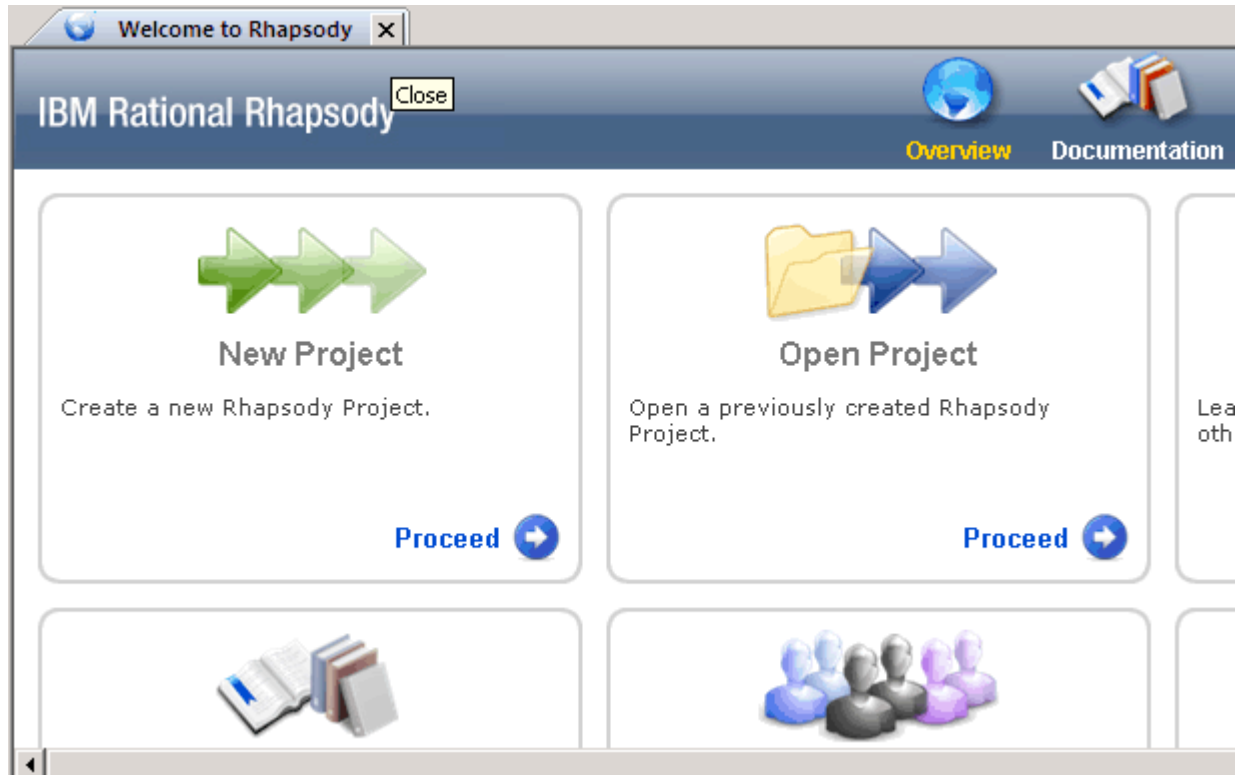
Scenario

A new `VendingMachine` project is created in Rational Rhapsody. Four *Money Management* requirements are added. Two separate Object Model (*requirement diagrams*) are created to show common diagram editing techniques. In a real development effort, you would take a less academic approach, and probably use just one diagram to illustrate the requirements hierarchy.

Task 1: Start a new project

In this task, you will start Rational Rhapsody and begin a new project.


1. Create a work directory where you can save Rational Rhapsody project files. For example, `P:\work`.
2. Launch Rational Rhapsody.
 - a. Select **Start > All Programs > IBM Rational > IBM Rational Rhapsody for C++ 8.0**
 - b. Close the **Welcome to Rhapsody** screen.



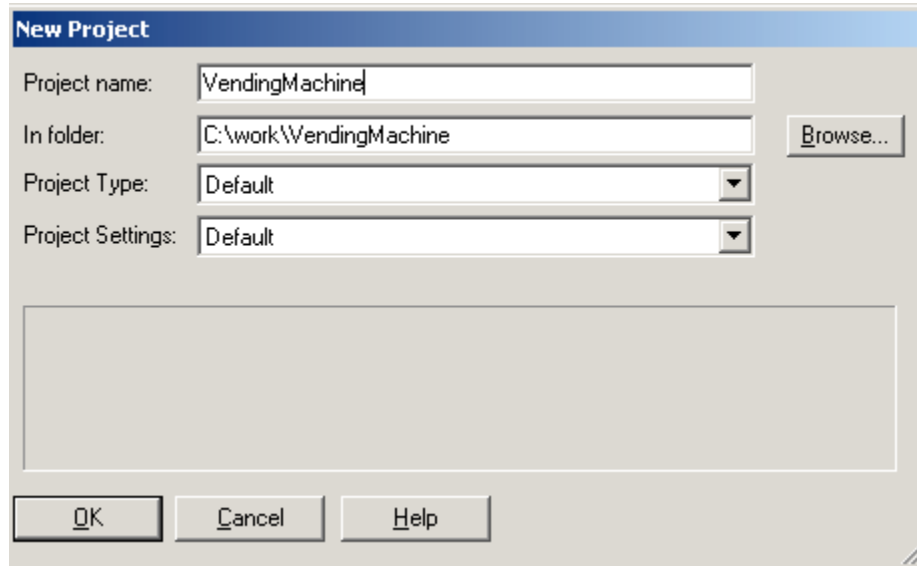
4. Start a new Rational Rhapsody project.

a. Select **File > New**

Or

b. Select the blank paper  icon

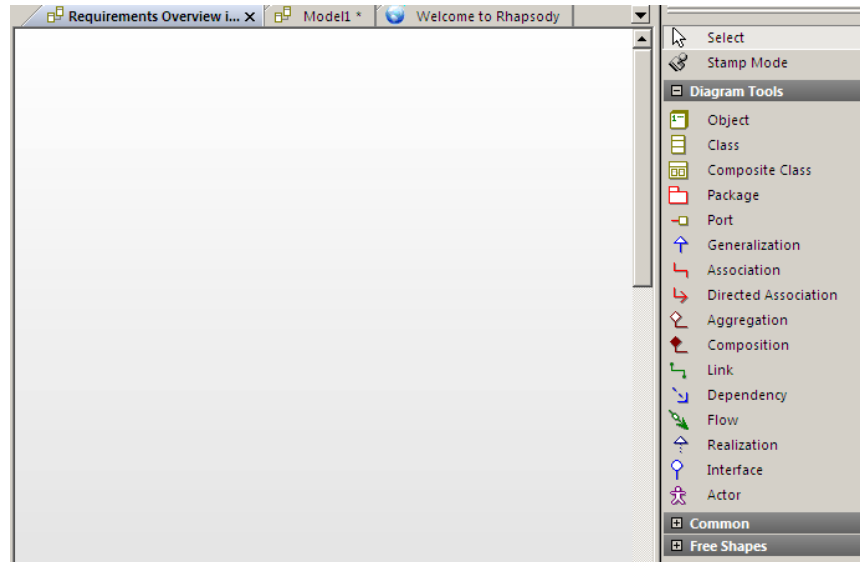
5. Name the project `VendingMachine` and select a writable directory, such as `P:\Work`. Keep **Default** as the Project Type and Project Settings.



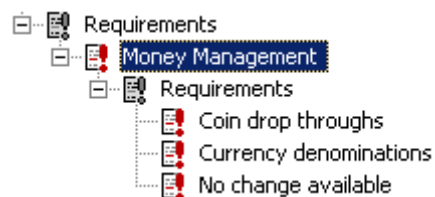
6. Click **OK** and **Yes** when prompted with a question about creating the new project directory. Rational Rhapsody creates an initial Object Model diagram.
7. Expand the **Packages** category. Change the name of the **Default** package to `Analysis`.
 - a. Double-click the package name and change the name in the **Features > Name** field, or right-click the name in the browser, right-click again and edit the name in the browser.
8. Right-click the **Packages** category, and click **Add new package**. Name the new package `Design`.
9. To add a description to the **Analysis** package, right-click the package name and select **Features > Description**. Add the following description for the **Analysis** package: `Systems level model including requirements and use case analysis`.

Task 2: Create a diagram


1. Create a subpackage under the **Analysis** package by right-clicking **Analysis** and selecting **Add new package**. Name it **Requirements**.
2. Right-click the **Requirements** package, select **Add New > Diagrams > Object Model Diagram** (Requirements diagram). Name it `Requirements Overview`.
3. Go to **View > Toolbars** and verify the **Drawing** toolbar is checked. Confirm that the **Drawing** toolbar appears to the right of the Object Model diagram.

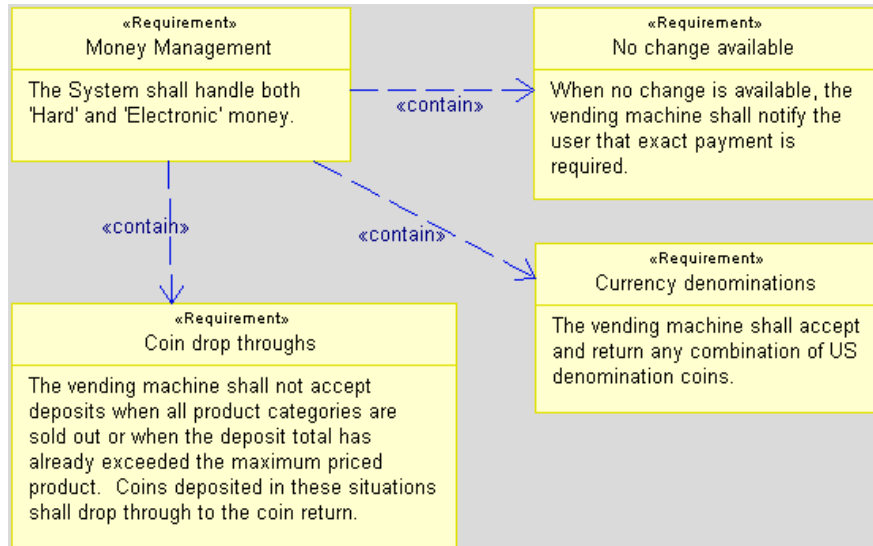


4. Use the Common section of the Drawing toolbar to find the requirement icon and add a requirement called **Money Management** to the Requirements Overview diagram. Right-click the **Money Management** requirement to get the Features. Add the following specification to the **Features > General>Specification**: The system shall handle 'Hard' and 'Electronic' money.
5. Right-click the Money Management requirement and select **Notation Style**. Confirm that **Box Style** is selected.
6. Add three new requirements under the **Money Management** requirement by right-clicking on **Money Management** in the browser and selecting **Add New>Requirement**. Name them as follows:
 - a. Coin drop throughs
 - b. Currency denominations
 - c. No change available
 - d. Confirm the new requirements are nested under the Money Management requirement as follows:

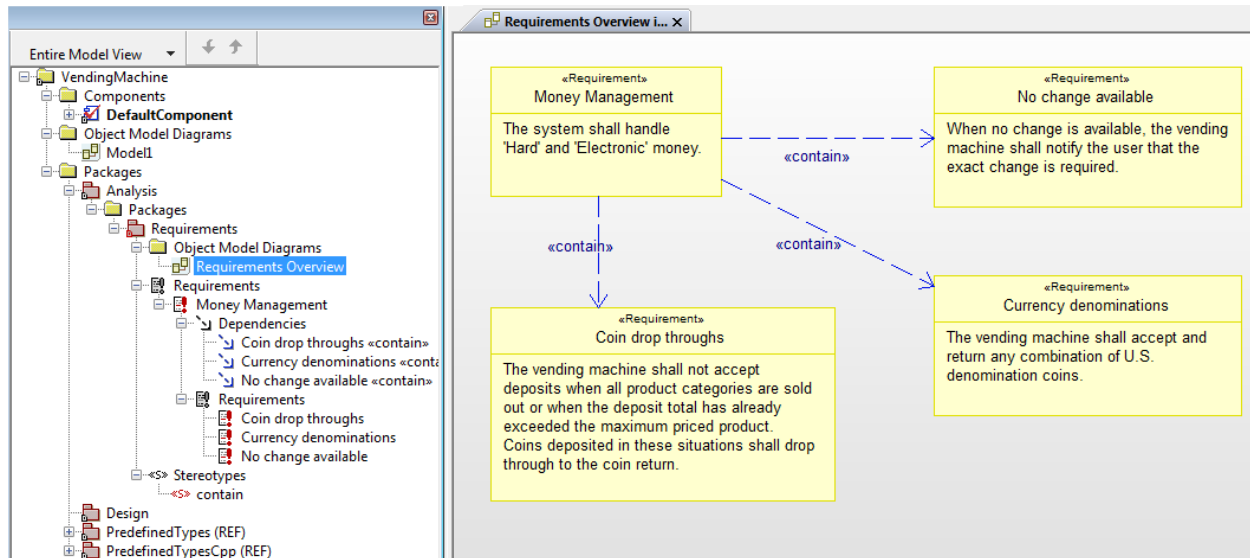


7. In the **Features > General** tab for each of the new requirements, enter the following text into the Specification field for the respective requirements (to match the order in step 6).
 - a. The vending machine shall not accept deposits when all product categories are sold out or when the deposit total has already exceeded the maximum priced product. Coins deposited in these situations shall drop through to the coin return.
 - b. The vending machine shall accept and return any combination of U.S. denomination coins.
 - c. When no change is available, the vending machine shall notify the user that the exact change is required.

- d. Drag the requirements from the browser onto the **Requirements Overview** diagram.
- 8. Add dependencies from the **Money Management** requirement to each of the other three requirements to visualize requirement containment.
 - a. Select the dependency icon  on the drawing toolbar and draw a dependency from the **Money Management** requirement to the **No change available** requirement. Draw another dependency from **Money Management** to **Currency denominations** and from **Money Management** to **Coin drop throughs**.




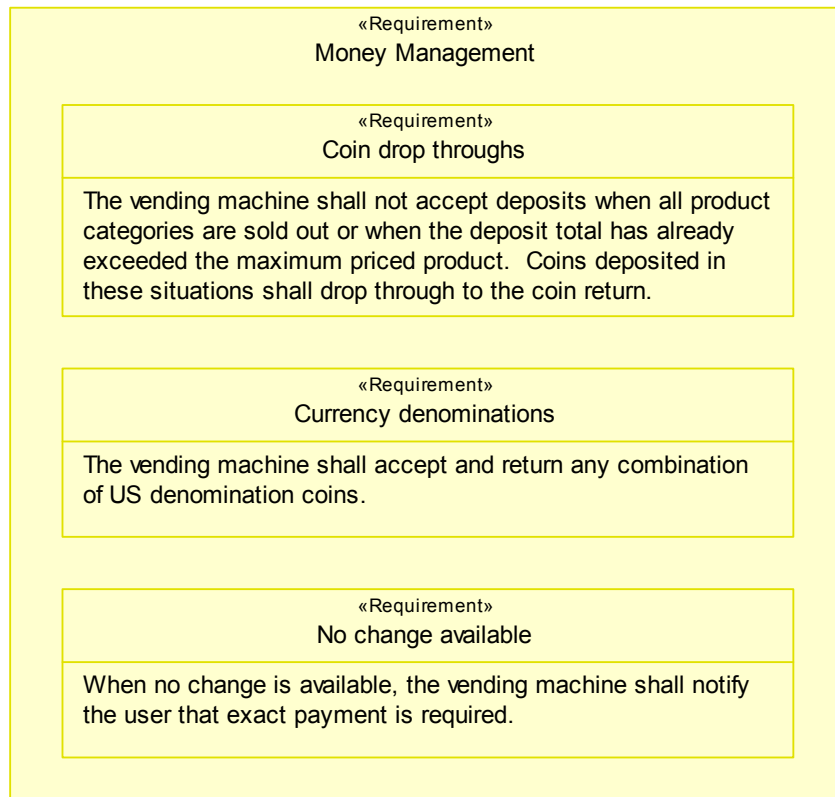
- b. Right-click each new dependency. In the **Features>General>Stereotype** field, select the radio button for **contain in Requirements**. If **«contain»** is not available in the selections, select the **New** radio button and name the new stereotype **contain**.



9. You can use the Layout toolbar to align the requirements on the diagram. You need to select two requirements on the diagram to activate the Layout



10. Create a new Object Model diagram called **Structured Requirements** in the Requirements package. This diagram is used to practice the editing of graphically nested elements.
11. Drag the **Money Management** requirement from the browser onto the new diagram and use the Specification/Structured view  icon on the Zoom toolbar to switch its view to **Structured View**. If the icon is greyed out, click an element in the diagram.
12. Enlarge the **Money Management** requirement by dragging a corner anchor.
13. Drag the three nested requirements from the browser into the diagram's Money Management requirement.



14. If the contain dependencies appear on your diagram, right-click each dependency and select **Remove from View**.
15. Select the border of the **Money Management** requirement and resize it. Note that the contents expand as the border expands.
16. To expand the container without resizing its contents, hold the Alt key before dragging. Alternatively, use **Edit > Resize Without Contained**.
17. Save the model.

Remember to use the Layout toolbar and **Resize without Contained** throughout the upcoming lab exercises.



Lab 2: Set properties and create diagrams

Objectives

After completing this lab, you will be able to:

- ▶ View and set a property for a project
- ▶ Create a use case diagram
- ▶ Create a black-box activity diagram
- ▶ Create a use case-level sequence diagram
- ▶ Allocate activities to subsystems

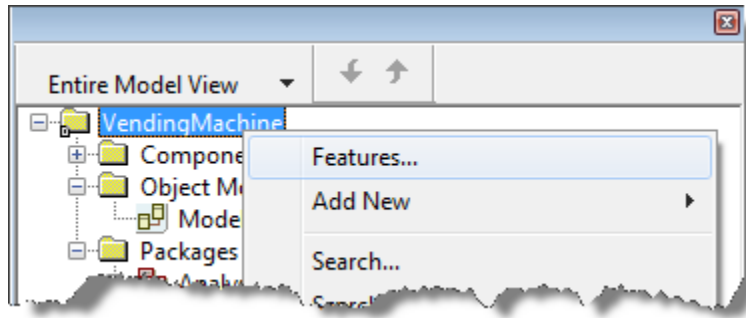
Scenario

You have started a new project and now you will build on it by adding more diagrams. A system-level use case diagram is created in this lab. The requirements will be realized by the design, but first are linked to use cases. Each use case describes a primary function of the vending machine system and is traced to one or more requirements to ensure all requirements are satisfied. The functional behavior of the black-box system is elaborated on using an activity diagram and sequence diagram to describe the services to be provided by the system. Three subsystems are introduced, and the activity diagram is updated to show the allocation of the services across the subsystems.

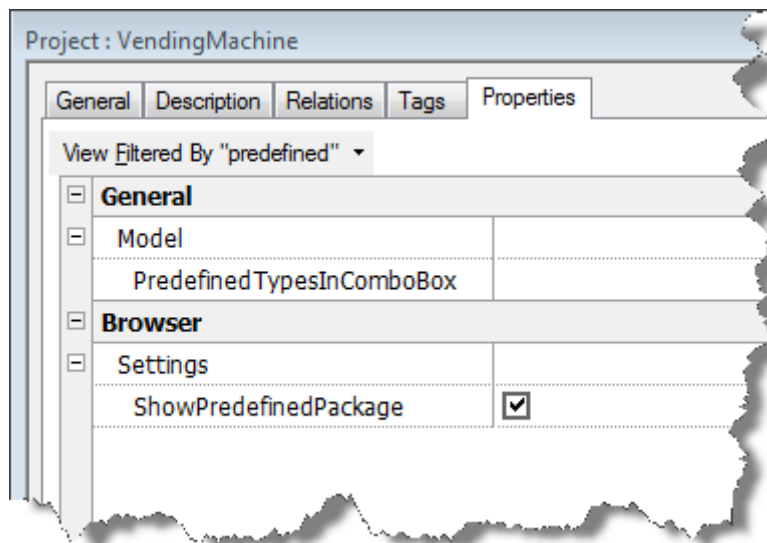
Task 1: Set a property

In this task, you will bring up the properties for the Rational Rhapsody project and make a modification.

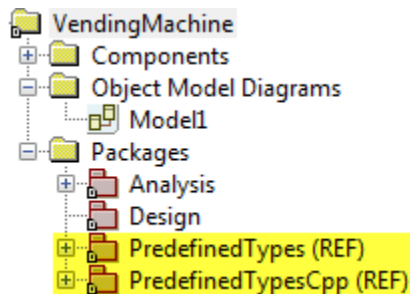
1. Your Vending Machine project is open. Right-click the project and select **Features**.



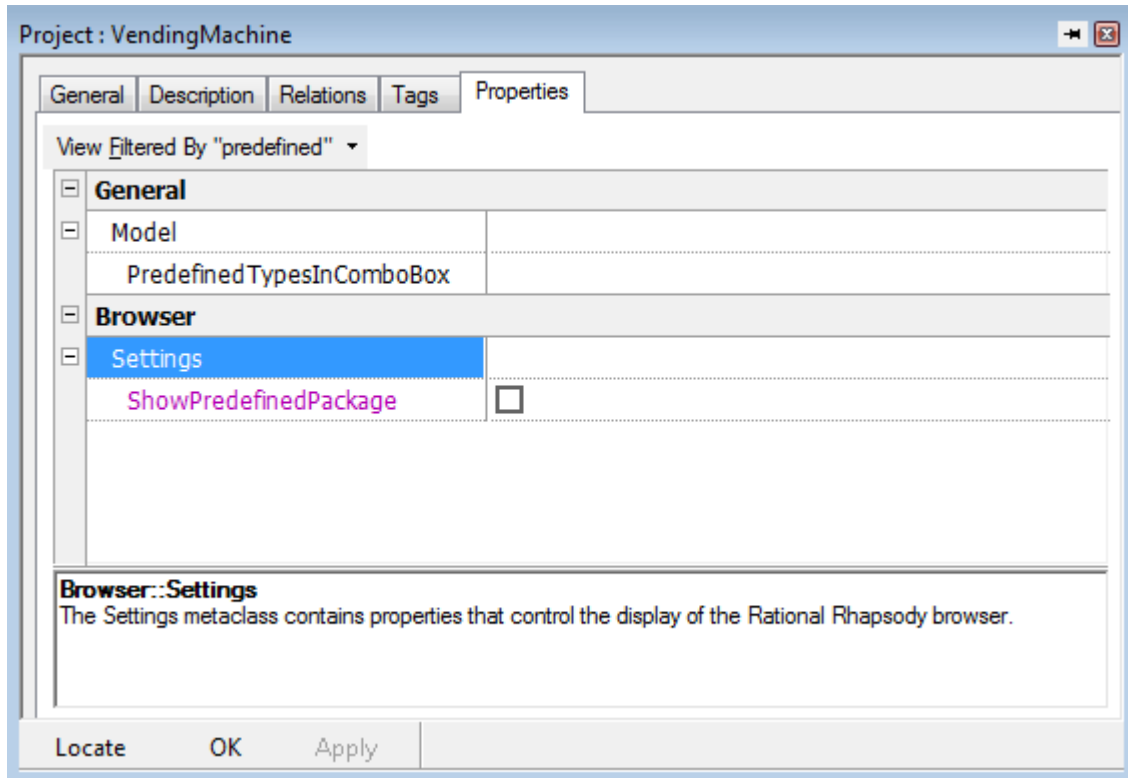
2. In the Features window, select **Properties**.
3. The property view filter makes it easy to locate properties. Select **View > Filter** and type `predefined` in the Filter text box. What do you see?



Note that the properties with `predefined` in the name appear as shown above. And currently the predefined types appear in the browser.



4. You can set a property to remove the predefined types packages from the browser view. Select the **Browser::Settings:ShowPredefinedPackage** property so the check mark is cleared. Click **OK**, and observe that the predefined types packages are no longer shown.
5. Click elsewhere in the Properties tab to confirm that the modified property is highlighted, and observe that the predefined types packages are no longer shown.

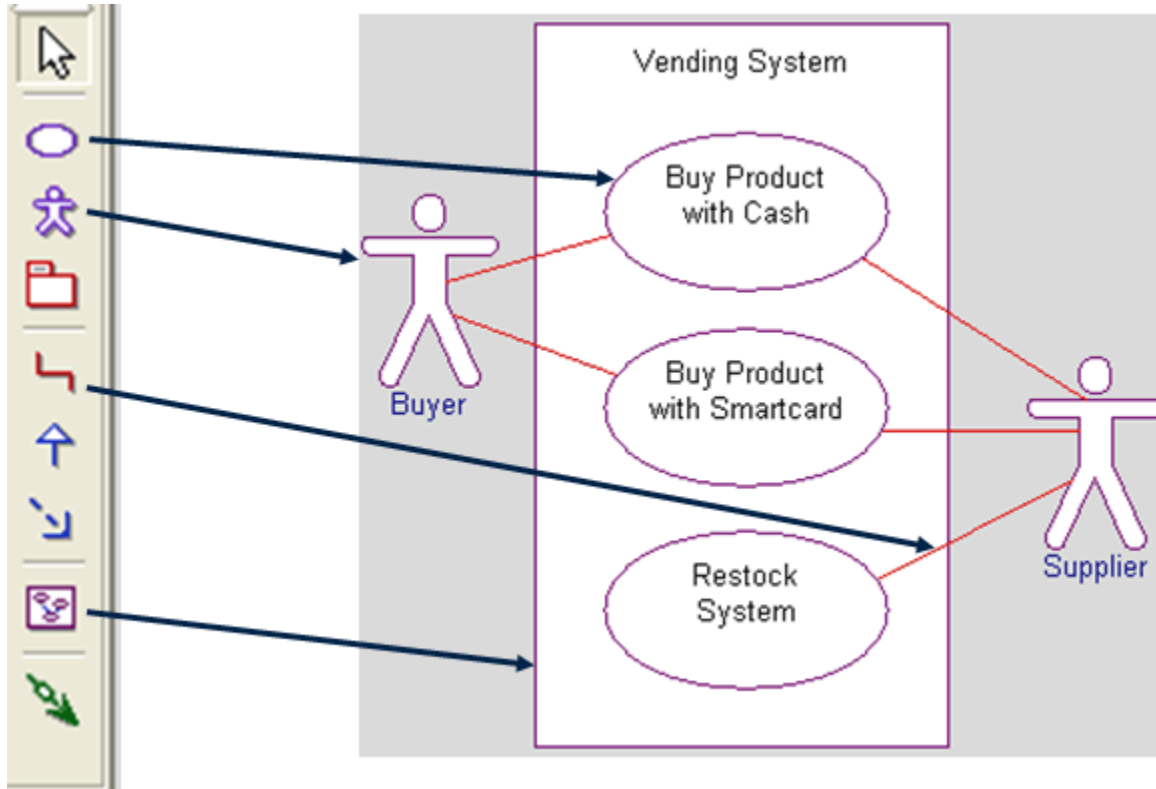


6. Select **View Locally Overridden** on the top left of the Properties dialog, to see all the properties that have been overridden at the project level.

Task 2: Create a use case diagram

In this task you create a use case diagram and add actors and use cases to it.

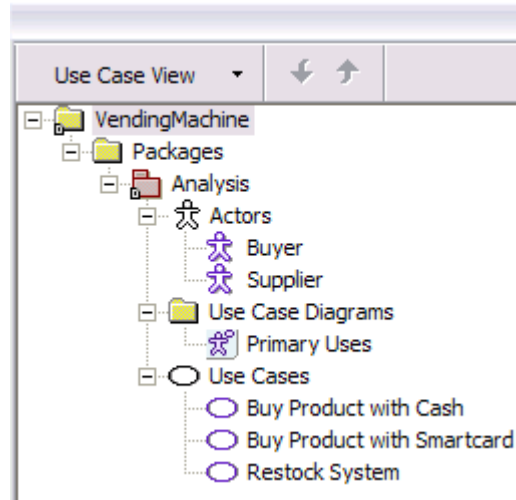
1. Add a new use case diagram called **Primary Uses** in the Analysis package.
 - a. Right-click the Analysis package and select **Add New > Diagrams > Use Case Diagram**.
2. Add actors, use cases, and associations to the diagram as shown here:



- a. Add two actors using the actor icon on the Drawing toolbar: name one **Buyer** and the other **Supplier**.
- b. Add three new use cases: **Buy Product with Cash**, **Buy Product with Smartcard**, **Restock System**.
- c. Use the Layout toolbar to align and space the use cases.
- d. Draw a boundary box around the use cases naming it **Vending System**. Here, the boundary box sets the context of the use cases as being the entire vending machine system.

Note - Later in development when the subsystems have been determined, other use case diagrams would be created for each subsystem. At the subsystem level, the boundary box would have the name of a subsystem and the use cases would be in the context of that subsystem, and the actors would be **Buyer**, **Supplier**, or another subsystem.

- e. Draw an association from the **Restock System** use case to the **Supplier**.
 - f. Draw an association from the **Buy Product with Smartcard** use case to the **Supplier** and to the **Buyer**.
 - g. Draw an association from the **Buy Product with Cash** use case to the **Buyer** and the **Supplier**.
3. Select the **Use Case View** in the Rational Rhapsody browser to view the newly created elements.



4. Switch back to the Entire Model view.
5. Right-click the project at the project level and select **Format**. Select UseCase as the Meta-class, select Format selected meta-class. Change the fill color of the use case to yellow. This will change all use cases in the project.
6. Change the format of the actors at the project level to a color of your choice.
7. Except for the Buy Product with Cash use case, add a description of your choice for the use cases.

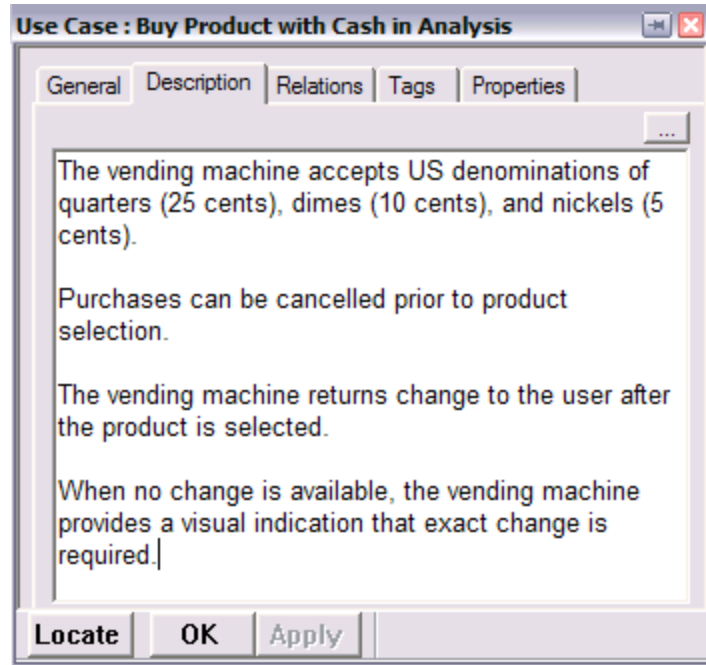
a. For the **Buy Product with Cash** use case, add the following description:

The vending machine accepts US denominations of quarters (25 cents), dimes (10 cents), and nickels (5 cents).

Purchases can be cancelled prior to product selection.

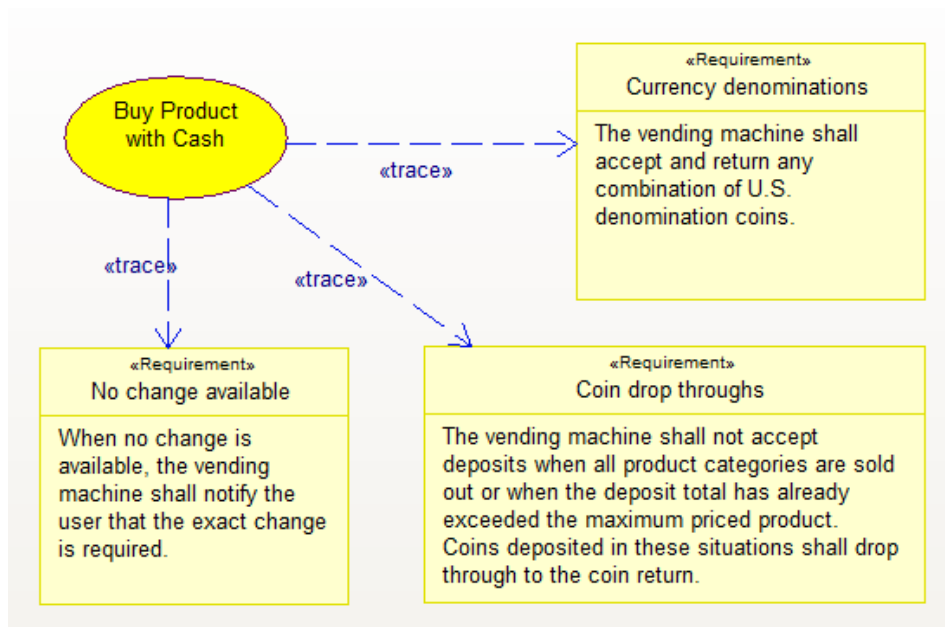
The vending machine returns change to the user after the product is selected.

When no change is available, the vending machine provides a visual indication that exact change is required.



You created a few requirements in the last lab. In a real development effort, you would have many more requirements. Each of the functional requirements should be satisfied by one or more use cases. If you were using SysML, you would use a <<satisfy>> dependency to show the linkage from a use case to a requirement. With UML, a <<trace>> dependency is used.

The requirements are organized into functional groups, with each functional group aligned with a use case. For example, the three requirements are satisfied by the **Buy Product with Cash** use case. This traceability linkage can be tracked and shown in Rational Rhapsody using a matrix, in the browser, or on a diagram as shown below.

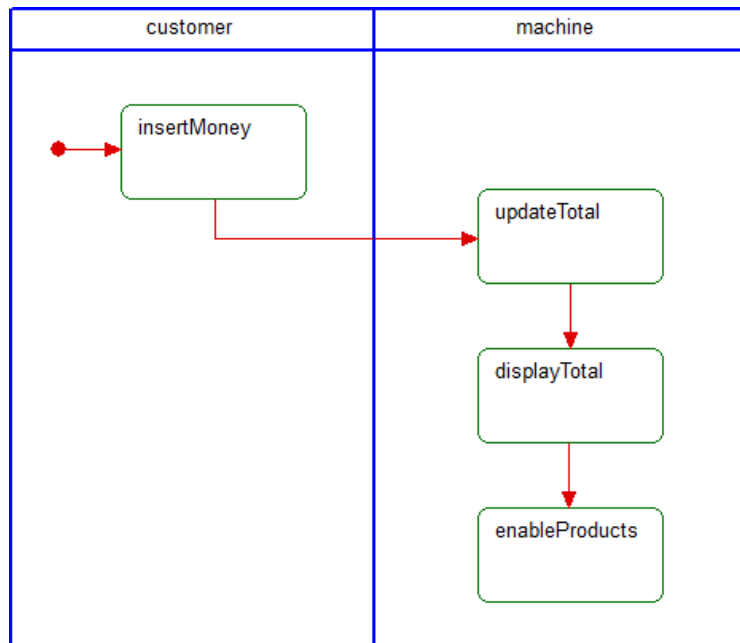


Task 3: Create a black-box activity diagram

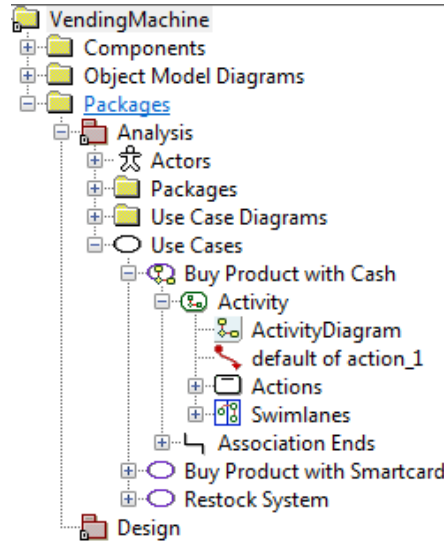
In this task you create a black-box activity diagram. A black-box activity diagram has a system-level scope, where

the internal components of the system are not being considered. In a black-box activity diagram, each action describes the activities (or services) provided by the system. If swimlanes are used, all but the system swimlane are represented by actors.

1. Add a new activity diagram to the **Buy Product with Cash** use case. Right-click the **Buy Product with Cash** use case and select **Add New > Diagrams > Activity**.
2. Add swimlanes, activities, and flows to the diagram as shown in the following figure:



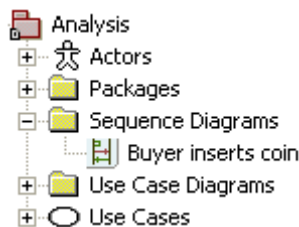
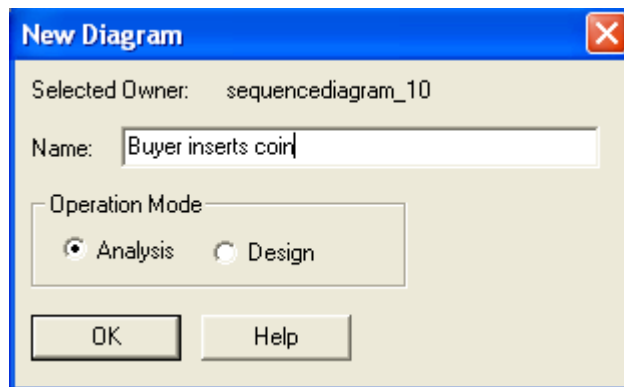
- a. Locate the action icon on the diagram tools and add the following four actions to the diagram: **insertMoney**, **updateTotal**, **displayTotal**, and **enableProducts**.
 - b. Add an initial flow to the **insertMoney** action.
 - c. Add a control flow from the **insertMoney** action to the **updateTotal** action.
 - d. Add a control flow from **updateTotal** to **displayTotal**, and from **displayTotal** to **enableProducts**.
 - e. Add a swimlane frame and a swimlane divider to create two swimlanes. Name the swimlanes **customer** and **machine**.
3. The **customer** swimlane will be represented by the **Buyer** actor from the use case diagram.
 - a. Open the customer swimlane’s features dialog and in the **General** tab select Represents:**Buyer in Analysis**.
 4. The project browser should look as follows:



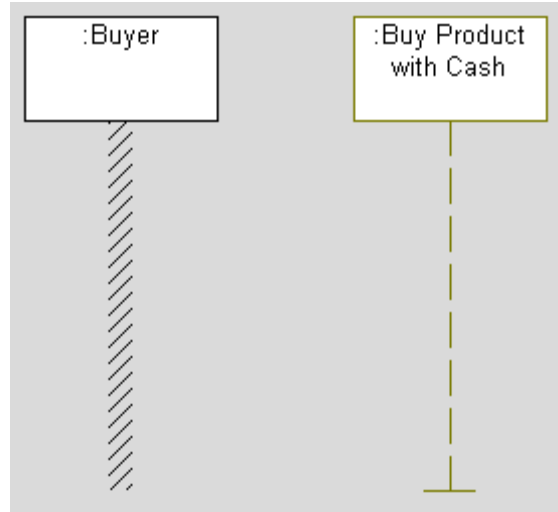
Task 4: Create a use-case level sequence diagram

Sequence diagrams can help describe the scenarios of a use case. Use case-level sequence diagrams have just one use case and one or more actors. The use case represents the system, but only in the context of the use case.

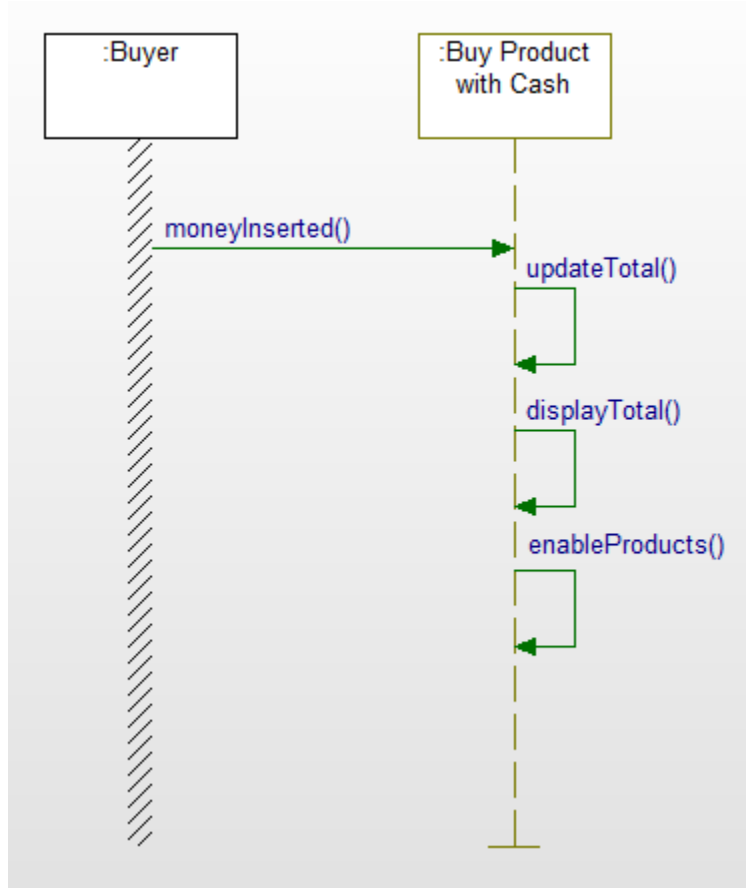
1. Add a new sequence diagram called `Buyer inserts coin` in the **Analysis** package.
2. Confirm that the **Operation Mode** is set to **Analysis**.



3. Drag the **Buyer** actor from the browser onto the sequence diagram.
4. Drag the **Buy Product with Cash** use case from the browser onto the diagram.



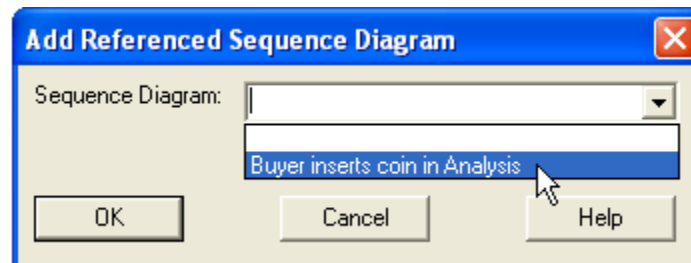
5. Use the Message icon on the diagram toolbar and draw the message from the **Buyer** to **Buy Product with Cash**. This will create a synchronous message. Name the message `moneyInserted`
6. Draw a message from Buy Product with Cash to itself and name the message `updateTotal`.
7. Draw another message from Buy Product with Cash to itself and name the message `displayTotal`.
8. Draw another message from Buy Product with Cash to itself and name the message `enableProducts`.



Note that each reflexive message matches the activities in the **Buy Product with Cash** activity diagram. Generally speaking, activity diagrams are preferred by systems engineers, and sequence diagrams are preferred by software developers. As you will see in the next lab, sequence diagrams play a key role in the transition from use case analysis to a software design.

Several sequence diagrams may be used to describe use case scenarios, and each can be set as a reference from the use case.

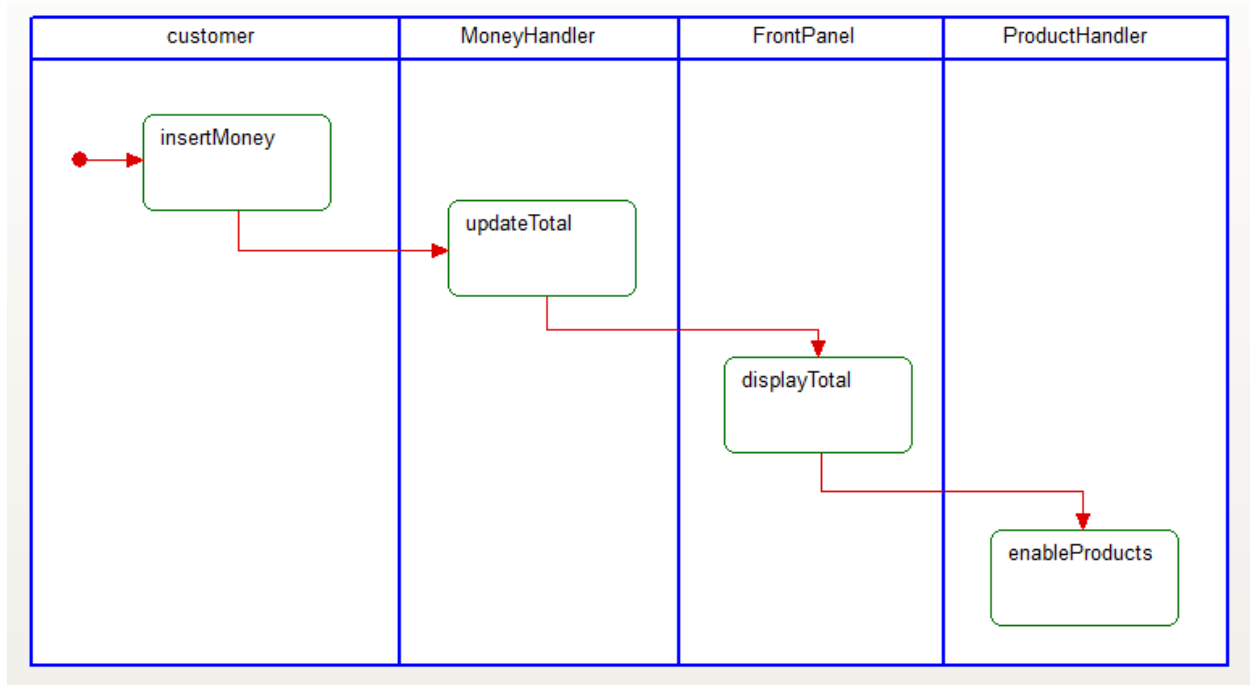
9. Right-click the **Buy Product with Cash** use case in the browser and select **Add New > Diagrams > Referenced Sequence diagram**. In the drop-down box select **Buyer inserts coin in Analysis**.



Task 5: Allocate activities to subsystems

In this task you will allocate the system-level activities to subsystems. Rational Rhapsody and SysML can be used to perform a trade study to evaluate candidate architectures. However, trade studies are outside the scope of this lesson. For this lab, assume that there are three primary subsystems: the MoneyHandler, FrontPanel, and ProductHandler subsystems.

- Using the swimlane divider, break the machine swimlane into two more swimlanes and name the swimlanes `MoneyHandler`, `FrontPanel`, `ProductHandler`. Replace the machine swimlane with either `MoneyHandler`, `FrontPanel`, or `ProductHandler`. Move the actions to be in the appropriate swimlane as shown in the picture below:



- Save the model.



Lab 3: Create an Object Model diagram

Objectives

After completing this lab, you will be able to:

- ▶ Create an Object Model diagram
- ▶ Create a class-level sequence diagram
- ▶ Generate and build source code

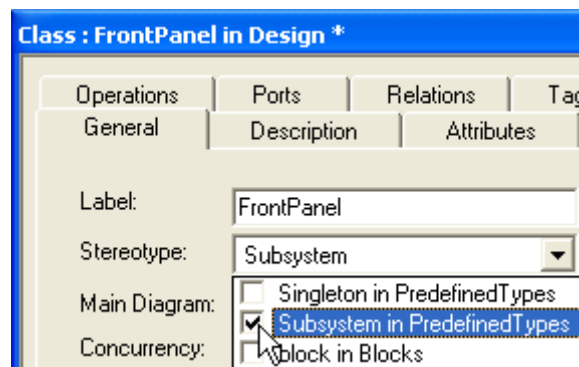
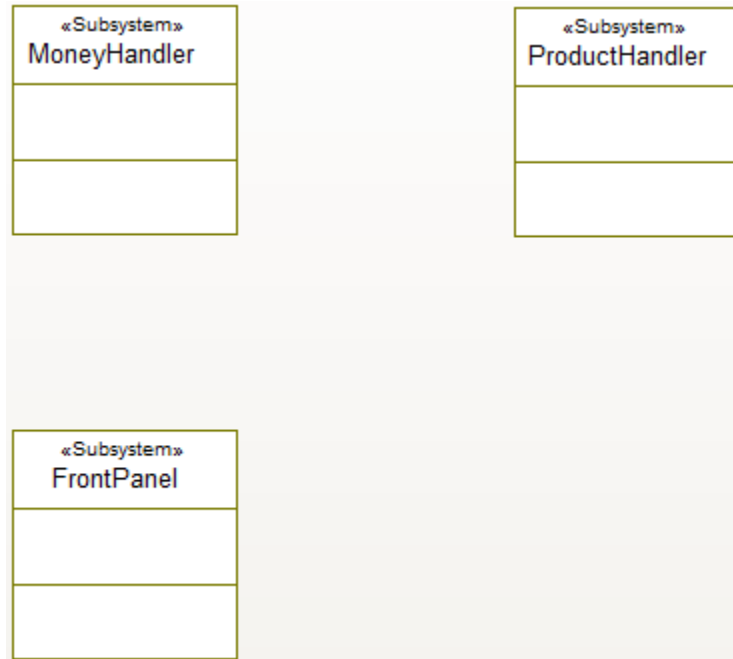
Scenario


A Rational Rhapsody Object Model diagram (OMD) is a general purpose structural diagram that can show classes, packages, objects, and the relationships between them. – A class diagram (object model diagram) is created in this lab showing the relations between the subsystems. A class-level sequence diagram is created to elaborate the messages defined in the Lab 2 use-case level sequence diagram. These messages are realized as new operations on the classes. You will see that Rational Rhapsody generates code for the classes and relations.

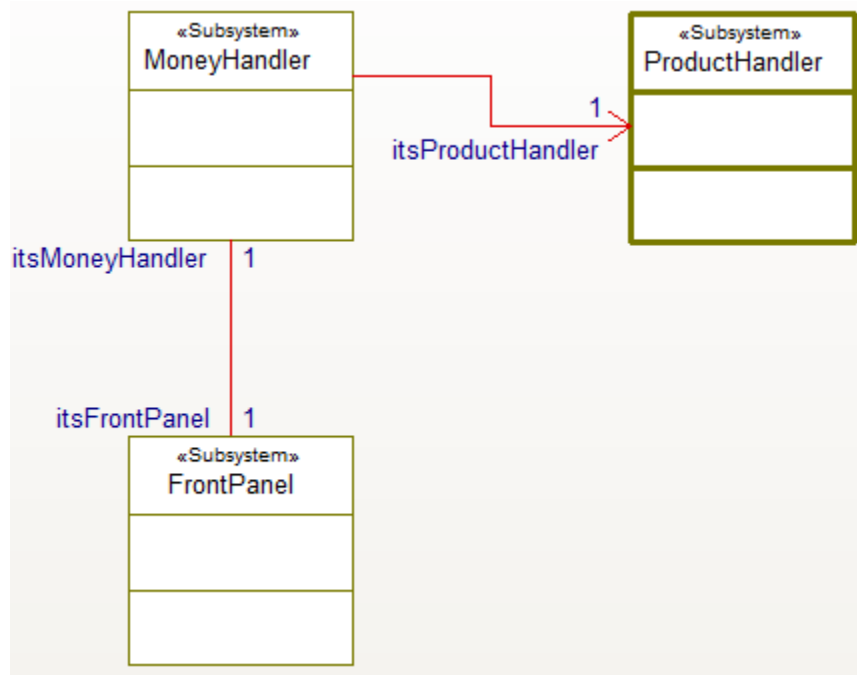
Task 1: Add a new Object Model diagram to the model

In this task you add a new Object Model diagram to the VendingMachine project.

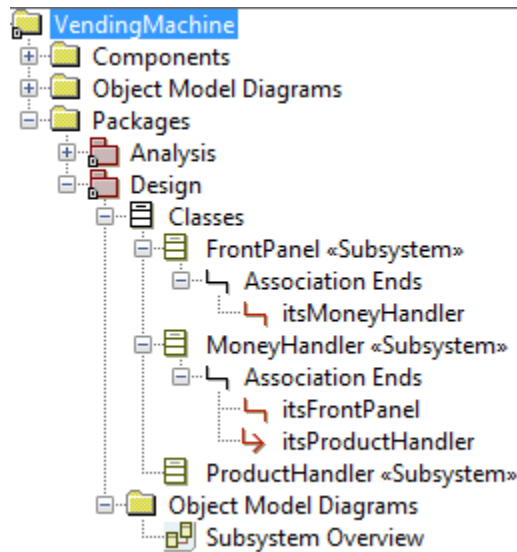
1. Add a new Object Model diagram called `Subsystem Overview` to the **Design** package.
2. Add three classes to the Object Model diagram. Name them `FrontPanel`, `ProductHandler`, `MoneyHandler`. Move the classes on the diagram so `MoneyHandler` is in the top left, `ProductHandler` in the top right, and `FrontPanel` in the bottom left. Resize the classes and align their edges using the Layout toolbar.
3. Set the stereotype of all three classes to **Subsystem**.



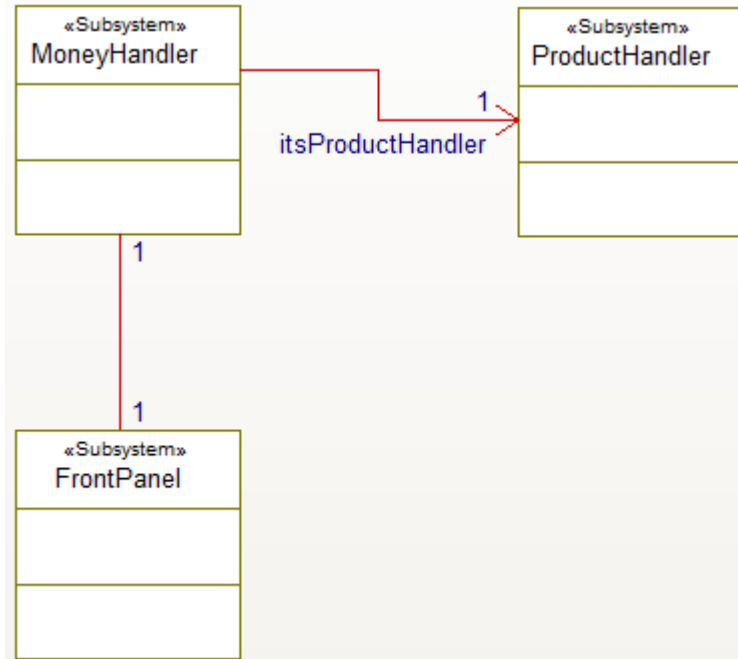
4. Add a directed association between the MoneyHandler and the ProductHandler class. End1 has multiplicity set to 1.
5. Add a symmetric association  between the MoneyHandler and the FrontPanel class. End1 and End2 of the association have multiplicity set to 1.



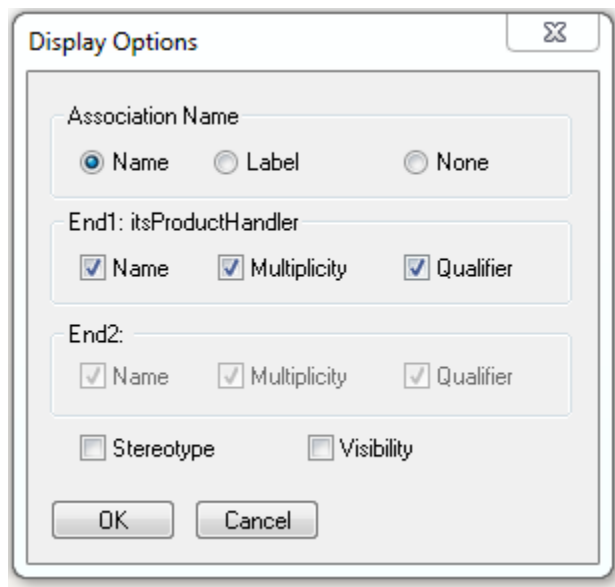
In the Subsystem Overview diagram, the associations will enable object communication in support of the **Buyer inserts coin** sequence diagram. Each instance of MoneyHandler communicates with one instance of ProductHandler, and knows that instance by the role name itsProductHandler.



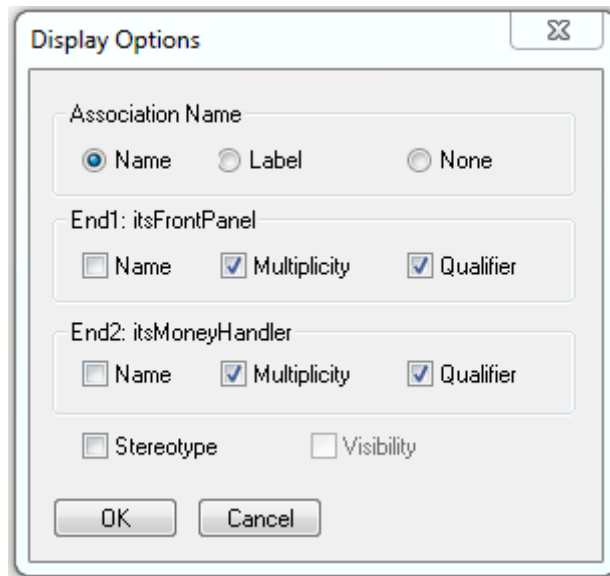
6. Select the association between **MoneyHandler** and **FrontPanel**, and use the association's **Display options** to turn off the display of role names of the association ends.



Following are the Display Options for the association between MoneyHandler and ProductHandler:



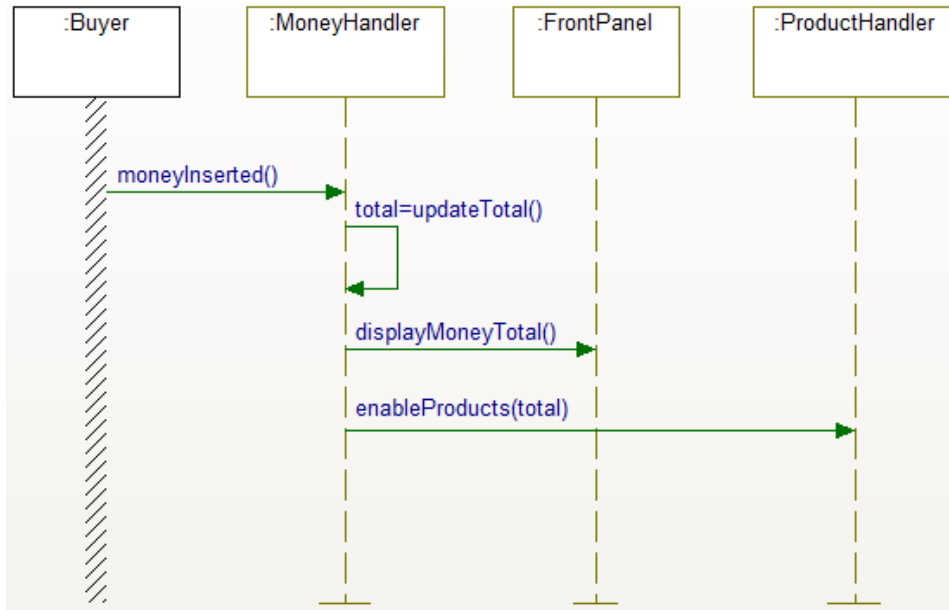
Following are the Display Options for the association between MoneyHandler and FrontPanel:



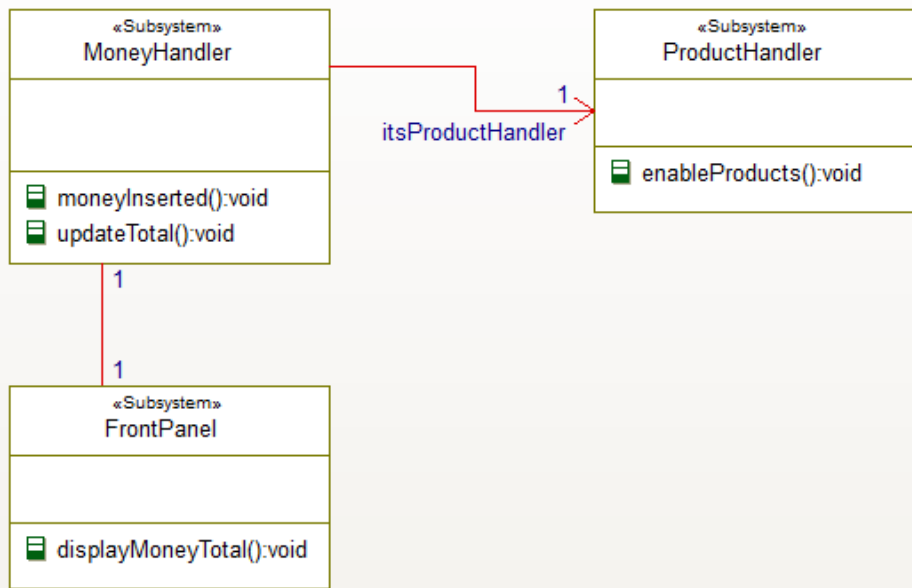
Task 2: Detail the sequence diagram

In this lab you will elaborate the use-case level sequence diagram to include classes and their messages. You also create operations by realizing the model, and you update the operations with arguments and return types.

1. Place a copy of the **Buyer inserts coin** sequence diagram into the Design package.
 - a. Select the diagram in the browser, press and hold the **Ctrl** key, and drag it to the Design package. Rename the copy to **Money inserted**.
2. Open the new **Design::Money inserted** sequence diagram.
3. Drag the **MoneyHandler**, **FrontPanel**, and **ProductHandler** classes from the browser onto the **Money inserted** sequence diagram, move the messages by dragging their end points, and change the message names as shown following. Change the text on the message by editing it right where it is shown on the diagram. When completed, remove from view the original **Buy Product with Cash** use case instance line.

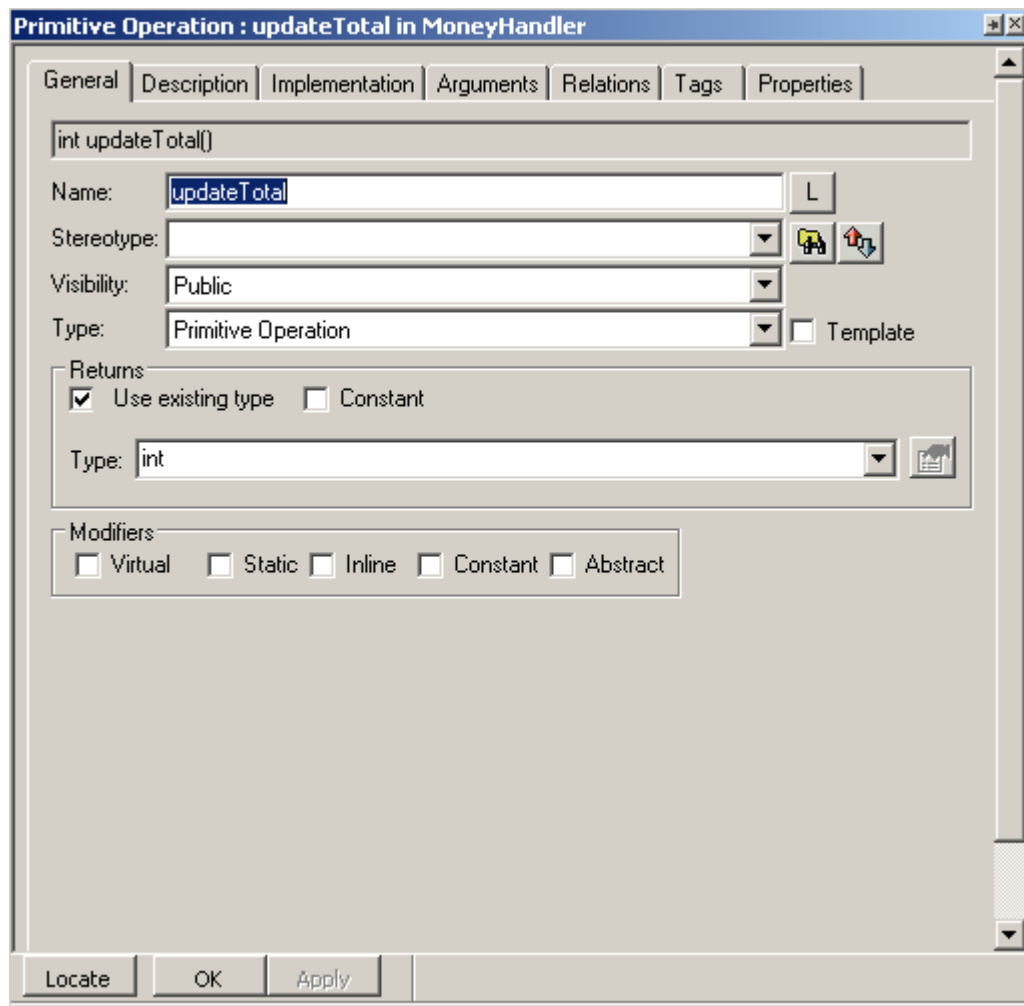


4. With the **Money inserted** sequence diagram as the active diagram, select **Edit > Select > Select Un-Realized** from the main menu.
5. Select **Edit > Auto Realize** from the main menu. This causes the messages to become realized as operations in the classes.
6. Confirm that the operations now show up on the **Subsystem Overview** Object Model diagram as shown below. You may need to change your Display options.

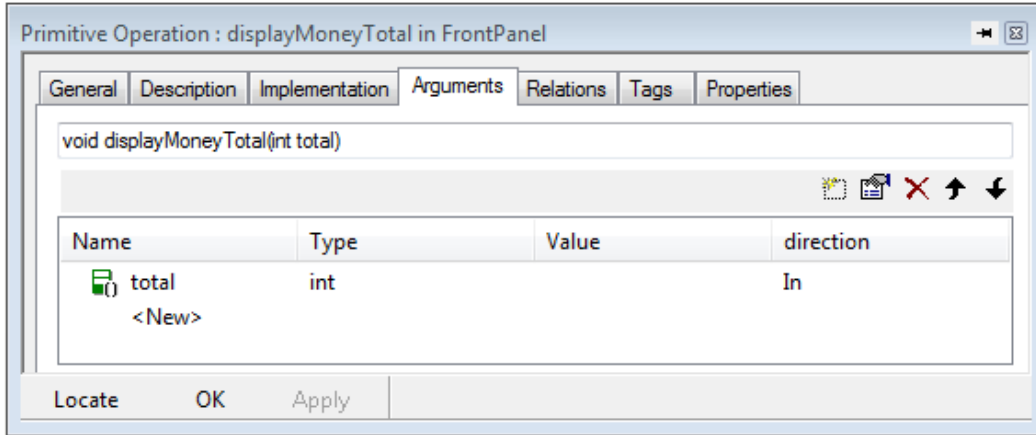


7. Resize and move the classes in the Object Model Diagram as needed.

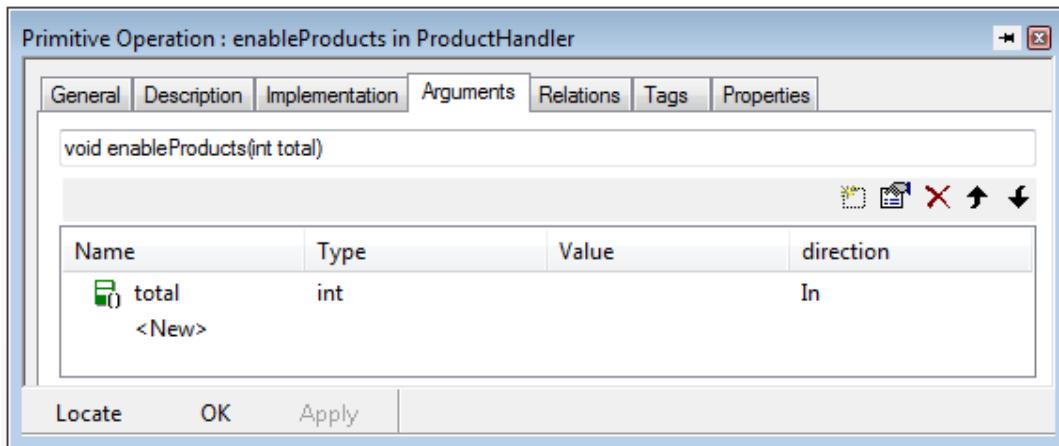
8. Update the `updateTotal` operation with a return type as follows:
 - a. Expand the `MoneyHandler` class in the Design package of the browser.
 - b. Select the `updateTotal` operation in the Operations category. In the **Features>General** tab, set the Returns Type to `int`. Click **OK**.



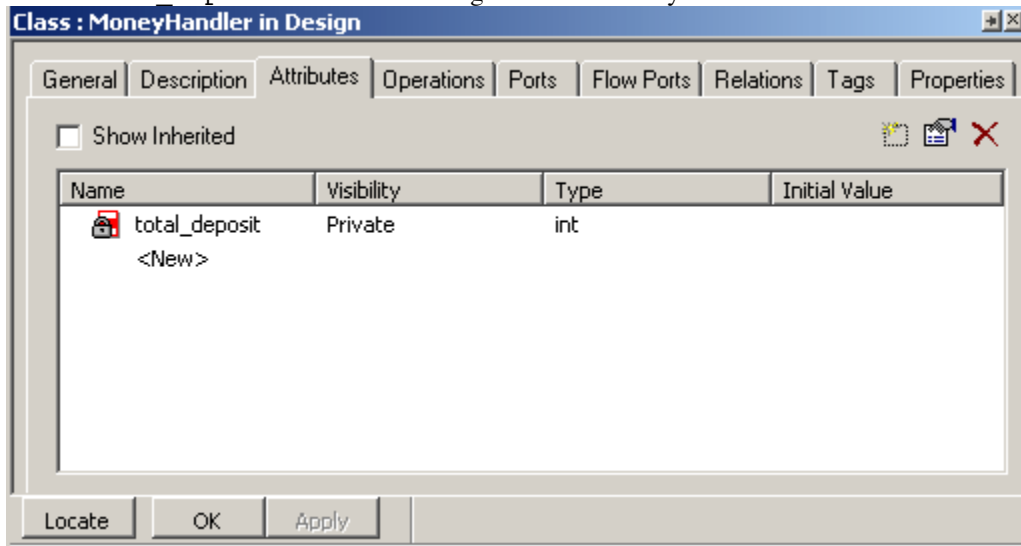
9. Add arguments to the `displayMoneyTotal` and `enableProducts` operations.
 - a. Expand the `FrontPanel` class in the Design package of the browser. Select the `displayMoneyTotal` operation in the Operations folder. In the **Features>Arguments** tab, add a new argument of type `int` and name the argument `total`. Click **OK**.



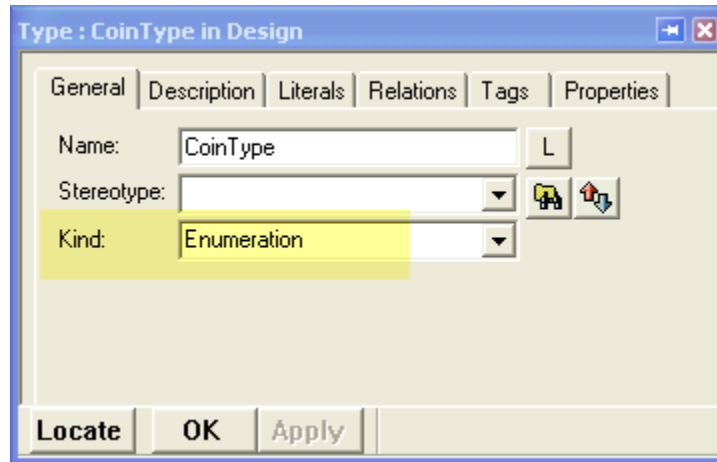
- b. Expand the **ProductHandler** class in the Design package of the browser. Add a `total` argument of type **int** to the **enableProducts** operation.



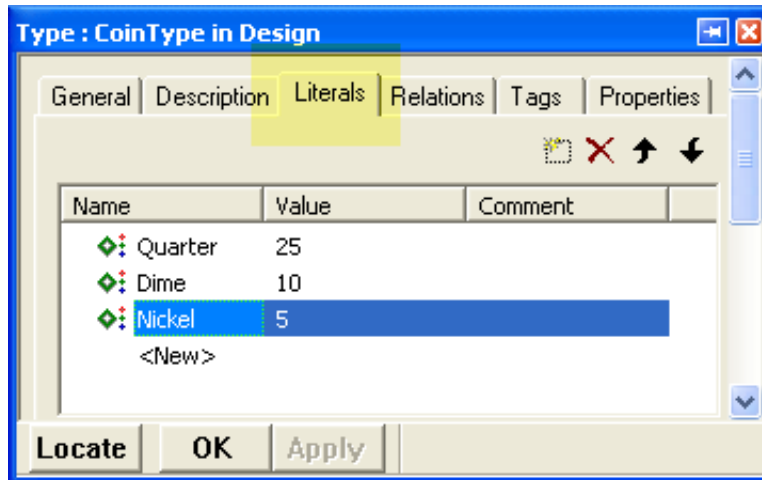
10. Double-click the **MoneyHandler** class and in the **Attributes** tab of the Features dialog, add a new attribute called `total_deposit` to the class. Assign Private visibility.



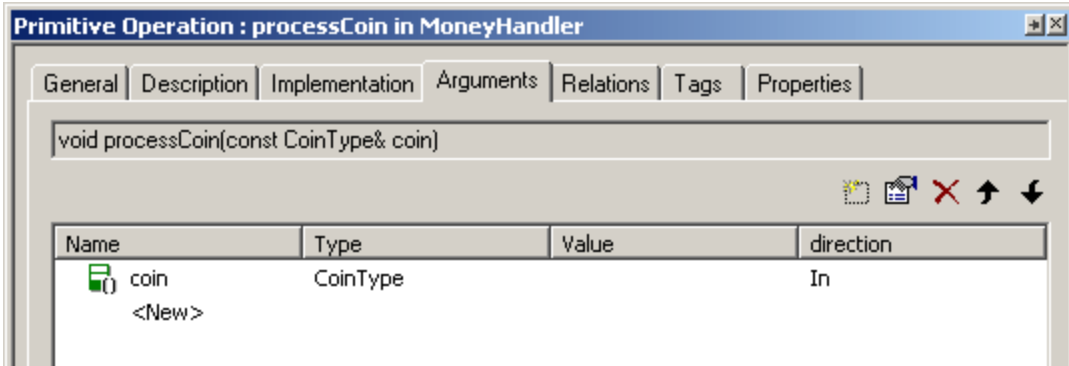
11. Add an enumerated type. Right-click the **Design** package and select **Add New>Type**. Name the new type **CoinType**.



- a. In the **Features>General>Kind** field of the type, select **Enumeration**.
- b. In the **Literals** tab, add enumerations as shown.



12. Right-click the **MoneyHandler** class and select **Add New > Operation** to create a new **processCoin** operation for it.
 - a. Add an argument **coin** of type **CoinType** to the **processCoin** operation.

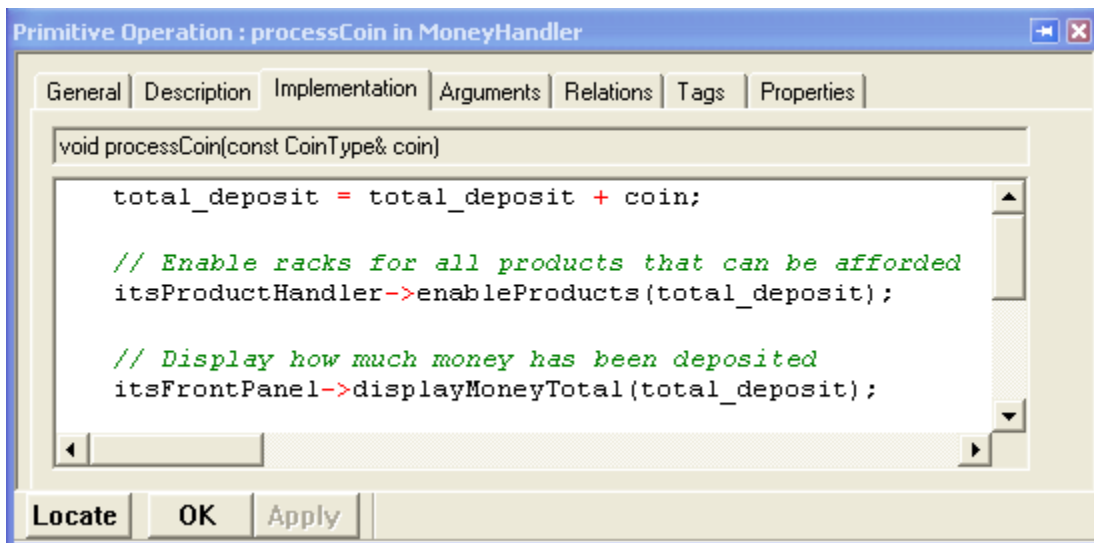


- b. Right-click the processCoin operation in the browser and select the **Features>Implementation** tab. Update the implementation of the **processCoin** operation as follows:

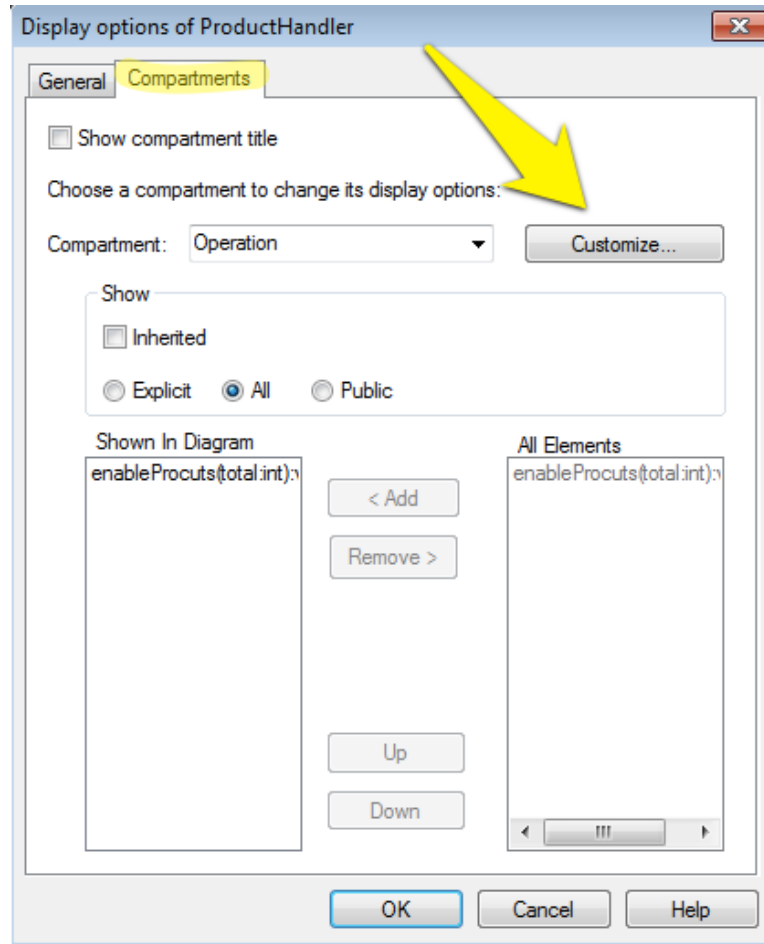
```
total_deposit = total_deposit + coin;

//Enable racks for all products that can be afforded
itsProductHandler->enableProducts(total_deposit);

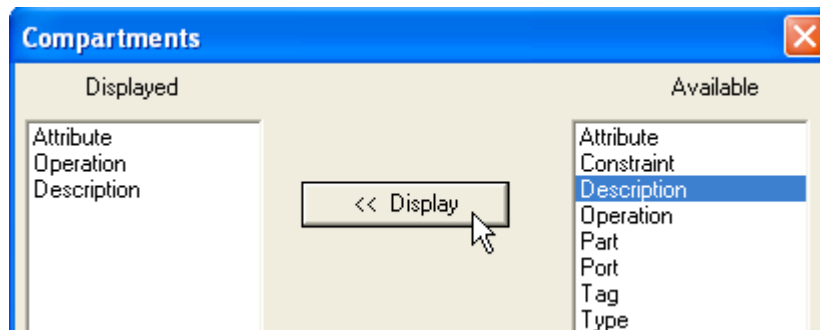
//Display how much money has been deposited
itsFrontPanel->displayMoneyTotal(total_deposit);
```



13. Add a description to the ProductHandler class as follows:
Controls and monitors all racks. Reports when a rack is empty.
14. Right-click the ProductHandler class on the Object Model diagram and select **Display Options**. Click the **Compartment** button and confirm that you can see the **Description** as Available.



15. Add **Description** as a third compartment.



16. Confirm that you can see the description displayed in a compartment of the `ProductHandler` class.

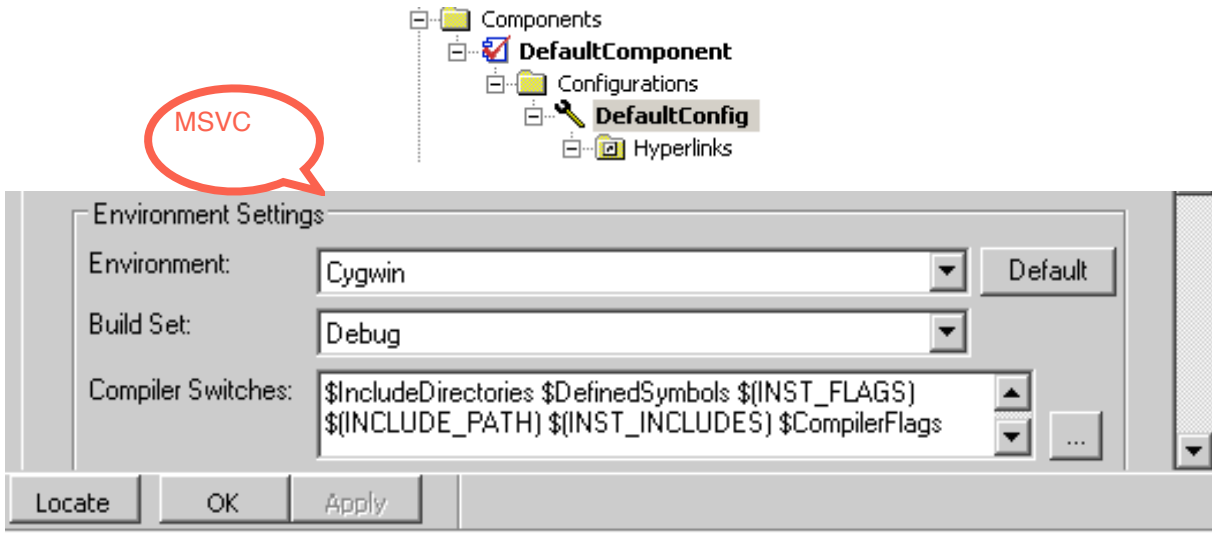
17. Save your work.


Task 3: Build the Model

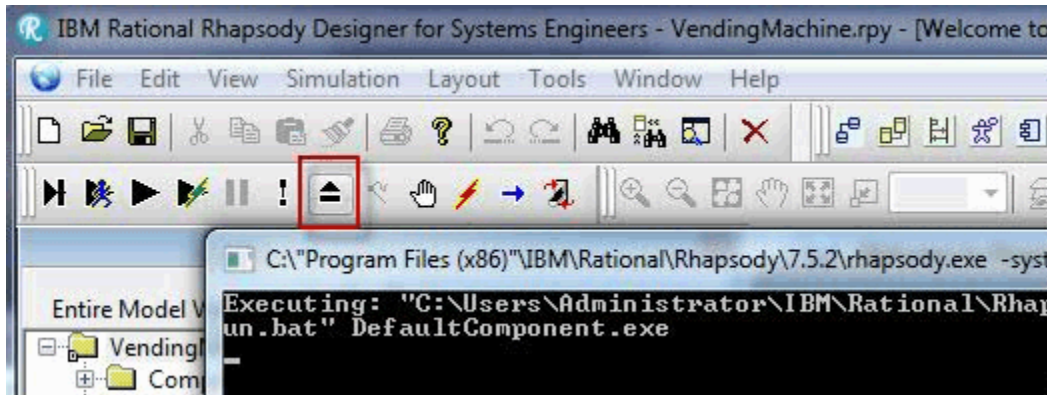
Generating and building the source code enables Rhapsody to play a key role in iterative software development. Just as you would frequently test the quality of your handwritten source code, use Rhapsody to iteratively generate code from the model, and to invoke the build. Consider this next task your first build iteration.

1. Save the Model.

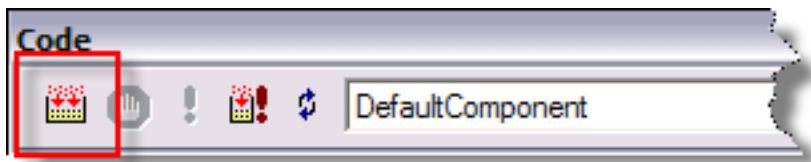
- In the browser at the top level of the project, right-click the **DefaultConfig**. Confirm the **DefaultConfig Features > Settings > Environment** is set to **Cygwin** or your chosen compiler.



- Select **Code > Generate > DefaultConfig**. (If you are using Rhapsody Designer for Systems Engineers, you can do Smart Build  - note that this build includes Make and Run. You will need to close the simulation – this will be covered in Lab 5.)



- When prompted to create a new directory, answer **Yes**. Save the model.
- Select the **Make** icon from the top toolbar.



- If the build fails, double-click the first line of the error message. Troubleshoot the errors starting at the top. After each error is fixed build the model again.
- Save the model.



Lab 4: Navigate the model and add a state machine diagram

Objectives

After completing this lab, you will be able to:

- ▶ Create hyperlinks to facilitate model navigation
- ▶ Create a state machine diagram
- ▶ Build the model

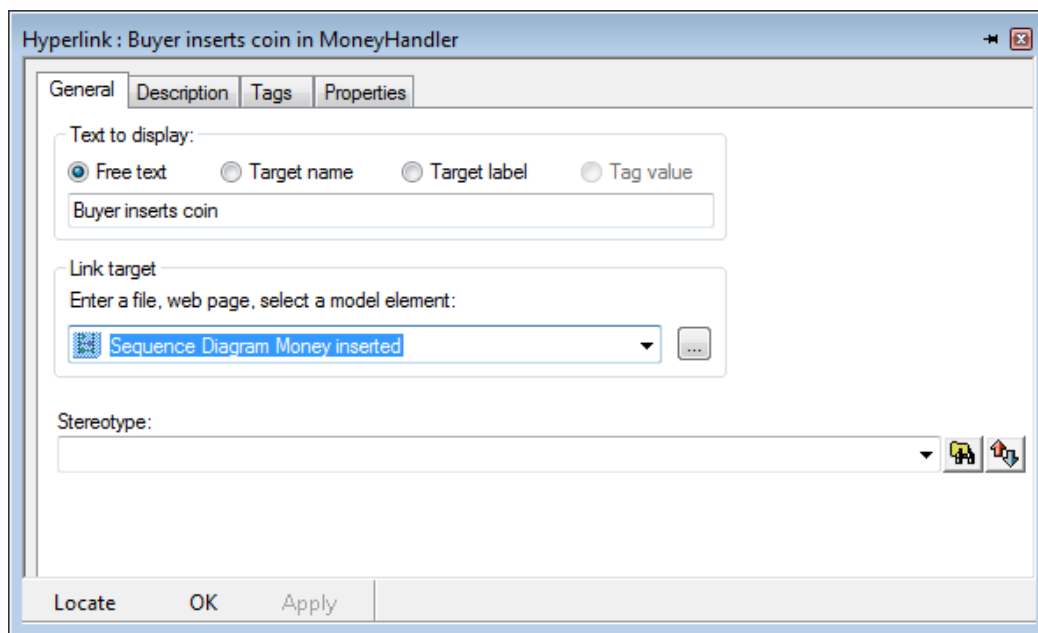
Scenario

In this lab several hyperlinks are added to the model to facilitate navigation throughout the model. Hyperlinks can add to the readability of a model when used to show related diagrams and model elements. A state machine is used to describe the behavior of the MoneyHandler class. You will see that Rhapsody generates code for the state machine.

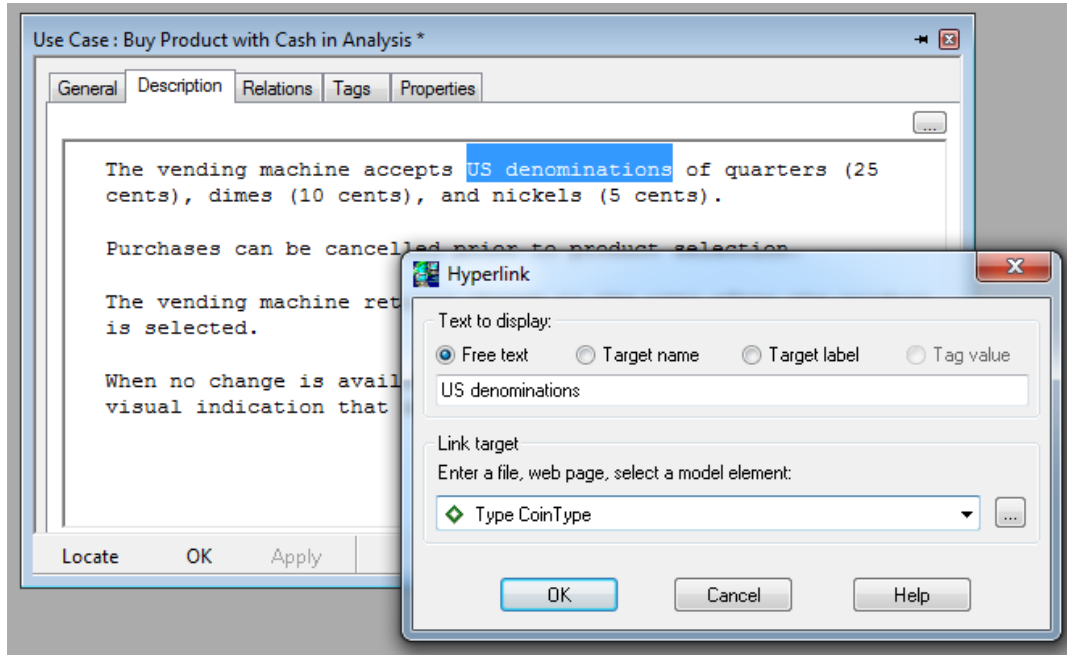
Task 1: Create a hyperlink


In this task you create several types of hyperlinks. Hyperlinks can point to diagrams, model elements, files, or web sites. Hyperlinks aid in model navigation and understanding, and are particularly helpful when used to tell a story by guiding the reader through the model.

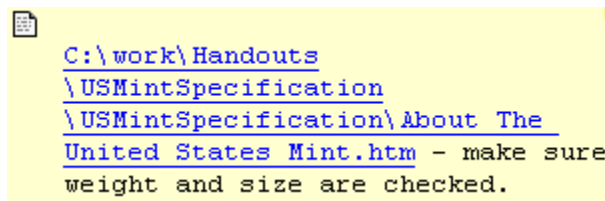
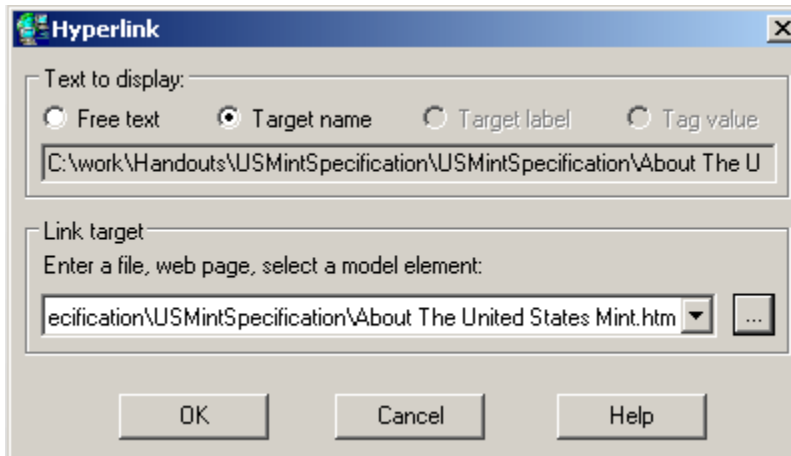
1. Find the **Design::MoneyHandler** class in the browser, right-click and select **Add new > Relations > Hyperlink**.
2. Open the Features of the hyperlink. For the **Text to display**, confirm the **Free Text** radio button is selected and enter `Buyer inserts coin`.
3. Browse to find the **Money inserted** sequence diagram and select it as the Link target. Click **OK**.



4. Double-click **Analysis::Buy Product with Cash** use case to launch the Features dialog.
5. Highlight some text in the use case description and create a hyperlink from the text to a model element of your choice. To do this, right-click the text and select **Hyperlink**.



6. Find the Comment icon  **Comment** in the Drawing:Common toolbar. Add the following comment to the **Design:SubSystem Overview** Object Model diagram.
 TODO – make sure weight and size are checked.
7. Highlight some text in the new comment, right-click the text and hyperlink it to the **About The United States Mint.htm** file found in your **C:/work/Handouts** directory. Set the Text to display to **Target name**.

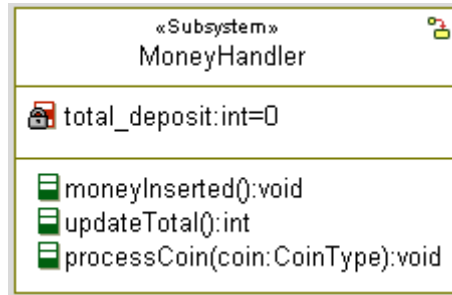


8. Test the link to confirm that it links correctly.

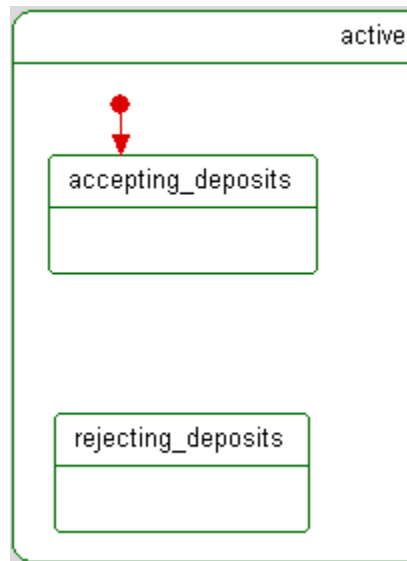
Task 2: Create a state machine diagram

In this task, you build a new state machine to describe the behavior of the MoneyHandler class..

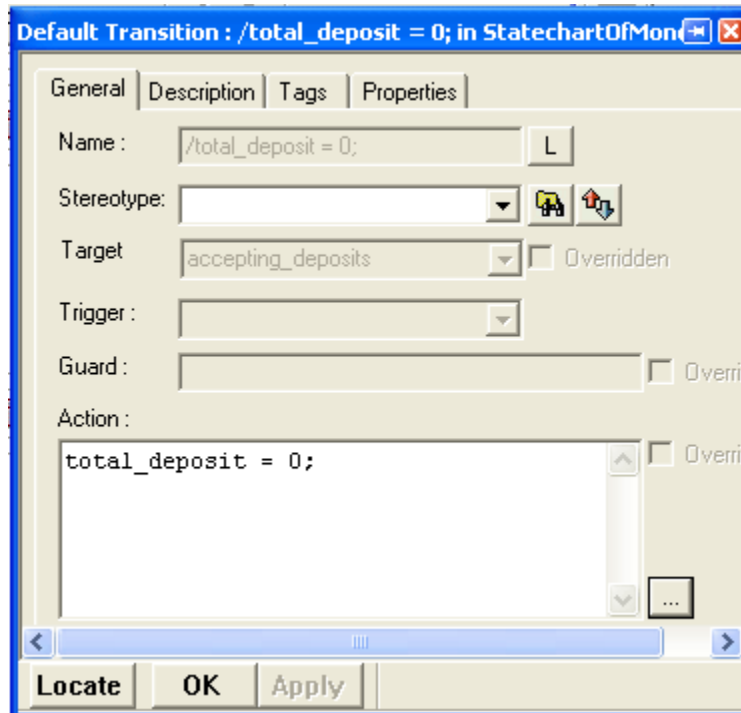
1. Right-click the MoneyHandler class (on a diagram or in the browser) and select **Add New>Diagrams>Statechart**.
2. Confirm that the icon to the link to the statechart appears on the class on the **Subsystem Overview OMD**.



3. Add an **active** state to the statechart using the state icon on the drawing toolbar.
4. Add two new states to the statechart inside the active state using the state icon on the drawing toolbar. The two states are **accepting_deposits** and **rejecting_deposits**.
5. Using the icon on the drawing toolbar, draw a default transition to the **accepting_deposits** state.

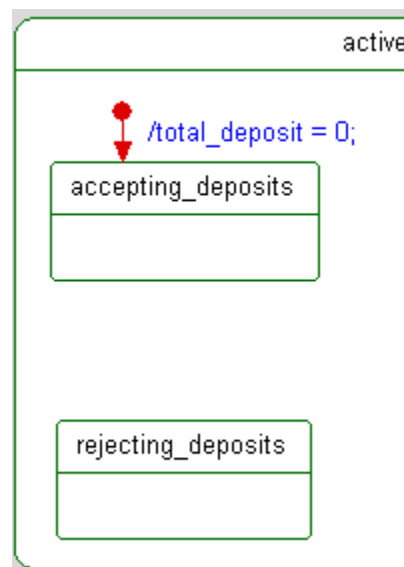



6. In the **Features > General** tab for the default transition, add an action to the action field. The action is `total_deposit = 0;`



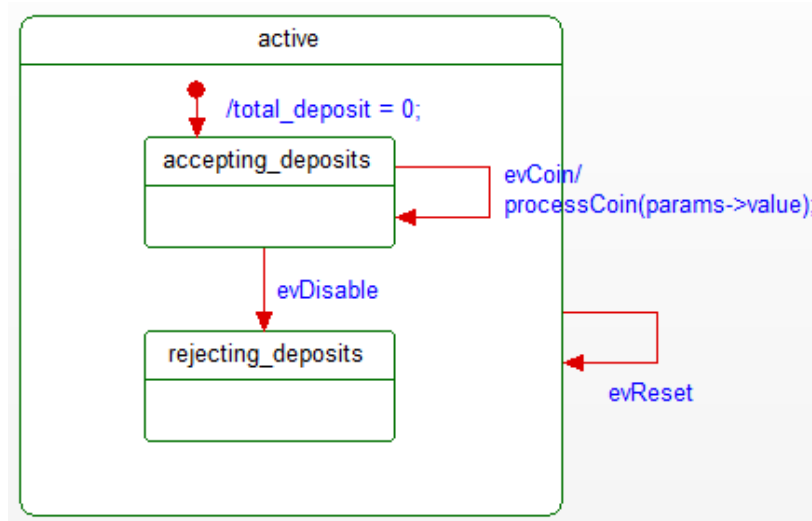
When the Features dialog is used, Rational Rhapsody automatically displays the trigger, guard, and action with proper syntax `trigger [guard] / action list`.

When the trigger, guard, and action list are entered directly on the diagram, you must manually enter the [] and /. Use Ctrl+Enter to complete an entry.

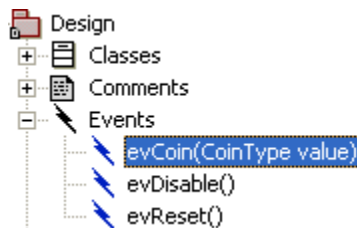
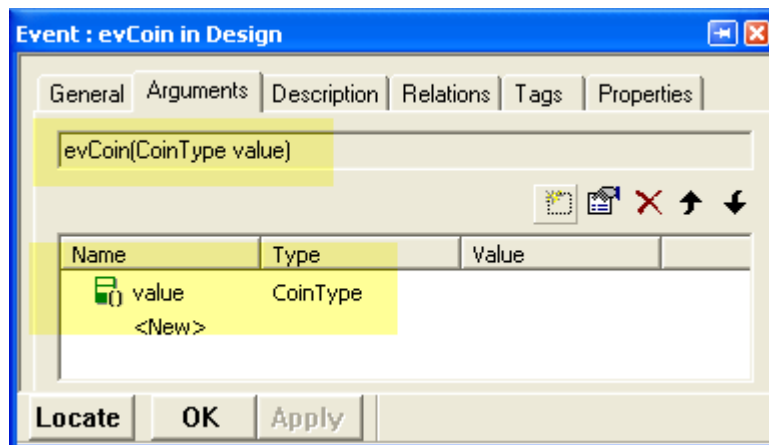


7. Draw a transition from the **accepting_deposits** action to **rejecting_deposits**. Immediately type `evDisable` on the transition when you are prompted to name it.. Press Ctrl+Enter to complete the entry. If you miss the opportunity to type immediately on the transition, click the transition label button  **Transition Label** on the drawing toolbar and then click the transition to type the text.

8. Draw a transition from the **accepting_deposits** action to itself. Immediately type `evCoin/processCoin(params->value);` You are creating an action by putting a slash in the name. The first part is a trigger and the second part is an action.
9. On the active state, add a transition to itself. Type `evReset` immediately on the transition.




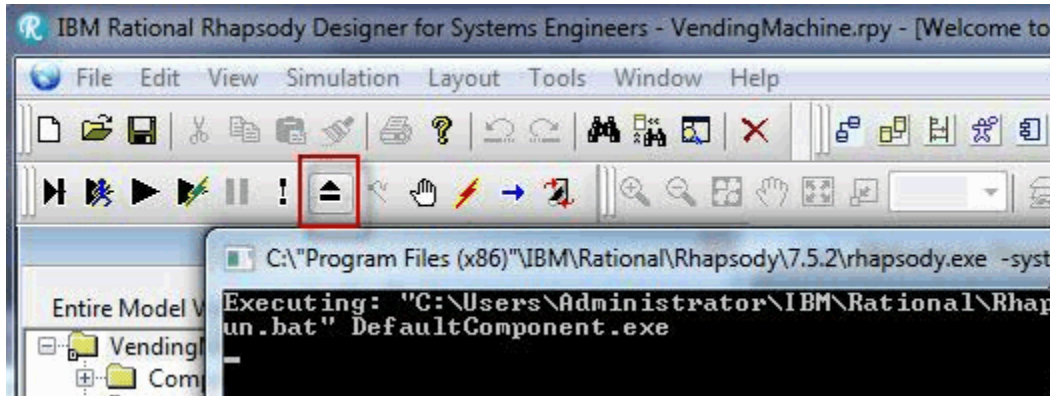
10. In the Rational Rhapsody browser under **Design:Events**, in the **Features > Arguments** tab add to the `evCoin` event an argument named `value` of type `CoinType`.



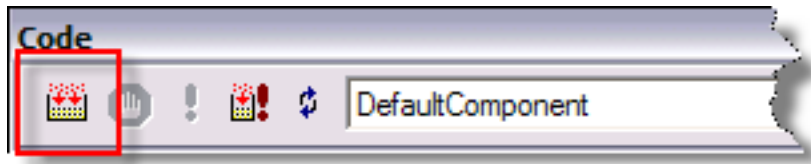
If an event has an argument, then Rational Rhapsody provides a pointer called `params` that can be used on the transition where the event occurs to get access to the argument.

Task 3: Build the Model

1. Save the model.
2. Select **Code > Generate > DefaultConfig**. (If you are using Rhapsody Designer for Systems Engineers, you can do Smart Build  - note that this build includes Make and Run. You will need to close the simulation. Simulation is covered in Lab 5.)



3. Select the **Make** icon from the top toolbar.



4. If the build fails, double-click the first line of the error message. Troubleshoot the errors starting at the top. After each error is fixed build the model again.
5. Save the model.



Lab 5: Model Execution

Objectives

After completing this lab, you will be able to:

- ▶ Build a report on the model
- ▶ Execute and test the model

Scenario

In this lab, two reports are generated on the model. The first is a simple report, generated using a fixed format. The second report uses ReporterPLUS to generate a heavily hyperlinked HTML report that includes diagrams with navigable hot spots. Source code is then generated from the model, built, and executed. Model execution is used to observe object interactions, and also to unit test a reactive class. Finally, the animated sequence diagram is compared to the expected scenario as captured in Lab 3, and two discrepancies are found.


Task 1: Report on model

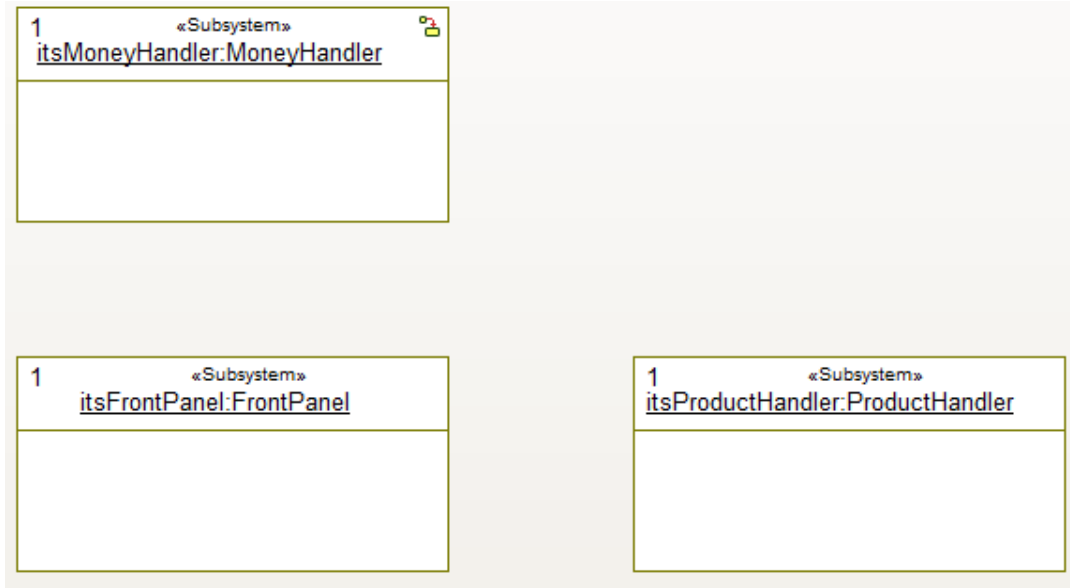
In this task you create a new report using the standard Report on Model and then you use one of the ReporterPLUS ready to use templates to create a report.

1. In the browser, right-click the top level VendingMachine project. Open the **Features>Properties** tab for the project. View all properties.
2. Confirm the `General.Graphics.ExportedDiagramScale` property for the project is set to **FitToOnePage**.
3. Select **Tools > Report on Model**.
4. Select the report settings of your choice and click **OK**.
5. A simple `rtf` format file is created. The **Report on Model** format cannot be customized by the user.
6. View and then close the new report.
7. Select **Tools > ReporterPLUS > Create/edit template with ReporterPLUS**.
8. When you are prompted with the question “What would you like to do?” answer **Generate HTML page**.
9. When you are prompted to select a template, browse to find the Rhapsody `HTML Exporter.tpl`.
10. Click **Next>Open** and browse to open the VendingMachine model to report on. Name the new report **mynewreport**. **Generate** the report. The report takes a few minutes to generate. When you are prompted with “Do you wish to open the report now” answer **Yes**.
11. Examine and close the HTML report.

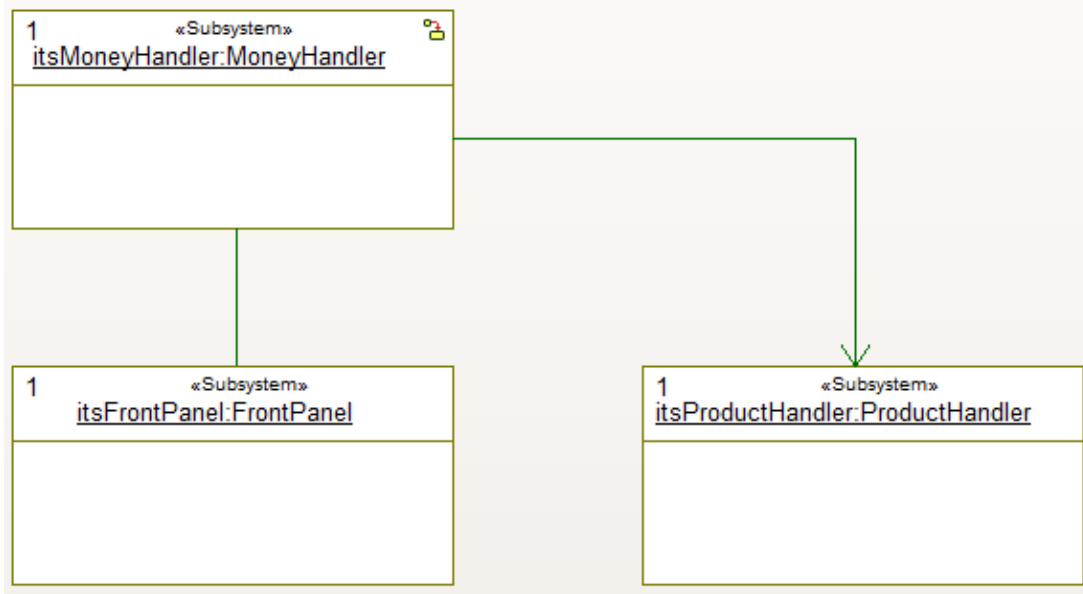
Task 2: Execute the model

In this task you execute the model to compile and generate code and validate the model to ensure correctness.

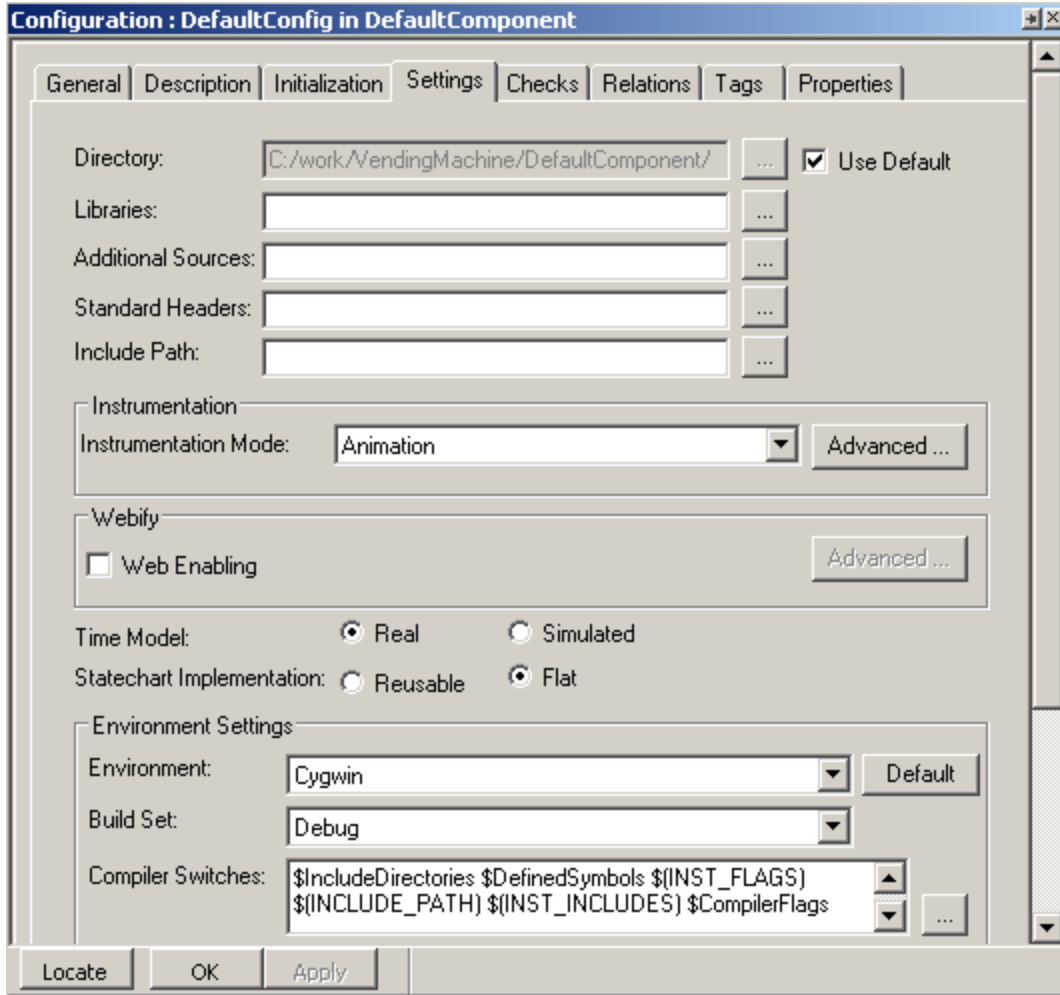
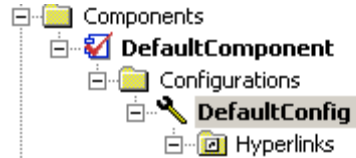
1. Create a new **Build Overview** Object Model diagram in the Design package.
2. Drag the `MoneyHandler`, `FrontPanel`, and `ProductHandler` classes from the browser onto the diagram.
3. Right-click on the two **red** associations that are on the diagram and select **Display Options>Remove from View**. Do not select **Delete from Model**.
4. Right-click each class on the diagram and select **Make an Object**. For each class, change to the structured view by clicking the structured/specification view icon . Resize and align the object boxes. Examine the diagram.





5. Add links between the classes on the diagram using the link icon. A link is an instantiation of an association, just like an object is an instantiation of a class. Draw a link from the `MoneyHandler` class to the `FrontPanel` class, and from the `MoneyHandler` class to the `ProductHandler` class.

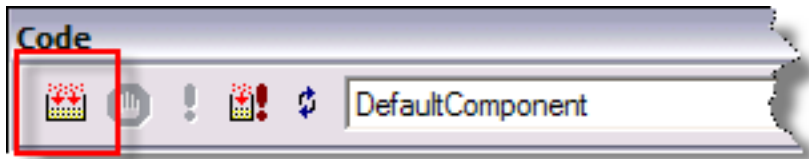


6. Save the model.
7. In the browser at the top level of the project, right-click the **DefaultConfig**. Set the **DefaultConfig** configuration **Features > Settings > Environment > Instrumentation mode** to **Animation**. Verify that the environment setting is **Cygwin** (or your chosen compiler).

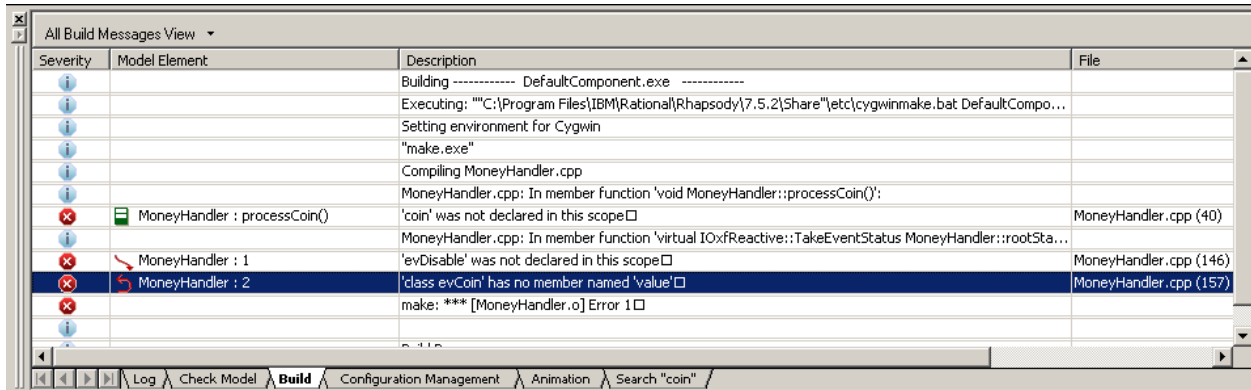



8. Select **Code > Generate > DefaultConfig**. (If you are using Rhapsody Designer for Systems Engineers you can do a Full Build  or a Smart Build  - note that these builds include Make and Run. The animation toolbar shown in Step 13 will appear after the Build.)

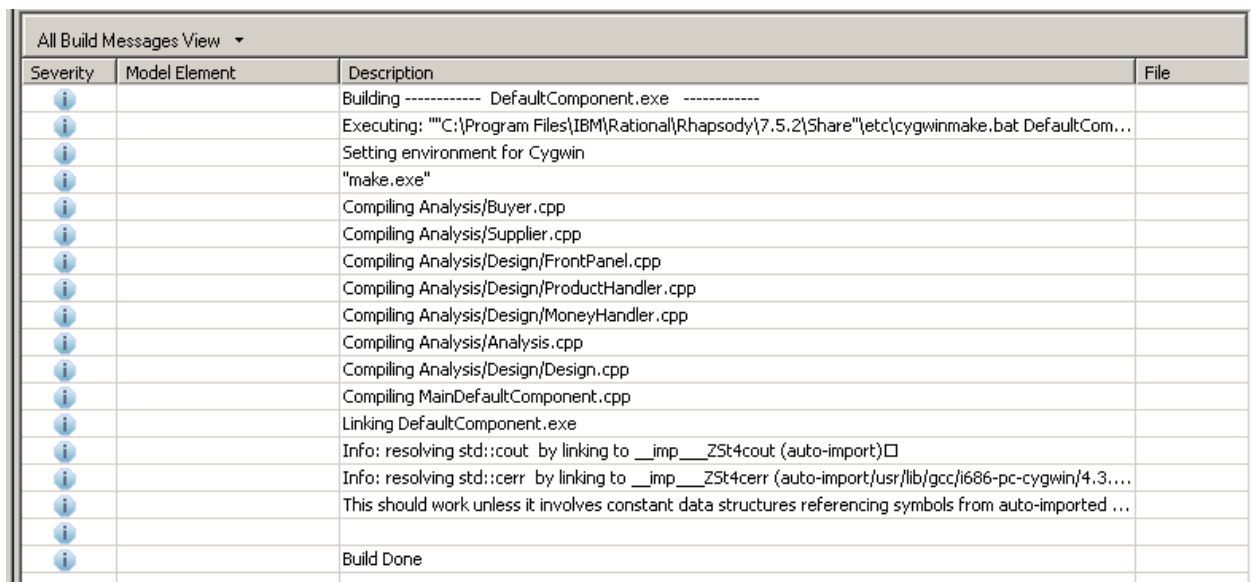
9. Select the **Make** icon from the top toolbar.



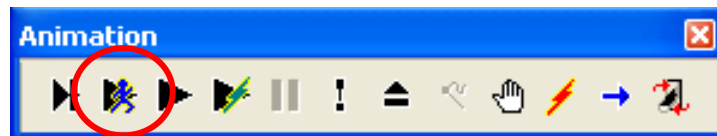
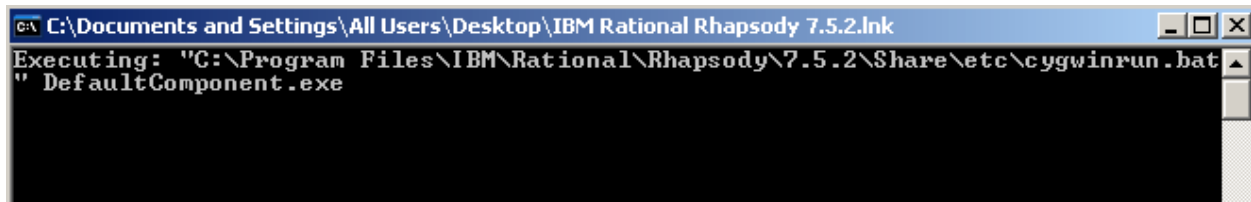
10. If the build fails, double-click the first line of the error message. Troubleshoot the errors starting at the top. After each error is fixed build the model again.



11. After the build is successful, run  the generated application.

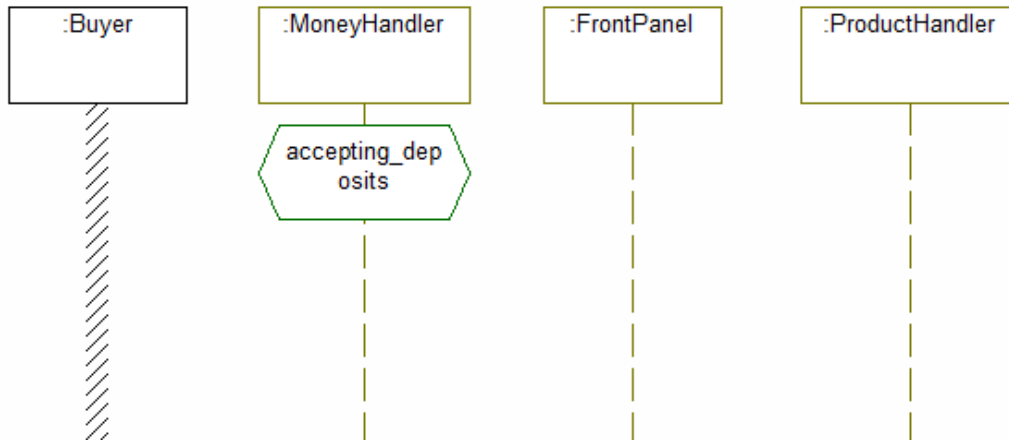



12. When the build and run are successful, the model executes, the black console displays and the animation toolbar is displayed.

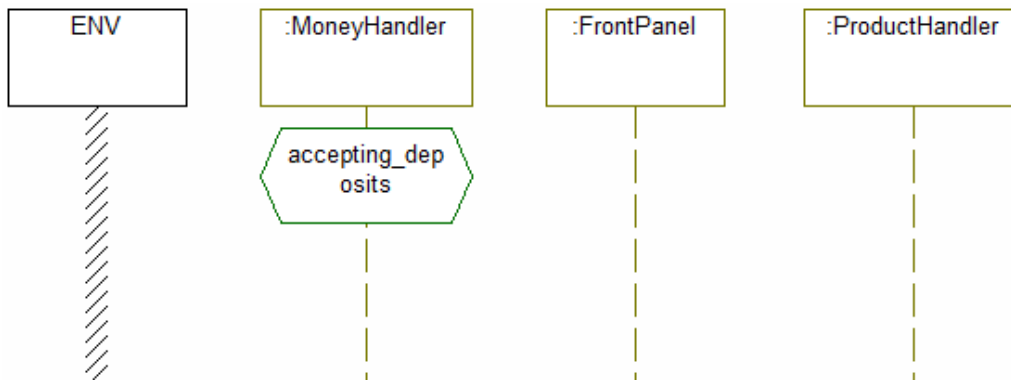


13. Click the **Go** button on the animation toolbar to start the execution.

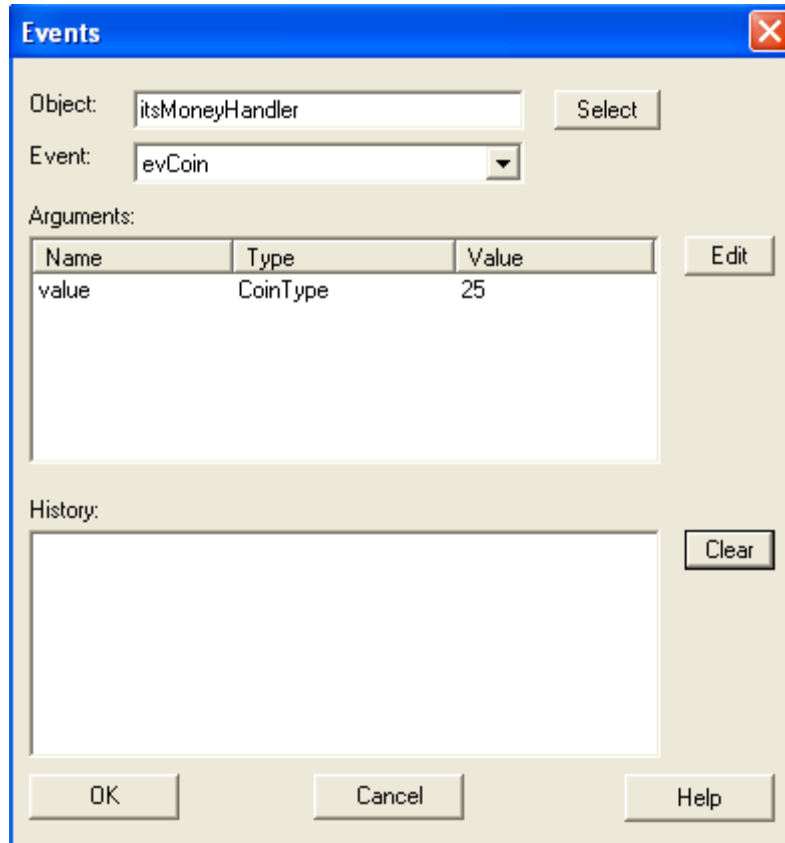
14. If it does not open automatically, open an animated **Money inserted** sequence diagram. To do this, select **Tools > Animated Sequence Diagram**. Select the **Money inserted** sequence diagram in the Design package. Note that if you are using Rhapsody Designer for Systems Engineers, it is called a simulated sequence diagram.
15. The animated sequence diagram has the lifelines of the Buyer inserts coin sequence diagram.
16. While the application is running, use the keyboard **Delete** key to delete the Buyer lifeline from the animated sequence diagram.



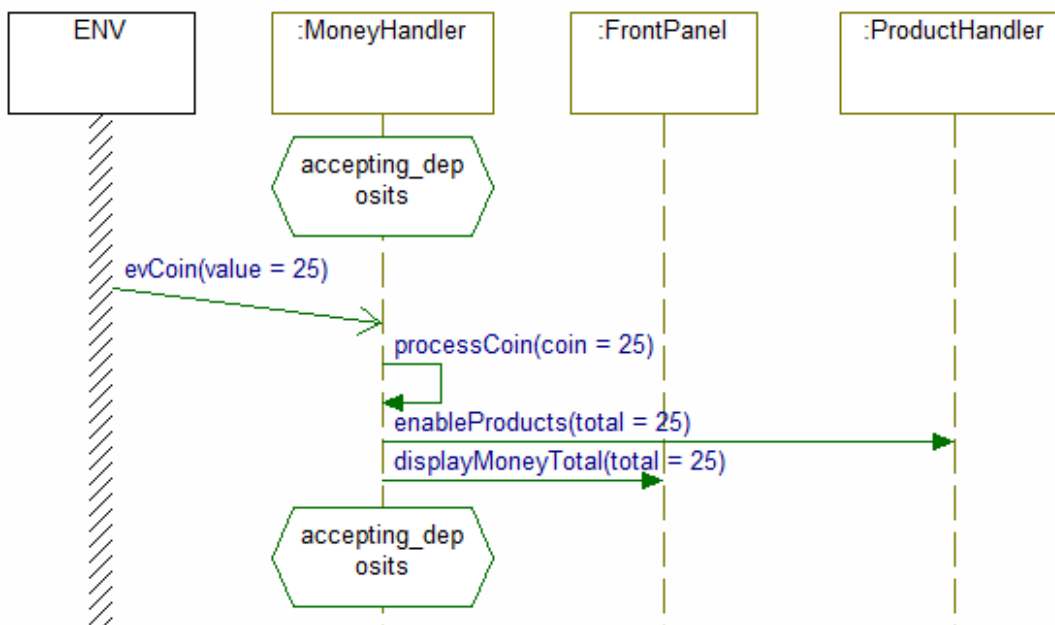
17. Add an Env lifeline to replace the Buyer lifeline only on the animated sequence diagram by clicking on the System border icon  System Border on the diagram tools and clicking in the diagram.



18. The model is in an idle state waiting for some operations to be called or events to be sent.
19. To generate the evCoin event, right-click the MoneyHandler lifeline in the animated sequence diagram, and select **Generate Event**.

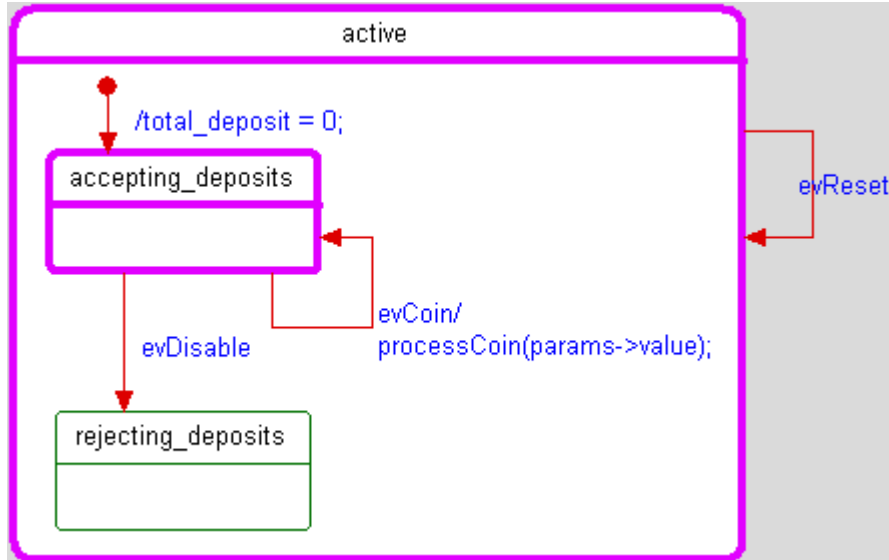


20. Select the `evCoin` event, and specify a value for the argument for `evCoin` by highlighting **value** and pressing **Edit**. Enter **25** as the number and click **Generate**. When the event is injected, the animated sequence diagram records the communication between the objects.

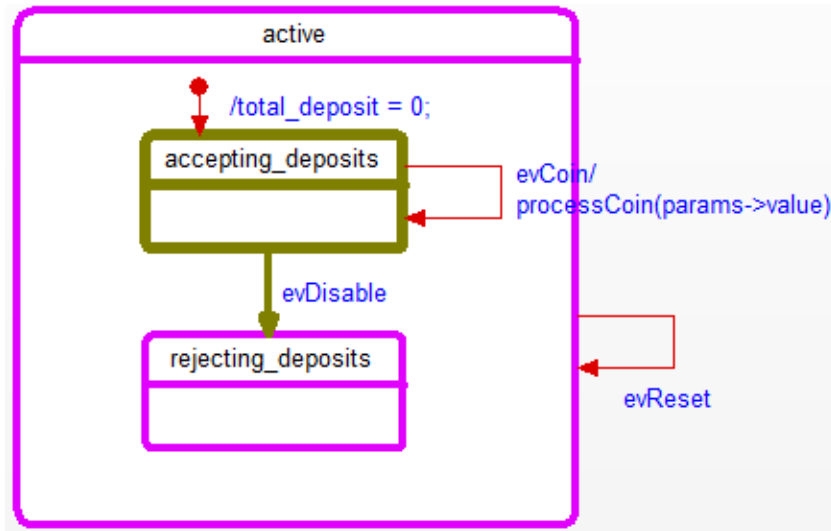



Note that the accepting deposits state is showing on the animated sequence diagram.

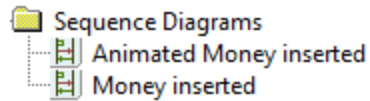
21. Reviewing the statechart explains why MoneyHandler called processCoin() in response to evCoin.
22. On the animated sequence diagram, right-click the MoneyHandler lifeline and select **Open animated Statechart**. Examine the animated statechart of the MoneyHandler object. The model is again idle and waiting for events.



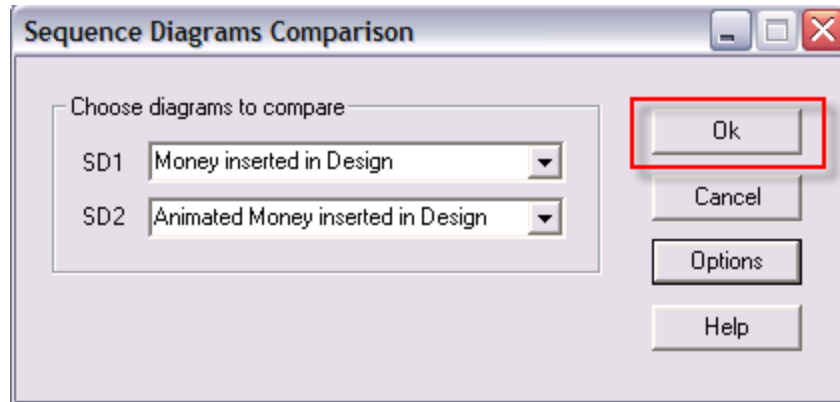
23. To generate the evDisable event, right-click the animated statechart, and select **Generate Event**. Confirm that MoneyHandler transitions to the rejecting_deposits state.



24. Stop  the animation.
25. Close the animated sequence diagram. When asked if you want to save it, click **Yes**. The animated sequence diagram is saved into the browser.



26. Select **Tools > Sequence Diagram Compare** and compare the animated results to see what happened.



Do your results show two significant differences?

- The `MoneyHandler::updateTotal()` operation is no longer needed since being replaced by `processCoin()`. You can delete the `updateTotal()` operation.
- The calls to `FrontPanel::displayMoneyTotal()` and `ProductHandler::enableProducts()` happened in reverse order.

27. Save the model. You do not need to save the animated diagrams.



Lab 6: Standard Ports

Objectives

After completing this lab, you will be able to:

- ▶ Add standard ports and contracts to the model

Scenario

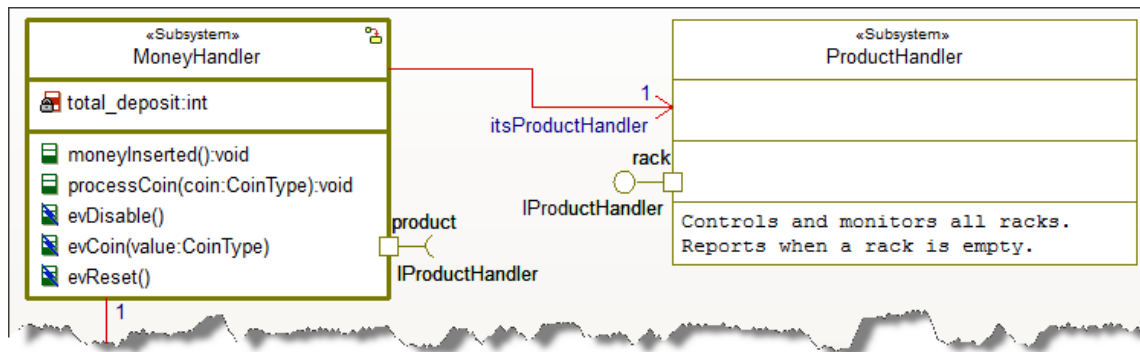
In this lab, some of the subsystem interfaces are formalized using standard ports. UML ports are named interaction points, and specify the provided and required services for the subsystem. Systems engineers commonly use ports to specify systems and subsystem interfaces, and software developers use ports to implement component-based development.

Task 1: Add a standard port with a contract that specifies a service

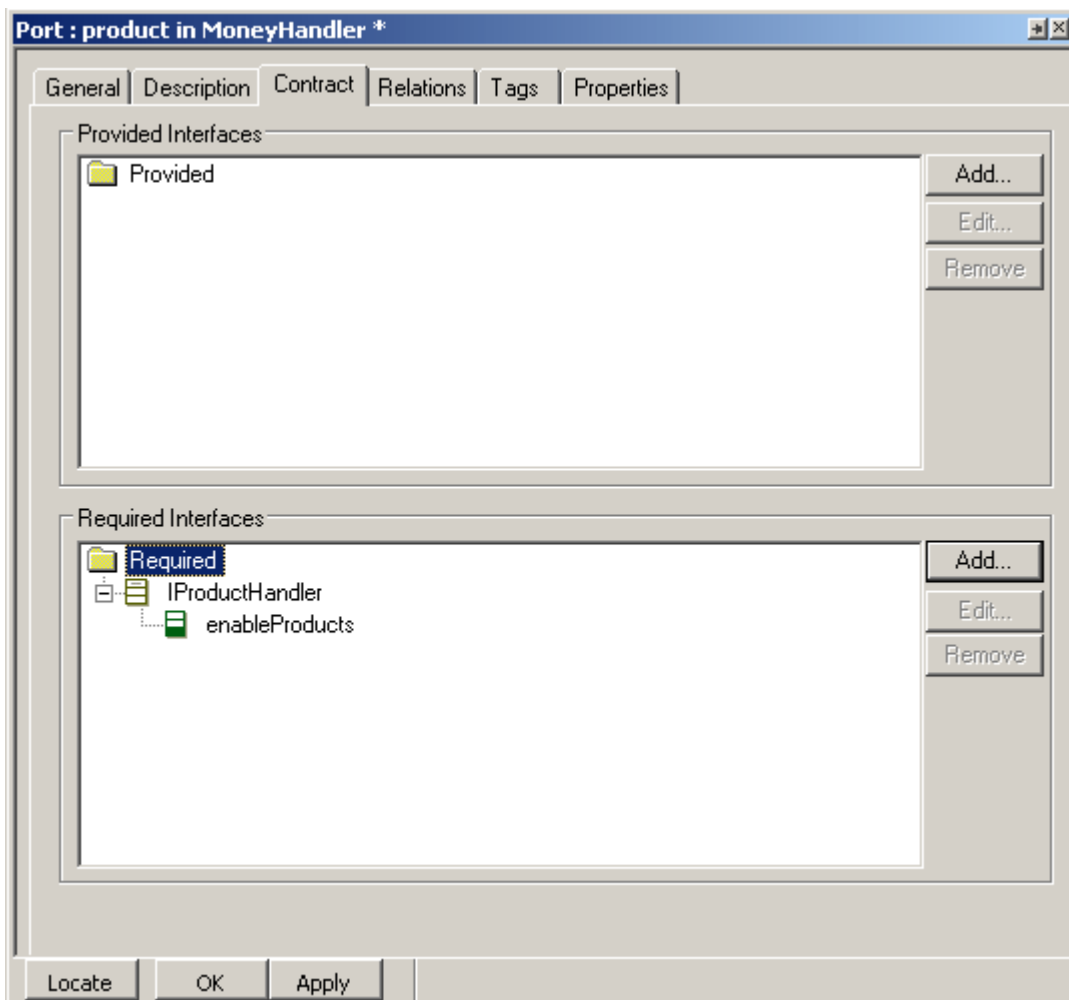
1. To add a new Interfaces package to the model, right-click the Packages category and select **Add New Package**.
2. Right-click the new Interfaces package and select **Add New>Interface**. Name it IProductHandler.
3. Move the enableProducts () operation from the ProductHandler class to the IProductHandler interface.



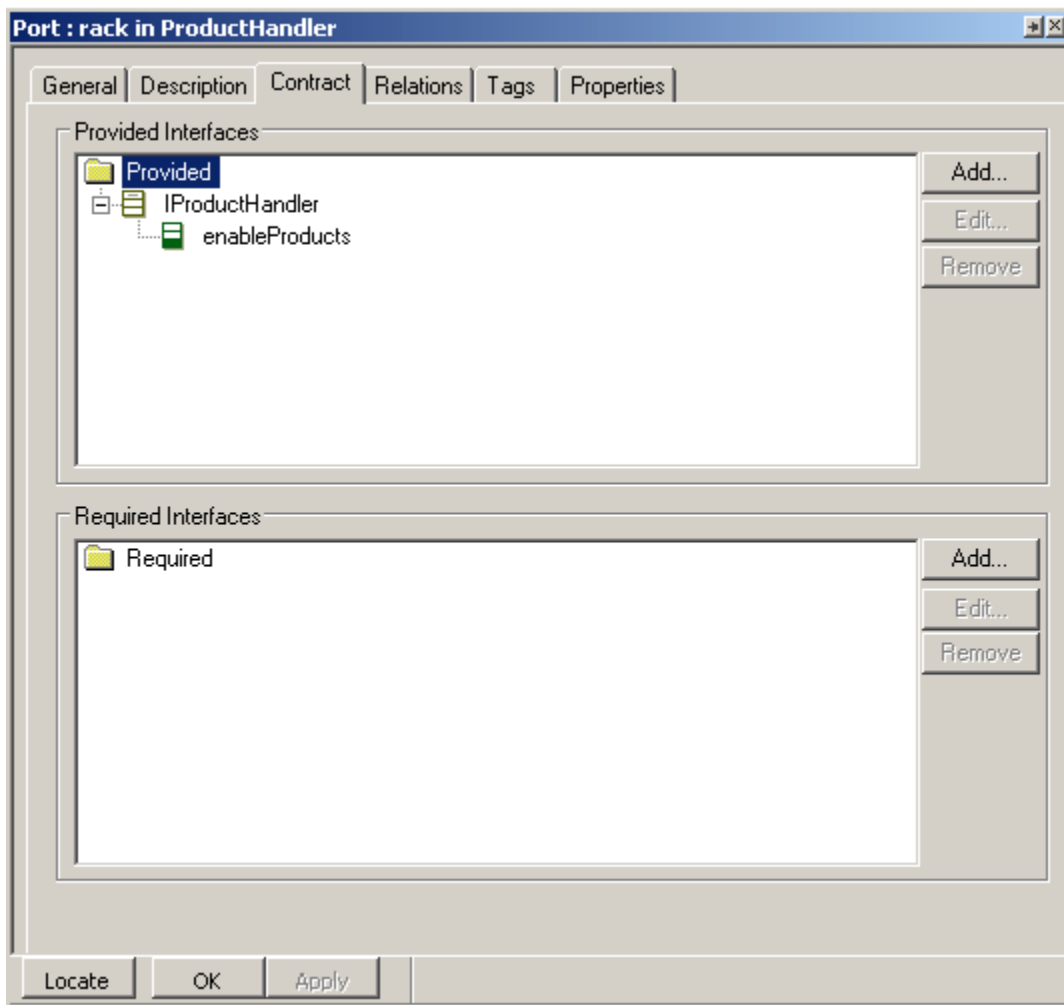
4. In the **Subsystem Overview** Object Model diagram, add ports with contracts as shown in the following figure.



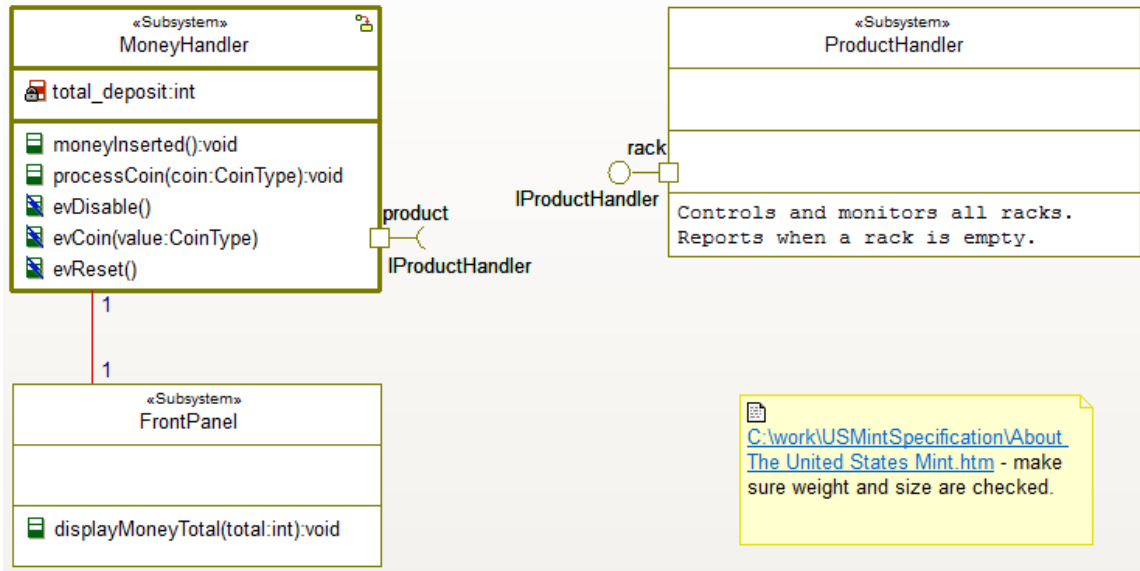
- a. Use the port icon to draw a port on the MoneyHandler. Name it product. In the **Features > Contract** tab, add the iProductHandler as a **Required** interface .



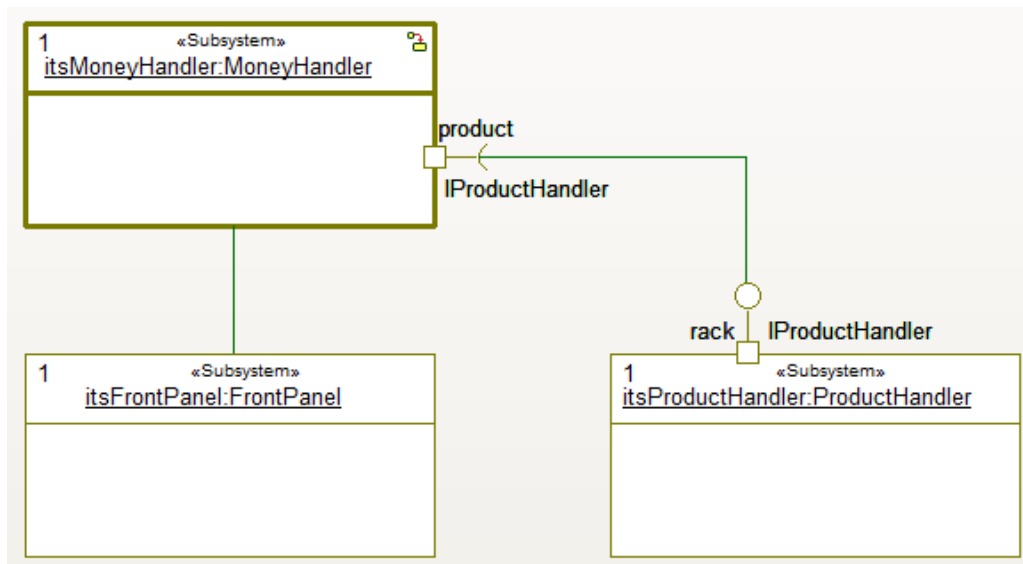
- b. Use the port icon to draw a port on the ProductHandler. Name it rack. In the **Features > Contract** tab add the IProductHandler as a **Provided** interface.



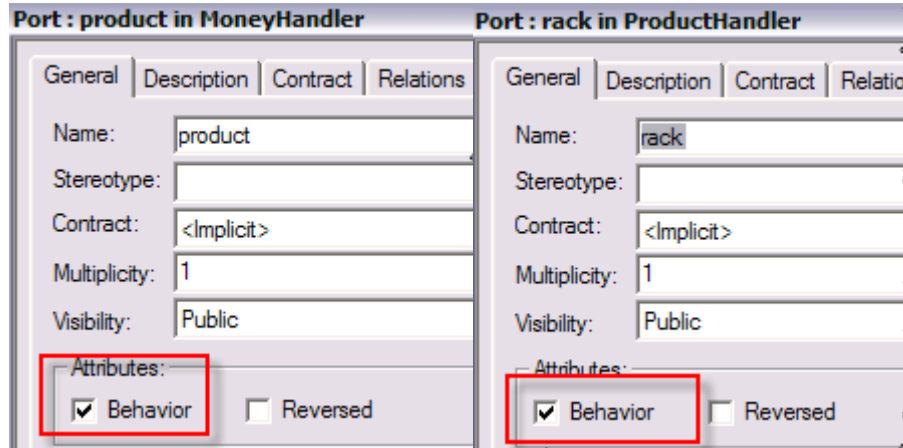
5. The unidirectional association between the classes **MoneyHandler** and **ProductHandler** is no longer needed. **Delete from Model** only this unidirectional association. The symmetric association between **MoneyHandler** and **FrontPanel** is still needed.



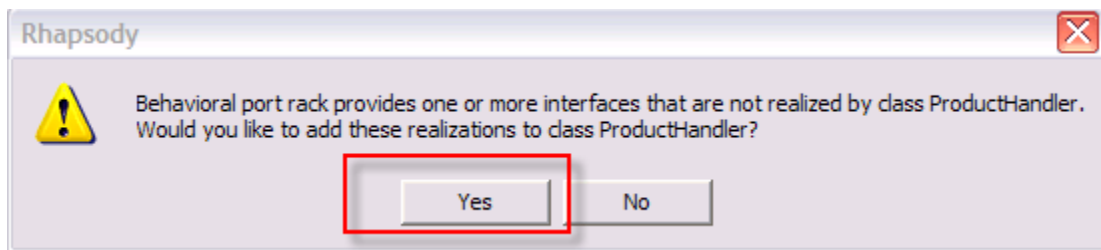
6. Close the **Subsystem Overview** Object Model diagram, and open the **Build Overview** Object Model diagram.
 - a. Move the ports as needed.
 - b. **Delete from Model** the uni-directional link between objects **itsMoneyHandler** and **itsProductHandler**.
 - c. Add a new link between the ports on **itsMoneyHandler** and **itsProductHandler**.



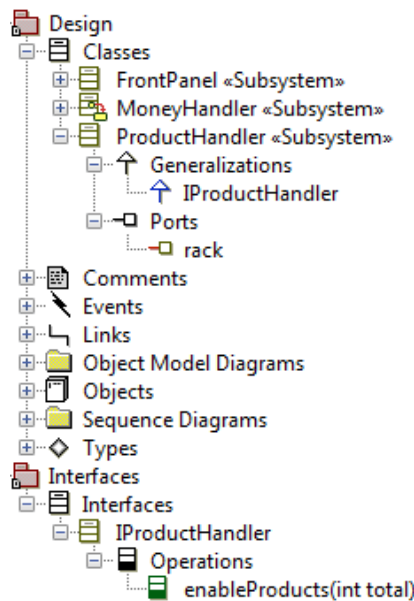
7. In the browser, click both ports (rack and product) and make them behavioral by selecting **Features > General > Attributes > Behavior** (behavioral means the services are provided by the class or block that contains the port).



- When you are prompted with a question, click **Yes** to add a realization of the interface to the `ProductHandler` class.



- The `ProductHandler` class now inherits from the interface **`IProductHandler`**.

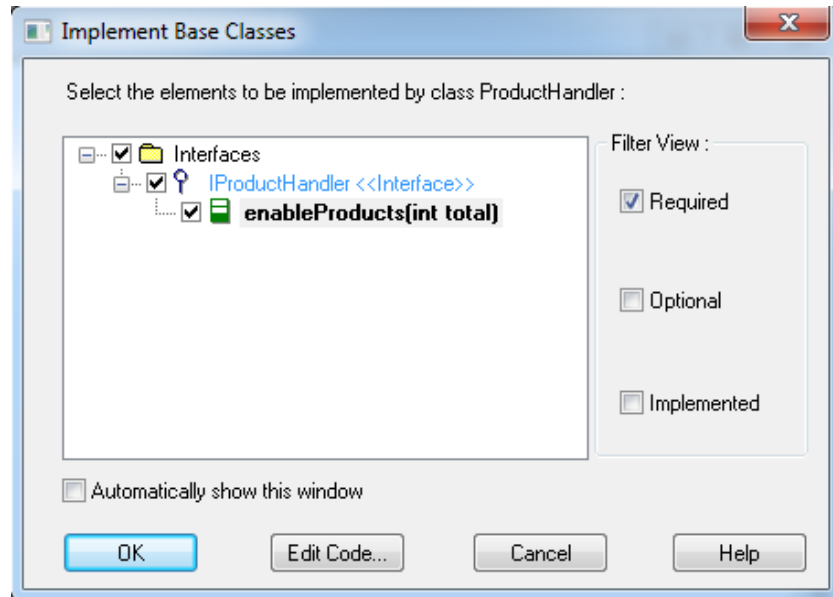


The interface `IProductHandler` is the base class of `ProductHandler`. `IProductHandler` declares an abstract operation `enableProducts (int)`.

- To add an implementation of `enableProducts (int)` to the `ProductHandler` class, right-click the `ProductHandler` class and select **Realize Base Classes**.

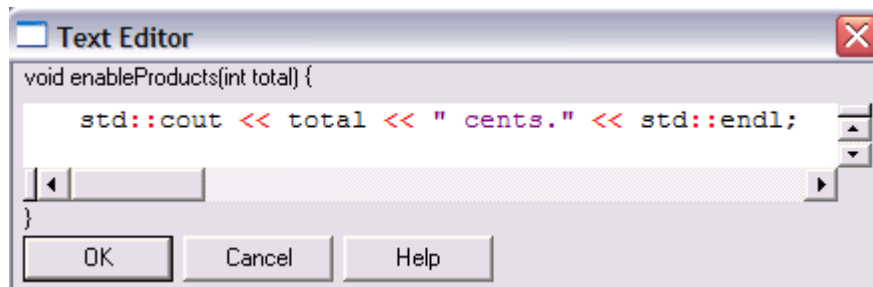
A dialog is invoked to allow you to implement the abstract operations defined in the interface classes.

11. Select the elements to be implemented by the class **ProductHandler** by setting the filter to **Required**, and then selecting the radio button for `enableProducts()`. The **Edit Code** button is now available, and you click **Edit Code** to invoke the editor.



12. Add the following code in the editor:

```
std::cout << total << " cents." << std::endl;
```



13. Click OK and OK again after you enter the code.

14. Now that the `itsMoneyHandler` communicates with `itsProductHandler` through a port, the `processCoin` operation's implementation must be edited. Open the **Features > Implementation** tab for `MoneyHandler::processCoin()`. Change the 2nd source line to use the product port.

```
total_deposit = total_deposit + coin;
```

```
//Enable racks for all products that can be afforded
```

```
OUT_PORT(product)->enableProducts(total_deposit);
```

```
//Display how much money has been deposited
```

```
itsFrontPanel->displayMoneyTotal(total_deposit);
```

```

Primitive Operation : processCoin in MoneyHandler
General Description Implementation Arguments Relations Tags Properties
void processCoin(const CoinType& coin)


    total_deposit = total_deposit + coin;

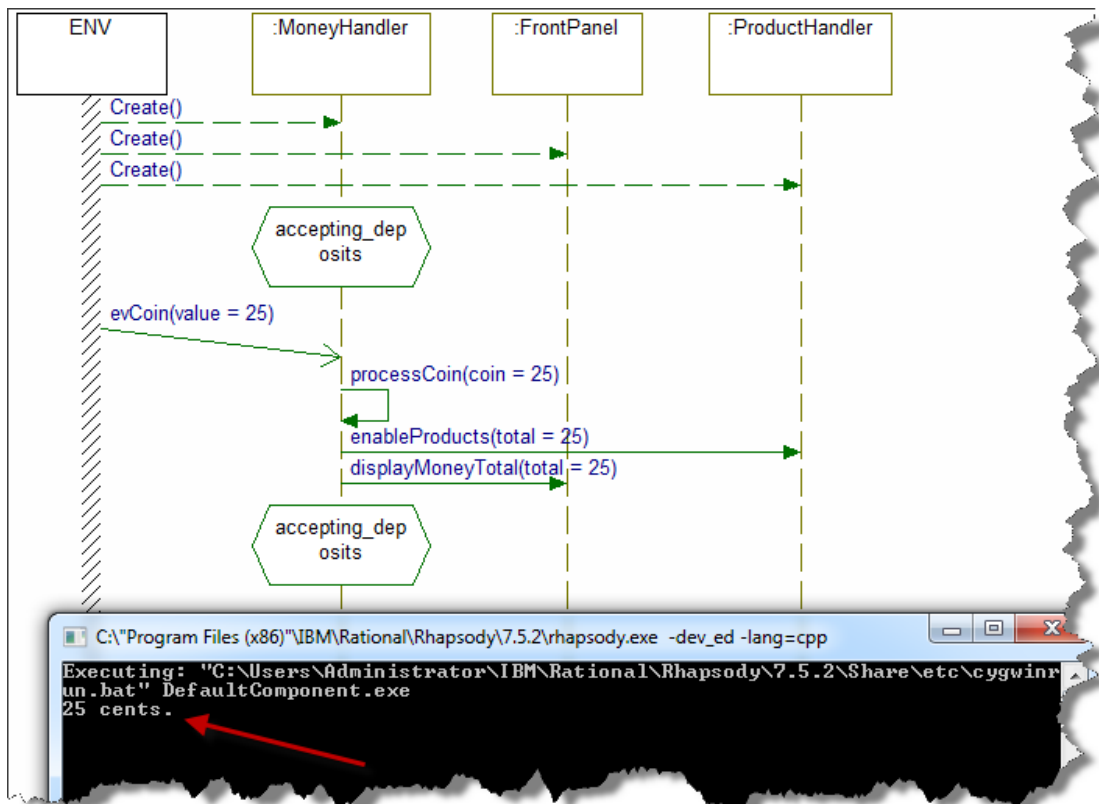
    //Enable racks for all products that can be afforded
    OUT_PORT (product) ->enableProducts (total_deposit);

    //Display how much money has been deposited
    itsFrontPanel->displayMoneyTotal (total_deposit);
    
```

15. Save the model.

16. Regenerate, Make, and Run the model. Open the animated **Money Inserted** sequence diagram. Click **Go**

 on the animation toolbar. On the animated **Money Inserted** sequence diagram right-click the **MoneyHandler** lifeline and inject the evCoin event with a value of 25. The animated sequence diagram should be the same as it was without ports. The console window should now display 25 cents., showing that the enableProducts () operation was called via the rack port on the itsProductHandler object.



17. Save and close the model.

You have now completed the hands-on lab exercise. You have used ports as named interaction points that are typed by interfaces. Ports specify the inputs and outputs of the port's class (subsystem). Ports provide a compact graphical notation, and are particularly useful for clearly defining the interfaces of subsystems and software components. In lab 6, ports of the same interface type were added to both the MoneyHandler and ProductHandler subsystems. MoneyHandler and ProductHandler are not dependent on one another. ProductHandler could be replaced by another subsystem as long as the other subsystem provides the same services as defined by the IProductHandler interface. The ability to swap in new components is the benefit of component-based development.