Architectural and
Use Case Analysis
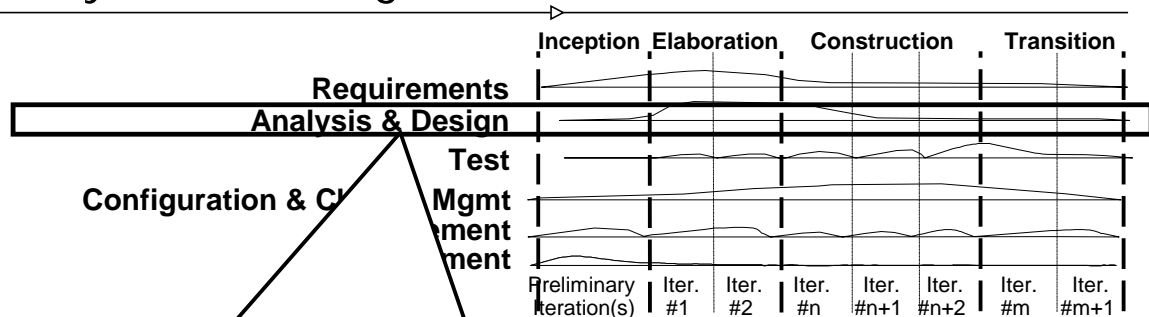
Ing. Giuseppe Calavaro, Ph.D.

1

Rational
the e-development company

In this module, we describe recommended software development practices and give the reasons for these recommendations.

Architectural and Use Case Analysis

Analysis and Design in Context

	Inception	Elaboration	Construction	Transition
Requirements				
Analysis & Design				
Test				
Configuration & Ch... Mgmt				
...ment				
...ment				

Preliminary Iteration(s)	Iter. #1	Iter. #2	Iter. #n	Iter. #n+1	Iter. #n+2	Iter. #m	Iter. #m+1

The purposes of Analysis and Design are:

- To transform the requirements into a design of the system to-be.

- To evolve a robust architecture for the system.

- To adapt the design to match the implementation environment, designing it for performance.

Rational
the e-development company


The purpose of Analysis and Design is to:

•Transform the requirements into a design of the system to-be.

•Evolve a robust architecture for the system.

•Adapt the design to match the implementation environment, designing it for performance.

The Analysis and Design workflow is related to other process workflows.

•The Business Modeling workflow provides an organizational context for the system.

•The Requirements workflow provides the primary input for Analysis and Design.

•The Test workflow tests system designed during Analysis and Design.

•The Environment workflow develops and maintains the supporting artifacts that are used during Analysis and Design.

•The Management workflow plans the project and each iteration (described in an Iteration Plan).

Architectural and Use Case Analysis
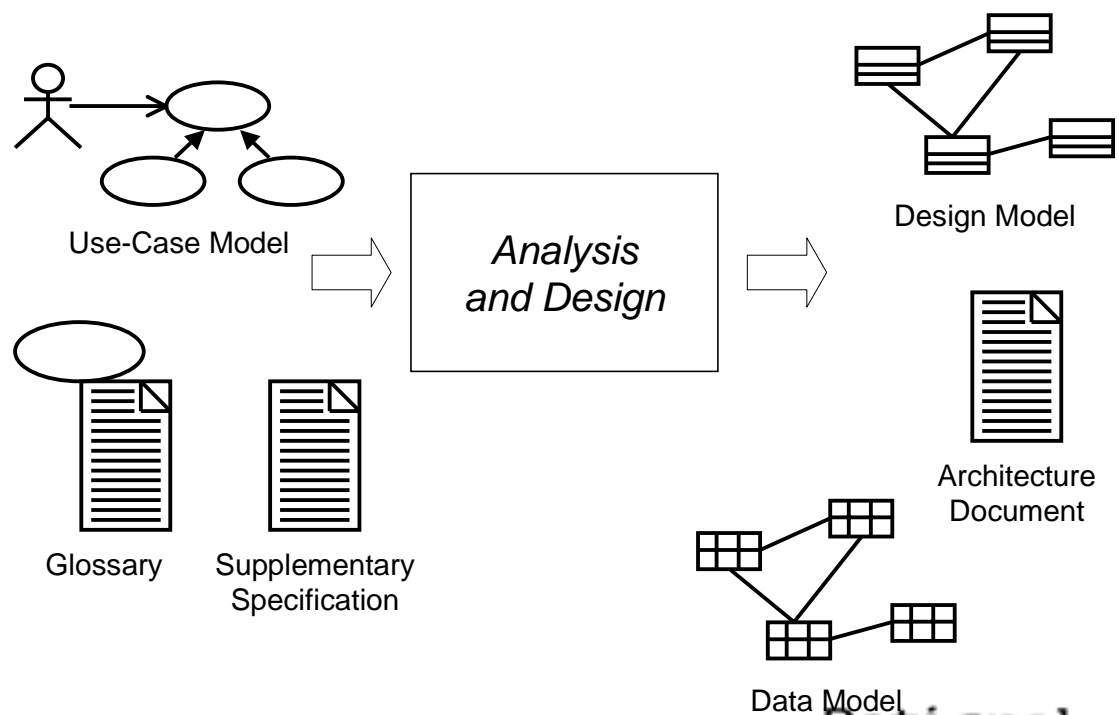
Analysis and Design Overview

Use-Case Model

Analysis and Design

Design Model

Glossary

Supplementary Specification

Architecture Document

Data Model

3

The input artifacts are the Use-Case Model, Glossary, and Supplementary Specification from the Requirements workflow. The result of analysis and design is a Design Model that serves as an abstraction of the source code; that is, the design model acts as a "blueprint" of how the source code is structured and written. The Design Model consists of design classes structured into design packages; it also contains descriptions of how objects of these design classes collaborate to perform use cases (use-case realizations).

The design activities are centered around the notion of architecture. The production and validation of this architecture is the main focus of early design iterations. Architecture is represented by a number of architectural views. These views capture the major structural design decisions. In essence architectural views are abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside. The architecture is an important vehicle not only for developing a good design model, but also for increasing the quality of any model built during system development. The architecture is documented in the Architecture Document.

The development of the Architecture Document is out of the scope of this course, but we will discuss it's contents and how to interpret them.

Data model development is out of the scope of this course.

Architectural and Use Case Analysis

Analysis Versus Design

◆ Analysis

- Focus on understanding the problem
- Idealized design
- Behavior
- System structure
- Functional requirements
- A small model

◆ Design

- Focus on understanding the solution
- Operations and Attributes
- Performance
- Close to real code
- Object lifecycles
- Non-functional requirements
- A large model

Rational
the e-development company

The differences between analysis and design are ones of focus and emphasis. The above slide lists the things that you focus on in analysis versus design.

The goal in Analysis is to understand the problem and to begin to develop a visual model of what you are trying to build, independent of implementation and technology concerns. Analysis focuses on translating the functional requirements into software concepts. The idea is to get a rough cut at the objects that comprise our system, but focusing on behavior (and therefore encapsulation). We then move very quickly, nearly seamlessly into "design" and the other concerns.

A goal of design is to refine the model with the intention of developing a design model that will allow a seamless transition to the coding phase. In design, we adapt to the implementation and the deployment environment. The implementation environment is the 'developer' environment, which is a software superset and a hardware subset of the deployment environment

In modeling, we start with an object model that closely resembles the real world (analysis), and then find more abstract (but more fundamental) solutions to a more generalized problem (design). The real power of software design is that it can create more powerful metaphors for the real world which change the nature of the problem, making it easier to solve.

Architectural and Use Case Analysis
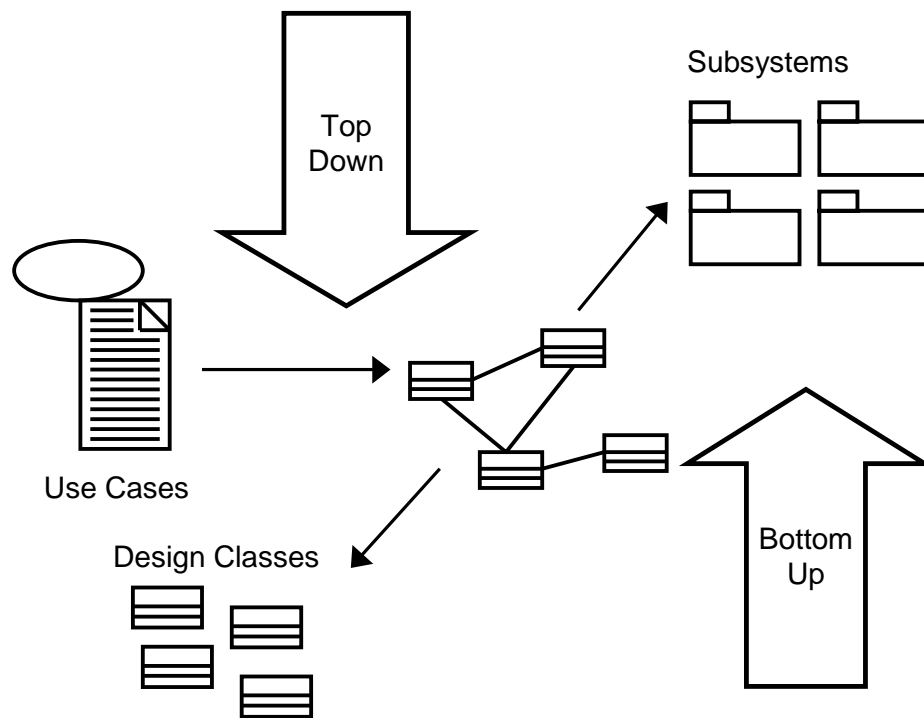
Analysis and Design is not Top-Down or Bottom-Up

Top Down

Subsystems

Use Cases

Design Classes

Bottom Up

Analysis and design is not top-down or bottom-up.

The use case comes in from the left and defines a middle level.

The analysis classes are not defined in a top-down pattern or a bottom-up pattern they are in the middle. From this middle level one may move up or down.
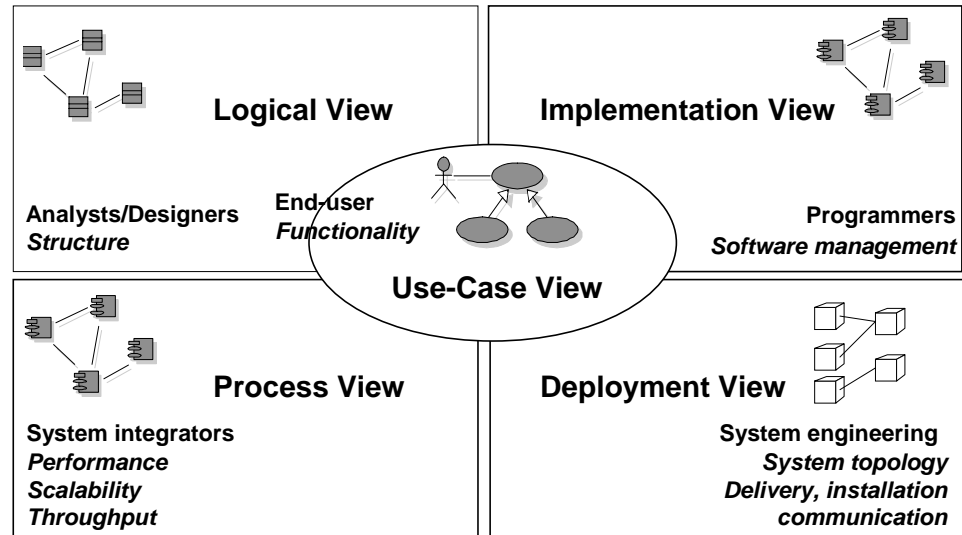
Defining subsystems is moving up and defining design classes is moving down.

Analysis is both top-to-middle, middle-up, middle-down and bottom-to-middle. There is no way of saying that one path is more important than the other - you have to travel on all paths to get the system right.

All of these four paths are equally important. That is why the bottom-up and top-down question can´t be solved.

Ing. Giuseppe Calavaro
Corso Informatica Industriale
Università degli studi di Roma "Tor Vergata"
AA 1999/2000

Architectural and Use Case Analysis

Software Architecture: The "4+1 View" Model

Logical View

Analysts/Designers
Structure

End-user
Functionality

Implementation View

Programmers
Software management

Use-Case View

Process View

System integrators
Performance
Scalability
Throughput

Deployment View

System engineering
System topology
Delivery, installation
communication

Rational
the e-development company

The above diagram describes the model Rational uses to describe the software architecture.

Architecture is many things to many different interested parties. On a particular project, there are usually multiple stakeholders, each with their own concerns and view of the system to be developed. The goal is to provide each of these stakeholders with a view of the system that addresses their concerns, and suppresses the other details.
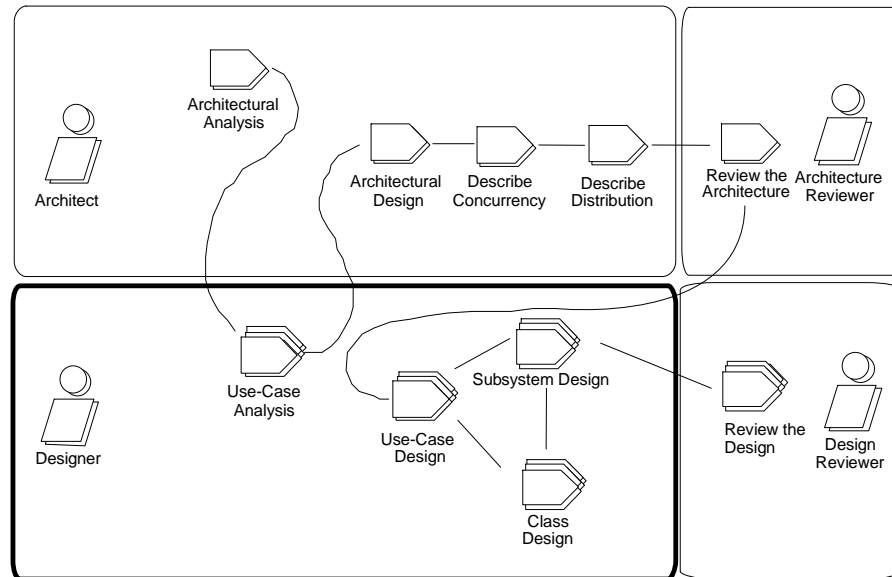
To address these different needs, Rational has defined the "4+1 view" architecture model. An architectural view is a simplified description (an abstraction) of a system from a particular perspective or vantage point, covering particular concerns, and omitting entities that are not relevant to this perspective. Views are "slices" of models.

Not all systems require all views (e.g., single processor: drop deployment view; single process: drop process view; small program: drop implementation view, etc.). A project may document all of these views or additional views. The number of views is dependent on the system you're building.

Each of these views, and the UML notation used to represent them, will be discussed in subsequent modules.

Architectural and Use Case Analysis

Analysis and Design Workflow

Architectural Analysis

Architect

Architectural Design — Describe Concurrency — Describe Distribution

Review the Architecture — Architecture Reviewer

Designer

Use-Case Analysis

Use-Case Design — Subsystem Design

Class Design

Review the Design — Design Reviewer

7

Rational the e-development company

The above diagram illustrates the workflow that we will be using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process. This course will be presented using the above workflow as a framework. This is to aid in presentation, and does not hinder the concepts from being applicable in other process contexts.
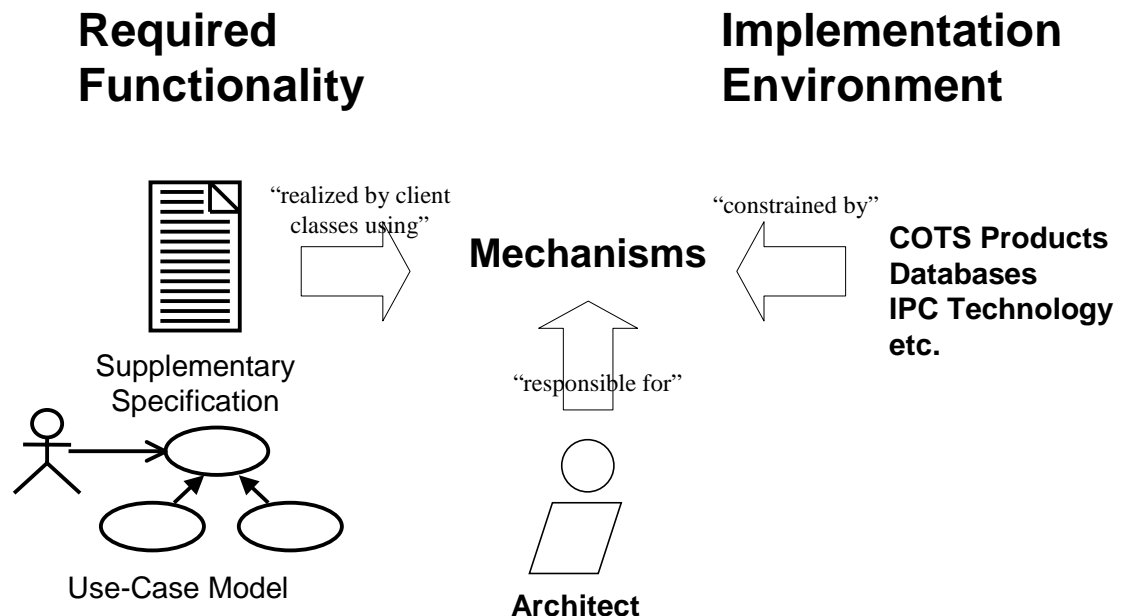
Remember, for analysis and design, we start out with the use-case model and the supplementary specifications from the Requirements workflow and end up with the design model that serves as an abstraction of the source code.

The design activities are centered around the notion of architecture. The production and validation of this architecture is the main focus of early design iterations. The architecture is an important vehicle not only for developing a good design model, but also for increasing the quality of any model built during system development.

The focus of this course is on the activities of the Designer. The Architect's activities will be discussed, but many of the architectural decisions will be "given". The activities of the Database Designer are considered out of scope for this course. Each of the Architect and Designer activities will be addressed in individual course modules.

Architectural and Use Case Analysis

Architectural Mechanisms: What Are They?

Required Functionality

Implementation Environment

"realized by client classes using"

"constrained by"

Mechanisms

COTS Products Databases IPC Technology etc.

Supplementary Specification

"responsible for"

Use-Case Model

Architect

In order to better understand what an analysis mechanism is, we have to understand what an architectural mechanism is.

An architectural mechanism is a strategic decision regarding common standards, policies, and practices. They are the realization of topics that should be standardized on a project. Everyone on the project should utilize these concepts in the same way and reuse the same mechanisms to perform the operations.

An architectural mechanism represents a common solution to a frequently encountered problem. They may be patterns of structure, patterns of behavior, or both. They are an important part of the "glue" between the required functionality of the system, and how this functionality is realized given the constraints of the implementation environment.

Support for architectural mechanisms needs to be "built in" to the architecture. Architectural mechanisms are coordinated by the architect. The Architect chooses the mechanisms, validates them by building or integrating them, verifies that they do the job, and then consistently imposes them upon the rest of the design of the system.

Architectural Mechanisms: Three Categories

- ◆ **Architectural Mechanism Categories**
  - ▪ Analysis Mechanisms (conceptual)
  - ▪ Design Mechanisms (concrete)
  - ▪ Implementation Mechanisms (actual)

Rational
the e-development company

There are three categories of architectural mechanisms, where the only difference between them is one of refinement.

Analysis mechanisms capture the key aspects of a solution in a way that is implementation independent. They provide specific behaviors to a domain-related class or component, or correspond to the implementation of cooperation between classes and/or components. They may be implemented as a framework. Examples include mechanisms to handle persistence, inter-process communication, error or fault handling, notification, and messaging, to name a few.

Design mechanisms are more concrete. They assume some details of the implementation environment, but are not tied to a specific implementation (as is an implementation mechanism).

Implementation mechanisms specify the exact implementation of the mechanism. Implementation mechanisms are are bound to a certain technology, implementation language, vendor, etc.

In a design mechanism, some specific technology is chosen (ex. RDBMS vs. ODBMS), whereas in an implementation mechanism, a VERY specific technology is chosen (Oracle vs. SYBASE).

The overall strategy for the implementation of analysis mechanisms must be built into the architecture. This will be discussed in more detail in Architectural Design when design and implementation mechanisms are discussed.

Architectural and Use Case Analysis

Sample Analysis Mechanisms

- Persistency
- Communication (IPC and RPC)
- Message routing
- Distribution
- Transaction management
- Process control and synchronization (resource contention)
- Information exchange, format conversion
- Security
- Error detection / handling / reporting
- Redundancy
- Legacy Interface

Analysis mechanisms provide specific behaviors to a domain-related class or component, or correspond to the implementation of cooperation between classes and/or components.

A persistent object is one that logically exists beyond the scope of the program that created it.

Examples of communication mechanisms would include inter-process communication (IPC) and inter-node communication (a.k.a. remote process communication or RPC). So with RPC, there is both a communication and a distribution aspect.

Mechanisms are perhaps easier to discuss when one talks about them as 'patterns' that are applied to the problem. So the inter-process communication pattern (i.e. "the application is partitioned into a number of communicating processes") interacts with the distribution pattern (i.e. "the application is distributed across a number of nodes") to produce the RPC pattern (i.e. "the application is partitioned into a number of processes, which are distributed across a number of nodes") which provides us a way to implement remote IPC.

Some examples of analysis mechanisms are listed on this slide. This list is not meant to be exhaustive.

Architectural and Use Case Analysis

Analysis Mechanism Characteristics

- ◆ **Persistency**
    - ▪ Granularity
    - ▪ Volume
    - ▪ Duration
    - ▪ Access mechanism
    - ▪ Access frequency (creation/deletion, update, read)
    - ▪ Reliability
- ◆ **Communication**
    - ▪ Latency
    - ▪ Synchronicity
    - ▪ Message Size
    - ▪ Protocol

Analysis mechanism characteristics capture some non-functional requirements of the system.

Persistency: For all classes whose instances may become persistent, we need to identify:

•Granularity: Range of size of the persistent objects

•Volume: Number of objects to keep persistent

•Duration: How long to keep persistent objects

•Access mechanism: How is a given object uniquely identified and retrieved?

•Access frequency: Are the objects more or less constant; are they permanently updated?

•Reliability: Shall the objects survive a crash of the process; the processor; or the whole system?

(**Inter-process) Communication**: For all model elements which needs to communicate with objects, components or services executing in other processes or threads, we need to identify:

•Latency: How fast must processes communicate with another?

•Synchronicity: Asynchronous communication

•Size of message: A spectrum might be more appropriate than a single number.

•Protocol, flow control, buffering, and so on.

Architectural and Use Case Analysis

Analysis Mechanism Characteristics (cont.)

- ◆ Legacy interface
  - ▪ Latency
  - ▪ Duration
  - ▪ Access mechanism
  - ▪ Access frequency
- ◆ Security
  - ▪ Data granularity
  - ▪ User granularity
  - ▪ Security rules
  - ▪ Privilege types
- ◆ etc.

For security:

•Data granularity: Package-level, Class-level, attribute level

•User granularity: Single Users, Roles/Groups

•Security Rules: Based on value of data, on algorithm based on data, algorithm based on user and data

•Privilege Types: Read, Write, Create, Delete, perform some other operation

Architectural and Use Case Analysis

Example: Course Registration Analysis Mechanisms

- ◆ Persistence
- ◆ Distribution
- ◆ Security
- ◆ Legacy Interface

The above are the selected analysis mechanisms for the Course Registration System.

Persistency: A means to make an element persistent (i.e., exist after the application that created it ceases to exist).

Distribution: A means to distribute an element across existing nodes of the system.

Security: A means to control access to an element.

Legacy Interface: A means to access a legacy system with an existing interface.

These are also documented in the Payroll Architecture Handbook, Architectural Mechanisms section.

Identify Key Abstractions

- ◆ **Define preliminary analysis classes (sources)**
  - ▪ Domain knowledge
  - ▪ Requirements
  - ▪ Glossary
  - ▪ Domain Model, or the Business Model (if exists)
- ◆ **Define analysis class relationships**
- ◆ **Model analysis classes and relationships on Class Diagrams**
  - ▪ Include brief description of analysis class
- ◆ **Map analysis classes to necessary analysis mechanisms**

Requirements and business modeling activities usually uncover key abstractions that the system must be able to handle. Because of the work already done, there is no need to repeat the identification work again during Use-Case Analysis. To take advantage of existing knowledge, identify preliminary analysis classes on the basis of general knowledge of the system, such as the Requirements, the Glossary, and in particular, the Domain Model, or the Business Model, if you have one. From the UML User's Guide by Booch et al: The Business Model "establishes an abstraction of the organization" and the Domain Model "establishes the context of the system".

While defining the initial analysis classes, you can also define any relationships that exist between them. The relationships are those that support the basic definitions of the abstractions. The objective is not to develop a complete class model at this point, but just to define some key abstractions and basic relationships to "kick off" the analysis effort and reduce any duplicate effort that may result when different teams analyze the individual use cases. Relationships defined at this point reflect the semantic connections between the defined abstractions, not the relationships necessary to support the implementation and required communication amongst abstractions.

The analysis classes identified at this point will probably change and evolve during the course of the project. The purpose of this step is not to identify a set of classes that will survive throughout design, but to identify the key abstractions the system must handle. Don't spend much time describing analysis classes in detail at this initial stage, because there is a risk that you identify classes and relationships that are not actually needed by the use cases. Remember that you will find more analysis classes and relationships when looking at the use cases.

Architectural and Use Case Analysis

Example: Key Abstractions
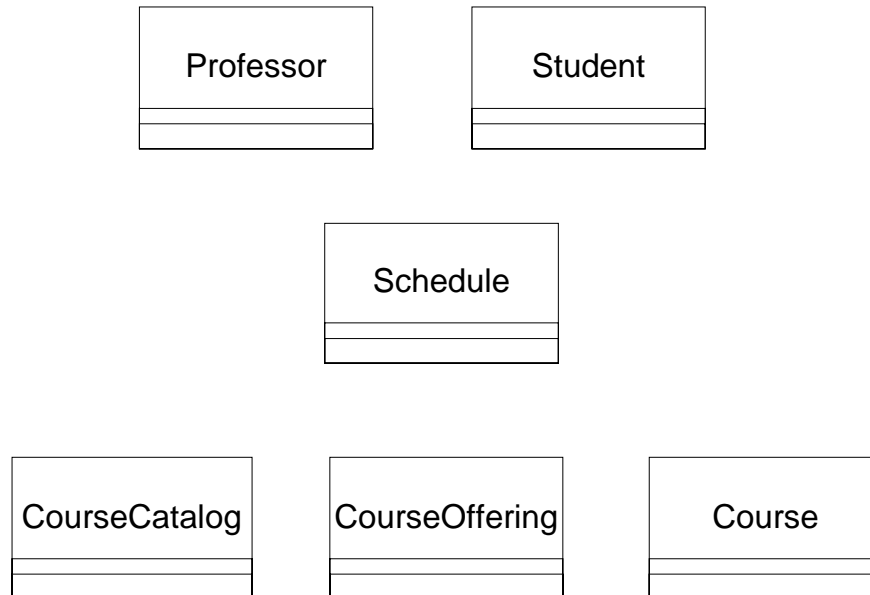
Professor	Student

Schedule

CourseCatalog	CourseOffering	Course

15

Rational
the e-development company

Professor - A person teaching classes at the university.

Student - A person enrolled in classes at the university.

Schedule - The courses a student has enrolled in for a semester.

CourseCatalog - Unabridged catalog of all courses offered by the university.

CourseOffering - A specific offering for a course, including days of the week and times.

Course - A class offered by the university.

Patterns and Frameworks

- ◆ Pattern
  - ▪ A common solution to a common problem in a context
- ◆ Analysis/Design Pattern
  - ▪ A solution to a narrowly-scoped technical problem
  - ▪ A fragment of a solution, or a piece of the puzzle
- ◆ Framework
  - ▪ Defines the general approach to solving the problem
  - ▪ Skeletal solution, whose details may be analysis/design patterns

OOAD Using the UML - Analysis and Design Overview, v 4.2
Copyright © 1998-1999 Rational Software, all rights reserved

16

Rati○nal
the e-development company

The selection of the upper-level layers may be affected by the choice of an architectural pattern or framework. Thus, it is important to define what these terms mean.

A **pattern** codifies specific knowledge collected from experience. Patterns provide examples of how good modeling solves real problems, whether you come up with it yourself or you reuse someone else's. Design patterns are discussed in more detail on the next slide.
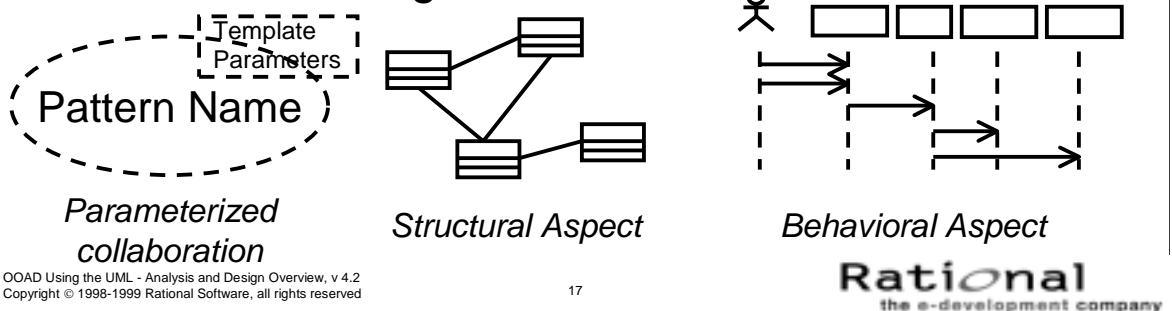
Frameworks differ from analysis and design patterns in their scale and scope. Frameworks describe a skeletal solution to a particular problem which may lack many of the details, which may be filled in by applying various analysis and design patterns.

A framework is a micro-architecture that provides an incomplete template for applications within a specific domain. Architectural frameworks provide the context in which the components run. They provide the infrastructure (plumbing, if you will) that allows the components to co-exist and perform in predictable ways. These frameworks may provide communication mechanisms, distribution mechanisms, error processing capabilities, transaction support, etc.

Frameworks may range in scope from persistence frameworks which describe the workings of a fairly complex but fragmentary part of an application, to domain-specific frameworks which are intended to be customized (such as Peoplesoft, SanFransisco, Infinity, SAP). SAP is a framework for manufacturing and finance.

Design Patterns

- ◆ A design pattern is a solution to a common design problem
  - Describes a common design problem
  - Describes the solution to the problem
  - Discusses the results and trade-offs of applying the pattern
- ◆ Design patterns provide the capability to reuse successful designs

Template Parameters

Pattern Name

Parameterized collaboration

Structural Aspect

Behavioral Aspect

We will look at a number of design patterns throughout this course. Thus, it is important to define what a design pattern is up front.

Design patterns are being collected and cataloged in a number of publications and mediums. You can use design patterns to solve issues in your design without "reinventing the wheel". You can also use design patterns to validate and verify your current approaches.

Using design patterns can lead to more maintainable systems, and increase productivity. They provide excellent examples of good design heuristics, and design vocabulary. In order to use design patterns effectively, you should become familiar with some common design patterns and the issues that they mitigate.

A design pattern is modeled in the UML as a parameterized collaboration. Thus it has a structural aspect and a behavioral aspect. The structural part is the classes whose instances implement the pattern, and their relationships (the static view). The behavioral aspect describes how the instance collaborate (e.g., send messages to each other) to implement the pattern (the dynamic view).

A parameterized collaboration is a template for a collaboration. The Template Parameters are what are used to adapt the collaboration for a specific usage. These parameters may be bound to different sets of abstractions, depending on how they are applied in the design.

Architectural Patterns

- ◆ Layers
- ◆ Model-view-controller (M-V-C)
- ◆ Pipes and filters
- ◆ Blackboard

Rational
the e-development company

Architectural Analysis is where you consider architectural patterns, as this choice affects the high-level organization of your object model.

Layers: The Layers pattern is where an application is decomposed into different levels of abstraction. The layers range from application-specific layers at the top to implementation/technology-specific layers on the bottom.

Model-View-Controller: The MVC pattern is where an application is divided into three partitions: The Model which is the business rules and underlying data, the View which is how information is displayed to the user, and the Controllers which process the user input.

Pipes and Filters: In the Pipes and Filters pattern, data is processed in streams that flow through pipes from filter to filter. Each filter is a processing step.
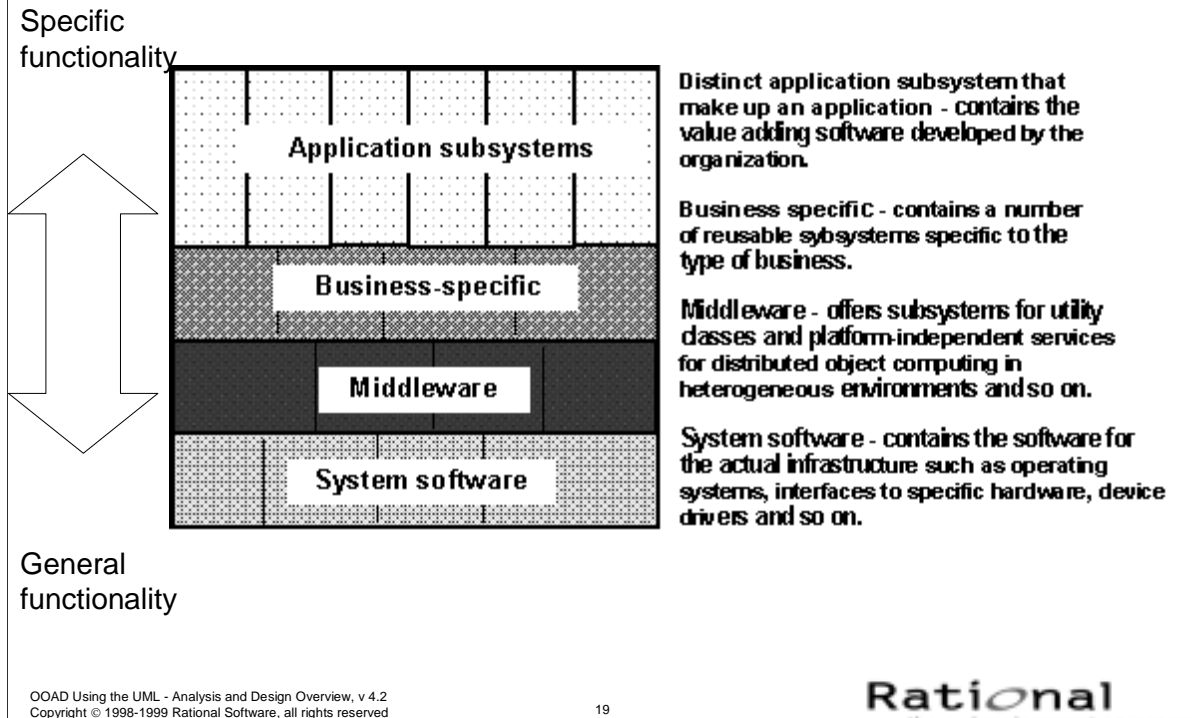
Blackboard: The Blackboard pattern is where independent specialized applications collaborate to derive a solution, working on a common data structure.

Architectural patterns can work together (e.g., more than one architectural pattern can be present in any one software architecture).

The architectural patterns listed above imply certain system characteristics, performance characteristics, and process and distribution architectures. Each solves certain problems but also poses unique challenges. For this course we will concentrate on the Layers architectural pattern.

Architectural and Use Case Analysis

Typical Layering Approach

Specific functionality

Application subsystems — Distinct application subsystem that make up an application - contains the value adding software developed by the organization.

Business-specific — Business specific - contains a number of reusable subsystems specific to the type of business.

Middleware — Middleware - offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

System software — System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

General functionality

Layering represents an ordered grouping of functionality, with the application-specific located in the upper layers, functionality that spans application domains in the middle layers, and functionality specific to the deployment environment at the lower layers.

Packages should be organized into layers with application-specific packages located in the upper layers of the architecture, hardware and operating-specific packages located in the lower layers of the architecture, and general-purpose services occupying the middleware layers. The advantage of partitioning in this way is that it provides a clear separation of concerns. By separating application (e.g., GUI) services from other services, the system's user interface can be changed without impacting the rest of the application. Similarly, by separating business services from other services, it's easier to change the business rules of your system with minimal impact to the rest of your system. Such separation of concerns results in more resilient systems.

Layers: How Do We Find Them?

- ◆ Level of abstraction
  - ▪ Group elements at the same level of abstraction
- ◆ Separation of concerns
  - ▪ Group like things together
  - ▪ Separate disparate things
  - ▪ Application vs. Domain model elements
- ◆ Resiliency
  - ▪ Loose coupling
  - ▪ Concentrate on encapsulating change
  - ▪ User interface, business rules, and retained data tend to have a high potential for change

Rati∩nal
the e-development company

Layers are used to encapsulate conceptual boundaries between different kinds of services and provide useful abstractions which makes the design easier to understand.

When layering, you should concentrate on grouping things that are similar together, as well as encapsulating change.

The number and composition of layers is dependent upon the complexity of both the problem domain and the solution space. There is generally only a single application-specific layer. Domains in which previous systems have been built, or in which large systems are composed in turn of inter-operating smaller systems, there is a strong need to share information between design teams. As a result, the Business-specific layer is likely to partially exist and may be structured into several layers for clarity. Solution spaces which are well-supported by middleware products and in which complex system software plays a greater role will have well-developed lower layers, with perhaps several layers of middleware and system software.

Remember, in Architectural Analysis, we are concentrating on the upper-level layers (the application- and business-specific layers). The lower level layers (infrastructure and vendor-specific layers) will be defined in Architectural Design.

Architectural and Use Case Analysis

Modeling Architectural Layers

- ◆ Architectural layers can be modeled using stereotyped packages
- ◆ <<layer>> stereotype
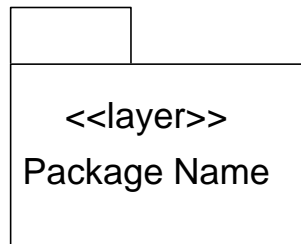
<<layer>>
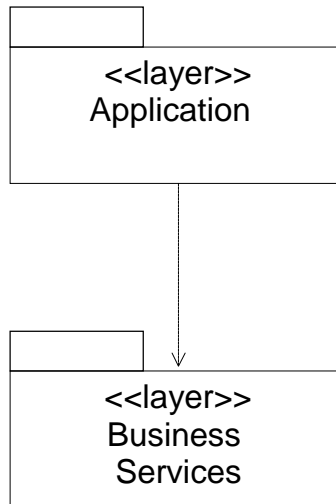Package Name

21

Rational
the e-development company

Layers can be represented in Rose as packages with the <<layer>> stereotype. The layer descriptions can be included in the documentation field of the specification of the package.

Architectural and Use Case Analysis

Example: High-Level Organization of the Model
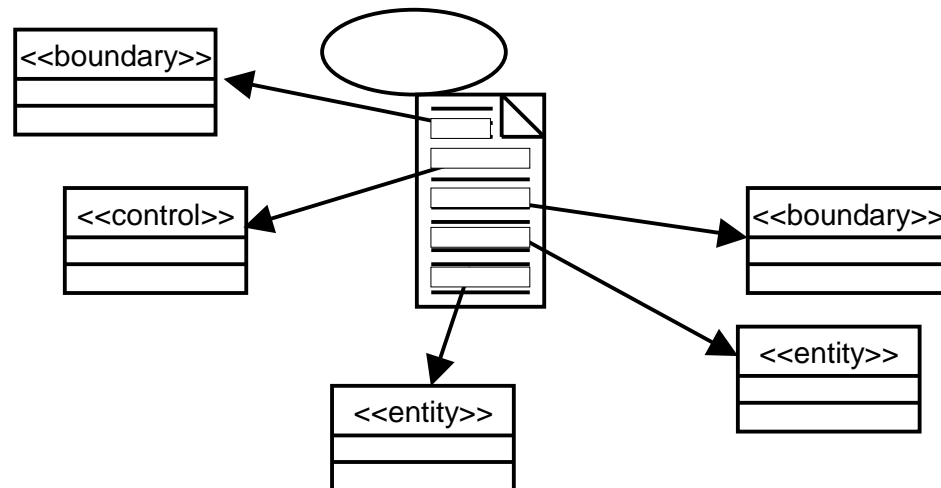
Rational
the e-development company

The above example includes the application- and business- specific layers for the Course Registration System.

The Application layer contains application-specific design elements.

We expect that multiple applications will share some key abstractions and common services. These have been encapsulated in the Business Services layer, that is accessible to the Application layer. The Business Services layer contains business-specific elements that are used in several applications.

Architectural and Use Case Analysis

Find Classes From Use-Case Behavior

- ◆ The complete behavior of a use case has to be distributed to analysis classes

<<boundary>>

<<control>>

<<boundary>>

<<entity>>

<<entity>>

23

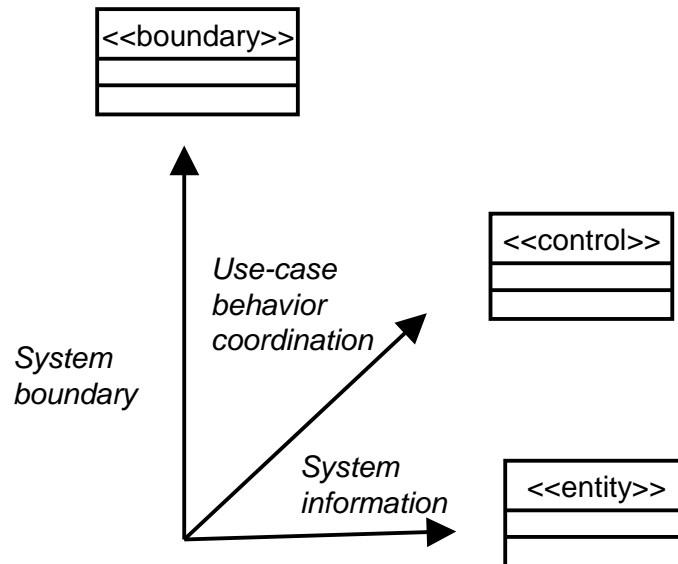**Rational**
the e-development company

The technique for finding analysis classes described in this module uses three different perspectives of the system to drive the identification of candidate classes. The three perspectives are that of the boundary between the system and its actors, the information the system uses, and the control logic of the system. The use of stereotypes to represent these perspectives (e.g., boundary, control and entity) results in a more robust model because they isolate those things most likely to change in a system: the interface/environment, the control flow and the key system entities. These stereotypes are conveniences used during analysis that disappear in design.

Identification of classes means just that: they should be identified, named, and described briefly in a few sentences.

The different stereotypes are discussed in more detail throughout this module.

Architectural and Use Case Analysis

What is an Analysis Class?

<<boundary>>

<<control>>

System boundary

Use-case behavior coordination

System information

<<entity>>

24

Rational
the e-development company

Analysis classes represent an early conceptual model for 'things in the system which have responsibilities and behavior'. Analysis classes are used to capture a 'first-draft', rough-cut of the object model of the system.

Analysis classes handle primarily functional requirements, and model objects from the problem domain. Analysis classes can be used to represent "the objects we want the system to support," without taking a decision on how much of them to support with hardware and how much with software.

There are three aspects of the system that are likely to change: the boundary between the system and its actors, the information the system uses, and the control logic of the system. In an effort to isolate the parts of the system that will change, different types of analysis classes are identified, each with a "canned" set of responsibilities: boundary, entity and control classes. Stereotypes may be defined for each type. These distinctions are used during analysis, but disappear in design.

The different types of analysis classes can be represented using different icons or with the name of the stereotype in guillemets (<< >>): <<boundary>>, << control>>, <<entity>>.

Each of these types of analysis classes are discussed on the following slides.

Architectural and Use Case Analysis

Analysis Classes: A First Step Towards Executables
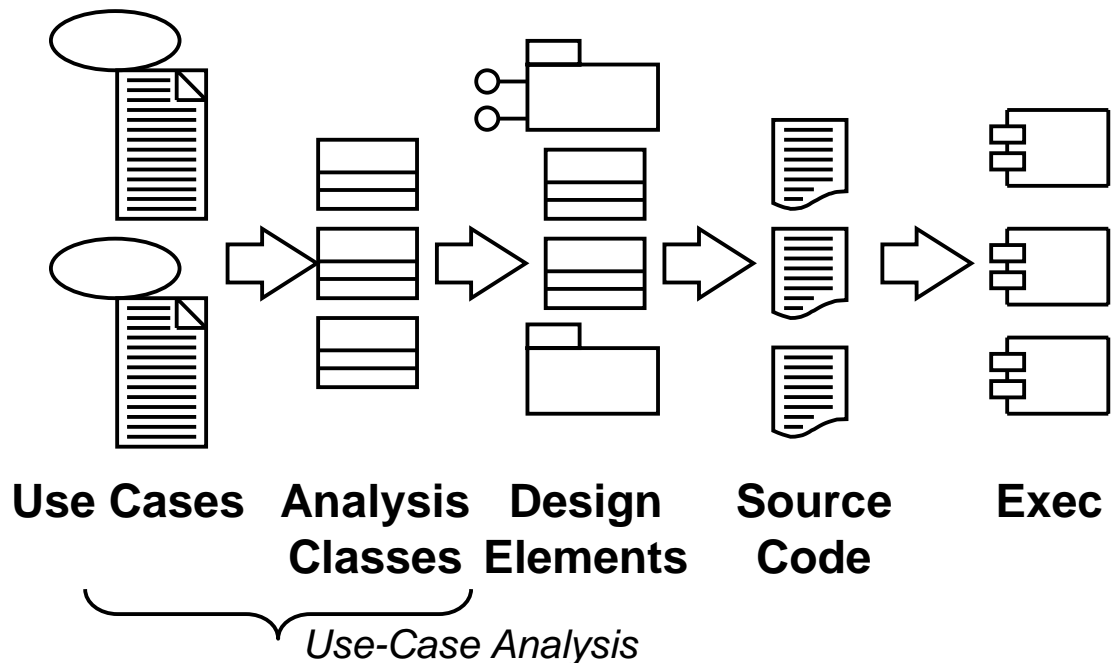
Use Cases   **Analysis Classes**   **Design Elements**   **Source Code**   **Exec**

Use-Case Analysis

Finding a candidate set of roles is the first step in the transformation of the system from a mere statement of required behavior to a description of how the system will work

The analysis classes, taken together, represent an early conceptual model of the system. This conceptual model evolves quickly and remains fluid for some time as different representations and their implications are explored. Formal documentation can impede this process, so be careful how much energy you expend on maintaining this 'model' in a formal sense; you can waste a lot of time polishing a model which is largely expendable. Analysis classes rarely survive into the design unchanged. Many of them represent whole collaborations of objects, often encapsulated by subsystems.

Analysis classes are 'proto-classes', which are essentially "clumps of behavior". These analysis classes are early conjectures of the composition of the system; they rarely survive intact into implementation. Many of the analysis classes morph into something else later on (subsystems, components, split classes, combined classes). They provide us with a way of capturing the required behaviors in a form that we can use to explore the behavior and composition of the system. Analysis classes allow us to "play" with the distribution of responsibilities, re-allocating, as necessary.

What is a Boundary Class?

- ◆ **Intermediates the interface to something outside the system**
- ◆ **Several Types**
  - ▪ User interface classes
  - ▪ System interface classes
  - ▪ Device interface classes
- ◆ *One boundary class per actor/use case pair*
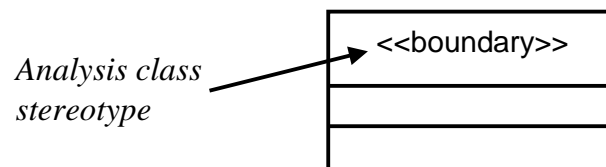
Analysis class stereotype

<<boundary>>

Environment Dependent

26

Rational
the e-development company

A boundary class intermediates the interface to something outside the system. Boundary class insulate the system from changes in the surroundings (changes in interfaces to other systems, changes in user requirements, etc.), keeping these changes from affecting the rest of the system.

A system may have several types of boundary classes:

User interface classes - Classes which intermediate communication with human users of the system.

System interface classes - Classes which intermediate communication with other systems. A boundary class which communicates with an external system is responsible for managing the dialogue with the external system; it provides the interface to that system for the system being built.

Device interface classes - Classes which provide the interface to devices which detect external events. These boundary classes capture the responsibilities of the device or sensor.

One recommendation for the initial identification of boundary classes is one boundary class per actor/use-case pair.

Architectural and Use Case Analysis

The Role of a Boundary Class
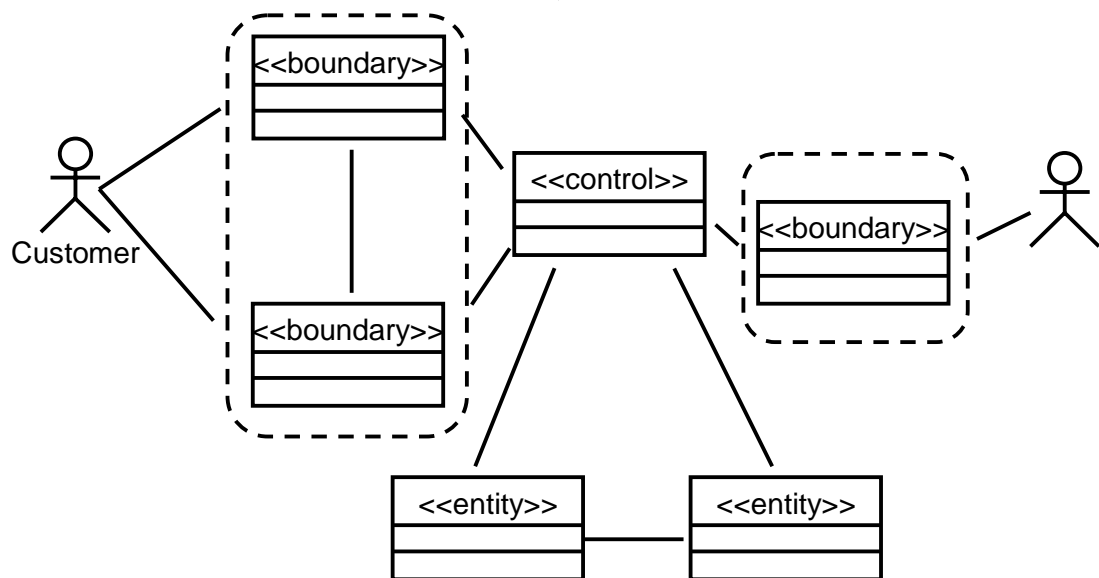
Model interaction between the system and its environment

Rational
the e-development company

A boundary class is a class used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as the interface).

Boundary classes model the parts of the system that depend on its surroundings. Entity classes and control classes model the parts that are independent of the system's surroundings. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes.

Actors can only communicate with boundary classes.

Boundary classes also make it easier to understand the system because they clarify the system's boundaries. They aid design by providing a good point of departure for identifying related services. For example, if you identify a printer interface early in the design, you will soon see that you must also model the formatting of printouts.

A boundary object (an instance of a boundary class) can outlive a use-case instance if, for example, it must appear on a screen between the performance of two use cases. Normally, however, boundary objects live only as long as the use-case instance.

Architectural and Use Case Analysis

Example: Finding Boundary Classes

◆ One boundary class per actor/use case pair
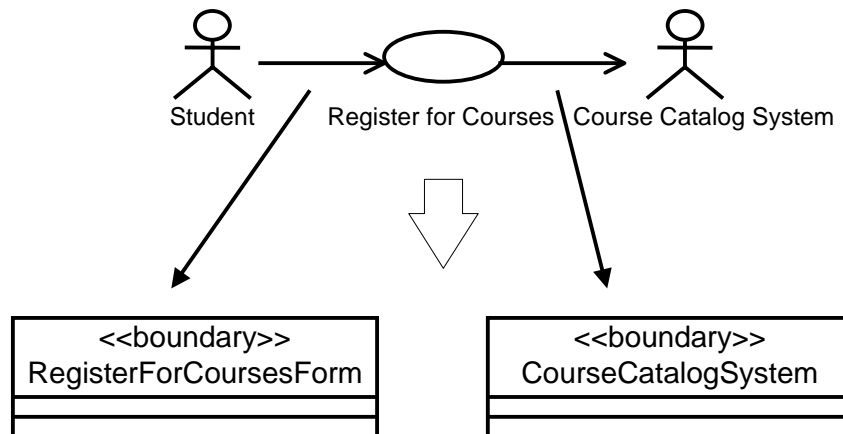
Student   Register for Courses   Course Catalog System

```
<<boundary>>
RegisterForCoursesForm
```

```
<<boundary>>
CourseCatalogSystem
```

The goal of analysis is to form a good picture of how the system is composed, not to design every last detail. In other words, identify boundary classes only for phenomena in the system or for things mentioned in the flow of events of the use-case realization.

Consider the source for all external events and make sure there is a way for the system to detect these events.

One recommendation for the initial identification of boundary classes is one boundary class per actor/use-case pair. This class can be viewed as having responsibility for coordinating the interaction with the actor. This may be refined as a more detailed analysis is performed. This is particularly true for window-based GUI applications, where there is typically one boundary class for each window, or one for each dialog.

In the above example:

•The RegisterForCoursesForm contains a Student's "schedule-in-progress". It displays a list of Course Offerings for the current semester from which the Student may select to be added to his/her Schedule.

•The CourseCatalogSystem interfaces with the legacy system that provides the unabridged catalog of all courses offered by the university. This class replaces the CourseCatalog abstraction originally identified in Architectural Analysis.

Guidelines: Boundary Class

- ◆ **User Interface Classes**
  - ▪ Concentrate on what information is presented to the user
  - ▪ Do NOT concentrate on the UI details
- ◆ **System and Device Interface Classes**
  - ▪ Concentrate on what protocols must be defined
  - ▪ Do NOT concentrate on how the protocols will be implemented

Concentrate on the responsibilities, not the details!
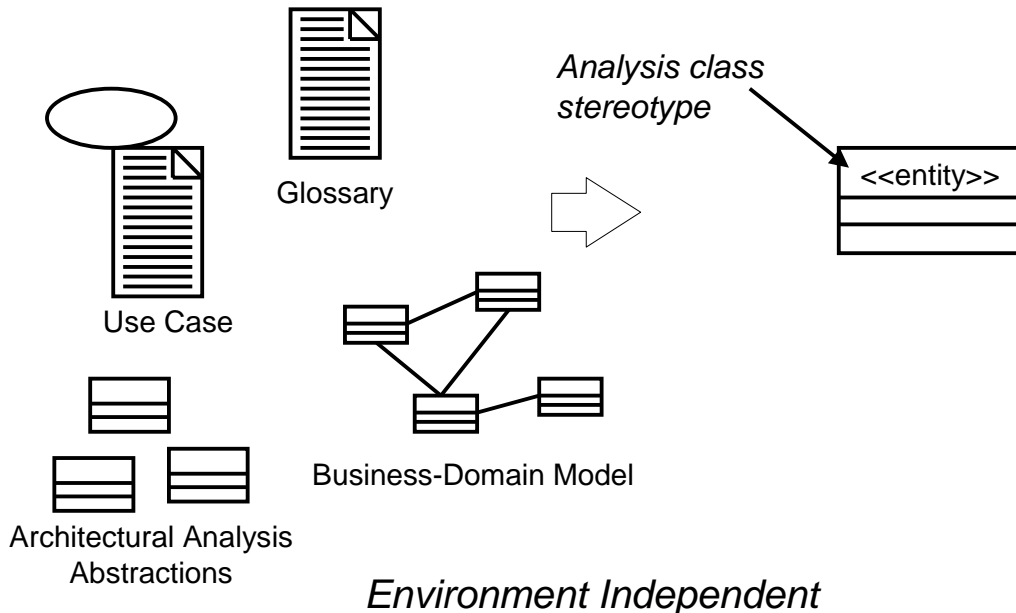
29

Rational
the e-development company

When identifying and describing analysis classes, be careful not too spend too much time on the details. Analysis classes are meant to be a first cut at the abstractions of the system. They help to clarify the understanding of the problem to be solved, and represent an attempt at an idealized solution (analysis has been called "idealized design").

User Interface Classes: Boundary classes may be used as "holding places" for GUI classes. The objective is not to do GUI design in analysis, but to isolate all environment-dependent behavior. The expansion, refinement and replacement of these boundary classes with actual user interface classes (probably derived from purchased UI libraries) is a very important activity of Class Design, and will be discussed in the Class Design module. Sketches, or screen dumps from a user-interface prototype, may have been used during the Requirements workflow to illustrate the behavior and appearance of the boundary classes. These may be associated with a boundary class. However, only model the key abstractions of the system; do not model every button, list and widget in the GUI.

System and Device Interface Classes: If the interface to an existing system or device is already well-defined, the boundary class responsibilities should be derived directly from the interface definition. If there is a working communication with the external system or device, make note of it for later reference during design.

Architectural and Use Case Analysis

What is an Entity Class?

◆ **Key abstractions of the system**

Glossary

Use Case

Architectural Analysis Abstractions

Business-Domain Model

Analysis class stereotype

<<entity>>

Environment Independent

Entity objects represent the key concepts of the system being developed. Entity classes provide another point of view from which to understand the system because they show the logical data structure, which can help you understand what the system is supposed to offer its users.

Frequent sources of inspiration for entity classes are the:

•Glossary (developed during requirements)

•Business-domain model (developed during business modeling, if business modeling has been performed)

•Use-case flow of events (developed during requirements)

•Key abstractions (identified in Architectural Analysis)

As mentioned earlier, sometimes there is a need to model information about an actor within the system. This is not the same as modeling the actor (actors are external. by definition).   These classes are sometimes called "surrogates".

Architectural and Use Case Analysis

The Role of an Entity Class
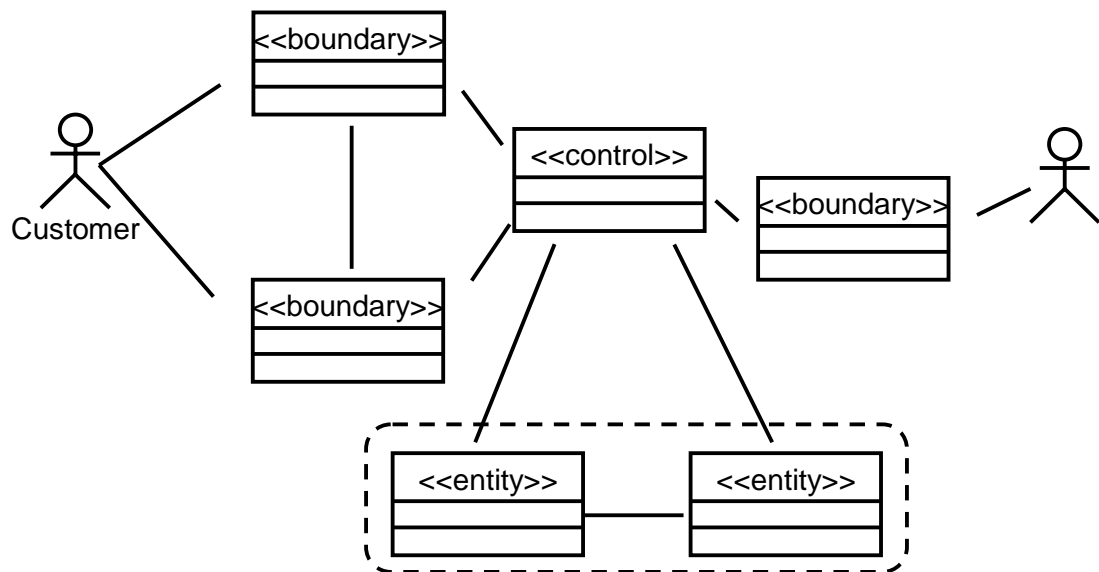
Store and manage information in the system

Rational
the e-development company

Entity classes represent stores of information in the system; they are typically used to represent the key concepts the system manages. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system.

The main responsibilities of entity classes are to store and manage information in the system.

An entity object is usually not specific to one use-case realization; sometimes, an entity object is not even specific to the system itself. The values of its attributes and relationships are often given by an actor. An entity object may also be needed to help perform internal system tasks. Entity objects can have behavior as complicated as that of other object stereotypes. However, unlike other objects, this behavior is strongly related to the phenomenon the entity object represents. Entity objects are independent of the environment (the actors).

Architectural and Use Case Analysis

Example: Finding Entity Classes

- ◆ Use use-case flow of events as input
- ◆ Key abstractions of the use case
- ◆ Traditional, filtering nouns approach
  - ▪ Underline noun clauses in the use-case flow of events
  - ▪ Remove redundant candidates
  - ▪ Remove vague candidates
  - ▪ Remove actors (out of scope)
  - ▪ Remove implementation constructs
  - ▪ Remove attributes (save for later)
  - ▪ Remove operations

Taking the use case flow of events as input, underline the noun phrases in the flow of events. These are the initial candidate list of analysis classes. Then go through a series of filtering steps where some candidate classes are eliminated. This is necessary due to the ambiguity of the English language. The result of the filtering exercise is a list of candidate entity classes.

While the filtering approach does add some structure to what could be an ad-hoc means of identifying classes, people generally filter as they go rather than blindly accepting all nouns and then filtering.

Example: Candidate Entity Classes

◆ Register for Courses (Create Schedule)
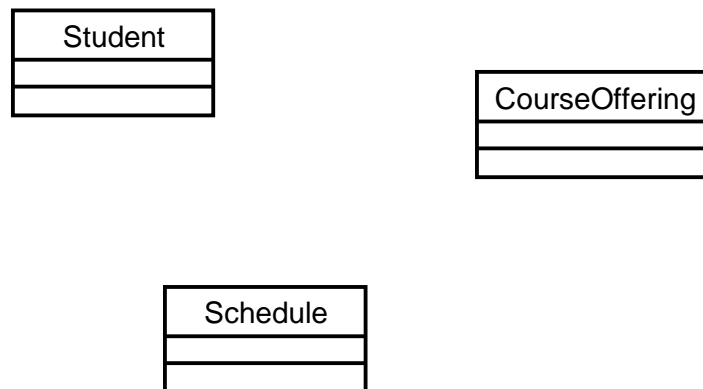
```
┌─────────────────────┐
│       Student       │
├─────────────────────┤
│                     │
├─────────────────────┤
│                     │
└─────────────────────┘
```

```
┌─────────────────────┐
│   CourseOffering    │
├─────────────────────┤
│                     │
├─────────────────────┤
│                     │
└─────────────────────┘
```

```
┌─────────────────────┐
│      Schedule       │
├─────────────────────┤
│                     │
├─────────────────────┤
│                     │
└─────────────────────┘
```

Rational
the e-development company

The following are the definitions for each of the classes shown in the above diagram:

CourseOffering - A specific offering for a course, including days of the week and times.

Schedule - The courses a student has selected for the current semester.

Student - A person enrolled in classes at the university.

As mentioned earlier, sometimes there is a need to model information about an actor within the system. This is not the same as modeling the actor (actors are external. by definition). These classes are sometimes called "surrogates".

For example, a course registration system maintains information about the student which is independent of the fact that the student also plays a role as an actor of the system. This information about the student that is stored in a 'Student' class is completely independent of the 'actor' role the student plays; the Student class will exist whether or not the student is an actor to the system.

Architectural and Use Case Analysis

Review: Generalization

- ◆ One class shares the structure and/or behavior of one or more classes
- ◆ "Is-a-kind of" relationship
- ◆ In analysis, use sparingly
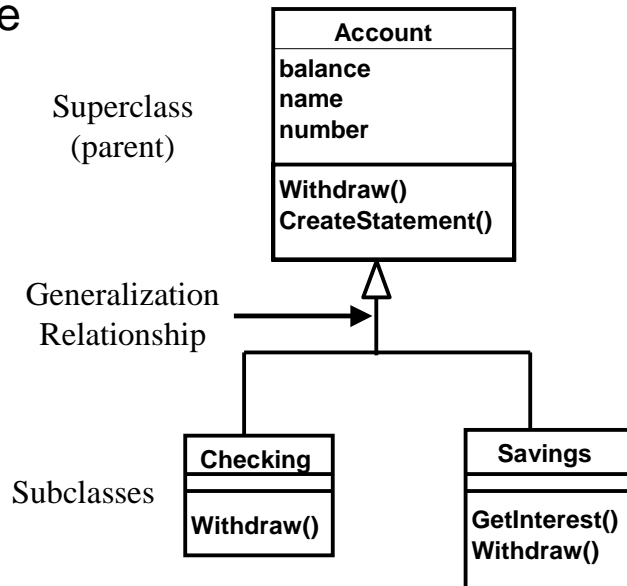
Superclass (parent)

Account

balance
name
number

Withdraw()
CreateStatement()

Generalization Relationship

Subclasses

Checking

Withdraw()

Savings

GetInterest()
Withdraw()

Rational
the e-development company

When identifying the analysis classes, especially the entity classes, inheritance relationships amongst the classes may be identified.

As discussed in the Introduction to Object Orientation, generalization is a relationship among classes where one class shares the structure and/or behavior of one or more classes.

Generalization refines a hierarchy of abstractions in which a subclass inherits from one or more superclasses.

Generalization is an "is-a-kind of" relationship. You should always be able to say that your generalized class 'is a kind of' the parent class.

In analysis, generalization should be used to model shared behavioral semantics only (i.e., generalization that passes the "is-a" test). Generalization to promote and support reuse will be defined in design. In analysis, the generalization should be used to reflect shared definitions/semantics and promote "brevity of expression" (i.e., the use of generalization makes the definitions of the abstractions easier to document and understand).

When generalization is found, a common super-class is created to contain the common attributes, associations, aggregations, and operations. The common behavior is removed from the classes which are to become sub-classes of the common super-class. A generalization relationship is drawn from the sub-class to the super-class.

Finding Generalization: Generalization of Classes

Savings | Checking | Stock

RealEstate | Bond

Asset

More general

BankAccount | Security | RealEstate

Savings | Checking | Stock | Bond
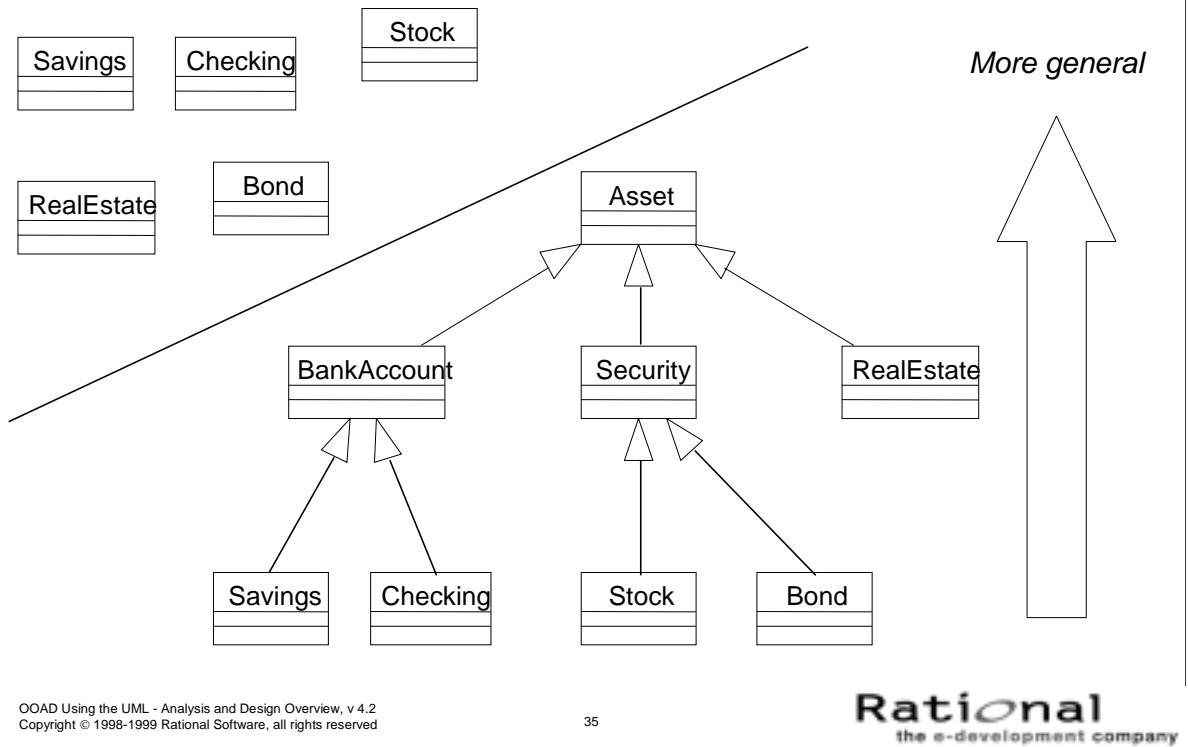
You may apply generalization when you have a set of classes that share some semantics and behavior. The shared semantics and behavior are included in the base class, the subclasses inherit from that base class, and the unique semantics and behavior are included in the subclasses.

General properties are placed in the upper part of the inheritance hierarchy, and special properties lower down.

Architectural and Use Case Analysis

Finding Generalization: Specialization of Classes
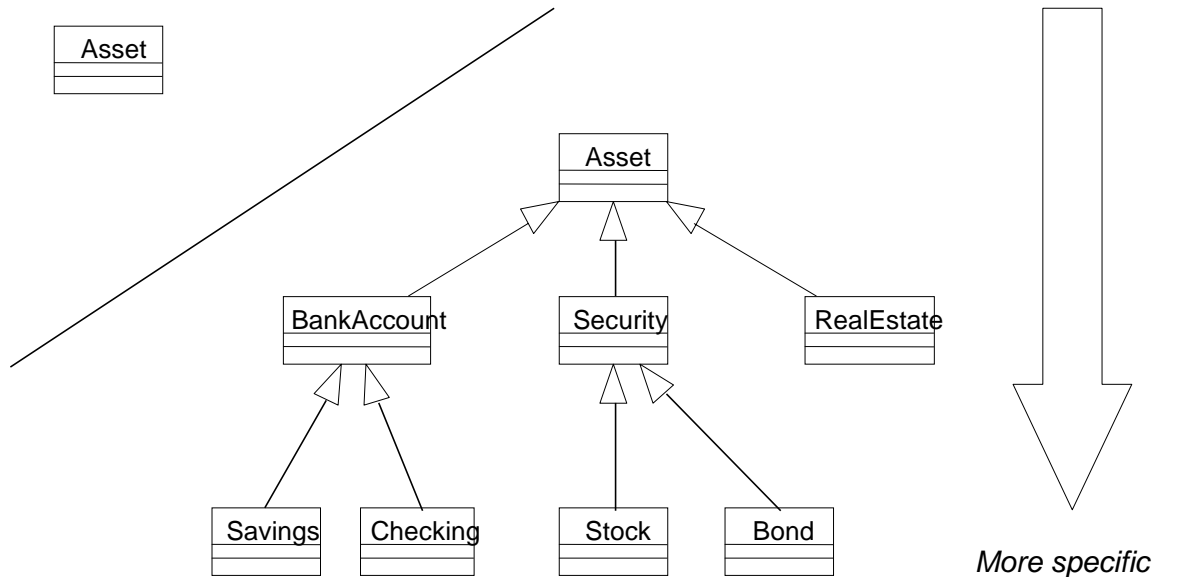
Asset

Asset

BankAccount | Security | RealEstate

Savings | Checking | Stock | Bond

More specific

36

Rational
the e-development company

You may apply generalization when you have a class with very general semantics and behavior, and you want to also model more specialized versions of the more general class. As with the previous example, the shared semantics and behavior are included in the base class, the subclasses inherit from the base class, and the unique semantics and behavior are included in the subclasses.

More specialized properties are placed in the lower part of the inheritance hierarchy.

Architectural and Use Case Analysis

Example: Generalization (Shared Semantics)
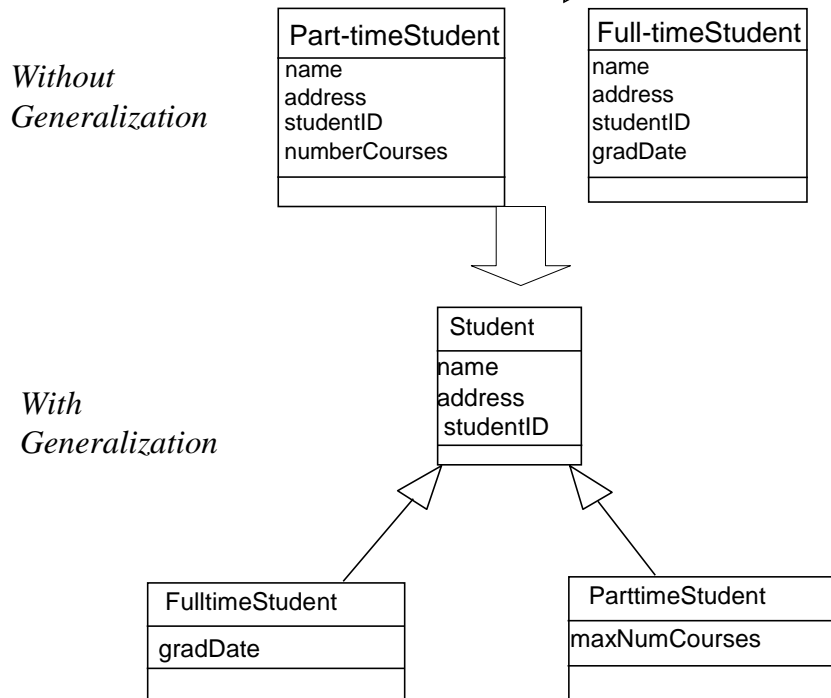
Without Generalization

Part-timeStudent
name
address
studentID
numberCourses

Full-timeStudent
name
address
studentID
gradDate

Student
name
address
studentID

With Generalization

FulltimeStudent
gradDate

ParttimeStudent
maxNumCourses

37

Rational
the e-development company

In the Course Registration System, there are two classifications of students, full- and part-time.

Full time students have an expected graduation date but part time students do not

Part time students may take a maximum of three courses where there is no maximum for full time students

Generalization can be used to model the common and unique semantics between these classifications.

What is a Control Class?

- ◆ Use-case behavior coordinator
- ◆ *One control class per use case*

Use Case

Analysis class stereotype → <<control>>

Use-case dependent, Environment independent

Rational
the e-development company

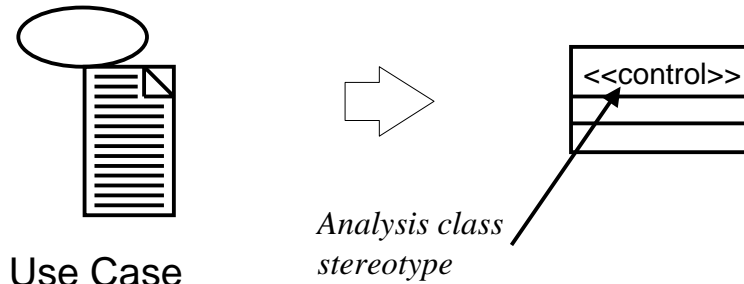Control classes provide coordinating behavior in the system. The system can perform some use cases without control classes (just using entity and boundary classes) - particularly use cases that involve only the simple manipulation of stored information. More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control classes include transaction managers, resource coordinators and error handlers.

Control classes effectively decouple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also decouple the use-case specific behavior from the entity objects, making them more reusable across use cases and systems.

Control classes provide behavior that:

•Is surroundings-independent (does not change when the surroundings change)

•Defines control logic (order between events) and transactions within a use case. Changes little if the internal structure or behavior of the entity classes changes

•Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes

•Is not performed in the same way every time it is activated (flow of events features several states)

One recommendation for the initial identification of control classes is one control class per use case.

Architectural and Use Case Analysis

The Role of a Control Class
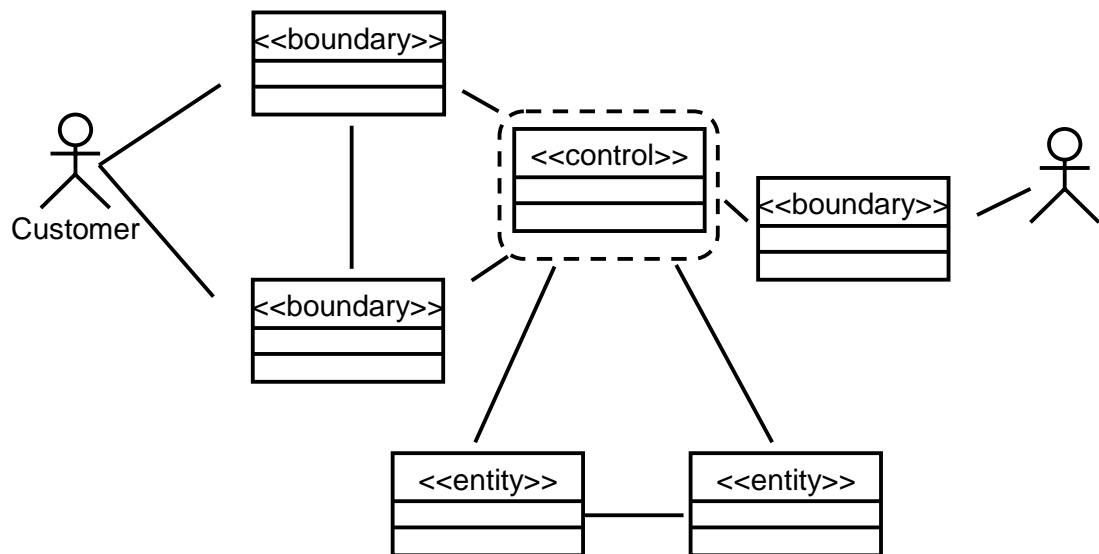
Coordinate the use-case behavior

Rational
the e-development company

A control class is a class used to model control behavior specific to one or more use cases. Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case-specific behavior.

The behavior of a control object is closely related to the realization of a specific use case. In many scenarios, you might even say that the control objects "run" the use-case realizations. However, some control objects can participate in more than one use-case realization if the use-case tasks are strongly related. Furthermore, several control objects of different control classes can participate in one use case. Not all use cases require a control object. For example, if the flow of events in a use case is related to one entity object, a boundary object may realize the use case in cooperation with the entity object. You can start by identifying one control class per use-case realization, and then refine this as more use-case realizations are identified and commonality is discovered.

Control classes can contribute to understanding the system because they represent the dynamics of the system, handling the main tasks and control flows.

When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

Architectural and Use Case Analysis

Example: Finding Control Classes

♦ One control class per use case
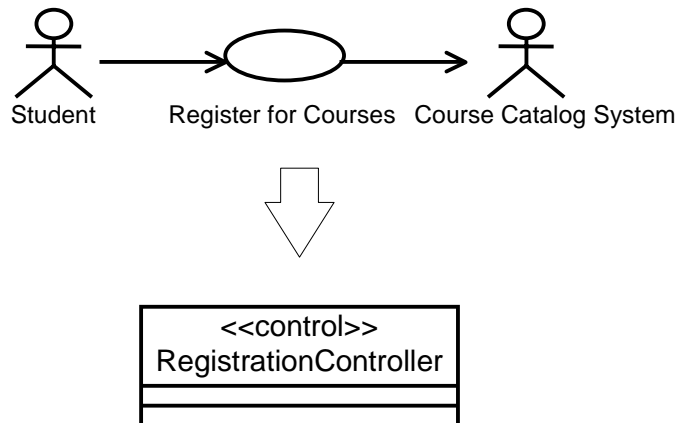
Student   Register for Courses   Course Catalog System

```
+-----------------------------+
|        <<control>>          |
|   RegistrationController    |
+-----------------------------+
|                             |
+-----------------------------+
```

One recommendation is to identify one control class per use case. Each control class is responsible for orchestrating/controlling the processing that implements the functionality described in the associated use case.

In the above example, the RegistrationController <<control>> class has been defined to orchestrate the Register for Courses processing within the system.

Architectural and Use Case Analysis

Example: Summary: Analysis Classes

Student → (Register for Courses) → Course Catalog System
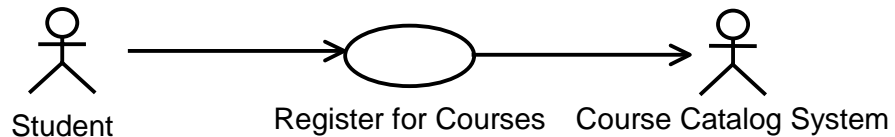
Use-Case Model

- -

Design Model
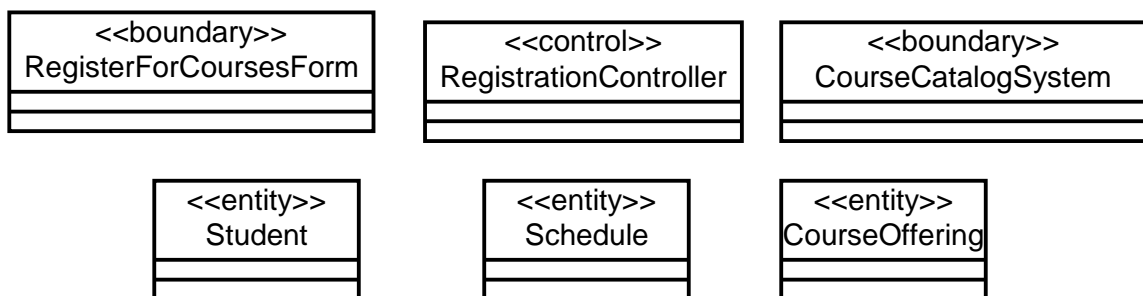
<<boundary>> RegisterForCoursesForm	<<control>> RegistrationController	<<boundary>> CourseCatalogSystem

<<entity>> Student	<<entity>> Schedule	<<entity>> CourseOffering

Rational the e-development company

For each use-case realization there is one or more class diagrams depicting its participating classes, along with their relationships. These diagrams help to ensure that there is consistency in the use-case implementation across subsystem boundaries. Such class diagrams have been called "View of Participating Classes" diagrams (VOPC, for short).

The diagram on this slide shows the classes participating in the "Register for Courses" use case. The Part-time Student and Full-time Student classes have been omitted for brevity (they both inherit from Student). Class relationships will be discussed later in this module.

Distribute Use-Case Behavior to Classes

- ◆ For each use-case flow of events:
    - Identify analysis classes
    - Allocate use-case responsibilities to analysis classes
    - Model analysis class interactions in interaction diagrams

Use Case

Sequence Diagrams

Collaboration Diagrams

Use-Case Realization

Rational
the e-development company
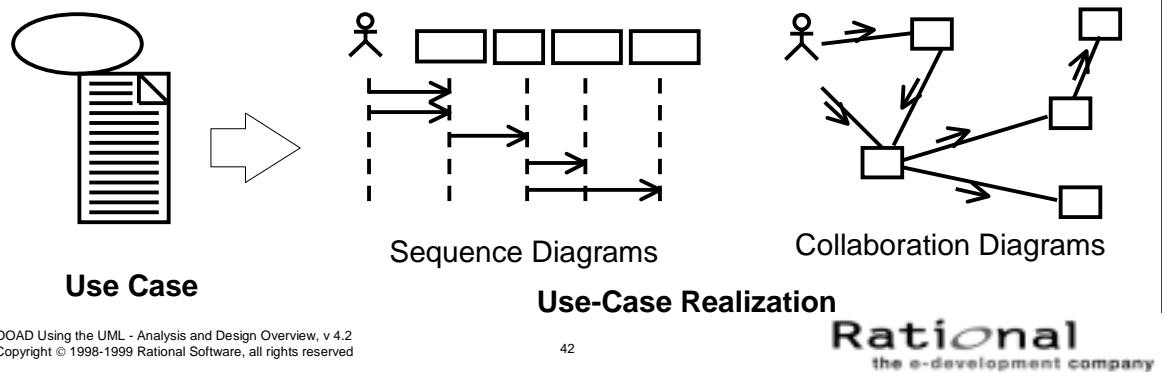
You can identify analysis classes responsible for the required behavior by stepping through the flow of events of the use case. In the previous step we outlined some classes, now it's time to see exactly where they are applied in the use-case flow of events.

In addition to the identified analysis classes, the interaction diagram should show interactions of the system with its actors (the interactions should begin with an actor, since an actor always invokes the use case). If you have several actor instances in the same diagram, try keeping them in the periphery of the diagram.

Interactions BETWEEN actors should NOT be modeled. By definition, actors are external, and are out of scope of the system being developed. Thus, you do not include interactions between actors in your system model. If you need to model interactions between entities that are external to the system you are developing (e.g., the interactions between a customer and an order agent for an order processing system), those interactions are best included in a Business Model that drives the System Model.

Guidelines for how to distribute behavior to classes is described on the next slide.

Guidelines: Allocating Responsibilities to Classes

◆ **Use analysis class stereotypes as a guide**
  ▪ Boundary Classes
    • Behavior that involves communication with an actor
  ▪ Entity Classes
    • Behavior that involves the data encapsulated within the abstraction
  ▪ Control Classes
    • Behavior specific to a use case or part of a very important flow of events

(continued)

Rational
the e-development company

The allocation of responsibilities in analysis is a crucial and sometimes difficult activity, and these stereotypes assist in that they provide a set of canned responsibilities that can be used to build a robust system by isolating the parts of the system that are most likely to change: the interface (boundary classes), the use-case flow-of-events (control classes) and the persistent data (entity classes).

Guidelines: Allocating Responsibilities to Classes (cont.)

◆ **Who has the data needed to perform the responsibility?**

- One class has the data, put the responsibility with the data
- Multiple classes have the data:
  - Put the responsibility with one class and add a relationship to the other
  - Create a new class, put the responsibility in the new class, and add relationships to classes needed to perform the responsibility
  - Put the responsibility in the control class, and add relationships to classes needed to perform the responsibility

Rational
the e-development company

A driving influence on where a responsibility should go is the location of the data needed to perform the operation.

The best case is that there is one class that has all the information needed to perform the responsibility. In that case, the responsibility goes with the data (after all, that's one of the tenets of OO -- data and operations together).

If this isn't the case, the responsibility may need to be allocated to a "third party" class that has access to the information needed to perform the responsibility. Classes and/or relationships may need to be created to make this happen. Be careful when adding relationships -- all relationships should be consistent with the abstractions they connect. Don't just add relationships to support the implementation without considering the overall affect on the model. Class relationships will be discussed later in this module.

When a new behavior is identified, check to see if there is an existing class that has similar responsibilities, reusing classes where possible. Only when sure that there is not an existing object that can perform the behavior should you create new classes.

Architectural and Use Case Analysis

The Anatomy of Sequence Diagrams

Client Object   *Supplier Object*

:Client   :Supplier

Object Lifeline

Reflexive Message

1: PerformResponsibility

This is a
sample script.

1.1: PerformAnother
Responsibility

Message

*Hierarchical Message
Numbering*

Focus of Control
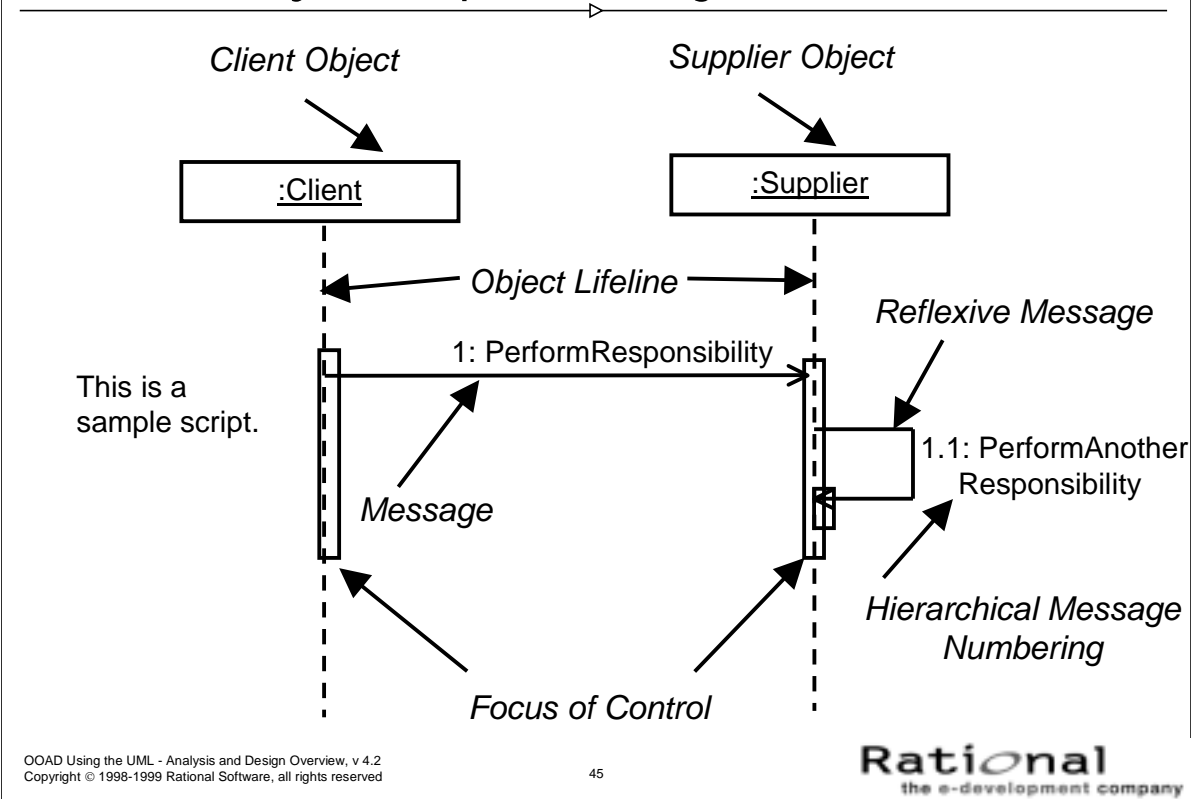
A sequence diagram describes a pattern of interaction among objects, arranged in a chronological order; it shows the objects participating in the interaction and the messages they send.

An **object** is shown as a vertical dashed line called the "lifeline". The lifeline represents the existence of the object at a particular time. An object symbol is drawn at the head of the lifeline, and shows the name of the object and its class separated by a colon and underlined.

A **message** is a communication between objects that conveys information with the expectation that activity will ensue; a message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. For a reflexive message, the arrow starts and finishes on the same lifeline. The arrow is labeled with the name of the message, and its parameters. The arrow may also be labeled with a sequence number.

Focus of control represents the relative time that the flow of control is focused in an object, thereby representing the time an object is directing messages. Focus of control is shown as narrow rectangles on object lifelines.

Hierarchical numbering bases all messages on a dependent message. The dependent message is the message whose focus of control the other messages originate in. For example, message 1.1 depends on message 1.

Scripts describe the flow of events textually.

Architectural and Use Case Analysis

Example: Sequence Diagram
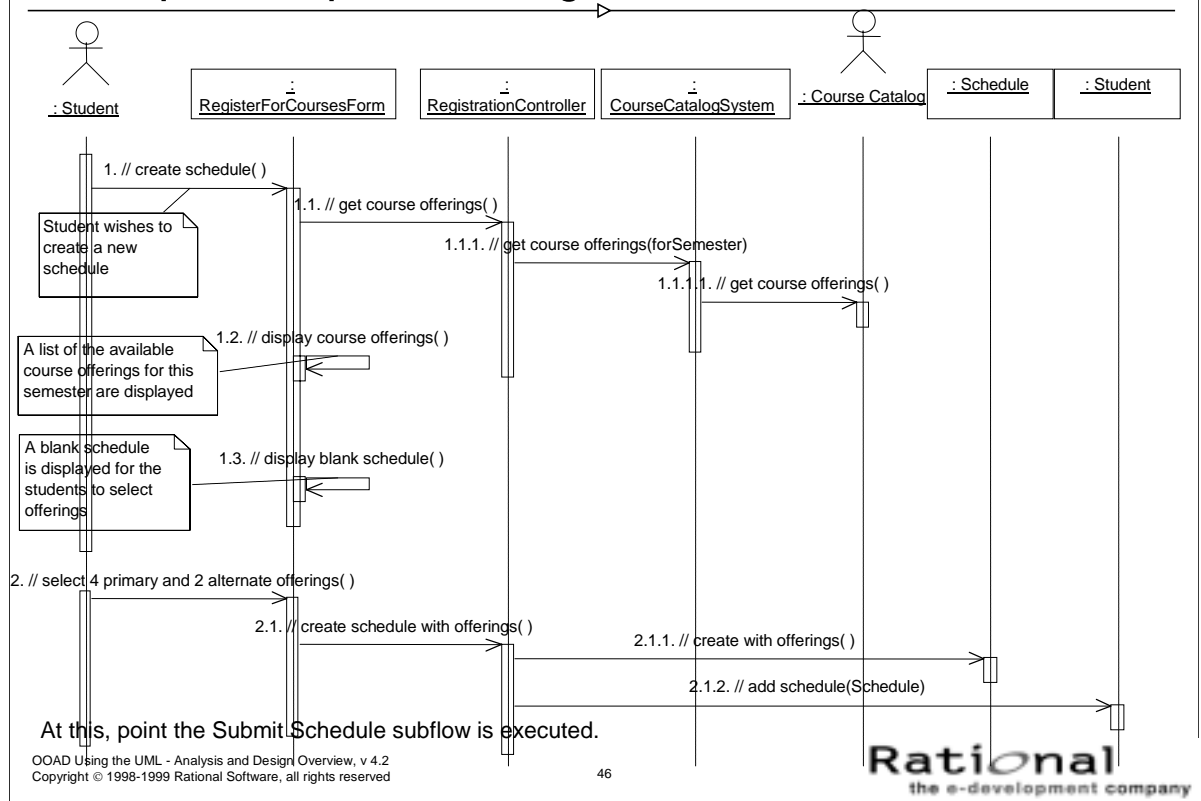
The above example shows the object interactions to support the Register for Courses use case, Create a Schedule subflow. Some responsibility allocation rationale is as follows:

The RegisterForCoursesForm knows what data it needs ton dispay and how to display it. It does not know where to go to get it. That's one of the RegistrationController's responsibilities.

Only the RegisterForCoursesForm interacts with the Student actor.

The RegistrationController understands how Students and Schedules are related.

Only the CourseCatalogSystem class interacts with the external legacy Course Catalog System.

Note the inclusion of the actors. This is important as it explicitly models what elements communicate with the "outside world".

Example: Sequence Diagram (cont.)

: Student   : RegisterForCoursesForm   : RegistrationController   : Schedule   : PrimaryScheduleOfferingInfob   : CourseOffering   : Student

1. // submit schedule( )

1.1. // submit schedule( )

1.1.1. // save( )

1.1.2. // submit( )

1.1.2.1. // is selected?( )

[is selected]

1.1.2.2. // has pre-requisites(CourseOffering)

Repeat for all primary course offerings.

An attempt is made to register the student for all selected course offerings

1.1.2.3. // still open?( )

1.1.2.4. // any conflicts?( )

[has pre-requisites, course offering open, and no schedule conflicts]

1.1.2.5. // add student(Schedule)

1.1.2.6. // mark as enrolled in( )

The above is the interaction diagram for the Submit Schedule subflow. It describes what occurs when a Student requests that an entered schedule be submitted. When a Schedule is submitted, an attempt is made to register the student for all selected primary courses.

Note the allocation of responsibility. The Schedule has been given the responsibility for performing all of the processing associated with submitting a Schedule. It orchestrates a series of checks (Student has pre-requisites, the course offering is still open, and the Student does not have any schedule conflicts), prior to enrolling the Student in the Course Offering.

Note the addition of the new class, PrimaryScheduleOfferingInfo. This class was needed to maintain the status of a particular Course Offering on a Schedule (as well as the associated grade). This will be discussed in more detail later in the module.

Architectural and Use Case Analysis

The Anatomy of Collaboration Diagrams
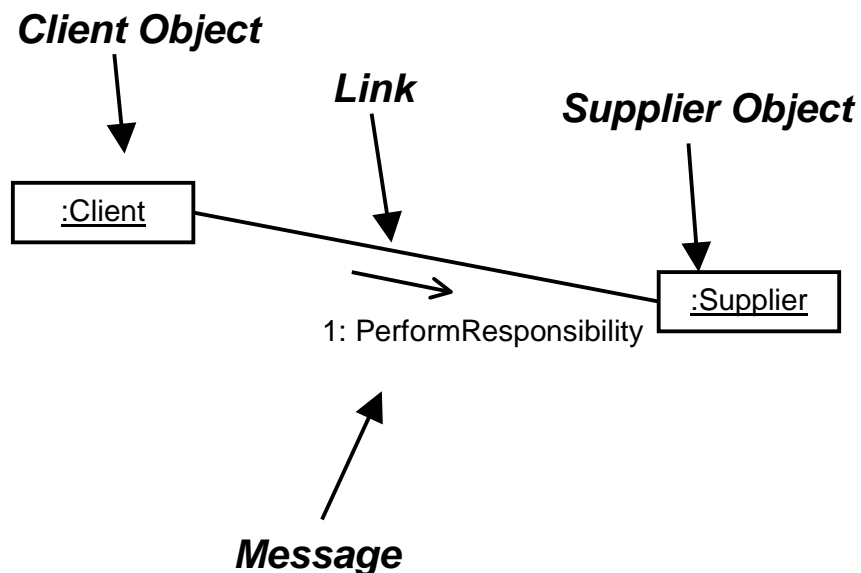
Client Object

Link

Supplier Object

:Client

1: PerformResponsibility

:Supplier

Message

Rational
the e-development company

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

An **object** is represented in three ways: Objectname:Classname, ObjectName, and :ClassName.

A **link** is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects. An object interacts with, or navigates to, other objects through its links to these objects. A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.

A **message** is a communication between objects that conveys information with the expectation that activity will ensue. In collaboration diagrams, a message is shown as a labeled arrow placed near a link. This means that the link is used to transport, or otherwise implement the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message). The arrow is labeled with the name of the message, and its parameters. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often used in collaboration diagrams, because they are the only way of describing the relative sequencing of messages. A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message. You can later assign the message by specifying the operation of the message's destination object. The specified operation will then replace the name of the message.

Architectural and Use Case Analysis
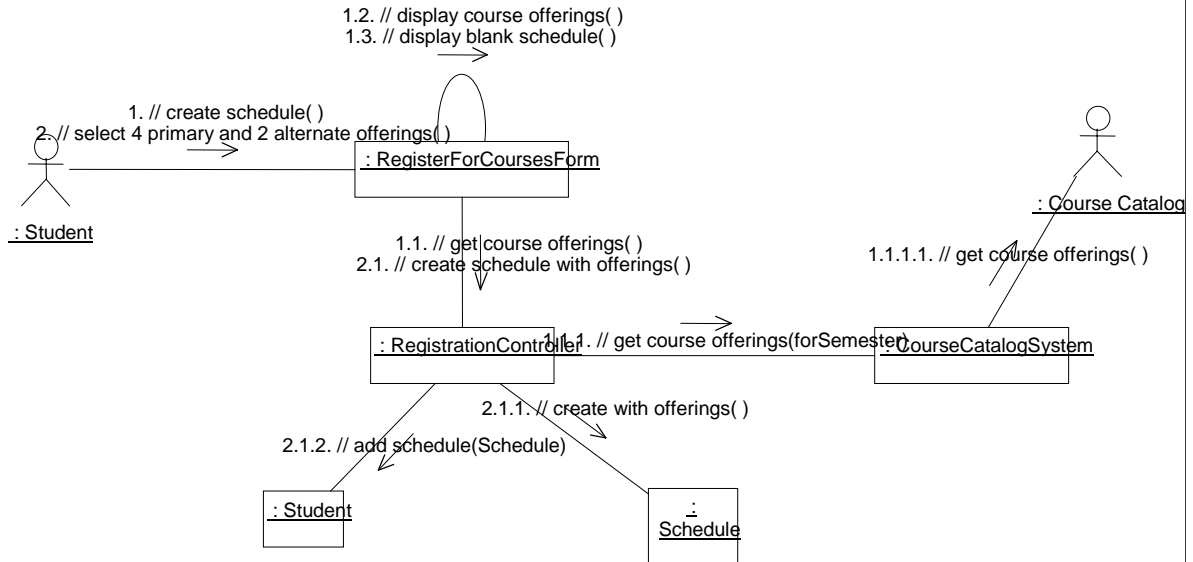
Example: Collaboration Diagram

1.2. // display course offerings( )
1.3. // display blank schedule( )

1. // create schedule( )
2. // select 4 primary and 2 alternate offerings( )

: RegisterForCoursesForm

: Student

: Course Catalog

1.1. // get course offerings( )
2.1. // create schedule with offerings( )

1.1.1.1. // get course offerings( )

: RegistrationController

1.1.1. // get course offerings(forSemester)

: CourseCatalogSystem

2.1.1. // create with offerings( )

2.1.2. // add schedule(Schedule)

: Student

: Schedule

The above example shows the collaboration of objects to support the Register for Courses use case, Create a Schedule subflow. It is the "collaboration diagram equivalent" of the sequence diagram shown earlier.

Architectural and Use Case Analysis

Example: Collaboration Diagram (cont.)

1. // submit schedule( )

: Student

: RegisterForCoursesForm

1.1. // submit schedule( )

1.1.2.4. // any conflicts?( )

1.1.1. // save( )
1.1.2. // submit( )

1.1.2.3. // still open?( )
1.1.2.5. // add student(Schedule)

:
RegistrationController

:
Schedule

:
CourseOffering

1.1.2.1. // is selected?( )
1.1.2.6. // mark as enrolled in( )

1.1.2.2. // has pre-requisites(CourseOffering)
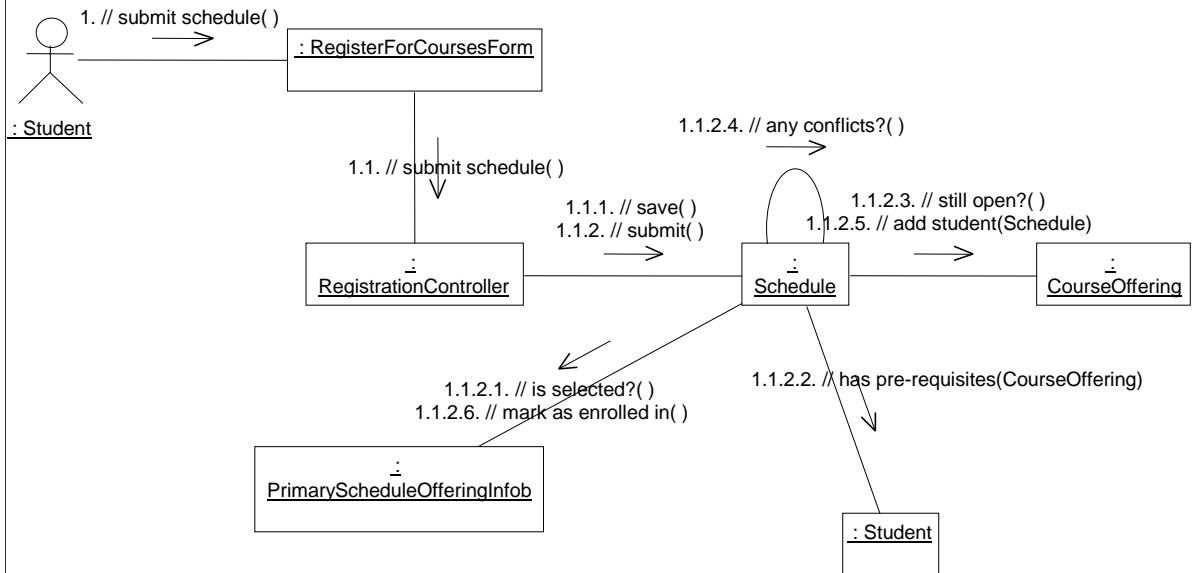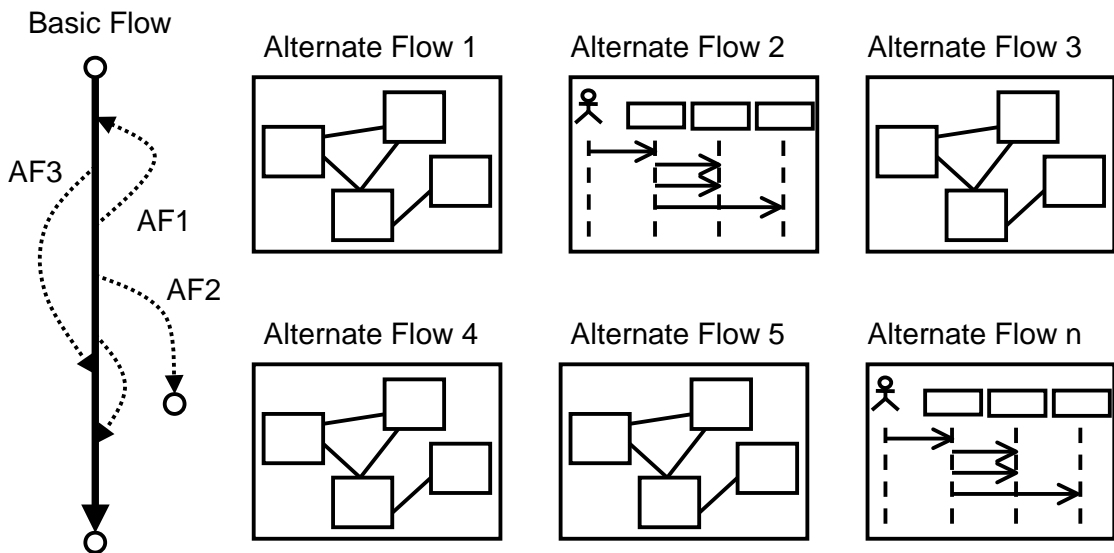
:
PrimaryScheduleOfferingInfob

: Student

Rational
the e-development company

The above example shows the collaboration of objects to support the Register for Courses use case, Submit Schedule subflow. It is the "collaboration diagram equivalent" of the sequence diagram shown earlier.

One Interaction Diagram Not Good Enough

Basic Flow

AF3

AF1

AF2

Alternate Flow 1

Alternate Flow 2

Alternate Flow 3

Alternate Flow 4

Alternate Flow 5

Alternate Flow n

51

Rational
the e-development company

You should model most of the flows of events to make sure that all requirements on the operations of the participating classes are identified. Start with describing the basic flow, which is the most common or most important flow of events. Then describe variants such as exceptional flows. You do not have to describe all the flows of events, as long as you employ and exemplify all operations of the participating objects. Very trivial flows can be omitted, such as those that concern only one object.

Examples of exceptional flows include the following.

•Error handling. What should the system do if an error is encountered?

•Time-out handling. If the user does not reply within a certain period, the use case should take some special measures

•Handling of erroneous input to the objects that participate in the use case (e.g., incorrect user input)

Examples of optional flows include the following:

•The actor decides-from a number of options-what the system is to do next

•The subsequent flow of events depends on the value of stored attributes or relationships

•The subsequent flow of events depends on the type of data to be processed

You can use either collaboration and sequence diagrams.

Architectural and Use Case Analysis

Collaboration Diagrams Vs Sequence Diagrams

- ◆ Collaboration Diagrams
  - Show relationships in addition to interactions
  - Better for visualizing patterns of collaboration
  - Better for visualizing all of the effects on a given object
  - Easier to use for brainstorming sessions

- ◆ Sequence Diagrams
  - Show the explicit sequence of messages
  - Better for visualizing overall flow
  - Better for real-time specifications and for complex scenarios

Rational
the e-development company

Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

Collaboration diagrams emphasize the structural collaboration of a society of objects and provide a clearer picture of the patterns of relationships and control that exist amongst the objects participating in a use case. Collaboration diagrams show more structural information (i.e., the relationships among objects). Collaboration diagrams are better for understanding all the effects on a given object and for procedural design.

Sequence diagrams show the explicit sequence of messages and are better for real-time specifications and for complex scenarios. A sequence diagram includes chronological sequences, but does not include object relationships. Sequence numbers are often omitted in sequence diagrams, in which the physical location of the arrow shows the relative sequence. On sequence diagrams, the time dimension is easier to read, the operations and parameters are easier to present, and the larger number of objects are easier to manage than in collaboration diagrams.

Both sequence and collaboration diagrams allow you to capture semantics of the use-case flow of events; they help identify objects, classes, interactions, and responsibilities; and they help validate the architecture.

Exercise: Use-Case Analysis, Part 1

- ◆ Given the following:
    - Use-Case Model, especially the use-case flows of events
    - Key abstractions/classes

(continued)

Rational
the e-development company

The goal of this exercise is to identify classes that must collaborate to perform a use case, allocate the use-case responsibilities to those classes, and diagram the collaborations.

Good sources for the analysis classes are the Glossary and any analysis classes defined during Architectural Analysis.

References to the givens:

•Use-Case Model: Payroll Requirements Document, Use-Case Model section

•Key abstractions: Payroll Exercise Solution, Architectural Analysis section.

Architectural and Use Case Analysis

Exercise: Use-Case Analysis, Part 1 (cont.)

- Identify the following for a particular use case:
    - The analysis classes, along with their:
        - Brief descriptions
        - Stereotypes
        - Responsibilities
    - The collaborations needed to implement the use case

(continued)

54

Rational
the e-development company

When identifying analysis classes from the use-case flows of events, use the analysis stereotypes to guide you (boundary, control, and entity).

Be sure to define the identified classes. These definitions become very important as you start to allocate responsibilities to those classes.

Architectural and Use Case Analysis

Exercise: Use-Case Analysis, Part 1 (cont.)

- ◆ Produce the following for a particular use case:
  - Use-case realization interaction diagram for at least one of the use-case flows of events

Start with diagramming the basic flow and then do the other subflows if you have time.

The interaction diagrams may be collaboration or sequence diagrams. On an interaction diagram, sending a message to an object means that you are allocating responsibility for performing that task to the object.

Be sure to use the "//" naming convention for responsibilities.

References to sample diagrams within the course that are similar to what should be produced are:

Use-case realization interaction diagram: Slide 35-36 or 38-39.

Architectural and Use Case Analysis

Use-Case Analysis Steps

- ◆ Supplement the Use-Case Descriptions
- ◆ For each use-case realization
  - ▪ Find Classes from Use-Case Behavior
  - ▪ Distribute Use-Case Behavior to Classes
- ◆ For each resulting analysis class
  - ☆ ▪ Describe Responsibilities
  - ▪ Describe Attributes and Associations
  - ▪ Qualify Analysis Mechanisms
- ◆ Unify Analysis Classes
- ◆ Checkpoints

At this point, analysis classes have been identified and use-case responsibilities have been allocated to those classes. This was done on a use-case-by-use-case basis, with a focus primarily on the use-case flows of events. Now it is time to turn our attention to each of the analysis classes and see what each of the use cases will require of them. A class and its objects often participate in several use-case realizations. It is important to coordinate all the requirements on a class and its objects that different use-case realizations may have.

The ultimate objective of these class-focused activities is to to document what the class knows and what the class does. The resulting Analysis Model gives you a big picture and a visual idea of the way responsibilities are allocated and what such an allocation does to the class collaborations. Such a view allows the analyst to spot inconsistencies in the way certain classes are treated in the system, for example, how boundary and control classes are used.

The purpose of "Describe Responsibilities" step is namely to describe the responsibilities of the analysis classes.
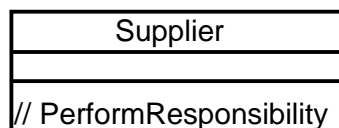
Architectural and Use Case Analysis

Describe Responsibilities

- ◆ What are responsibilities?
- ◆ How do I find them?

Interaction Diagram

| :Client | | :Supplier |

// PerformResponsibility

Class Diagram

Supplier
// PerformResponsibility

Rational the e-development company

A responsibility is a statement of something an object can be asked to provide. Responsibilities evolve into one (or more) operations on classes in design; they can be characterized as:

•The actions that the object can perform

•The knowledge that the object maintains and provides to other objects

Responsibilities are derived from messages on interaction diagrams. For each message, examine the class of the object to which the message is sent. If the responsibility does not yet exist, create a new responsibility that provides the requested behavior.

Other responsibilities will derive from non-functional requirements. When you create a new responsibility, check the non-functional requirements to see if there are related requirements which apply. Either augment the description of the responsibility, or create a new responsibility to reflect this.

Analysis class responsibilities can be documented in one of two ways:

•As "analysis" operations.
When this approach is chosen, it is important that some sort of naming convention be used. This naming convention indicates that the operation is being used to describe the responsibilities of the analysis class and that these "analysis" operations WILL PROBABLY change/evolve in design).

•Textually, in the description of the analysis classes.

For the OOAD course example, we will use the "analysis" operation approach. The naming convention that will be used is that the "analysis" operation name will be preceded by '//'.

Example: View of Participating Classes (VOPC) Class Diagram

<<boundary>>
RegisterForCoursesForm

// submit schedule()
// display course offerings()
// display schedule()
// create schedule()
// select 4 primary and 2 alternate offerings()
// display blank schedule()

<<control>>
RegistrationController

// get course offerings()
// submit schedule()
// create schedule with offerings()

<<boundary>>
CourseCatalogSystem

// get course offerings()

<<entity>>
Student

// add schedule()
// has pre-requisites()

<<entity>>
Schedule

// create with offerings()
// submit()
// save()

<<entity>>
CourseOffering
number : String = "100"
startTime : Time
endTime : Time
days : Enum

// add student()
// still open?()
// save()

<<entity>>
ScheduleOfferingInfo
status

// mark as selected()
// mark as cancelled()
// is selected?()

<<entity>>
PrimaryScheduleOfferingInfo
grade

// is enrolled in?()
// mark as enrolled in()

58

Rational
the e-development company

The View of Participating Classes (VOPC) class diagram contains the classes whose instances participate in the use-case realization interaction diagrams, as well as the relationships required to support the interactions. We will discuss the relationships later in this module. Right now, we are most interested in what classes have been identified, and what responsibilities have been allocated to those classes.

The generalization relationship has been shown because it is important to understand the relationship between the two new classes, ScheduleOfferingInfo and PrimaryScheduleOfferingInfo.

The PrimaryScheduleOfferingInfo class contains information about a primary CourseOffering that is schedule-specific. For example, the grade the student received in the CourseOffering, as well as the status of the CourseOffering on the Schedule (e.g., has the CourseOffering been enrolled in yet, or has it just been selected on the Schedule).

The ScheduleOfferingInfo class was created because status will need to be maintained for alternate CourseOfferings, as well as primary CourseOfferings, with the only difference being that Students can only be enrolled in and receive a grade in a primary CourseOffering. Thus, generalization was used to model the commonality amongst the different types of CourseOffering information.

Maintaining Consistency: What to Look For

- ◆ In order of criticality
  - ▪ Redundant responsibilities across classes
  - ▪ Disjoint responsibilities within classes
  - ▪ Class with one responsibility
  - ▪ Class with no responsibilities
  - ▪ Better distribution of behavior
  - ▪ Class that interacts with many other classes

59

Rational
the e-development company

Examine classes to ensure they have consistent responsibilities. When a class's responsibilities are disjoint, split the object into two or more classes. Update the interaction diagrams accordingly.

Examine classes to ensure that there are not two classes with similar responsibilities. When classes have similar responsibilities, combine them and update the interaction diagrams accordingly.

Sometimes you have to get back to a previous interaction diagram and redo it: a better distribution of behavior may have become evident while you were working on another interaction diagram. It is better (and easier) to change things now than later in design. Take the time to set the diagrams right, but don't get hung-up trying to optimize the class interactions.

A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed. Be prepared to challenge and justify the existence of all classes.

Architectural and Use Case Analysis

Use-Case Analysis Steps

- ◆ Supplement the Use-Case Descriptions
- ◆ For each use-case realization
  - ▪ Find Classes from Use-Case Behavior
  - ▪ Distribute Use-Case Behavior to Classes
- ◆ For each resulting analysis class
  - ▪ Describe Responsibilities
  - ▪ Describe Attributes and Associations
  - ▪ Qualify Analysis Mechanisms
- ◆ Unify Analysis Classes
- ◆ Checkpoints

60

Rational
the e-development company

Now that we have defined the analysis classes and their responsibilities, and have an understanding of how they need to collaborate, we can continue our documentation of the analysis classes by describing their attributes and associations.

The purpose of "Describe Attributes and Operations" is to

•Identofy the other classes on which the analysis class depend

•Define the events in other analysis classes that the class must know about

•Define the information that the analysis class is responsible for maintaining

Describe Attributes and Associations

- ◆ Define Attributes
- ◆ Establish Aggregations and Associations

61

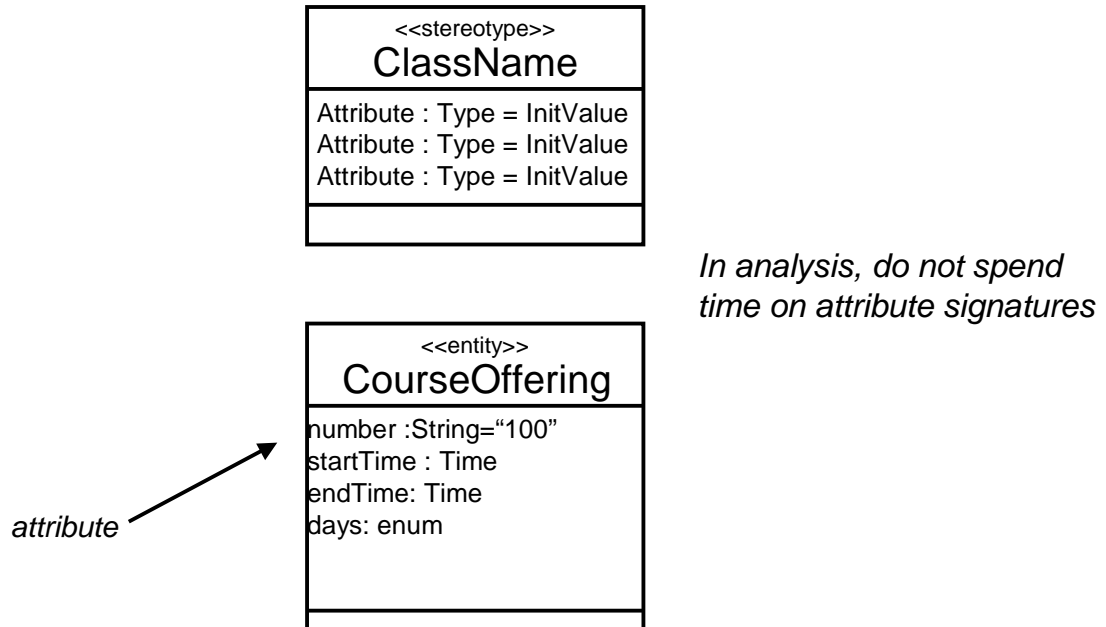**Rational**
the e-development company

In order to carry-out their responsibilities, classes frequently depend on other classes to supply needed behavior. Associations document the inter-class dependencies and help us to understand class coupling; better understanding of class coupling, and reduction of coupling where possible, can help us build better, more resilient systems.

Architectural and Use Case Analysis

Review: What is an Attribute?

```
              <<stereotype>>
              ClassName
   ──────────────────────────────
   Attribute : Type = InitValue
   Attribute : Type = InitValue
   Attribute : Type = InitValue
   ──────────────────────────────

```

In analysis, do not spend time on attribute signatures

```
              <<entity>>
            CourseOffering
   ──────────────────────────────
   number :String="100"
   startTime : Time
   endTime: Time
   days: enum
   ──────────────────────────────

```

attribute

Rational
the e-development company

Attributes are used to store information. Attributes are atomic things with no responsibilities.

The attribute name should be a noun that clearly states what information the attribute holds. The description of the attribute should describe what information is to be stored in the attribute; this can be optional when the information stored is obvious from the attribute name.

During analysis, the attribute types should be from the domain, and not adapted to the programming language in use. For example, in the above diagram, enum will need to be replaced with a true enumeration that describes the days the CourseOffering is offered (e.g., MWF, TR, etc.).

Finding Attributes

- ◆ Properties/characteristics of identified classes
- ◆ Information retained by identified classes
- ◆ "Nouns" that did not become classes
    - ▪ Information whose value is the important thing
    - ▪ Information that is uniquely "owned" by an object
    - ▪ Information that has no behavior

Rational
the e-development company

Sources of possible attributes: domain knowledge, requirements, glossary, domain model, business model, etc.

Attributes are used instead of classes where :

•Only the value of the information, not it's location, is important

•The information is uniquely "owned" by the object to which it belongs; no other objects refer to the information

•The information is accessed by operations which only get, set or perform simple transformations on the information; the information has no "real" behavior other than providing its value

If, on the other hand, the information has complex behavior, or is shared by two or more objects the information should be modeled as a separate class.

Attributes are domain dependent (an object model for a system, includes those characteristics that are relevant for the problem domain being modeled).

Remember, the process is use-case-driven. Thus, all discovered attributes should support at least one use case. For this reason, the attributes that are discovered are affected by what functionality/domain being modeled.

Review: What is an Association?

- ◆ Models a semantic connection among instances

```
┌─────────────────┐                          ┌─────────────────┐
│   <<entity>>    │                          │   <<entity>>    │
│    Student      │                          │  CourseOffering │
├─────────────────┤        Simple            ├─────────────────┤
├─────────────────┤       association        ├─────────────────┤
└─────────────────┘                          └─────────────────┘
              ↘                          ↙
                    ┌─────────────────┐              is a pre-requisite of
                    │   <<entity>>    │           ┌──────┐
                    │    Schedule     │           │      ┌─────────────────┐
                    ├─────────────────┤           │      │   <<entity>>    │
                    ├─────────────────┤           └──────│     Course      │
                    └─────────────────┘         ↗        ├─────────────────┤
                                     Reflexive           ├─────────────────┤
                                    association           └─────────────────┘
```
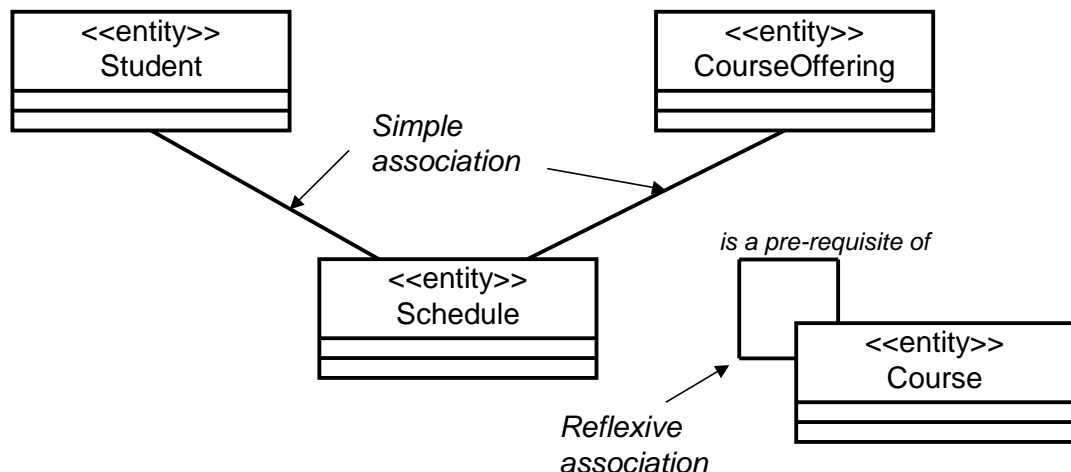
Association is a structural relationship

Rational
the e-development company

Associations represent structural relationships between objects of different classes, information that must be preserved for some duration and not simply procedural dependency relationships (e.g., one object has a permanent association to another object).

You can use associations to show that objects know about other objects. Sometimes, objects must hold references to each other to be able to interact, for example send messages to each other; thus, in some cases associations may follow from interaction patterns in sequence diagrams or collaboration diagrams.

Most associations are simple (exist between exactly two classes), and are drawn as solid paths connecting pairs of class symbols. Ternary relationships are also possible. Sometimes, a class has an association to itself. This does not necessarily mean that an instance of that class has an association to itself; more often, it means that one instance if the class has associations to other instances of the same class.

An association may have a name that is placed on, or adjacent to the association path. The name of the association should reflect the purpose of the relationship and be a verb phrase; the name of an association can be omitted, particularly if role names are used (see next slide).

Avoid names like "has" and "contains", as they add no information about what the relationships are between the classes.

In the above example, Students and CourseOfferings are related via the Schedule class. This is because in the Course Registration System, Schedule is a first class citizen. Students are enrolled in a CourseOffering if there is a relationship between the Student's Schedule and the CourseOffering.

Review: What are Roles?

- ◆ The "face" that a class plays in the association
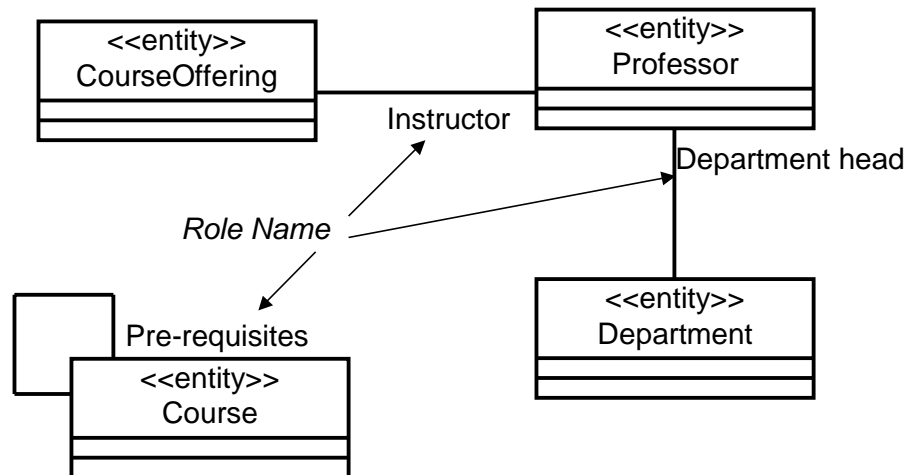
Each end of an association is a role specifying the face that a class plays in the association. Each role must have a name, and the role names opposite a class must be unique. The role name should be a noun indicating the associated object's role in relation to the associating object.

The use of association names and role names is mutually exclusive: one would not use both an association name and a role name. For each association, decide which conveys more information.
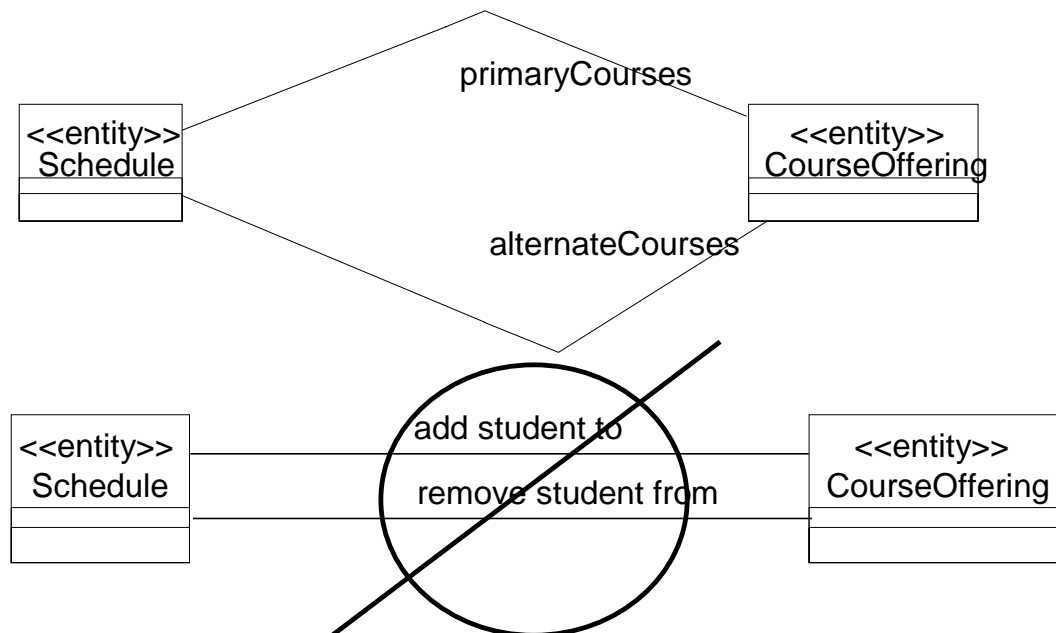
The role name is placed next to the end of the association line of the class it describes.

In the case of self-associations, role names are essential to distinguish the purpose for the association.

In the above example, the Professor participates in two separate association relationships, playing a different role in each.

Architectural and Use Case Analysis

Example: Multiple Associations

primaryCourses

<<entity>>
Schedule

<<entity>>
CourseOffering

alternateCourses

add student to

remove student from

<<entity>>
Schedule

<<entity>>
CourseOffering

Multiple associations must reflect multiple roles

Rational
the e-development company

There can be multiple associations between the same two classes, but they should represent distinct relationships, DIFFERENT ROLES; they should not be just for invoking different operations.

If there is more than one association between two classes then they MUST be named.

It is unusual to find more than one association between the same two classes. Occurrences of multiple associations should be carefully examined.

To determine if multiple associations are appropriate, look at instances of the classes. If ClassA and ClassB have 2 associations between them, Assoc1 and Assoc2. If an instance of ClassA has a link with two SEPARATE instances of ClassB, then multiple associations are valid.

In the above example, the top diagram is an appropriate use of multiple associations, the bottom diagram is not. In the valid case, two associations are required between Schedule and CourseOffering as a Schedule can contain two kind of CourseOfferings, primary and alternate. These must be distinguishable, so two separate associations are used. In the invalid case, the two relationships represent two operations of CourseOffering, not two roles of CourseOffering.

Remember, Students and CourseOfferings are related via the Schedule class. Students are enrolled in a CourseOffering if there is a relationship between the Student's Schedule and the CourseOffering.

Architectural and Use Case Analysis

Review: Multiplicity

- Unspecified

- Exactly one

- Zero or more (many, unlimited)

- One or more

- Zero or one (optional scalar role)

- Specified range

- Multiple, disjoint ranges

_____

_____
1

_____
0..*

_____
*

_____
1..*

_____
0..1

_____
2..4

_____
2, 4..6

Rational
the e-development company

For each role you can specify the multiplicity of its class, how many objects of the class can be associated with one object of the other class.

Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges. A range is indicated by an integer (the lower value), two dots, and an integer (the upper value). A single integer is a valid range.

During analysis, assume a multiplicity of 0..* (zero to many) unless there is some clear evidence of something else. A multiplicity of zero implies that the association is optional; make sure you mean this; if an object might not be there, operations which use the association will have to adjust accordingly. Narrower limits for multiplicity may be specified (such as 2..4).

Within multiplicity ranges, probabilities may be specified. Thus, if the multiplicity is 0..*, is expected to be between 10 and 20 in 85% of the cases, make note of it; this information will be of great importance during design. For example, if persistent storage is to be implemented using a relational database, narrower limits will help better organize the database tables.

Architectural and Use Case Analysis

Example: Multiplicity
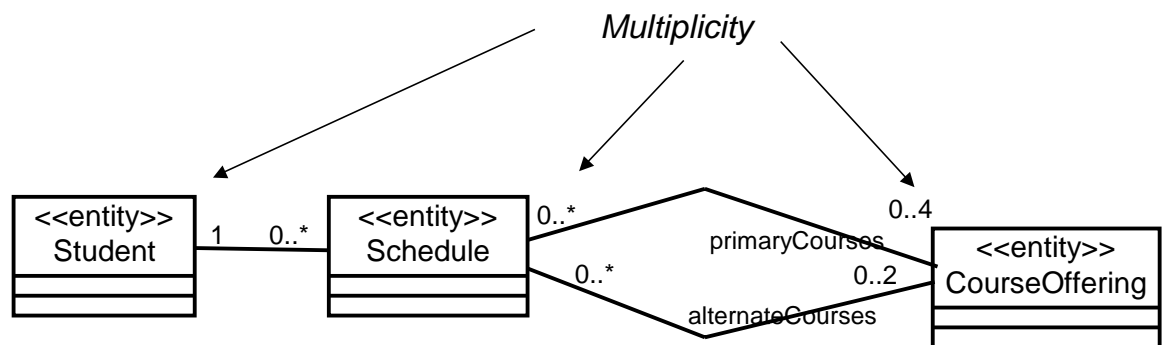
Multiplicity

<<entity>> Student	1 0..*	<<entity>> Schedule	0..* primaryCourses 0..4

<<entity>> CourseOffering

0..*   alternateCourses   0..2

A Student can have zero or more Schedules. It is zero when the Student has not registered for any course offerings yet. It is more than one because the system retains schedules for multiple semesters.

A Schedule is only associated with one Student.

A Schedule can contain up to four primary course offerings and up to two alternate course offerings.

A CourseOffering can appear on any number of Schedules, as either a primary course or as an alternate course.

A CourseOffering does not have to appear on any Schedule.

Remember, Students and CourseOfferings are related via the Schedule class. Students are enrolled in a CourseOffering if there is a relationship between the Student's Schedule and the CourseOffering.
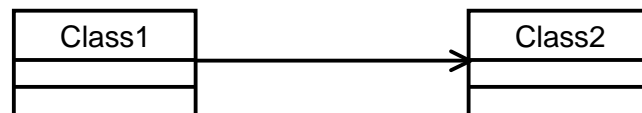
Review: Navigability

- ◆ **Possible to navigate from an associating class to the target class**

Bi-directional
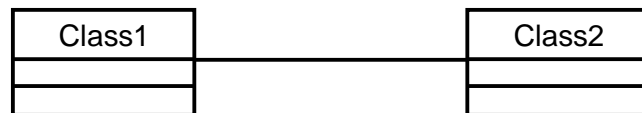
| Class1 | | Class2 |

Uni-directional

| Class1 | → | Class2 |

The navigability property on a role indicates that it is possible to navigate from an associating class to the target class using the association.

Navigability is indicated by an open arrow, which is placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is true (e.g., associations are bi-directional, by default).

Note that usually arrows are suppressed for associations and aggregations with navigability in both directions (e.g., when no arrowheads are shown, the association is assumed to be navigable in both directions). Arrows are shown only for associations with one-way navigability.

The interaction diagrams can be used to identify the needed navigation between classes. Navigability is inherently a design and implementation property, though it can be specified in analysis, with the understanding that it may need to be refined in design (Class Design, to be specific). In analysis, many associations are modeled as being bi-directional. During design, we look at whether it is really necessary to navigate in both directions.

Architectural and Use Case Analysis

Example: Navigability
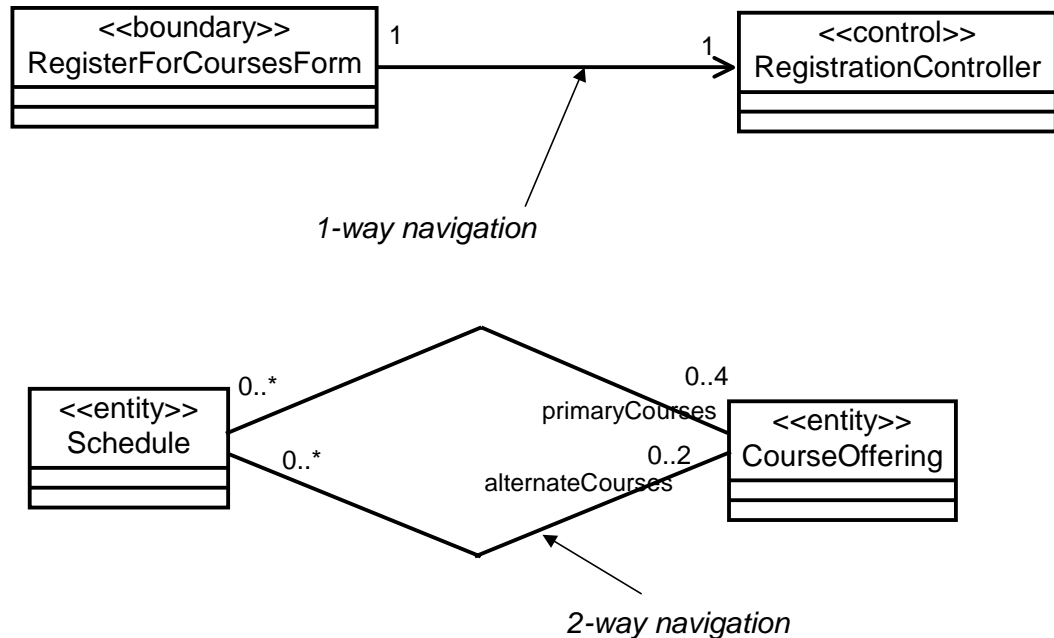
<<boundary>> RegisterForCoursesForm	1 → 1	<<control>> RegistrationController

1-way navigation

| <<entity>> Schedule | 0..* primaryCourses 0..* alternateCourses | 0..4 0..2 <<entity>> CourseOffering |

2-way navigation

The top part of the above diagram demonstrates one-way navigation: A RegisterForCoursesForm invokes a single RegistrationController that will process the registration for the current Student (the Student that is building the current Schedule). The RegistrationController will never need to communicate directly to the RegisterForCoursesForm.

The bottom part of the above diagram demonstrates two-way navigation: You can ask a Schedule what CourseOfferings it contains and you can ask a CourseOffering what Schedules it appears on. This is necessary, so that when a CourseOffering is cancelled, the appropriate Schedules can be updated.

Architectural and Use Case Analysis

Review: What is Aggregation?

♦ A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts

Whole/aggregate   part

```
<<entity>>     1   0..*  <<entity>>   0..*        primaryCourses    0..4   <<entity>>
Student  ◇————————————  Schedule                                          CourseOffering
                              0..*   alternateCourses   0..2
```

Rational
the e-development company

Aggregation is a stronger form of association which is used to model a whole-part relationship between model elements. The whole/aggregate has an aggregation association to the its constituent parts. A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.

Since aggregation is a special form of association, the use of multiplicity, roles, navigation, etc. is the same as for association.

Sometimes, a class may be aggregated with itself. This does not mean that an instance of that class is composed of itself (this would be silly), it means that one instance if the class is an aggregate composed of other instances of the same class.

Some situations where aggregation may be appropriate:

•An object is physically composed of other objects (e.g. car being physically composed of an engine and four wheels).

• An object is a logical collection of other objects (e.g., a family is a collection of parents and children).

• An object physically contains other objects (e.g., an airplane physically contains a pilot).

In the above example, the relationship from Student to Schedule is modeled as an aggregation because a Schedule is inherently tied to a particular Student. A Schedule outside of the context of a Student makes no sense in this Course Registration System. The relationship from Schedule to CourseOffering is an association because CourseOfferings may appear on multiple Schedules.

Association or Aggregation?

◆ Consideration

 ▪ Context, independent identity of Class2

Class1		association		Class2

Class1	◇	aggregation		Class2
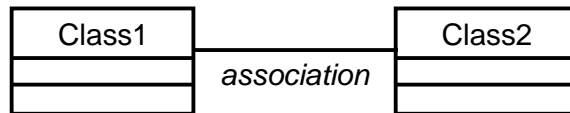
When in doubt use association

OOAD Using the UML - Analysis and Design Overview, v 4.2
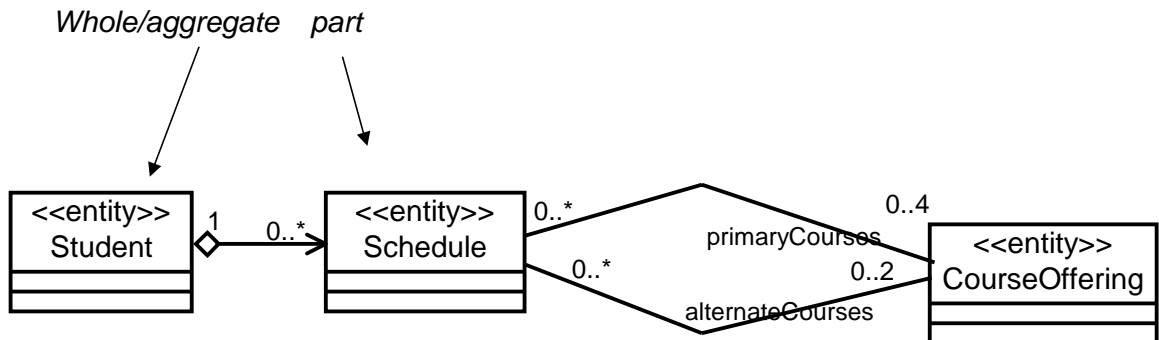Copyright © 1998-1999 Rational Software, all rights reserved   72

Rati⊘nal
the e-development company

Aggregation should be used only where the "parts" are incomplete outside the context of the whole. If the classes can have independent identity outside the context provided by other classes, if they are not parts of some greater whole, then the association relationship should be used.

When in doubt, an association is more appropriate. Aggregations are generally obvious. A good aggregate should be a natural, coherent part of the model. The meaning of aggregates should be simple to understand from the context. Choosing aggregation is only done to help clarify, it is not something that is crucial to the success of the modeling effort.

The use of aggregation versus association is dependent on the application you are developing. For example, if you are modeling a car dealership, the relationship between a car and its wheels might be modeled as aggregation because the car always comes with wheels and the wheels are never sold separately. However, if you are modeling a car parts store, you might model the relationship as an association, as the car (the body) might be independent of the wheels.

Association Class

- ◆ A class "attached" to an association
- ◆ Contains properties of a relationship
- ◆ One instance per link
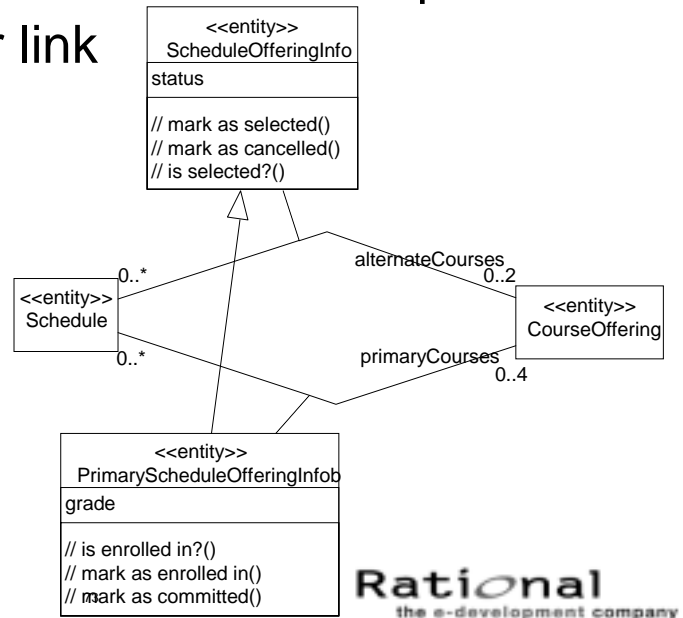
```
                          <<entity>>
                       ScheduleOfferingInfo
                       status
                       // mark as selected()
                       // mark as cancelled()
                       // is selected?()
```

```
                                        alternateCourses
                         0..*                              0..2
   <<entity>>                                                     <<entity>>
   Schedule                                                      CourseOffering
                         0..*          primaryCourses
                                                          0..4
```

```
                       <<entity>>
                 PrimaryScheduleOfferingInfob
                 grade
                 // is enrolled in?()
                 // mark as enrolled in()
                 // mark as committed()
```

Rational
the e-development company

An association class is a class that is connected to an association. It is a full-fledged class and can contain attributes, operations and other associations.

Association classes allow you to store information about the relationship itself. Such information is not appropriate, or does not belong, within the classes at either end of the relationship.

There is an instance of the association class for every instance of the relationship (e.g., for every link).
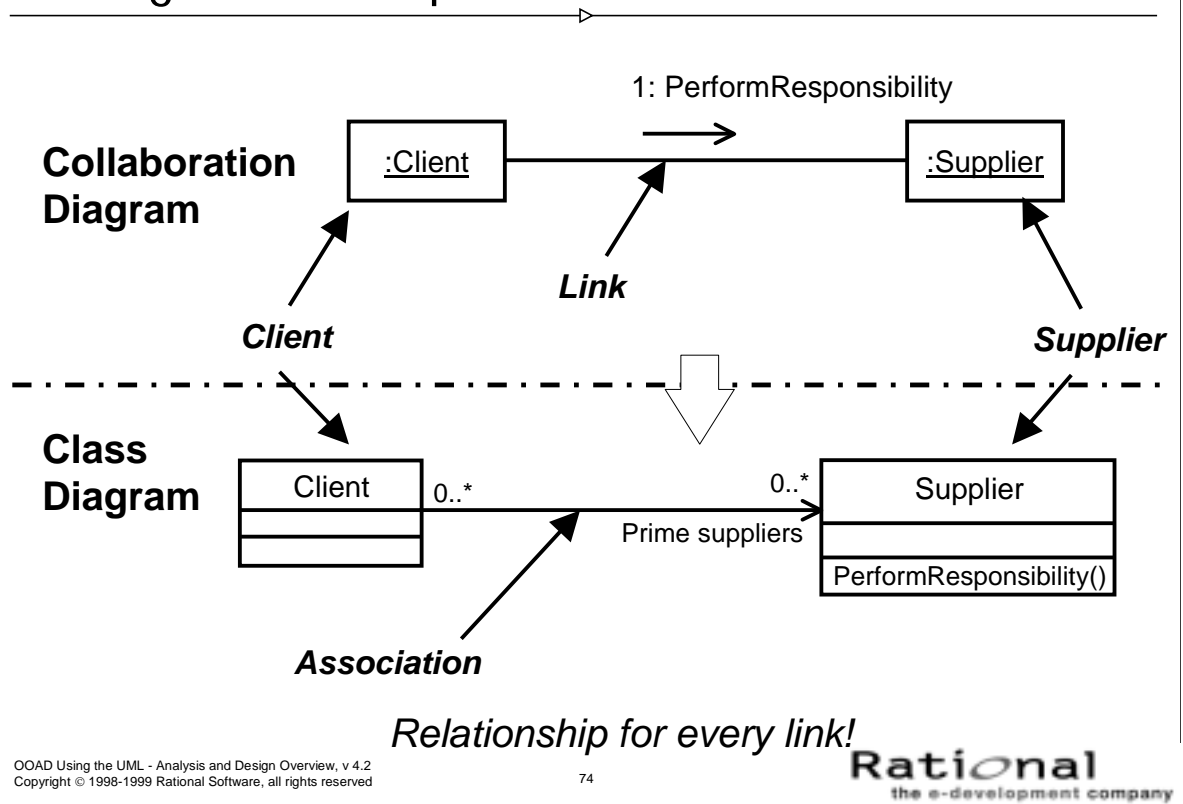
In many cases, association classes are used to resolve many-to-many relationships, as shown in the example above. In this case, a Schedule includes multiple primary CourseOfferings and a CourseOffering can appear on multiple schedules as a primary. Where would a Student's grade for a primary CourseOffering "live"? It cannot be stored in Schedule because a Schedule contains multiple primary CourseOfferings. It cannot be stored in CourseOffering because the same CourseOffering can appear on multiple Schedules as primary. Grade is really an attribute of the relationship between a Schedule and a primary CourseOffering.

The same is true of the status of a CourseOffering, primary or alternate, on a particular Schedule.

Thus, association classes were created to contain such information. Two classes related by generalization were created to leverage the similarities between what must be maintained for primary and alternate CourseOfferings. Remember, Students can only enrol in and receive a grade in a primary CourseOffering, not an alternate.

Architectural and Use Case Analysis

Finding Relationships

Collaboration Diagram

1: PerformResponsibility

:Client → :Supplier

Client

Link

Supplier

Class Diagram

Client 0..* — 0..* Supplier

Prime suppliers

PerformResponsibility()

Association

Relationship for every link!

Rational
the e-development company

To find relationships, start studying the links in the collaboration diagrams. Links between classes indicate that objects of the two classes need to communicate with one another to perform the use case. Thus, an association or an aggregation is needed between the associated classes (the difference between these relationship types and when to use which was discussed earlier).

Reflexive links do not need to be instances of reflexive relationships; an object can send messages to itself. A reflexive relationship is needed when two different objects of the same class need to communicate.

The navigability of the relationship should support the required message direction. In the above example, if navigability was not defined from the Client to the Supplier, then the PerformResponsibility message could not be sent from the Client to the Supplier.
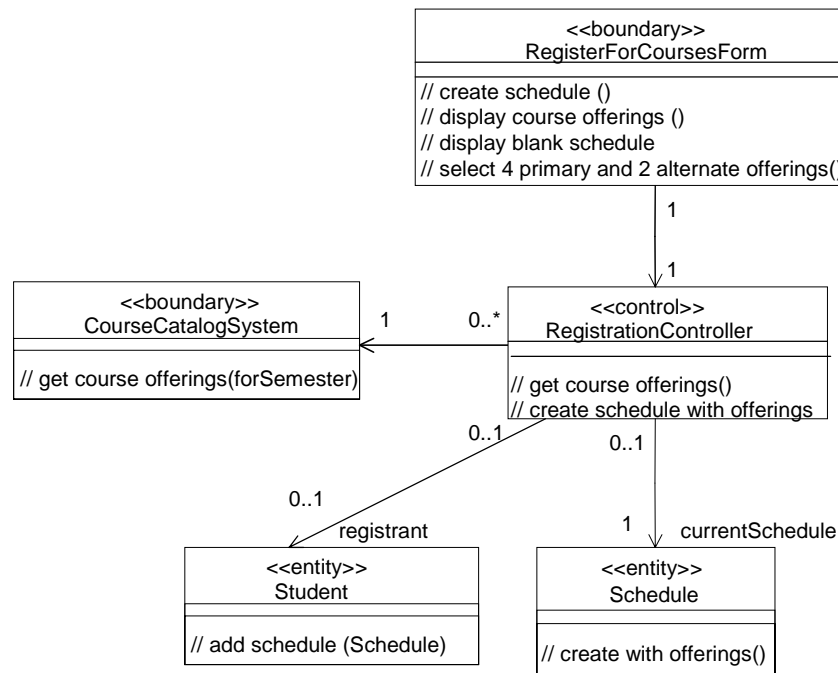
Focus only on associations needed to realize the use cases; don't add association you think "might" exist unless they are required based on the interaction diagrams.

Remember to give the associations role names and multiplicities. You can also specify navigability, though this will be refined in Class Design.

Write a brief description of the association to indicate how the association is used, or what relationships the association represents.

Example: VOPC: Finding Relationships

```
                                    <<boundary>>
                                RegisterForCoursesForm
                          ─────────────────────────────────────
                          // create schedule ()
                          // display course offerings ()
                          // display blank schedule
                          // select 4 primary and 2 alternate offerings()
                                                │ 1
                                                │
                                                │ 1
         <<boundary>>                       <<control>>
       CourseCatalogSystem      1    0..*  RegistrationController
   ──────────────────────────   ◁──────   ──────────────────────────
   // get course offerings(forSemester)    // get course offerings()
                                           // create schedule with offerings
                           0..1                    0..1
                    0..1
                         registrant            1    currentSchedule
                  <<entity>>                  <<entity>>
                    Student                    Schedule
              ─────────────────           ─────────────────
              // add schedule (Schedule)   // create with offerings()
```

The diagram on this slide shows classes that collaborate to perform the "Register for Courses" use case, along with their relationships. Again, this diagram is called a View of Participating Classes (VOPC) diagram.

Note: The complete VOPC was too big to fit on a single slide, so some of the classes are shown above and the remaining appear on the next slide.

The relationships were defined based on the interaction diagrams for the "Register for Courses" use case provided earlier in this module.

Rationale for relationships:

•From RegisterForCoursesForm to RegistrationController: There is one controller for each Schedule being created (e.g., each Student registration session).

•From RegistrationController to CourseCatalogSystem: There's only one CourseCatalogSystem instance for possibly many RegistrationControllers. This serializes access to the legacy system.

•From RegistrationController to Student. A RegistrationController deals with one Student at a time (the Student currently registering for courses). Note the use of the "registrant" role name.

•From RegistrationController to Schedule. A RegistrationController deals with one Schedule at a time (the current Schedule for the Student registering for courses). Note the use of the "currentSchedule" role name.

Note: Many RegisterForCoursesForms can be active at one time (for different sessions/students), each with their own RegistrationController.

Architectural and Use Case Analysis

Example: VOPC: Finding Relationships (contd)
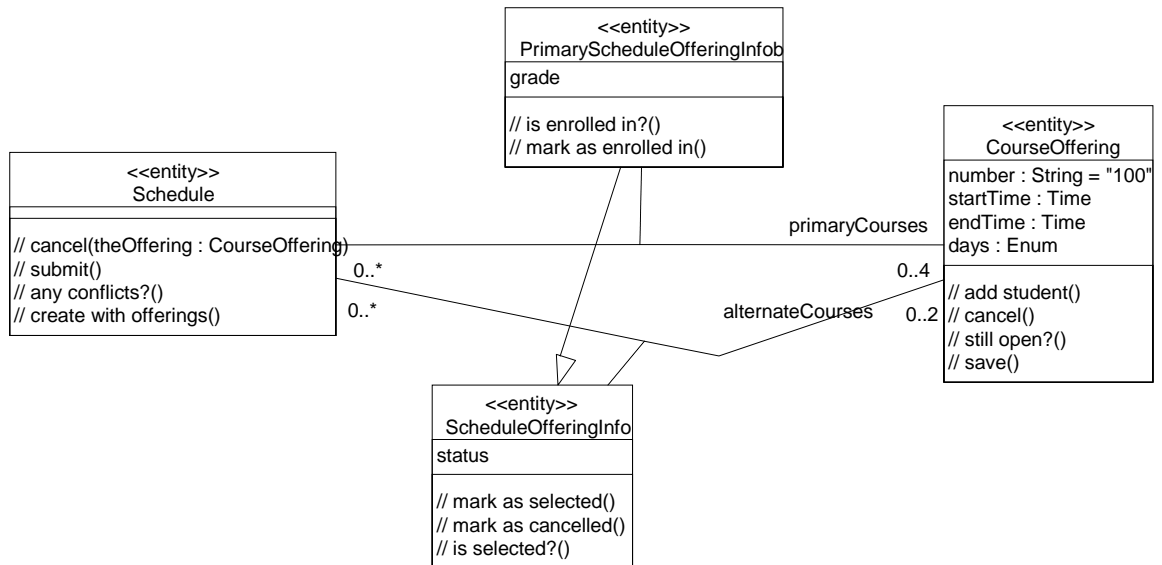
The above are the remaining classes from the Register for Courses VOPC. The complete VOPC was too big to fit on a single slide, so some of the classes are shown above and the remaining were shown on the previous slide.

The relationships were defined based on the interaction diagrams for the "Register for Courses" use case provided earlier in this module.

Rationale for relationships:

• From Schedule to Course Offering:
Each Schedule may have up to four primary Course Offerings, and up to two alternate Course Offerings.
A particular Course Offering may appear on many Schedules, as either a primary or an alternate.

• Association classes, ScheduleOfferingInfo and PrimaryScheduleOfferingInfo: Status information must be maintained for each CourseOffering on each Schedule, and for primary CourseOfferings, the Student's grade in the CourseOffering must be maintained. Thus, the ScheduleOfferingInfo class was created because status will need to be maintained for alternate Course Offerings, as well as primary Course Offerings, with the only difference being that Students can only be enrolled in and receive a grade in a Primary Course Offering. Thus, generalization was used to model the commonality amongst the different types of CourseOffering information.

Use-Case Analysis Steps

- ◆ Supplement the Use-Case Descriptions
- ◆ For each use-case realization
  - ▪ Find Classes from Use-Case Behavior
  - ▪ Distribute Use-Case Behavior to Classes
- ◆ For each resulting analysis class
  - ▪ Describe Responsibilities
  - ▪ Describe Attributes and Associations
  - ☆ ▪ Qualify Analysis Mechanisms
- ◆ Unify Analysis Classes
- ◆ Checkpoints

OOAD Using the UML - Analysis and Design Overview, v 4.2
Copyright © 1998-1999 Rational Software, all rights reserved   77

Rational
the e-development company

At this point, we have a pretty good understanding of the analysis classes, their responsibilities, and the collaborations required to support the functionality described in the use cases.

Now we must look into how each of the defined analysis classes implements the analysis mechanisms identified in Architectural Analysis.

The purpose of "Qualify Analysis Mechanisms" is to:

•Identify analysis mechanisms (if any) used by the class

•Provide additional information about how the class applies the analysis mechanism

For each such mechanism, qualify as many characteristics as possible, giving ranges where appropriate, or when there is still much uncertainty

Different architectural mechanisms will have different characteristics, so this information is purely descriptive and need only be as structured as necessary to capture and convey the information. During analysis, this information is generally quite speculative, but capturing has value since conjectural estimates can be revised as more information is uncovered. The analysis mechanism characteristics should be documented with the class.

Architectural and Use Case Analysis

Describing Analysis Mechanisms

- ◆ Collect all analysis mechanisms in a list
- ◆ Draw a map of the client classes to the analysis mechanisms

Analysis Class	Analysis Mechanism(s)

- ◆ Identify characteristics of the Analysis Mechanisms

78

Rational
the e-development company

In Architectural Analysis, the possible analysis mechanisms were identified and defined.

From that point on, as classes are defined, the required analysis mechanisms and analysis mechanism characteristics should be identified and documented. Not all classes will have mechanisms associated with them. Also, it is not uncommon for a client class to require the services of several mechanisms.

A mechanism has characteristics, and a client class uses a mechanism by qualifying these characteristics; this is to discriminate across a range of potential designs. These characteristics are part functionality, and part size and performance.

Architectural and Use Case Analysis

Example: Describing Analysis Mechanisms

♦ Analysis class to analysis mechanism map

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

79

Rational
the e-development company

As analysis classes are identified, it is important to identify the analysis mechanisms that apply to the identified classes.

The classes that must be persistent are mapped to the Persistency mechanism.

The classes that are maintained within the legacy Course Catalog system are mapped to the Legacy Interface mechanism.

The classes for which access must be controlled (i.e., control who is allowed to read and modify instances of the class) are mapped to the Security mechanism. Note: The Legacy Interface classes do not require additional security as they are read-only and are considered readable by all.

The classes that are seen to be distributed are mapped to the Distribution mechanism. The distribution identified during analysis is that which is specified/implied by the user in the initial requirements. Distribution will be discussed in detail in the Describe Distribution module. For now, just take it as an architectural given that all control classes are distributed for the OOAD course example and exercise.

Example: Describing Analysis Mechanisms (cont.)

- ◆ Analysis mechanism characteristics
- ◆ Persistency for Schedule class:
  - Granularity: 1 to 10 Kbytes per product
  - Volume: up to 2,000 schedules
  - Access frequency
    - Create: 500 per day
    - Read: 2,000 access per hour
    - Update: 1,000 per day
    - Delete: 50 per day
  - Etc.

Rational
the e-development company

The above is just an example of how the characteristics for an analysis mechanism would be documented for a class. For scoping reasons, the analysis mechanisms and their characteristics are not provided for all of the analysis classes.

Checkpoints: Analysis Classes

- ◆ Are the classes reasonable?
- ◆ Does the name of each class clearly reflect the role it plays?
- ◆ Does the class represent a single well-defined abstraction?
- ◆ Are all attributes and responsibilities functionally coupled?
- ◆ Does the class offer the required behavior?
- ◆ Are all specific requirements on the class addressed?

81

(Continued)

The above checkpoints for the analysis classes might be useful.

Note: All checkpoints should be evaluated with regards to the use cases being developed for the current iteration.

The class should represent a single well-defined abstraction. If not, consider splitting it.

The class should not define any attributes and responsibilities that are not functionally coupled to the other attributes or responsibilities defined by that class.

The classes should offer the behavior the use-case realizations and other classes require.

The class should address all specific requirements on the class from the requirement specification.

Remove any attributes and relationships if they are redundant or are not needed by the use-case realizations.

Checkpoints: Use-Case Realizations

- ◆ Have all the main and/or sub-flows been handled, including exceptional cases?
- ◆ Have all the required objects been found?
- ◆ Has all behavior been unambiguously distributed to the participating objects?
- ◆ Has behavior been distributed to the right objects?
- ◆ Where there are several interaction diagrams, are their relationships clear and consistent?

Rational
the e-development company

The above checkpoints for the Use-Case realizations might be useful.

Note: All checkpoints should be evaluated with regards to the use cases being developed for the current iteration.

The objects participating in a use-case realization should be able to perform all of the behavior of the use case.

If there are several interaction diagrams for the use-case realization, it is important that it is easy to understand which interaction diagrams relates to which flow of events. Make sure that it is clear from the Flow of Events description how the diagrams are related to each other.

Architectural and Use Case Analysis

Review: Use-Case Analysis

- What is the purpose of Use-Case Analysis?
- What is an analysis class? Name and describe the three analysis stereotypes.
- What is a use-case realization?
- Describe some considerations when allocating responsibilities to analysis classes.
- How many interaction diagrams should be produced during Use-Case Analysis?

83

Rational
the e-development company

Exercise: Use-Case Analysis, Part 2

- ◆ Given the following:
  - The Requirements artifacts, especially the Supplementary Specification
  - The possible analysis mechanisms
  - The flow of events interaction diagrams for a particular use case

(continued)

84

Rational
the e-development company

The goal is to complete the Use-Case Analysis of one the use cases that we started in Part 1. In Part 1, we identified the analysis classes that collaborate to perform the use case, we allocated the use-case responsibilities to those classes, and we diagrammed the collaborations. In Part 2, we will complete the Use-Case Analysis of the use cases by defining the relationships that must exist between the analysis classes to support the collaborations.

References to the givens:

•Supplementary Specification: Payroll Requirements Document, Supplementary Specification section.

•The analysis mechanisms we are concentrating on in this course include: persistency, distribution, security, legacy interface). See the Payroll Architecture Handbook, Architectural Mechanisms, Analysis Mechanisms section for more information on these analysis mechanisms.

•Use-case interaction diagrams: Payroll Exercise Solution, Use-Case Analysis, Exercise: Use-Case Analysis, Part 1 section.

Architectural and Use Case Analysis

Exercise: Use-Case Analysis, Part 2 (cont.)

- ◆ Identify the following for a particular use case:
    - Analysis class attributes and relationships
    - Analysis class analysis mechanisms

(continued)

85

Rational
the e-development company

The relationships to be identified are those needed to support the collaborations modeled in the use-case interaction diagrams developed in Part 1. Be sure to include multiplicity, navigability, and relationship and/or role names.

The attributes to be identified are the "obvious" properties of the identified classes. More attributes may be defined during later Class Design.

For each identified analysis class, determine if any of the analysis mechanisms apply. To make this decision, the supplementary specification may be needed.

Architectural and Use Case Analysis

Exercise: Use-Case Analysis, Part 2 (cont.)

- ◆ Produce the following diagrams:
  - ▪ VOPC class diagram, containing the analysis classes, their stereotypes, responsibilities, attributes, and relationships
  - ▪ Analysis class to analysis mechanism map

86

Rational
the e-development company

A View of Participating Classes (VOPC) is a class diagram that contains all the classes whose instances collaborate to perform a use case, as well as the relationships needed to support the collaborations. The class stereotype and and any attributes should be included on the VOPC.

For each relationship, be sure to include multiplicity, role or relationship names, and navigability (where known).

The analysis class to analysis mechanism map should now contain a list of all analysis classes and their associated analysis mechanism (if any).

References to sample diagrams within the course that are similar to what should be produced are:

•VOPC: Slides 64 and 65

•Analysis class to analysis mechanisms map: Slide 68