► ► ► **Module 11**
Use-Case Design

Mastering Object-Oriented Analysis
and Design with UML
Module 11: Use-Case Design

Topics

Objectives: Use-Case Design

Objectives: Use-Case Design

- Define the purpose of Use-Case Design and when in the lifecycle it is performed
- Verify that there is consistency in the use-case implementation
- Refine the use-case realizations from Use-Case Analysis using defined Design Model elements

Rational®
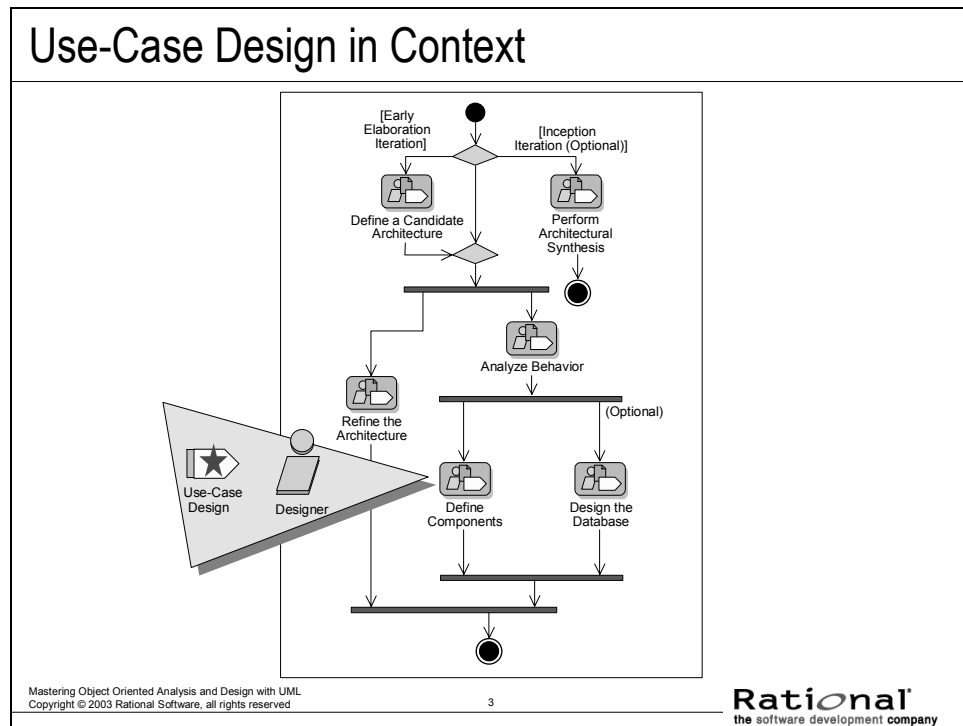the software development company

Use-Case Design is when the use-case implementations are verified for consistency. This means that for all the use-case realizations for each use case, the following items are verified:

- All the necessary behavior to support a use-case implementation has been distributed among the appropriate participating classes.

- The use case flows naturally over the participating design elements.

- All associations between design elements (classes or subsystems) needed for the use-case realizations have been defined.

- All of the attributes needed for the use-cases have been defined.

To assist in this verification, use-case realizations initially developed in Use-Case Analysis are refined to include the defined Design model elements from the originally defined Analysis Model elements. The Design Model elements are the design classes and subsystems that the analysis classes "morphed into."

In addition, any applicable architectural mechanisms should be incorporated into the use-case realizations.

Use-Case Design in Context

Use-Case Design in Context

[Early Elaboration Iteration]

[Inception Iteration (Optional)]

Define a Candidate Architecture

Perform Architectural Synthesis

Analyze Behavior

Refine the Architecture

(Optional)

Use-Case Design

Designer

Define Components

Design the Database

Mastering Object Oriented Analysis and Design with UML
Copyright © 2003 Rational Software, all rights reserved

3

Rational
the software development **company**

As you may recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process.
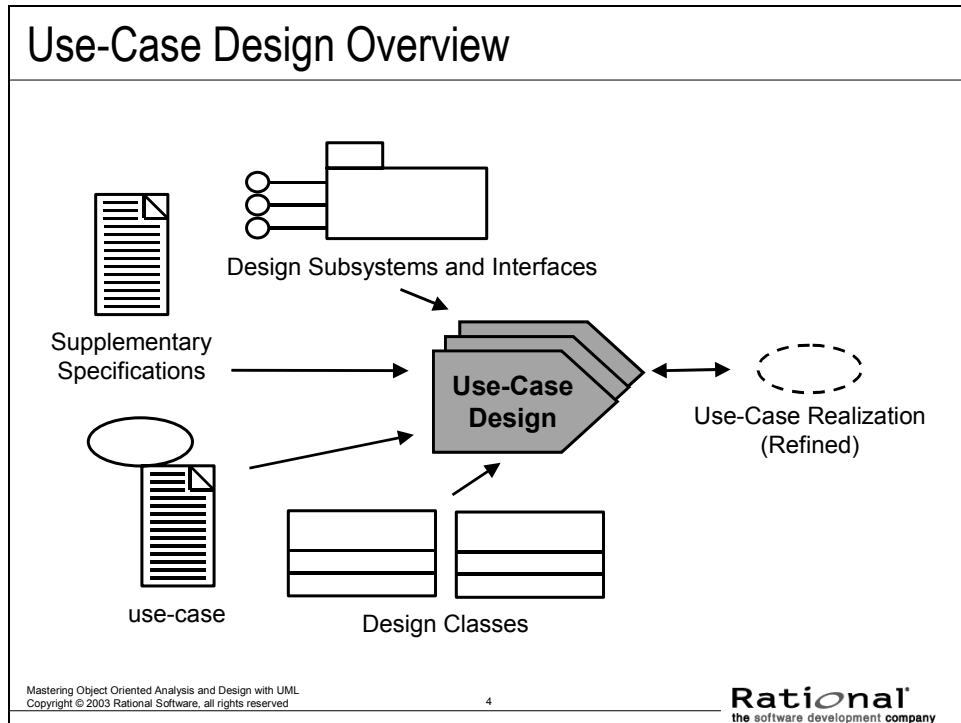
At this point, you have made an initial attempt at defining the architecture. You have defined the major elements of your system (that is, the subsystems, their interfaces, the design classes, the processes and threads) and their relationships, and you have an understanding of how these elements map into the hardware on which the system will run.

In **Use-Case Design**, you are going to concentrate on how a use-case has been implemented and make sure that there is consistency from beginning to end. You will also be verifying that nothing has been missed (that is, you will make sure that what you have done in the previous design activities is consistent with regards to the use-case implementation).

Use-Case Design is where the design elements (design classes and subsystems) meet the architectural mechanisms. The use-case realization initially defined in Use-Case Analysis is refined to include the design elements, using the patterns of interaction defined for the architectural mechanisms.

You might need to do some **Use-Case Design** before Subsystem Design, because after Analysis and Identify Design Elements, you usually only have sketchy notions of responsibilities of classes and subsystems. The real details need to get worked out in **Use-Case Design**, before you will be really ready to design the classes and subsystems. The detailed design activities (for example, Subsystem Design, Class Design and Use-Case Design) are tightly bound and tend to alternate between one another.

Use-Case Design Overview

Use-Case Design Overview

Design Subsystems and Interfaces

Supplementary
Specifications

**Use-Case
Design**

Use-Case Realization
(Refined)

use-case

Design Classes

Rational
the software development company

Use-Case Design is performed by the designer, once per use case.

Purpose:

- To refine use-case realizations in terms of interactions.

- To refine requirements on the operations of design classes.

- To refine requirements on the operations of design subsystems and/or their interfaces.

Input Artifacts:

- Supplementary Specifications

- Use cases

- Use-case realizations

- Design classes

- Design subsystems

- Interfaces

Resulting Artifacts:

- Use-case realizations, described with class diagrams, interaction diagrams, and flow of events refined.

Use-Case Design Steps

Use-Case Design Steps

- ◆ Describe interaction among design objects
- ◆ Simplify sequence diagrams using subsystems
- ◆ Describe persistence-related behavior
- ◆ Refine the flow of events description
- ◆ Unify classes and subsystems
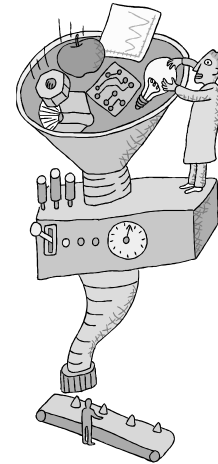
Mastering Object Oriented Analysis and Design with UML
Copyright © 2003 Rational Software, all rights reserved   5

Rati∘nal
the software development **company**

The above slide shows the major steps of the **Use-Case Design** activity.

Use-case realizations evolve from interactions among analysis classes to interactions among design elements (for example, design classes and subsystems).

The purpose of **Use-Case Design** is to make sure these are consistent and make sense. During **Use-Case Design**, the focus is on the use case, and this includes crossing subsystem boundaries. The use-case designer needs to make sure the use case is implemented completely and consistently across the subsystems.

Anything done from the use-case perspective is part of **Use-Case Design**.

This module will address each of these steps listed on the slide.

Describe Interaction Among Design Objects

Use-Case Design Steps

☆ ◆ Describe interaction among design objects
- ◆ Simplify sequence diagrams using subsystems
- ◆ Describe persistence-related behavior
- ◆ Refine the flow of events description
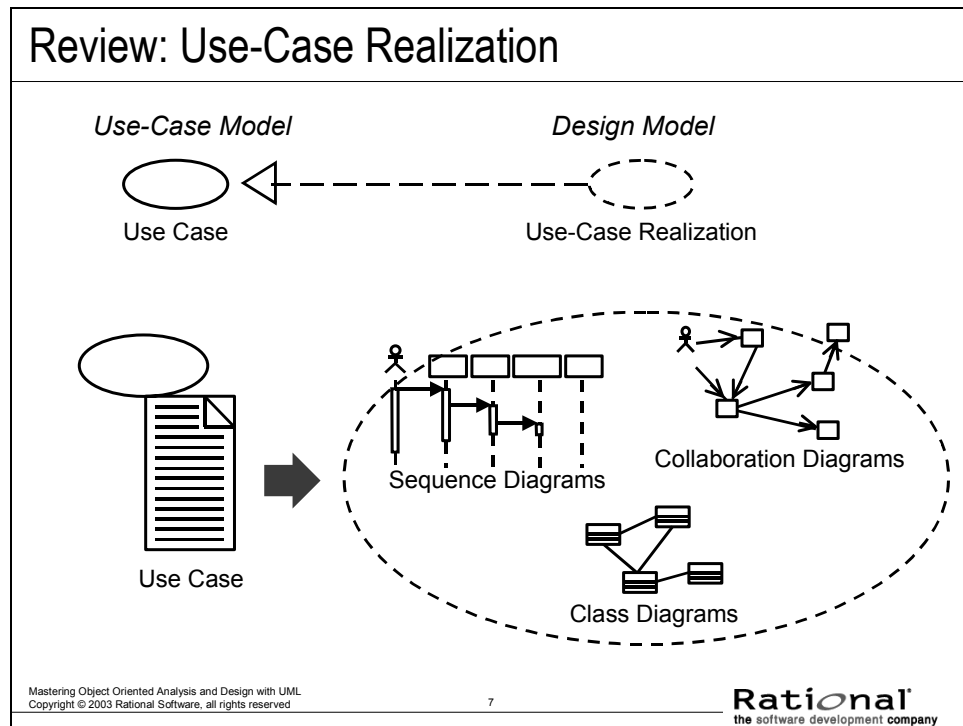- ◆ Unify classes and subsystems

6

Rational
the software development company

In this step, describe the use-case flow of events in terms of the Design Model (that is, interactions between design classes and/or subsystems). This involves replacing analysis classes with the design elements that they were refined into during Identify Design Elements, as well as incorporating any applicable architectural mechanisms.

Review: Use-Case Realization

Review: Use-Case Realization

Use-Case Model   *Design Model*

Use Case   Use-Case Realization

Sequence Diagrams
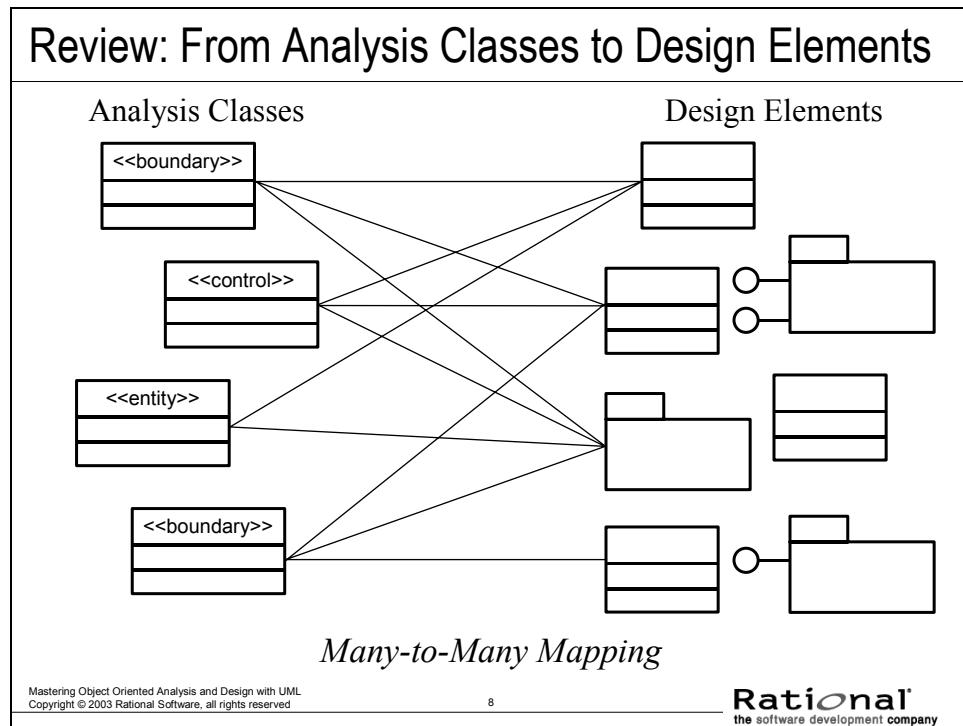
Collaboration Diagrams

Use Case

Class Diagrams

Before discussing the steps of the **Use-Case Design** activity, it is important to review what a use-case realization is, since refining a use-case realization is the focus of the activity.

Use-case realizations were first introduced in the Analysis and Design Overview module. The initial development of use-case realizations was discussed in the Use-Case Analysis module. The above diagram is included here as a review. For details on the use-case realization, refer to that module.

As stated in that module, a designer is responsible for the integrity of the use-case realization. He or she must coordinate with the designers responsible for the design elements (that is, classes and subsystems) and the relationships employed in the use-case realization.

Review: From Analysis Classes to Design Elements

Review: From Analysis Classes to Design Elements

Analysis Classes   Design Elements

<<boundary>>

<<control>>

<<entity>>

<<boundary>>

Many-to-Many Mapping

Mastering Object Oriented Analysis and Design with UML
Copyright © 2003 Rational Software, all rights reserved   8

Rational
the software development company

Before moving forward in **Use-Case Design,** it is important that you revisit what happened to the analysis classes that were identified in Use-Case Analysis. These analysis classes were mapped to design elements that were identified by the architect during the Identify Design Elements activity. You will be incorporating these design elements into your models in **Use-Case Design**.

Identify Design Elements is where the analysis classes identified during Use-Case Analysis are refined into design elements (for example, classes or subsystems). Analysis classes primarily handle functional requirements and model objects from the "problem" domain; design elements handle nonfunctional requirements and model objects from the "solution" domain.

It is in Identify Design Elements that you decide which analysis "classes" are really classes, which are subsystems (that must be further decomposed), and which are existing components and do not need to be "designed" at all.

Once the design classes and subsystems have been created, each must be given a name and a short description. The responsibilities of the original analysis classes should be transferred to the newly created subsystems. In addition, the identified design mechanisms should be linked to design elements.

Use-Case Realization Refinement

Use-Case Realization Refinement

- ◆ Identify participating objects
- ◆ Allocate responsibilities among objects
- ◆ Model messages between objects
- ◆ Describe processing resulting from messages
- ◆ Model associated class relationships
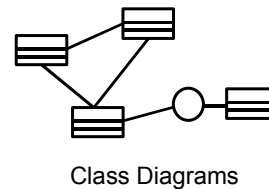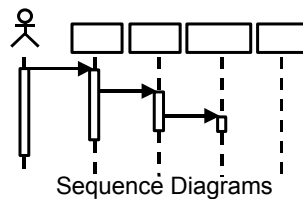
Sequence Diagrams   Class Diagrams

Rational
the software development **company**

Each use-case realization should be refined to describe the interactions between participating design objects as follows:

- Identify each object that participates in the use-case flow of events. These objects can be instances of design classes and subsystems, or they can be instances of actors that the participating objects interact with.

- Represent each participating object in an interaction diagram. Subsystems can be represented by instances of the subsystem's interface(s).

- Illustrate the message-sending between objects by creating messages (arrows) between the objects. The name of a message should be the name of the operation invoked by it. For messages going to design classes, the operation is a class operation. For messages going to subsystems, the operation is an interface operation.

- Describe what an object does when it receives a message. This is done by attaching a script or note to the corresponding message. When the person responsible for an object's class assigns and defines its operations, these notes or scripts will provide a basis for that work.

For each use-case realization, illustrate the class relationships that support the collaborations modeled in the interaction diagrams by creating one or more class diagrams.

Use-Case Realization Refinement Steps

Use-Case Realization Refinement Steps

- ◆ Identify each object that participates in the flow of the use case
- ◆ Represent each participating object in a sequence diagram
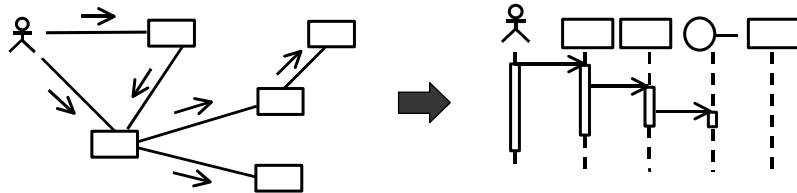
- ◆ Incrementally incorporate applicable architectural mechanisms

Rational®
the software development company

Look at the interaction diagrams.

For each class that has been refined into a subsystem, replace the class with the associated subsystem interface. Any interactions that describe *how* the subsystem should implement the service should be deferred until subsystem design.
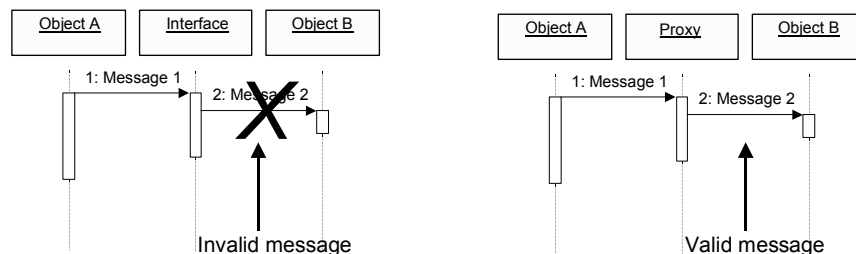
Incrementally incorporate any applicable architectural mechanisms, using the patterns of behavior defined for them during the architectural activities. This may include the introduction of new design elements and messages.

Any updates need to be reflected in both the static and dynamic parts of the use-case realization (that is, the interaction diagrams and the VOPC class diagram).

Representing Subsystems on a Sequence Diagram
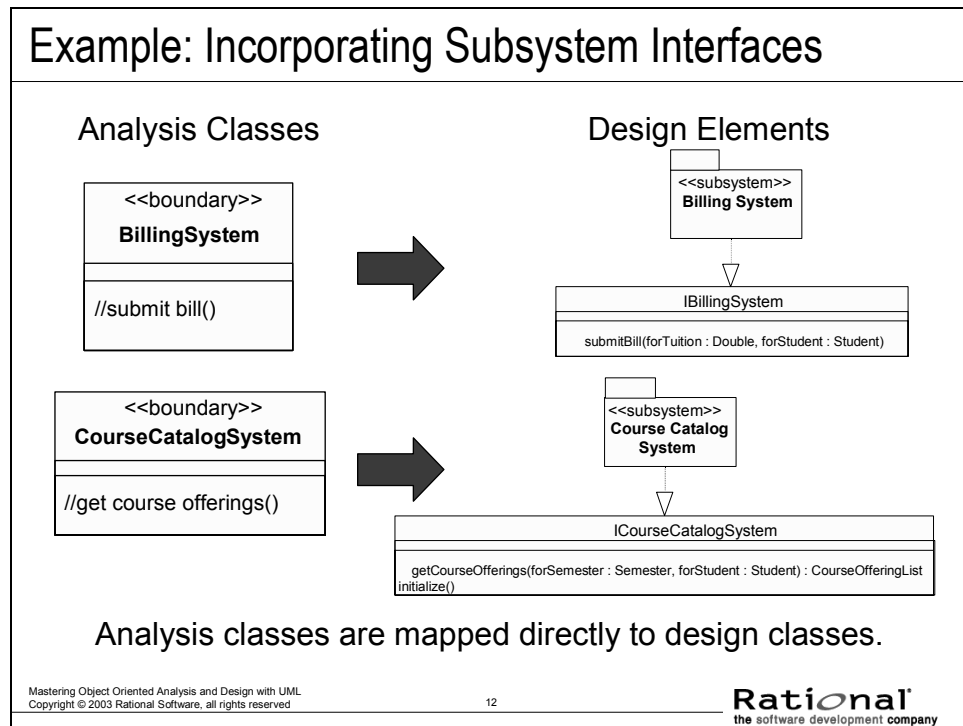

You have two choices for representing the subsystems:

- You can use the interfaces realized by the subsystem. This is the better choice in cases where any model element that realizes the same interface can be used in place of the interface. If you choose to show interfaces on the sequence diagram, be aware that you will want to ensure that no messages are sent from the interface to other objects. The reason for this is that interfaces completely encapsulate the internal realization of their operations. Therefore, you cannot be certain that all model elements that realize the interface will in fact actually be designed the same way. So on sequence diagrams, no messages should be shown being sent from interfaces.

- You can use a proxy class (which is discussed in the Subsystem Design Module) to represent the subsystem on sequence diagrams. This proxy class is contained within the subsystem and is used to represent the subsystem in diagrams that do not support the direct use of packages and subsystems as behavioral elements. The proxy class should be used in cases where a specific subsystem responds to a message. Messages can be sent from the subsystem proxy to other objects.

For this class, you will use interfaces to represent subsystems in the design.

Example: Incorporating Subsystem Interfaces

Example: Incorporating Subsystem Interfaces

Analysis Classes

<<boundary>>
BillingSystem

//submit bill()

<<boundary>>
CourseCatalogSystem

//get course offerings()

Design Elements

<<subsystem>>
Billing System

IBillingSystem

submitBill(forTuition : Double, forStudent : Student)

<<subsystem>>
Course Catalog System

ICourseCatalogSystem

getCourseOfferings(forSemester : Semester, forStudent : Student) : CourseOfferingList
initialize()

Analysis classes are mapped directly to design classes.

In Identify Design Elements, it was determined by the architects of the Course Registration System that the interactions to support external system access were going to be more complex than could be implemented in a single class. Thus, subsystems were identified to encapsulate the access to these external systems. The above diagram includes these subsystems, as well as their interfaces.

The BillingSystem subsystem provides an interface to the external Billing System. It is used to submit a bill when registration ends and students have been registered in courses.

The CourseCatalogSystem subsystem encapsulates all the work that goes on in communicating to the legacy Course Catalog System. The system provides access to the unabridged catalog of all courses and course offerings offered by the university, including those from previous semesters.

All other analysis classes map directly to design classes in the Design Model.

Example: Incorporating Subsystem Interfaces (Before)
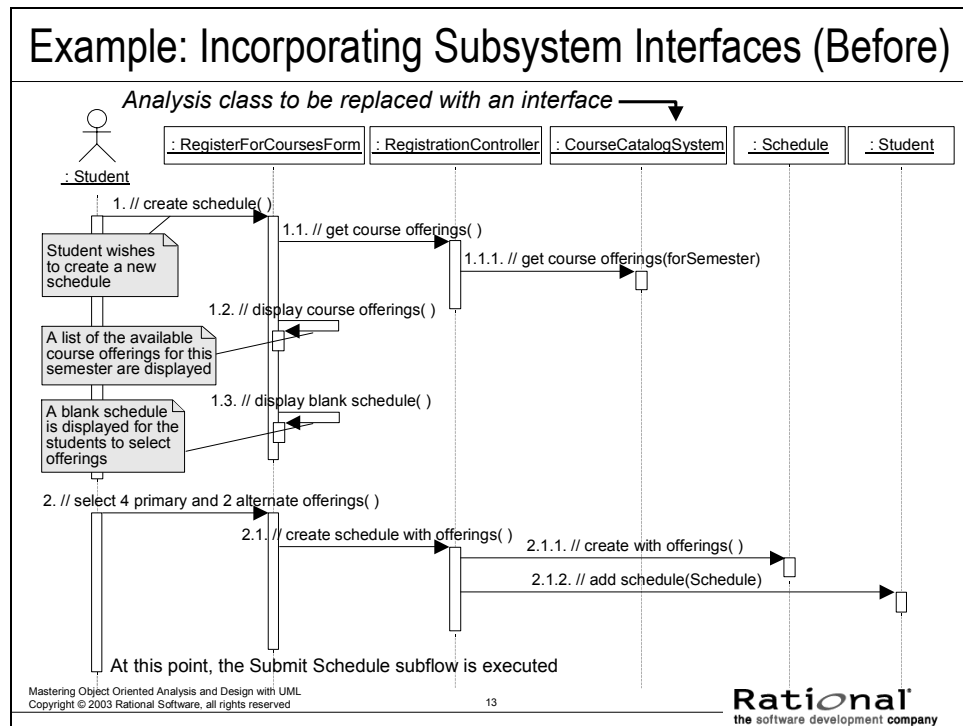
The above slide shows part of the Register for Courses use-case realization developed during Use-Case Analysis.

You know from Identify Design Elements that a CourseCatalogSystem subsystem has been defined to encapsulate access to the external legacy CourseCatalogSystem. Thus, you need to refine this interaction diagram and replace the original CourseCatalogSystem boundary class with the associated subsystem interface, ICourseCatalogSystem.

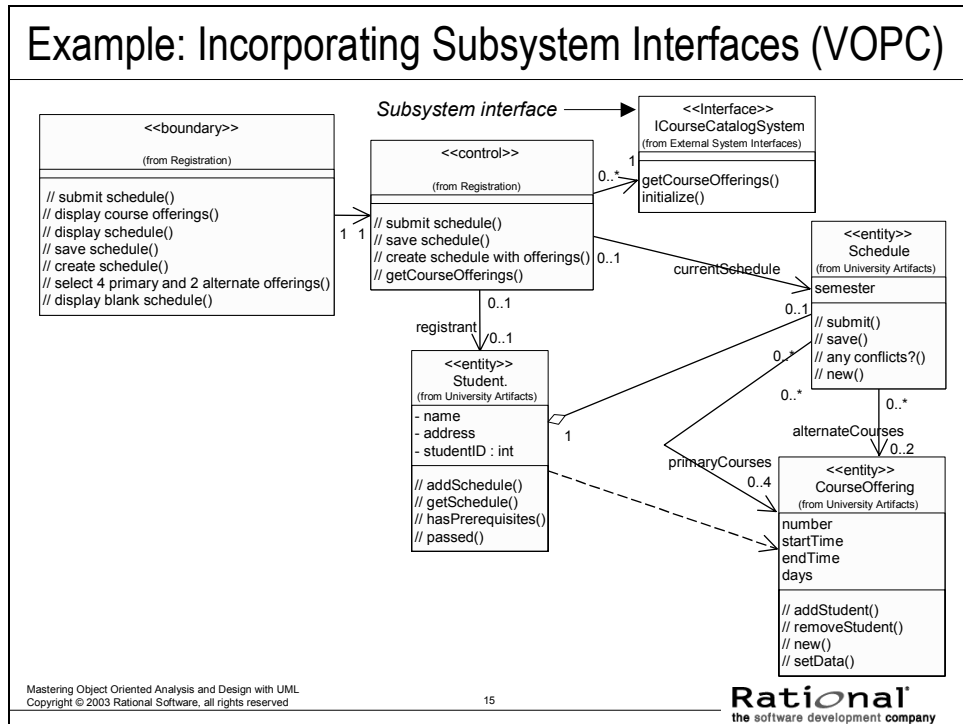## Example: Incorporating Subsystem Interfaces (After)
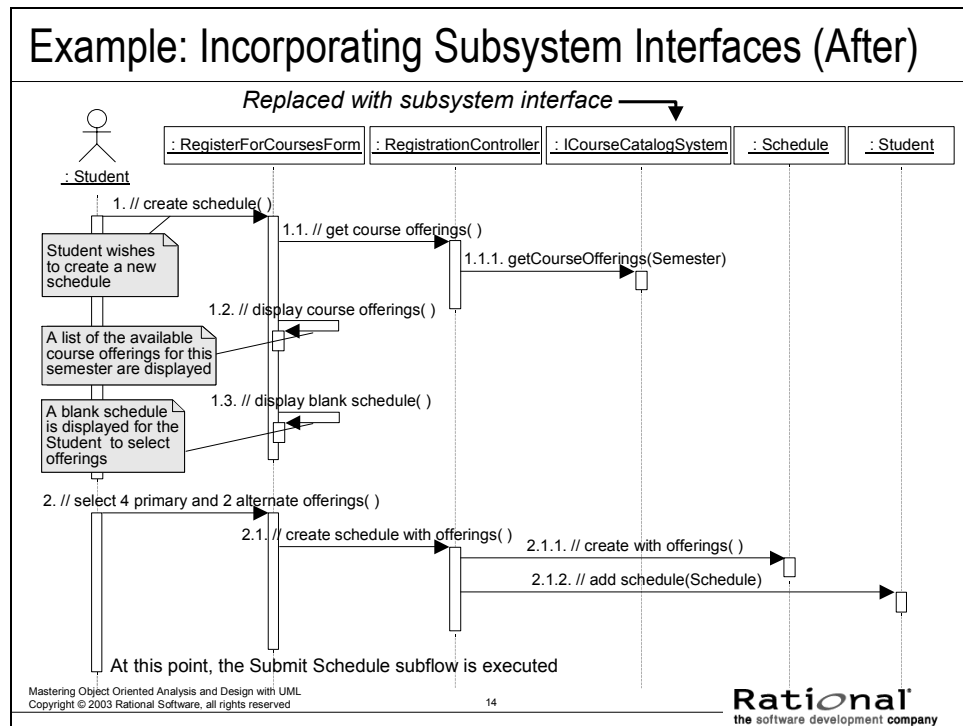
The above is a fragment of the sequence diagram from the Register for Courses use-case realization. It demonstrates how interactions are modeled between design elements, where one of the elements is a subsystem.

You know from Identify Design Elements that a CourseCatalogSystem subsystem was defined to encapsulate access to the external legacy CourseCatalogSystem. Thus, the original use-case realization (shown on the previous slide) was refined, and the original CourseCatalogSystem boundary class was replaced with the associated subsystem interface, ICourseCatalogSystem.

In this module, you will not flesh out the internals of the CourseCatalogSystem subsystem. That is the purpose of Subsystem Design. The above diagram will become the subsystem context diagram in Subsystem Design. In Subsystem Design, you will concentrate on the internals of the subsystem.

Example: Incorporating Subsystem Interfaces (VOPC)

Example: Incorporating Subsystem Interfaces (VOPC)

15

The above example is the VOPC after incorporating the design elements. The original CourseCatalogSystem boundary class has been replaced with the associated subsystem interface, ICourseCatalogSystem.

Incorporating Architectural Mechanisms: Security

<table>
<tr><td colspan="2">Incorporating Architectural Mechanisms: Security</td></tr>
<tr><td colspan="2">♦ Analysis-Class-to-Architectural-Mechanism Map from Use-Case Analysis</td></tr>
</table>

Analysis Class	Analysis Mechanism(s)
Student	Persistency, *Security*
Schedule	Persistency, *Security*
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution

Details are in the appendix.

Rational
the software development **company**

During Use-Case Analysis, applicable mechanisms for each identified analysis class were documented. This information, along with the information on what analysis classes became what design elements, allows the applicable mechanisms for a design element to be identified.

Since we have been concentrating on course registration, the above table contains only the classes for the Register for Courses use-case realization that have analysis mechanisms assigned to them.

The details of incorporating the security mechanism are provided in the Additional Information Appendix in the Security Mechanism section.

Incorporating Architectural Mechanisms: Distribution

Incorporating Architectural Mechanisms: Distribution

◆ Analysis-Class-to-Architectural-Mechanism
Map from Use-Case Analysis

Analysis Class	Analysis Mechanism(s)
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	*Distribution*

17

Rational
the software development **company**

We started the discussion of the distribution mechanism in the Describe Distribution module. Now we will see how to incorporate this mechanism into the use-case realizations.

Review: Incorporating RMI: Steps

Review: Incorporating RMI: Steps

- ◆ Provide access to RMI support classes (e.g., Remote and Serializable interfaces, Naming Service)
- √ ▪ *Use java.rmi and java.io package in Middleware layer*
- ◆ For each class to be distributed:
- √ ▪ *Controllers to be distributed are in Application layer*
- √ ▪ *Dependency from Application layer to Middleware layer is needed to access java packages*
  - ▪ Define interface for class that realizes Remote
  - ▪ Have class inherit from UnicastRemoteObject

√ - Done

Mastering Object Oriented Analysis and Design with UML
Copyright © 2003 Rational Software, all rights reserved   18

Rati○nal
the software development company

The next few slides contain a summary of the steps that can be used to implement the RMI distribution mechanism described in this module. The italicized text describes the architectural decisions made with regards to RMI for our Course Registration example.

These steps were first discussed in the Describe Distribution module. They are repeated here for convenience. The check marks indicate what steps have been completed.

In **Use-Case Design**, you will continue to incorporate this mechanism. You will define the distributed class interfaces, and the supporting generalization and realization relationships. As pointed out in the Describe Distribution module, for any class that is to be distributed, an interface must be defined that realizes the Java Remote interface. The distributed class will need to realize that interface, as well as inherit from UnicastRemoteObject.

As previously decided, for the Course Registration System, the control classes will be distributed. (The classes to be distributed were tagged with the analysis mechanism, distribution.) The interface classes that will be defined for the distributed control classes should be placed in the same package as the associated distributed control classes.

The remaining steps are discussed on the next two slides.

Review: Incorporating RMI: Steps (cont.)

Review: Incorporating RMI: Steps (cont.)

- ◆ Have classes for data passed to distributed objects realize the Serializable interface
  - √ ▪ *Core data types are in Business Services layer*
  - √ ▪ *Dependency from Business Services layer to Middleware layer is needed to get access to java.rmi*
    - ▪ Add the realization relationships
- ◆ Run pre-processor – out of scope

√ *- Done*

19

Rational
the software development company

- • Any class whose instances will be passed between the client and the server needs to realize the Serializable interface.
  For the Course Registration System, most of the data passed is of one of the core data types. The core data types were allocated to the Business Services layer of the architecture (specifically, the University Artifacts package) in Identify Design Elements. Thus, a dependency exists from the Business Services layer to the Middleware layer so the core data classes can access to Remote interface. Now we will define the realization relationships from the classes to be passed and the Serializable interface.

- • The developer must run the compiled distributed class through the rmic compiler provide by Sun to generate the stubs and skeletons for all classes that realize the Remote interface. These classes handle the communication that must occur to support distribution (see the previous slide). Once a class is run through RMIC, you can access it as if it were a local class; the client does not know the difference. This is really implementation, which is out of the scope of this course.

The remaining steps are discussed on the next slide.

Review: Incorporating RMI: Steps (cont.)

Review: Incorporating RMI: Steps (cont.)

♦ Have distributed class clients look up the remote objects using the Naming service

√ ▪ *Most Distributed Class Clients are forms*

√ ▪ *Forms are in Application layer*

√ ▪ *Dependency from Application layer to Middleware layer is needed to get access to java.rmi*

   ▪ Add relationship from Distributed Class Clients to Naming Service

♦ Create/update interaction diagrams with distribution processing (optional)

√ *- Done*

20

Rati♦**nal**
the software development **company**

Clients of distributed classes will need to lookup the location of the remote object using the Naming service. The look up returns a reference to the distributed class interface.

Now we will define the dependency relationships from the distributed class clients and the Naming Service. You will also develop interaction diagrams that model the distribution functionality.

Example: Incorporating RMI

Example: Incorporating RMI
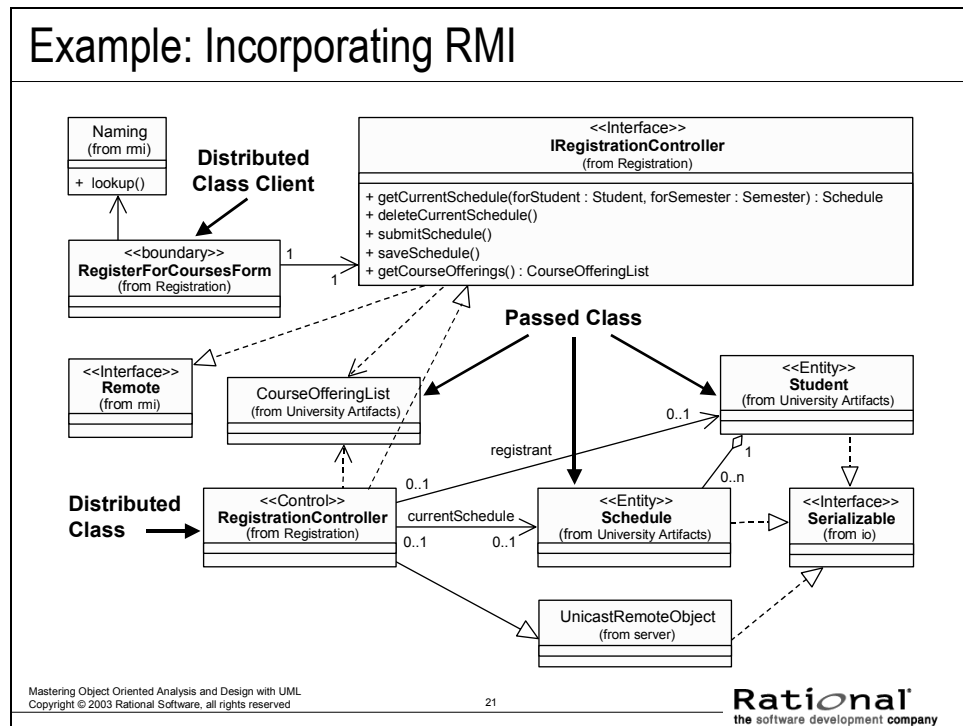
The above diagram provides a static view of the classes needed to incorporate the RMI distribution mechanism into the Course Registration System design.

The RegistrationController class is distributed, so an interface was defined, IRegistrationController, that realizes the Remote interface. The distributed class, RegistrationController realizes this new interface, and inherits from the UnicastRemoteObject.

Instances of the Student, Schedule, and CourseOfferingList classes are passed to and from the distributed class (note the operation signatures for the IRegistrationController interface), so they will realize the Serializable interface.

The RegisterForCoursesForm needs to look up the location of the RegistrationController using the Naming service, so a dependency was added from the RegisterForCoursesForm to Naming.

The remaining steps are discussed on the next two slides.

Example: Incorporating RMI (cont.)
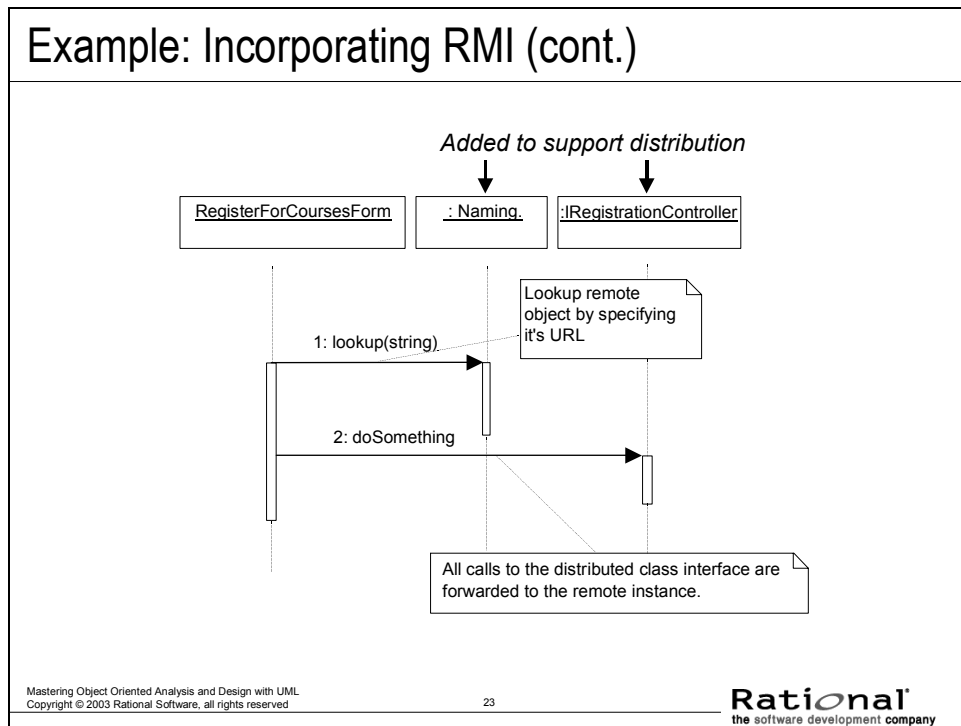

Example: Incorporating RMI (cont.)
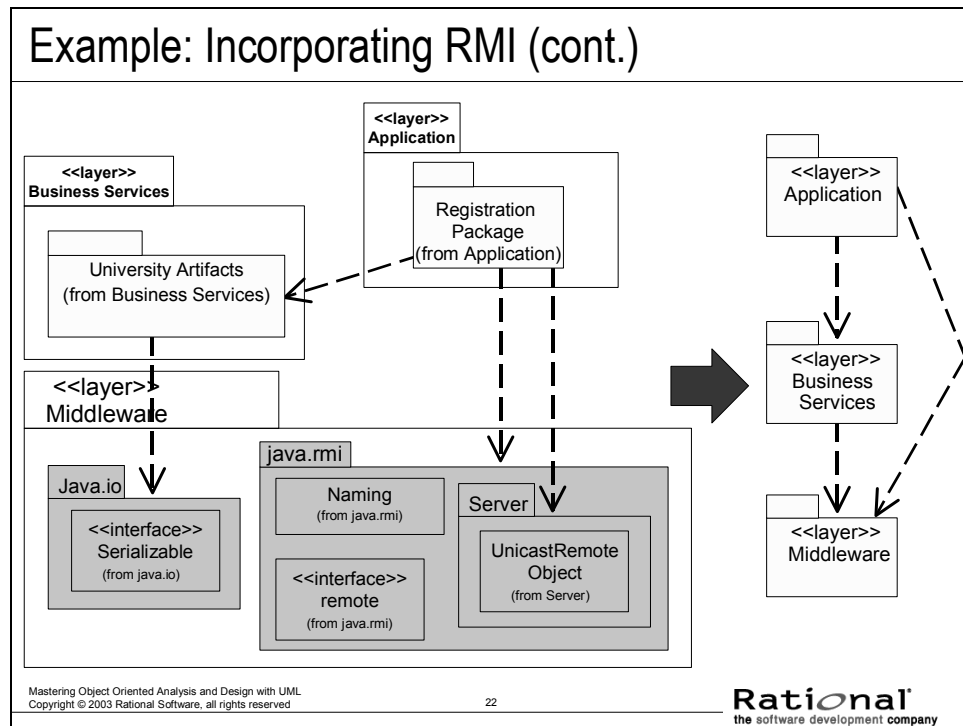
The above diagram describes the package dependencies needed to support the distribution pattern described in this module (and the class relationships shown on the previous slide). This diagram is very similar to the one included in the Describe Distribution module, but the generic Sample Application package has been replaced with the Registration package. The Registration package contains the RegistrationController class that needs to be distributed; the created IRegistrationController interface; and the distributed class client, the RegisterForCoursesForm class. As discussed in the Describe Distribution module, the following package dependencies were added to support distribution:

- The java.rmi package contains the classes that implement the RMI distribution mechanism. This package is commercially available with most standard Java IDEs.
- Dependency from the Application packages to java.rmi provides access to the Remote interface for distributed controller interfaces, and to the Naming service for the distributed controller clients.
- Dependency from the Application packages to the Java Server package provides access to the UnicastRemoteObject class for distributed controllers.
- Dependency from the University Artifacts package to java.io provides access to the Serializable interface for classes, whose instances must be passed for distributed objects.

The layer dependencies that support the package dependencies are shown on the right side of diagram.

Note: In the above diagram, only a subset of the packages are shown. The remaining packages have been omitted for clarity. The remaining steps are discussed on the next slide.

Example: Incorporating RMI (cont.)

Example: Incorporating RMI (cont.)

Added to support distribution

RegisterForCoursesForm	: Naming.	:IRegistrationController

Lookup remote object by specifying it's URL

1: lookup(string)

2: doSomething

All calls to the distributed class interface are forwarded to the remote instance.

23

Rational
the software development **company**

The above diagram provides an example of what you would include in an interaction diagram to model the distribution functionality.

Notice the addition of a call to the Naming utility to locate the distributed class instance as well as the replacement of the original RegistrationController control class with the IRegistrationController interface. (Naming returns a reference to an IRegistrationController.) The remainder of the interaction diagram remains the same as before the distribution mechanism was incorporated.

Simplify Sequence Diagrams Using Subsystems

Use-Case Design Steps

- ◆ Describe interaction among design objects
- ☆ ◆ Simplify sequence diagrams using subsystems
- ◆ Describe persistence-related behavior
- ◆ Refine the flow of events description
- ◆ Unify classes and subsystems

Rational
the software development company

When a use case is realized, the flow of events is usually described in terms of the executing objects, that is, as interactions between design objects. To simplify diagrams and to identify reusable behavior, there might be a need to encapsulate a subflow of events within a subsystem. When this is done, large subsections of the interaction diagram are replaced with a single message to the subsystem. Within the subsystem, a separate interaction diagram might illustrate the internal interactions within the subsystem that provide the required behavior. These subsystem interaction diagrams are developed during Subsystem Design.

At first glance, this step may appear similar to the previous one, Describe Interactions among Design Objects. However, they differ in perspective. In the case of Describe Interactions among Design Objects, the common subflows are identified outside-in. (Common collaborations have already been encapsulated within the subsystems identified in Identify Design Elements.) In the case of Simplify Interaction Diagrams Using Subsystems, the common subflows are discovered inside-out — after modeling the flows of events using design elements, you recognize common subflows. This step is optional if common subflows are not discovered.

Encapsulating Subsystem Interactions

Encapsulating Subsystem Interactions

- ◆ Interactions can be described at several levels
- ◆ Subsystem interactions can be described in their own interaction diagrams
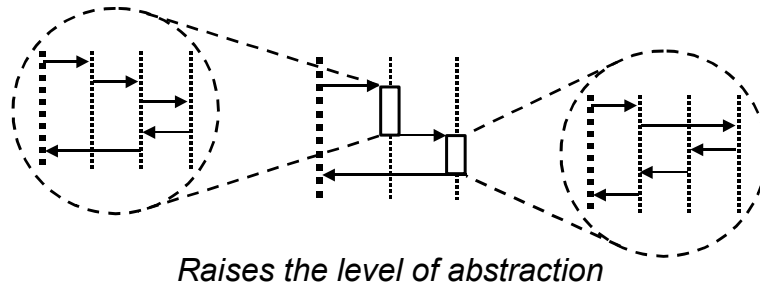
Raises the level of abstraction

Rati⊘nal
the software development company

A use-case realization can be described, if necessary, at several levels in the subsystem hierarchy.

In the above example, the lifelines in the middle diagram represent subsystems; the interactions in the circles represent the internal interaction of subsystem members in response to the message.

This approach raises the level of abstraction of the use-case realization flows of events.

The advantages of this approach are described on the next three slides.

When to Encapsulate Subflows in a Subsystem

When to Encapsulate Subflows in a Subsystem

Encapsulate a Subflow when it:
- Occurs in multiple use-case realizations
- Has reuse potential
- Is complex and easily encapsulated
- Is responsibility of one person or team
- Produces a well-defined result
- Is encapsulated within a single Implementation Model component
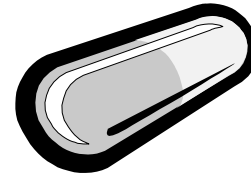
Ratio**nal**
the software development **company**

Encapsulate a subflow within a subsystem when it:

- Occurs repeatedly in different use-case realizations: Similar messages are sent to similar objects, providing the same end result. The word "similar" is used because some design work may be needed to make the behavior reusable.

- Occurs in only one use-case realization but it is expected to be performed repeatedly in future iterations, or in similar systems in the future. The behavior may make a good reusable component.

- Occurs in only one use-case realization but is complex and easily encapsulated, needs to be the responsibility of one person or a team, and provides a well-defined result. In these kinds of situations, the complex behavior usually requires special technical knowledge, or special domain knowledge. As a result, it is wellsuited to encapsulation within a subsystem.

- Is encapsulated within a component in the Implementation Model. In this case, a subsystem is the appropriate representation for the component within the Design Model.

Make sure that what you are abstracting is worth abstracting.

Guidelines: Encapsulating Subsystem Interactions

To achieve true substitutability of subsystems that realize the same interface, only their interfaces can be visible in the interaction diagrams; otherwise all diagrams will need to be changed whenever a subsystem is substituted for another.

On an interaction diagram, sending a message to an interface lifeline means that any subsystem that realizes the interface can be substituted for the interface in the diagram.

In many cases, the interface lifeline does not have messages going out from it, since different subsystems realizing the interface may send different messages. However, if you want to describe what messages should be sent (or are allowed to be sent) from any subsystem realizing the interface, such messages can originate from the interface lifeline.

With this approach, when describing the interactions, the focus remains on the services, not on how the services are implemented within the design elements. This is known as "Design by Contract" and is one of the core tenets of robust software development using abstraction and encapsulation mechanisms.

Describing how the services are implemented is the focus of Subsystem Design for the design subsystems and Class Design for the design classes.

Advantages of Encapsulating Subsystem Interactions

Advantages of Encapsulating Subsystem Interactions

Use-case realizations:
- Are less cluttered
- Can be created before the internal designs of subsystems are created (parallel development)
- Are more generic and easier to change (Subsystems can be substituted.)

28

Rational
the software development **company**

The advantages of encapsulating subsystem interactions over modeling the entire system at once are:

- Use-case realizations become less cluttered, especially if the internal design of some subsystems is complex.

- Use-case realizations can be created before the internal designs of subsystems are created. This can be used to make sure that use-case functionality has not been "lost" between the allocation of use-case responsibility in Use-Case Analysis and the identification of design elements (subsystems and design classes) in Identify Design Elements, and before Subsystem Design is performed.

- Use-case realizations become more generic and easier to change, especially if a subsystem needs to be substituted for another subsystem.

Encapsulating subsystem interactions raises the level of abstraction of the use-case realization flows of events.

Parallel Subsystem Development

Parallel Subsystem Development

- ◆ Concentrate on requirements that affect subsystem interfaces
- ◆ Outline required interfaces
- ◆ Model messages that cross subsystem boundaries
- ◆ Draw interaction diagrams in terms of subsystem interfaces for each use case
- ◆ Refine the interfaces needed to provide messages
- ◆ Develop each subsystem in parallel

Use subsystem interfaces as synchronization points

Rational
the software development company

In some cases, it is appropriate to develop a subsystem more or less independently and in parallel with the development of other subsystems. To achieve this, we must first find subsystem dependencies by identifying the interfaces between them.

This work can be done as follows:

1. Concentrate on the requirements that affect the interfaces between the subsystems.

2. Make outlines of the required interfaces, showing the messages that are going to pass over the subsystem borders.

3. Draw interaction diagrams in terms of subsystem interfaces for each use case.

4. Refine the interfaces needed to provide messages.

5. Develop each subsystem in parallel using the interfaces as synchronization instruments between development teams.

You can also choose whether to arrange the interaction diagrams in terms of subsystems or in terms of their interfaces only. In some projects, it might even be necessary to implement the classes providing the interfaces before you continue with the rest of the modeling.

The detailed design of the subsystem "internals" is done during Subsystem Design. The interfaces are what ensure compatibility between the Use-Case Design and the Subsystem Design.

Describe Persistence Related Behavior

Use-Case Design Steps

- ◆ Describe interaction among design objects
- ◆ Simplify sequence diagrams using subsystems
- ☆ ◆ Describe persistence-related behavior
- ◆ Refine the flow of events description
- ◆ Unify classes and subsystems
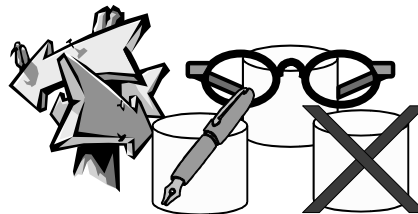
30

Rational
the software development company

You will now take a closer look at the incorporation of the persistency mechanism.

Use-Case Design Steps: Describe Persistence-Related Behavior

◆ **Describe Persistence-Related Behavior**
- Modeling Transactions
- Writing Persistent Objects
- Reading Persistent Objects
- Deleting Persistent Objects

In practice, there may be times when the application needs to control various aspects of persistence. These include:

- When determining how transactions are managed.

- When persistent objects are written — either the initial time when the object is written to the persistent object store or the subsequent times when the object is updated.

- When persistent objects are read. Retrieval of objects from the persistent object store is necessary before the application can send messages to that object. You need to send a message to an object that knows how to query the database, retrieve the correct objects, and instantiate them.

- When persistent objects are deleted. Unlike transient objects, which simply disappear when the process that created them dies, persistent objects exist until they are explicitly deleted. So, it is important to delete the object when it is no longer being used. However, this is hard to determine. Just because one application is done with an object does not mean that all applications, present and future, are done. And, because objects can and do have associations that even they do not know about, it is not always easy to figure out if it is okay to delete an object. The persistence framework may also provide support for this.

We will look at each one of these situations on the next two slides.

Modeling Transactions

Modeling Transactions

- ◆ What is a transaction?
  - ▪ Atomic operation invocations
  - ▪ "All or nothing"
  - ▪ Provide consistency
- ◆ Modeling options
  - ▪ Textually (scripts)
  - ▪ Explicit messages
- ◆ Error conditions
  - ▪ Rollback
  - ▪ Failure modes
  - ▪ May require separate interaction diagrams
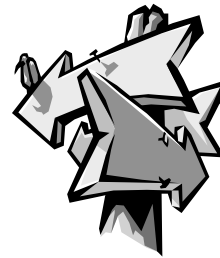
32

Rational
the software development company

Before we discuss the modeling of the persistence-related behavior, we need to define what transactions are. Transactions define a set of operation invocations that are atomic: They are either all performed, or none of them are performed. In the context of persistence, a transaction defines a set of changes to a set of objects that are either all performed or none are performed. Transactions provide consistency, ensuring that sets of objects move from one consistent state to another. If all operations specified in a transaction cannot be performed (usually because an error occurred), the transaction is aborted, and all changes made during the transaction are reversed.

Anticipated error conditions often represent exceptional flows of events in use cases. In other situations, error conditions occur because of some failure in the system. Error conditions should be documented in interactions. Simple errors and exceptions can be shown in the interaction where they occur; complex errors and exceptions might require their own interactions. Failure modes of specific objects can be shown on statecharts. Conditional flow of control handling of these failure modes can be shown in the interaction in which the error or exception occurs.

Transactions can be represented either textually using scripts or via explicit messages. The examples provided on the next slide demonstrate the use of separate messages.

Incorporating the Architectural Mechanisms: Persistency

Incorporating the Architectural Mechanisms: Persistency

♦ Analysis-Class-to-Architectural-Mechanism Map from Use-Case Analysis

Analysis Class	Analysis Mechanism(s)	
Student	*Persistency*, Security	***OODBMS***
Schedule	*Persistency*, Security	***Persistency***
CourseOffering	Persistency, Legacy Interface	***RDBMS***
Course	Persistency, Legacy Interface	***Persistency***
RegistrationController	Distribution	

Legacy persistency (RDBMS) is deferred to Subsystem Design.

Details in Appendix

During Use-Case Analysis, applicable mechanisms for each identified analysis class were documented. This information, along with the information on what analysis classes became what design elements, allows the applicable mechanisms for a design element to be identified.

In our example, we have been concentrating on course registration. Thus, the above table contains only the classes for the Register for Courses use-case realization that have analysis mechanisms assigned to them.

The legacy interface mechanism distinguishes the type of persistency. Remember, legacy data is stored in an RDBMS. The RDMBS JDBC mechanism is described in the Identify Design Mechanisms module.

The details of incorporating the ObjectStore mechanism are provided in the Additional Information Appendix in the ObjectStore Mechanism section.

RDBMS persistency is deferred to Subsystem Design since access to the legacy systems has been encapsulated within a subsystem.

Refine The Flow Of Events Description

Use-Case Design Steps

- ◆ Describe interaction among design objects
- ◆ Simplify sequence diagrams using subsystems
- ◆ Describe persistence-related behavior
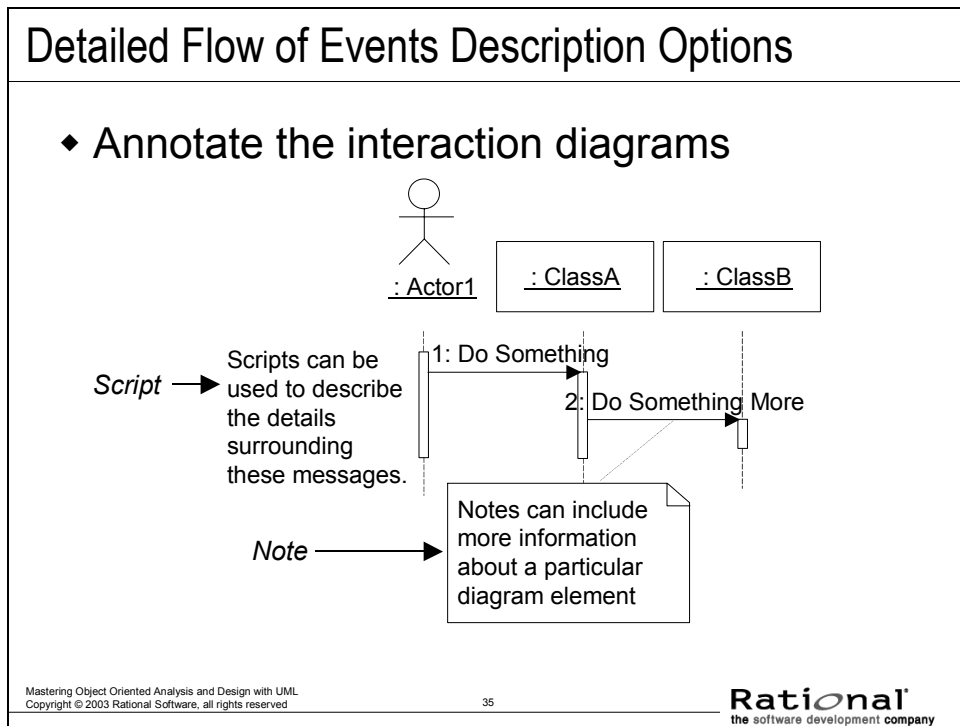☆ ◆ Refine the flow of events description
- ◆ Unify classes and subsystems

Rati⊘nal
the software development company

In this step you refine the flow of events originally outlined in Use-Case Analysis as needed to clarify the developed interaction diagrams.

This will make it easier for external observers to read the diagrams.

Detailed Flow of Events Description Options

In those cases where the flow of events is not fully clear from just examining the messages sent between participating objects, you might need to add additional descriptions to the interaction diagrams.

These steps are taken in cases where timing annotations, notes on conditional behavior, or clarification of operation behavior is needed to make it easier for external observers to read the diagrams.

Often, the name of the operation is not sufficient to understand why the operation is being performed. Textual notes or scripts in the margin of the diagram might be needed to clarify the interaction diagram. Textual notes and scripts might also be needed to represent control flow such as decision steps, looping, and branching. In addition, textual tags might be needed to correlate extension points in the use case with specific locations in interaction diagrams.

Unify Classes And Subsystems

Use-Case Design Steps

- ◆ Describe interaction among design objects
- ◆ Simplify sequence diagrams using subsystems
- ◆ Describe persistence-related behavior
- ◆ Refine the flow of events description
- ☆ ◆ Unify classes and subsystems

Rational
the software development company

At this point, you have a pretty good understanding of the design elements, their responsibilities, and the collaborations required to support the functionality described in the use cases. Now you must review your work to make sure that it is as complete and as consistent as possible before moving on to the detailed design activities of Subsystem and Class Design.

The purpose of Unify Classes and Subsystems is to ensure that each design element represents a single well-defined concept, with non-overlapping responsibilities. It is important to unify the identified classes and subsystems to ensure homogeneity and consistency in the model.

The next slide describes some of the considerations that a designer for a particular use case is concerned with (for example, consistency among collaborating design elements, and between the use-case flows of events and the Design Model). This is where you make sure that everything hangs together and fix it if does not.

Design Model Unification Considerations

Design Model Unification Considerations

- ◆ Model element names should describe their function
- ◆ Merge similar model elements
- ◆ Use inheritance to abstract model elements
- ◆ Keep model elements and flows of events consistent
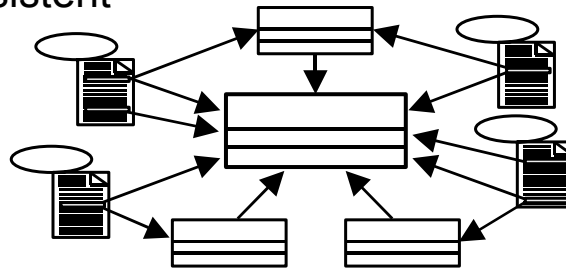
Points to consider:

- Names of model elements should describe their function. Avoid similar names and synonyms, because they make it difficult to distinguish between model elements.

- Merge model elements that define similar behaviors or that represent the same phenomenon.

- Merge classes that represent the same concept or have the same attributes, even if their defined behaviors are different.

- Use inheritance to abstract model elements, which tends to make the model more robust.

- When updating a model element, also update the affected use-case realizations.

Checkpoints: Use-Case Design

Checkpoints: Use-Case Design

- ◆ Is package/subsystem partitioning logical and consistent?
- ◆ Are the names of the packages/subsystems descriptive?
- ◆ Do the public package classes and subsystem interfaces provide a single, logically consistent set of services?
- ◆ Do the package/subsystem dependencies correspond to the relationships between the contained classes?
- ◆ Do the classes contained in a package belong there according to the criteria for the package division?
- ◆ Are there classes or collaborations of classes that can be separated into an independent package/subsystem?

Rational®
the software development company

The Design Model as a whole must be reviewed to detect glaring problems with layering and responsibility partitioning. The purpose of reviewing the model as a whole is to detect large-scale problems that a more detailed review would miss.

We want to ensure that the overall structure for the Design Model is well-formed, as well as detect large-scale quality problems that might not be visible by looking at lower-level elements.

The above checkpoints are important, because new packages/subsystems might be created when common subflows are identified.

Five packages and 1,000 classes is probably a sign that something is wrong.

Checkpoints: Use-Case Design

Checkpoints: Use-Case Design

- ◆ Have all the main and/or subflow for this iteration been handled?
- ◆ Has all behavior been distributed among the participating design elements?
- ◆ Has behavior been distributed to the right design elements?
- ◆ If there are several interaction diagrams for the use-case realization, is it easy to understand which collaboration diagrams relate to which flow of events?

Rati○nal
the software development company

Once the structure of the Design Model is reviewed, the behavior of the model needs to be scrutinized. First, make sure that there are no missing behaviors by checking to see that all scenarios for the current iteration have been completely covered by use-case realizations. All of the behaviors in the relevant use-case subflows must be described in the completed use-case realizations.

Next, make sure the behavior of the use-case realization is correctly distributed between model elements in the realizations. Make sure the operations are used correctly, that all parameters are passed, and that return values are of the correct type.

We want to ensure that the behavior of the system (as expressed in use-case realizations) matches the required behavior of the system (as expressed in use cases). Is it complete? We also want to ensure that the behavior is allocated appropriately among model elements, that is, is it correct?

Participating objects must be present to perform all the behaviors of the corresponding use case.

You need to make sure that the flow of events description clarifies how the diagrams are related to each other.

Review

Review: Use-Case Design

- ◆ What is the purpose of Use-Case Design?
- ◆ What is meant by encapsulating subsystem interactions? Why is it a good thing to do?

40

Rational
the software development company

Exercise: Use-Case Design

Exercise: Use-Case Design

- ◆ Given the following:
  - ▪ Analysis use-case realizations (VOPCs and interaction diagrams)
  - ▪ The analysis-class-to-design-element map
  - ▪ The analysis-class-to-analysis-mechanism map
  - ▪ Analysis-to-design-mechanism map
  - ▪ Patterns of use for the architectural mechanisms
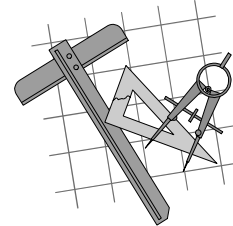
(continued)

41

Rational®
the software development **company**

The goal of this exercise is to refine the use-case realizations developed in Use-Case Analysis to incorporate the design classes and subsystems, as well as to (optionally) incorporate the patterns of use for the architectural mechanisms (persistency, security, and distribution). This exercise can also include the incorporation of the implementation mechanism.

References to the givens listed on this slide:

- **Use-case realizations:** Payroll Exercise Solution, Use-Case Analysis section

- **Analysis-class-to-design-element map:** Payroll Exercise Solution, Identify Design Elements section

The following are not needed if no mechanisms are being applied:

- **Analysis-class to analysis mechanism map:** Payroll Exercise Solution, Use-Case Analysis

- **Analysis-to-design-mechanism map:** Payroll Architecture Handbook, Architectural Mechanisms section

- **The patterns of use for the architectural mechanisms:**
  Payroll Architecture Handbook, Architectural Mechanisms: Implementation Mechanisms section.

Additional information on the incorporation of the ObjectStore OODBMS persistency mechanism for the Payroll System is provided in the Payroll Architectural Handbook, Logical View: Architectural Design section.

Exercise: Use-Case Design (cont.)

Exercise: Use-Case Design (cont.)

- ◆ Identify the following:
  - The design elements that replaced the analysis classes in the analysis use-case realizations
  - The architectural mechanisms that affect the use-case realizations
  - The design element collaborations needed to implement the use case
  - The relationships between the design elements needed to support the collaborations
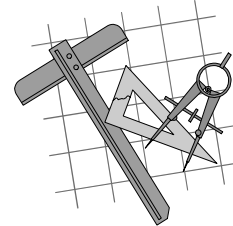
(continued)

Rational®
the software development company

You can determine which design elements have replaced the original analysis classes by referring back to the analysis-class-to-design-element map defined in the Identify Design Elements module.

You can determine which architectural mechanisms affect the use-case realization, by looking at the:

- Analysis-class-to-analysis-mechanism map

- Analysis-class-to-design-element map

- Analysis-to-design-to-implementation-mechanism map

Using this information, you can determine which architectural mechanisms affect the design elements involved in the use-case realization.

Based on what design elements have replaced the analysis classes and what architectural mechanisms must be incorporated, the analysis use-case realizations will need to be refined, including both the interaction diagrams and the VOPCs. As use-case responsibilities were allocated among analysis classes in Use-Case Analysis, use-case responsibilities must now be reallocated among the defined design elements.

Exercise: Use-Case Design (cont.)

Exercise: Use-Case Design (cont.)

- ◆ Produce the following:
  - ▪ Design use-case realization
    - • Interaction diagram(s) per use-case flow of events that describes the design element collaborations required to implement the use case
    - • Class diagram (VOPC) that includes the design elements that must collaborate to perform the use case, and their relationships
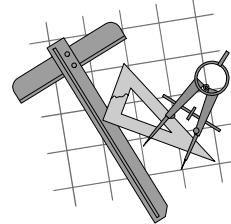
(continued)

Rational
the software development company

The produced interaction diagrams might be collaboration or sequence diagrams, but they should show the necessary collaborations. The VOPC class diagrams should reflect the relationships needed to support the collaborations.

Design elements can be either design classes or subsystems.

Where the interactions involve subsystems, the interaction diagram should only "go to" the subsystem interface and not "step within" the subsystem. The interactions within the subsystem will be developed in the Subsystem Design module.

Where necessary, include notes and scripts to clarify the use-case flow of events.

Refer to the following slides:

Exercise: Review

Exercise: Review

♦ Compare your use-case realizations
- ♦ Have all the main and subflows for this iteration been handled?
- ♦ Has all behavior been distributed among the participating design elements?
- ♦ Has behavior been distributed to the right design elements?
- ♦ Are there any messages coming from the interfaces?

Rati♀nal
the software development **company**

After completing a model, it is important to step back and review your work. Some helpful questions are the following:

- Have all the main and subflows for this iteration been handled?

- Has all behavior been distributed among the participating design elements? This includes design classes and interfaces.

- Has behavior been distributed to the right design elements?

- Are there any messages coming from the interface? Remember, messages should not come from an interface, since the behavior is realized by the subsystem.