IBM Software Group | Rational Software France

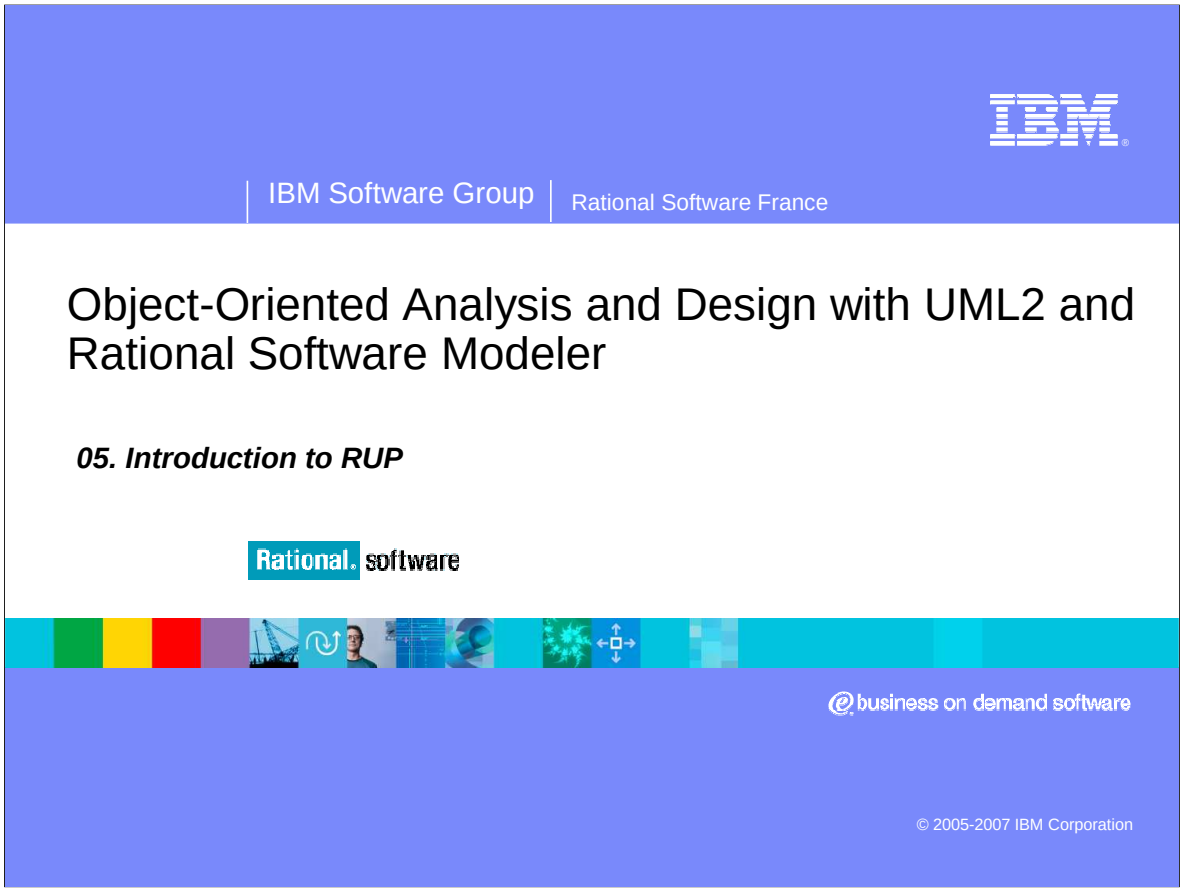
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

PART II – Object-Oriented Analysis

Rational. software

e business on demand software

Table of Contents



Success Rates of Software Development Projects

"Standish Group" CHAOS Chronicles	Year	Success Rate
First "Chaos" Report	1994	16 %
"Extreme Chaos"	2000	28 %
Last "Chaos" Report	2003	34 %

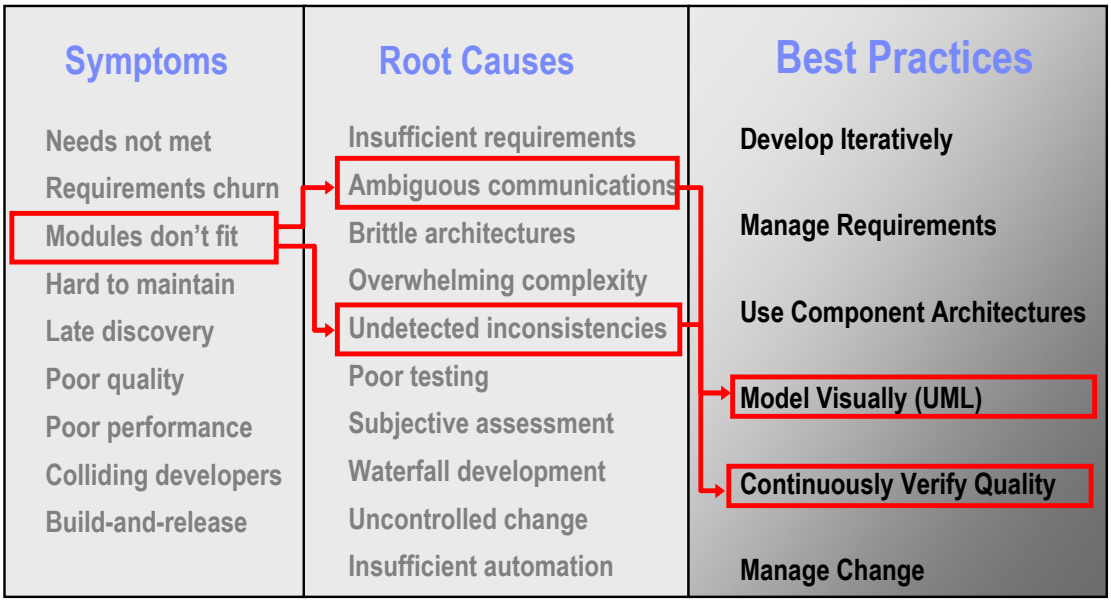
- Success = project delivered on time, within budget and meeting the needs of the users

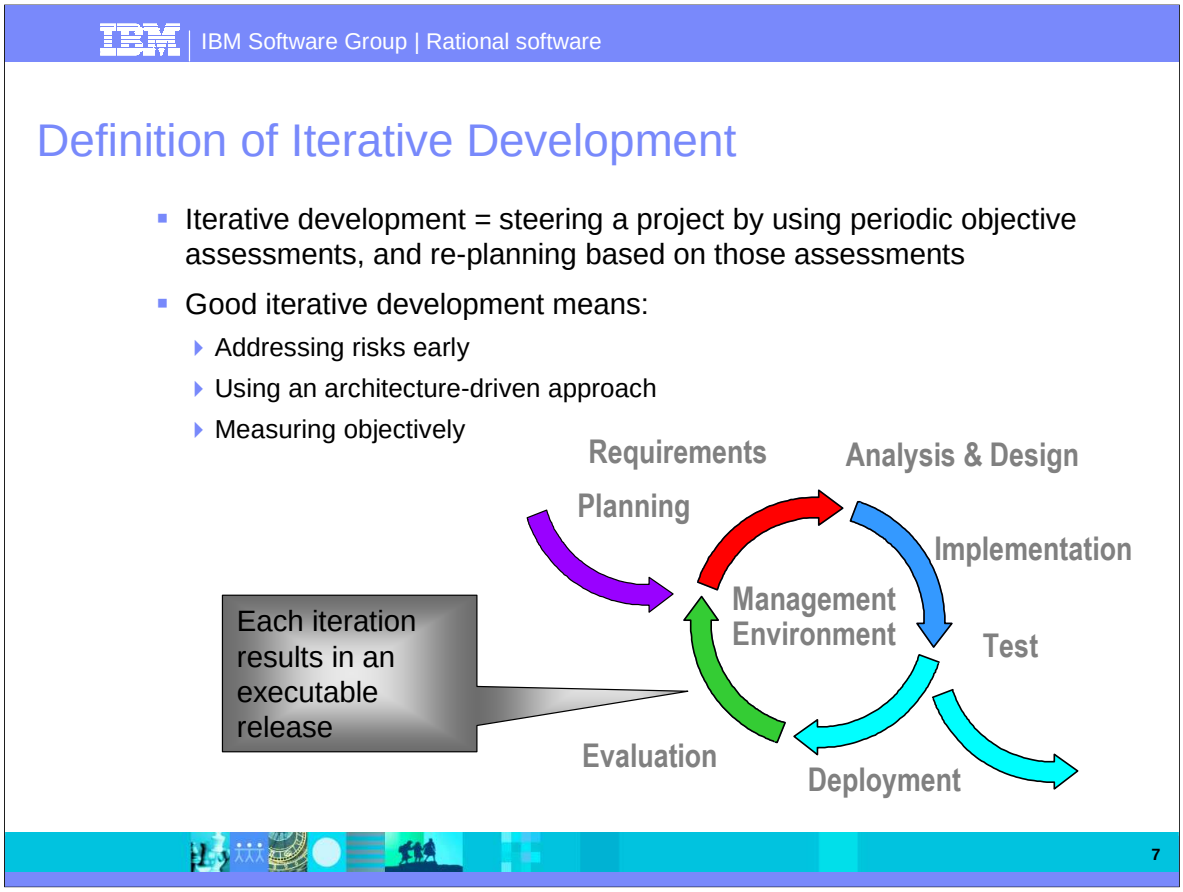
"We know why projects fail, we know how to prevent their failure -- so why do they still fail?" - Martin Cobb

Symptoms of Software Development Problems

- ✓ User or business needs not met
- ✓ Requirements not addressed
- ✓ Modules not integrating
- ✓ Difficulties with maintenance
- ✓ Late discovery of flaws
- ✓ Poor quality of end-user experience
- ✓ Poor performance under load
- ✓ No coordinated team effort
- ✓ Build-and-release issues

Trace Symptoms to Root Causes





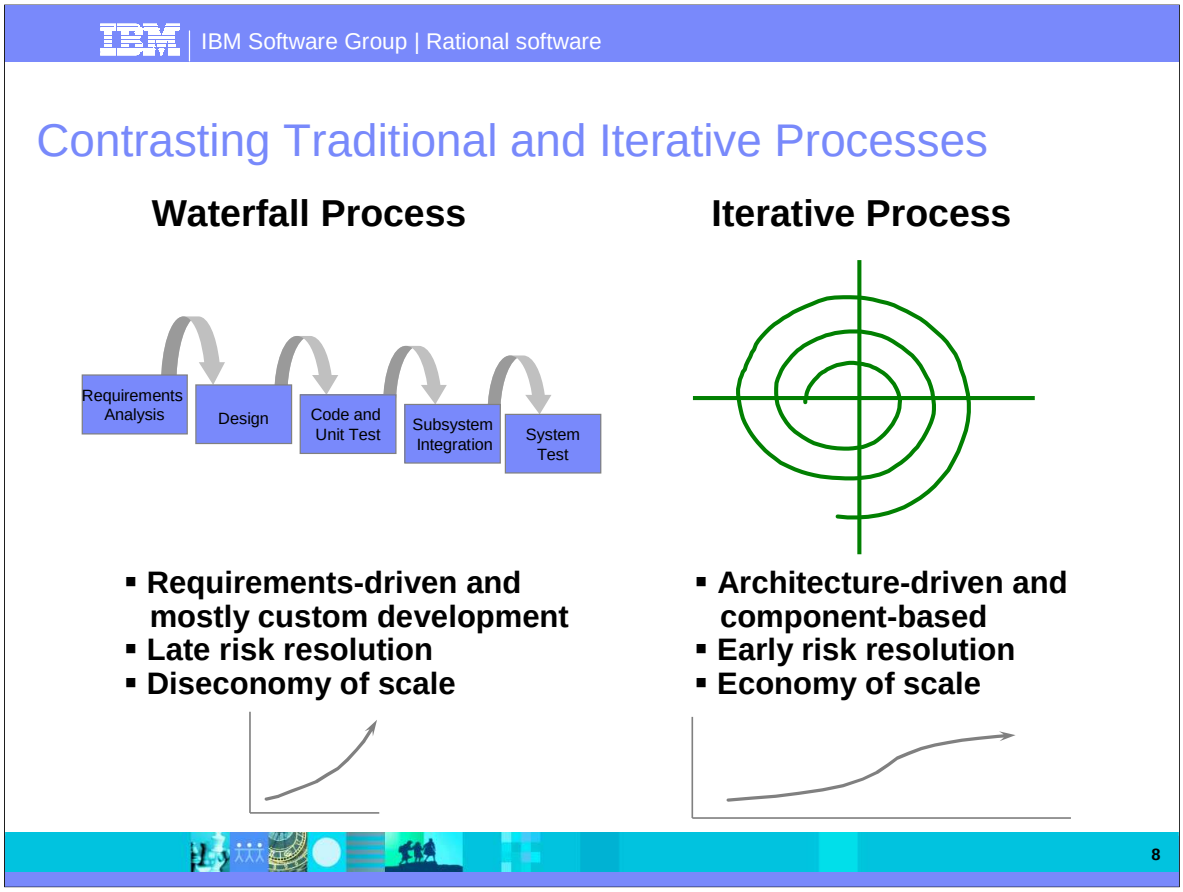
Developing iteratively is a technique that is used to deliver the functionality of a system in a successive series of releases of increasing completeness. Each release is developed in a specific, fixed time period called an **iteration**.

Each iteration is focused on defining, analyzing, designing, building, and testing a set of requirements.

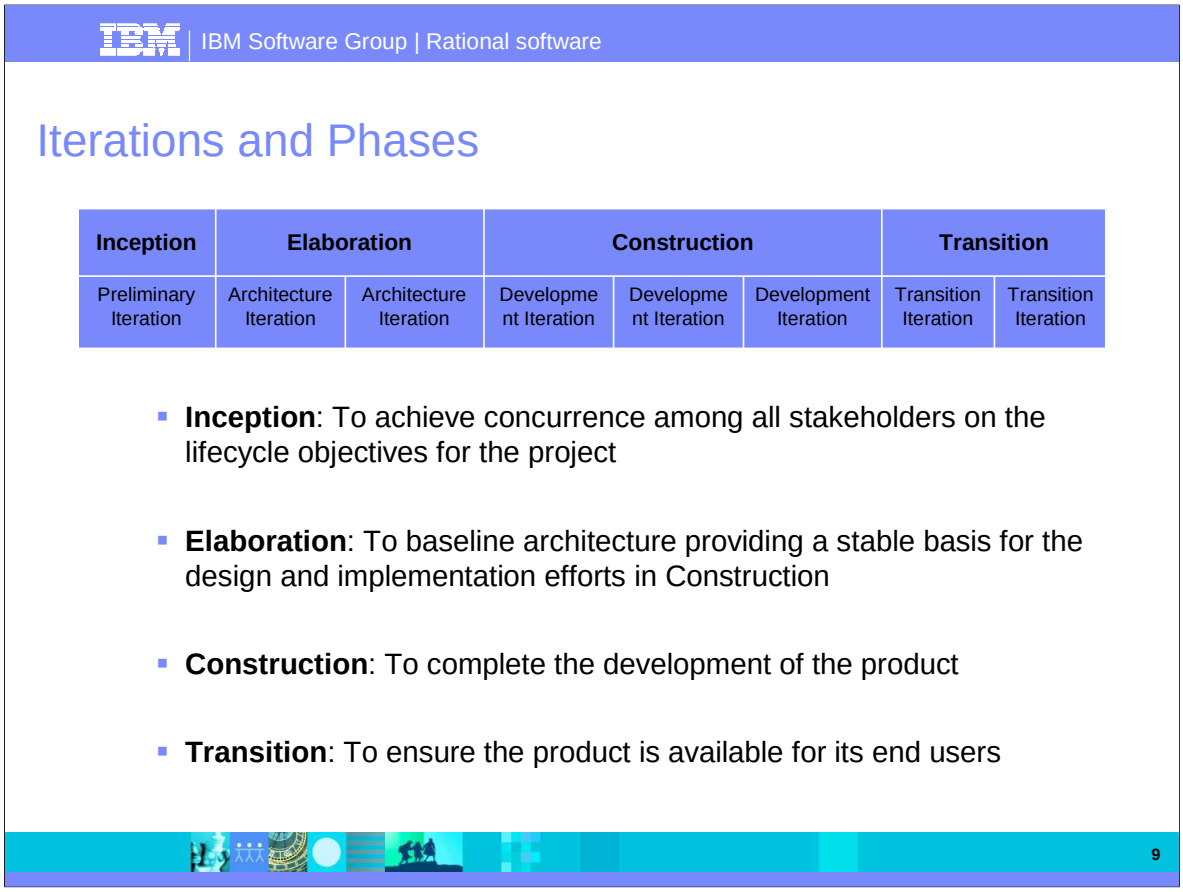
The earliest iterations address the greatest risks. Each iteration includes integration and testing and produces an executable release. Iterations help:

- •Resolve major risks before making large investments.
- •Enable early user feedback.
- •Make testing and integration continuous.
- •Define a project's short-term objective milestone.
- •Make deployment of partial implementations possible.

Instead of developing the whole system in lock step, an increment (for example, a subset of system functionality) is selected and developed, then another increment, and so on. The selection of the first increment to be developed is based on risk, with the highest priority risks first. To address the selected risk(s), choose a subset of use cases. Develop the *minimal* set of use cases that will allow objective verification (that is, through a set of executable tests) of the risks that you have chosen. Then, select the next increment to address the next-highest risk, and so on.



In the waterfall process, there is usually a diseconomy of scale. This means that the larger the size of the software, the higher it costs per unit to build. In an iterative process, there is an improvement in the economy of scale. That is, the software becomes cheaper to build per unit as you build more.



The overriding goal of the **Inception** phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. The Inception phase is of significance primarily for new development efforts, in which there are significant business and requirements risks which must be identified before the project can proceed.

The goal of the **Elaboration** phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the construction phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risk. The stability of the architecture may be evaluated through one or more architectural prototypes. Other equally important risks are also the business risks.

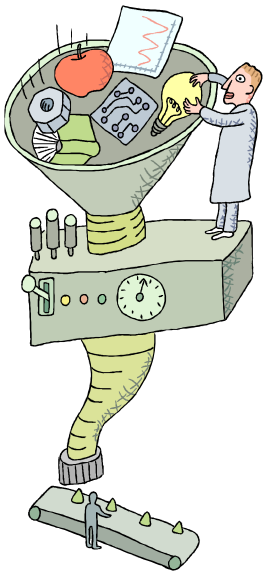
The goal of the **Construction** phase is implementing the remaining requirements and completing the development of the system based upon the baselined architecture. The Construction phase is in some sense a manufacturing process, where emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality.

The focus of the **Transition** Phase is to ensure that software is available for its end users. The Transition Phase can span several iterations, and includes testing the product in preparation for release, and making minor adjustments based on user feedback.

IBM Software Group | Rational software

Managing Requirements

- Ensures that you:
  - Solve the right problem
  - Build the right system
- By taking a systematic approach to
  - Eliciting
  - Organizing
  - Documenting
  - Managing
- The changing requirements of a software application



10

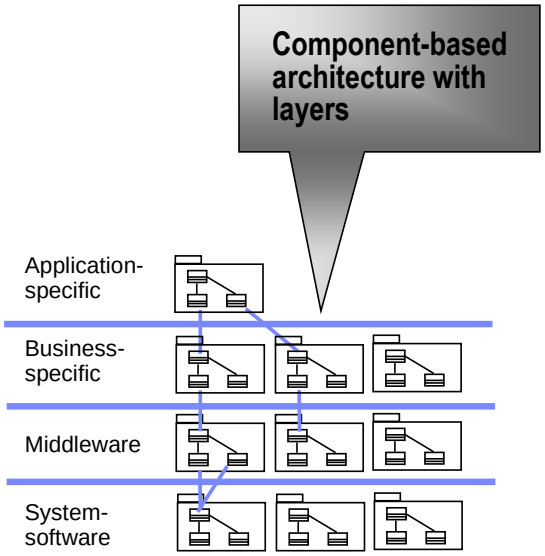
The 1994 report from the Standish Group confirms that a distinct minority of software development projects is completed on time and on budget. In their report, the success rate was only 16.2%, while challenged projects (operational, but late and over budget) accounted for 52.7%. Impaired (canceled) projects accounted for 31.1%. These failures are attributed to incorrect requirements definition from the start of the project and poor requirements management throughout the development lifecycle. (Source: *Chaos Report*, http://www.standishgroup.com)

Aspects of requirements management:

- Analyze the problem
- Understand user needs
- Define the system
- Manage scope
- Refine the system definition
- Manage changing requirements

Use Component-Based Architectures

- Basis for reuse
  - Component reuse
  - Architecture reuse
- Basis for project management
  - Planning
  - Staffing
  - Delivery
- Intellectual control
  - Manage complexity
  - Maintain integrity



Architecture is a part of Design. It is about making decisions on how the system will be built. But it is not all of the design. It stops at the major abstractions, or, in other words, the elements that have some pervasive and long-lasting effect on system performance and ability to evolve.

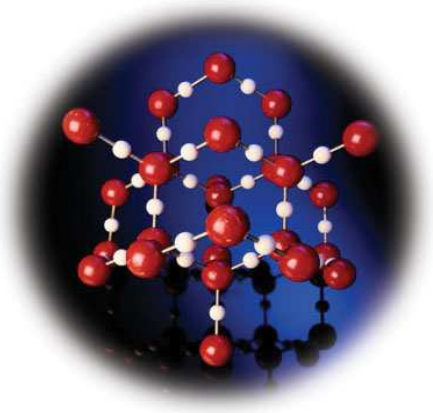
A software system's architecture is perhaps the most important aspect that can be used to control the iterative and incremental development of a system throughout its lifecycle.

The most important property of an architecture is resilience –flexibility in the face of change. To achieve it, architects must anticipate evolution in both the problem domain and the implementation technologies to produce a design that can gracefully accommodate such changes. Key techniques are abstraction, encapsulation, and object-oriented Analysis and Design. The result is that applications are fundamentally more maintainable and extensible.

Software architecture is the development product that gives the highest return on investment with respect to quality, schedule, and cost, according to the authors of *Software Architecture in Practice* (Len Bass, Paul Clements, and Rick Kazman [1998] Addison-Wesley). The Software Engineering Institute (SEI) has an effort underway called the Architecture Tradeoff Analysis (ATA) Initiative that focuses on software architecture, a discipline much misunderstood in the software industry. The SEI has been evaluating software architectures for some time and would like to see architecture evaluation in wider use. As a result of performing architecture evaluations, AT&T reported a 10% productivity increase (from news@sei, Vol. 1, No. 2).

Model Visually (UML)

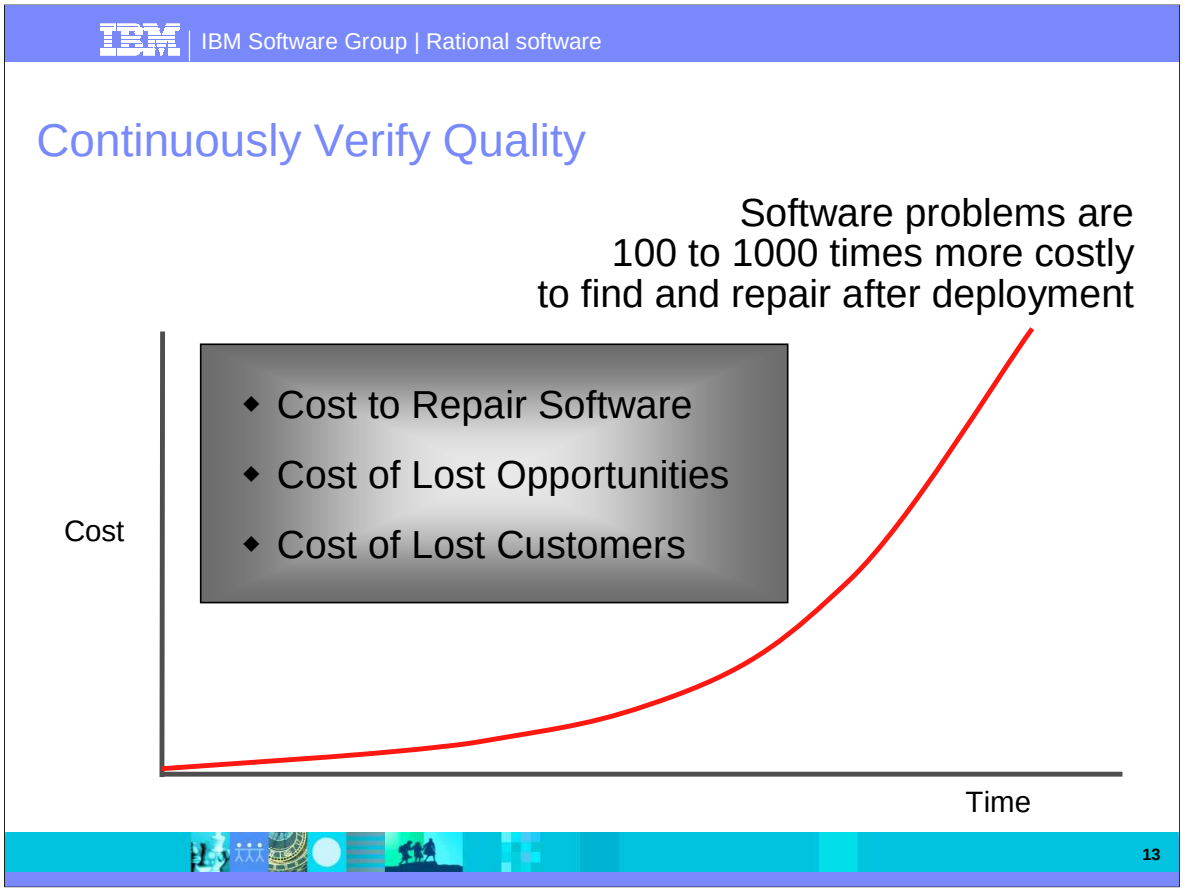
- Captures structure and behavior
- Shows how system elements fit together
- Keeps design and implementation consistent
- Hides or exposes details as appropriate
- Promotes unambiguous communication
  - The UML provides one language for all practitioners



A **model** is a simplification of reality that provides a complete description of a system from a particular perspective. We build models so that we can better understand the system we are building. We build models of complex systems because we cannot comprehend any such system in its entirety.

Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of system architecture. Using a standard modeling language such as the UML (the Unified Modeling Language), different members of the development team can communicate their decisions unambiguously to one another.

Using visual modeling tools facilitates the management of these models, letting you hide or expose details as necessary. Visual modeling also helps you maintain consistency among system artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team's ability to manage software complexity.



Quality, as used within the RUP, is defined as “The characteristic of having demonstrated the achievement of producing a product which meets or exceeds agreed-upon requirements, as measured by agreed-upon measures and criteria, and is produced by an agreed-upon process." Given this definition, achieving quality is not simply “meeting requirements" or producing a product that meets user needs and expectations. Quality also includes identifying the measures and criteria (to demonstrate the achievement of quality) and the implementation of a process to ensure that the resulting product has achieved the desired degree of quality (and can be repeated and managed).

This principle is driven by a fundamental and well-known property of software development: It is a lot less expensive to correct defects during development than to correct them after deployment.

Tests for key scenarios ensure that all requirements are properly implemented.

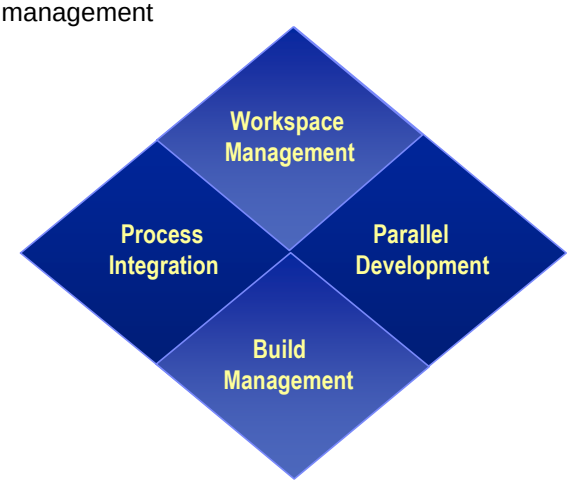
- Poor application performance hurts as much as poor reliability.
- Verify software reliability by checking for memory leaks and bottlenecks.
- Test every iteration by automating testing.

Inception, Elaboration, Construction, and Transition are all RUP terms that will be discussed shortly.

Manage Change

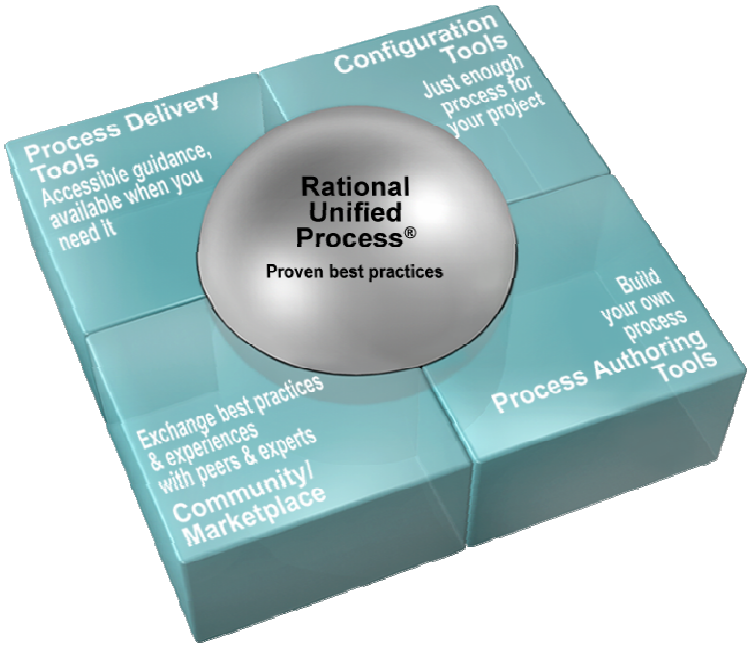
- To avoid confusion, have:
  - Secure workspaces for each developer
  - Automated integration/build management
  - Parallel development

Configuration Management is more than just check-in and check-out



Rational Unified Process Implements Best Practices

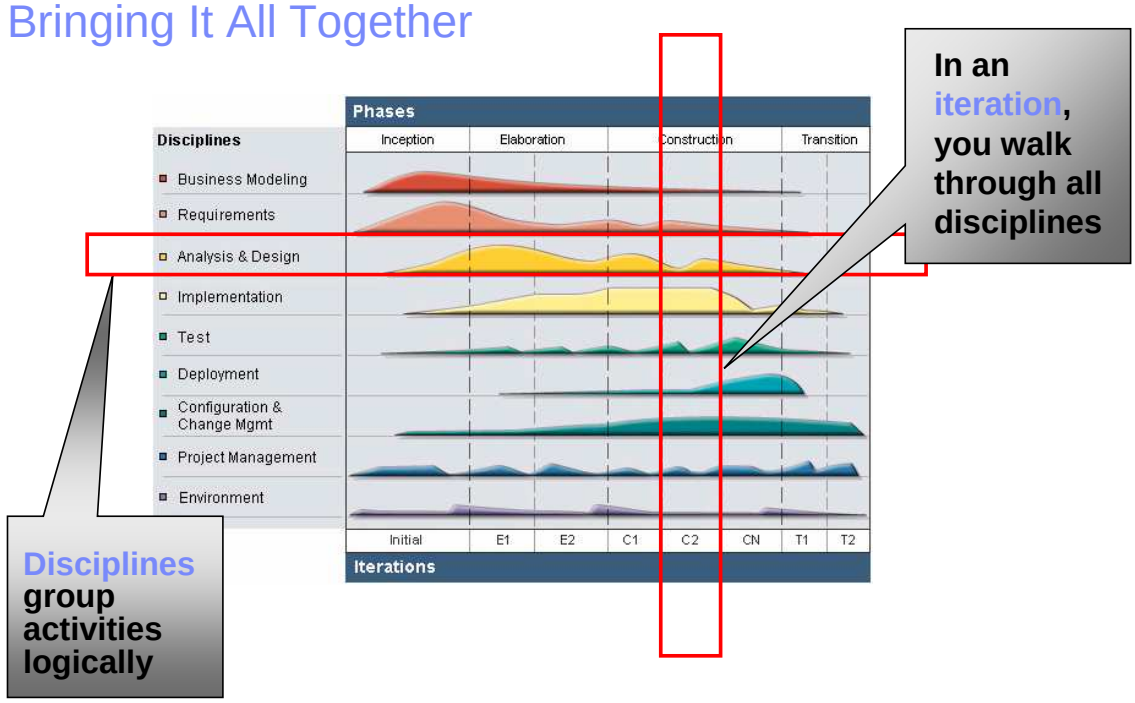
- Iterative approach
- Guidance for activities and artifacts
- Process focus on architecture
- Use cases that drive design and implementation
- Models that abstract the system

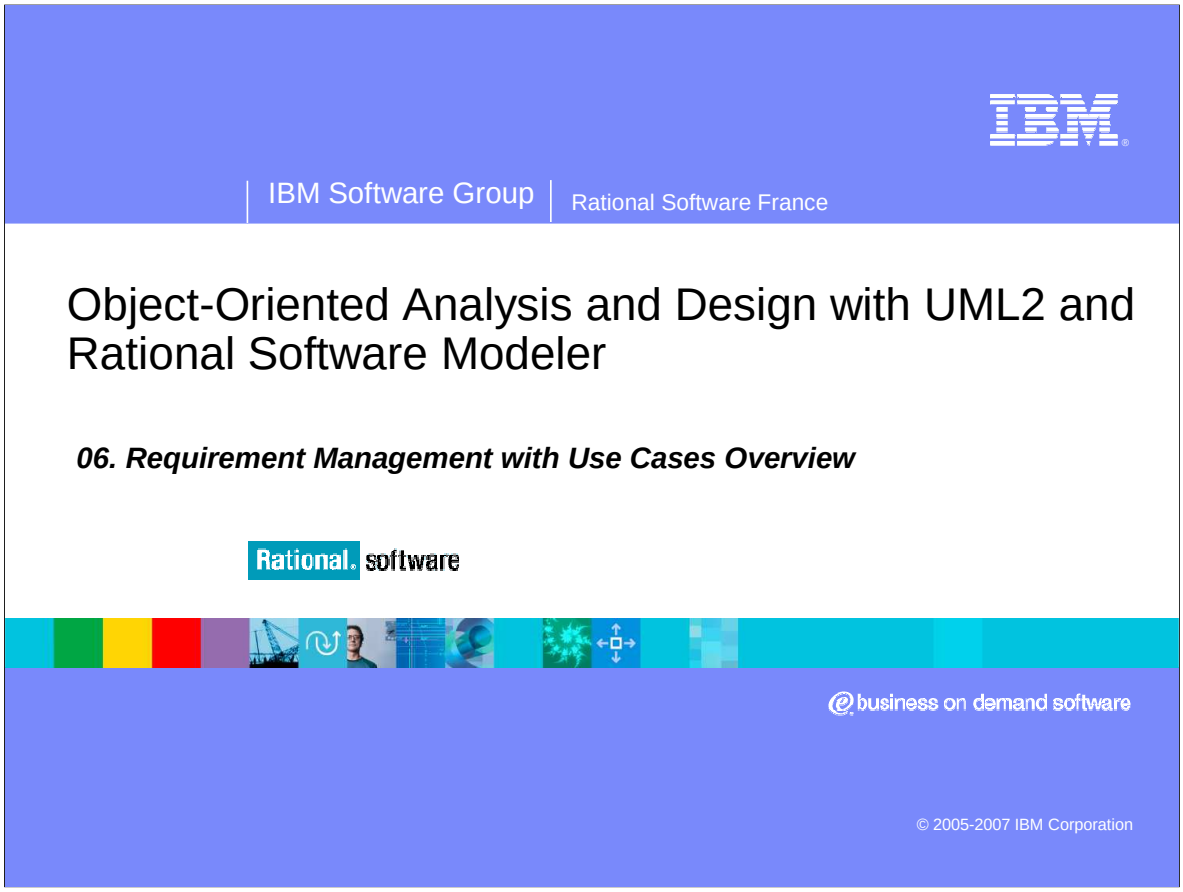


The Rational Unified Process (RUP) is a generic business process for object-oriented software engineering. It describes a family of related software-engineering processes sharing a common structure and a common process architecture. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget. The RUP captures the Best Practices in modern software development in a form that can be adapted for a wide range of projects and organizations.

The UML provides a standard for the artifacts of development (semantic models, syntactic notation, and diagrams): the things that must be controlled and exchanged. But the UML is not a standard for the development *process*. Despite all of the value that a common modeling language brings, you cannot achieve successful development of today's complex systems solely by the use of the UML. Successful development also requires employing an equally robust development process, which is where the RUP comes in.

Bringing It All Together



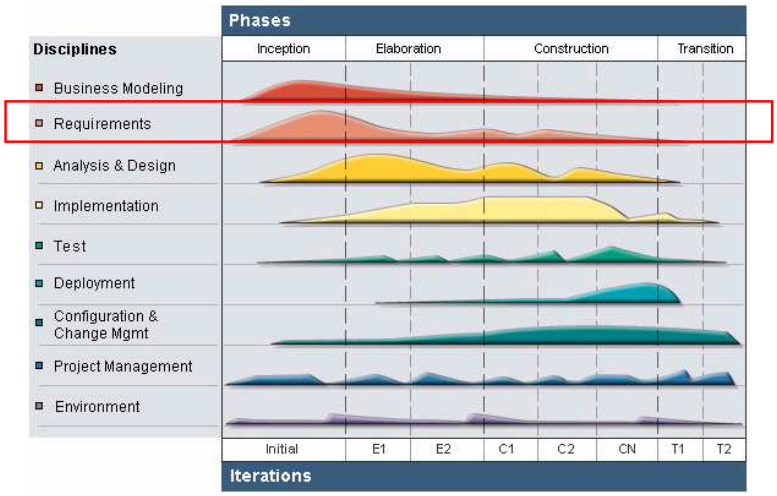


Where Are We?

- Introduction to Use-Case Modeling
  - Find Actors and Use Cases
  - Use-Case Specifications

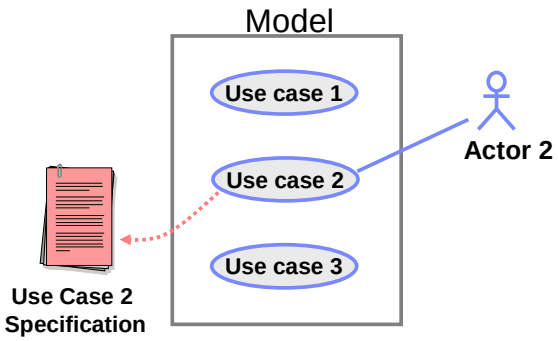
Requirements in Context

- The purpose of Requirements is to:
  - Elicit stakeholder requests and transform them into a set of requirements work products that scope the system to be built and provide detailed requirements for what the system must do
- RUP recommends a use-case driven approach



What Is Use-Case Modeling?

- Links stakeholder needs to software requirements
- Defines clear boundaries of a system
- Captures and communicates the desired behavior of the system
- Identifies who or what interacts with the system
- Validates/verifies requirements
- Is a planning instrument

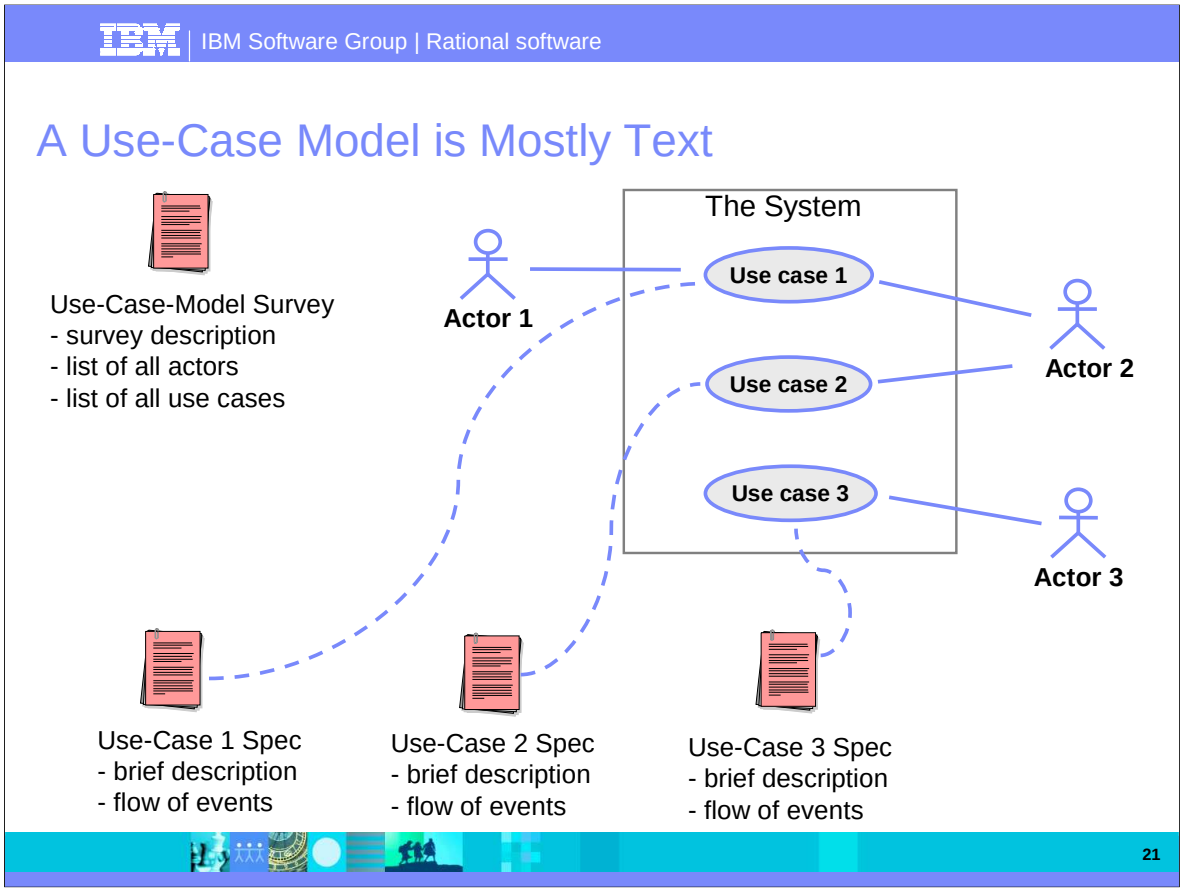


A use-case model describes a system's functional requirements in terms of use cases.

It is a model of the system's intended functionality (use cases) and its environment (actors). Use cases enable you to relate what you need from a system to how the system delivers on those needs.

Think of a use-case model as a menu, much like the menu you'd find in a restaurant. By looking at the menu, you know what's available to you, the individual dishes as well as their prices. You also know what kind of cuisine the restaurant serves: Italian, Mexican, Chinese, and so on. By looking at the menu, you get an overall impression of the dining experience that awaits you in that restaurant. The menu, in effect, "models" the restaurant's behavior.

Because it is a very powerful planning instrument, the use-case model is generally used in all phases of the development cycle by all team members.



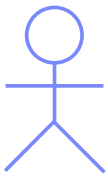
The use-case model consists of both diagrams and text. The diagrams give a visual overview of the system. The text gives descriptions of the actors and the use cases.

Use cases involve writing text. Drawing the pictures is only a small part of the effort. Typically, more than 80 percent of all effort during requirements capture is to write the textual description of what happens in each use case, the non-functional requirements, and rules. The description of what happens is called the flow of events.

Activity diagrams are another useful tool you can use to describe a use case. It is quite common to use an activity diagram to describe complex flows of events. When using activity diagrams, it is advisable to use partitions to represent each actor and the system. Without partitions, the activities float in a “semantic emptiness” and quickly become meaningless.

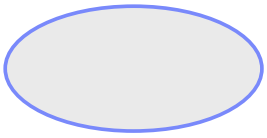
Major Concepts in Use-Case Modeling

- An actor represents anything that interacts with the system



Actor

- A use case is a sequence of actions a system performs that yields an observable result of value to a particular actor



Use Case

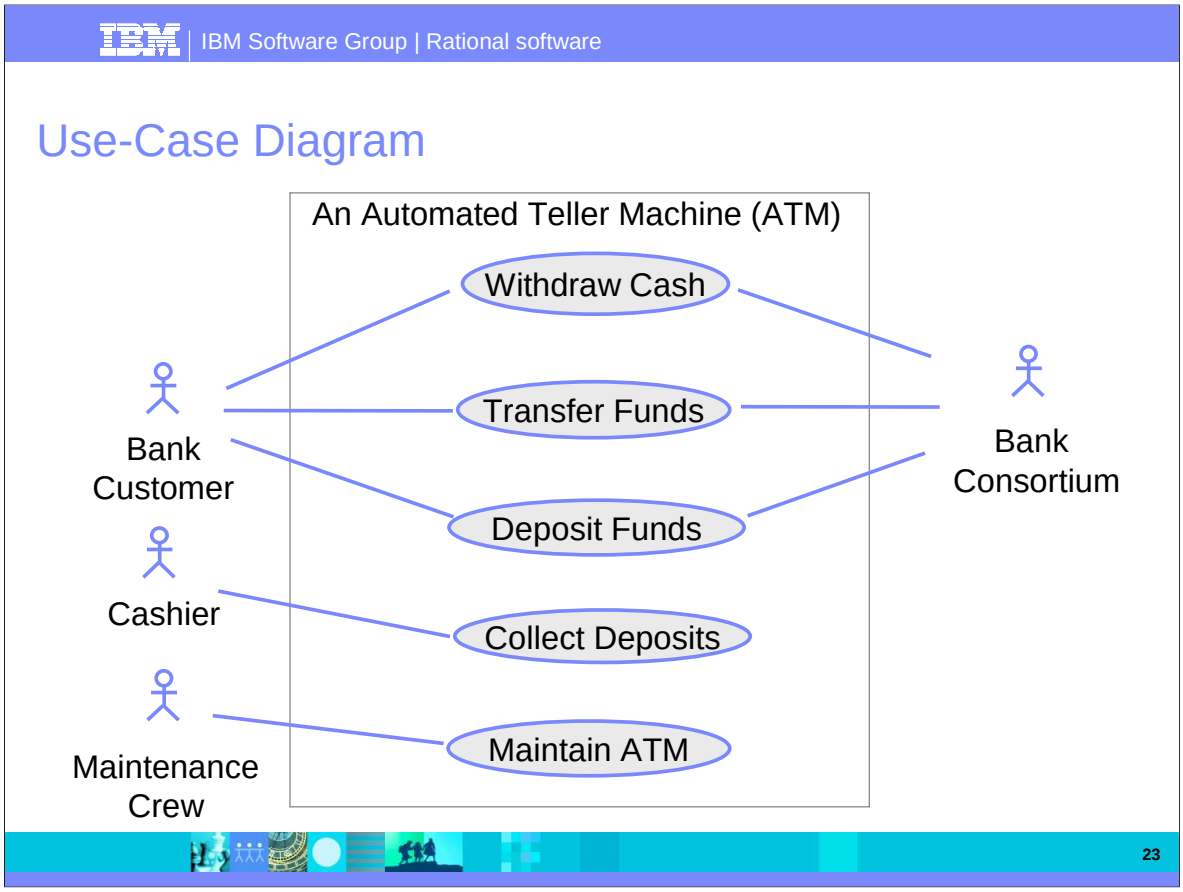
An **actor** represents a role that interacts with the system.

A **use case** describes a sequence of interactions between actors and the system that occur when an actor uses the system to achieve a certain business goal.

A use case describes:

- The system, its environment, and the relationship between them.
- How things outside the system interact with the system.
- The desired behavior for the system.

Use cases are containers for contextually related requirements of the system under development. They are containers because they group all requirements related to achieving a particular goal into a single story of how that is achieved.



A use-case model shows what the system is supposed to do (use cases), the system's surroundings (actors), and the relationship between actors and use cases.

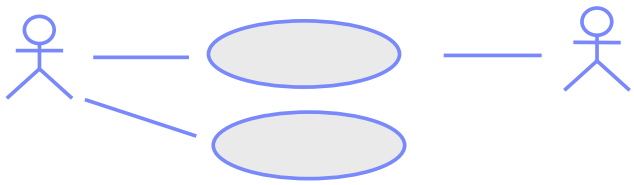
The use-case diagram is a graphical description of the use-case model. Can you tell at a glance what users can do with an ATM?

What do you think the priorities for the use cases above would be? The diagram already helps to determine these priorities; without the first one we do not have an ATM, so we better get it right, (e.g. during elaboration); other use cases like deposit funds are nice to have, but would also have quite an architectural impact.

It is also useful to distinguish between primary use case and secondary use cases: Primary use cases support the customers/actors and business. Secondary use cases we get, because we decided on a particular technical solution and thus have to specify "extra" functionality to run this technical solution; thus, Maintain ATM, Print logs, Start, Stop, Backup the System, etc are required. These could be also be organized in a separate diagram.

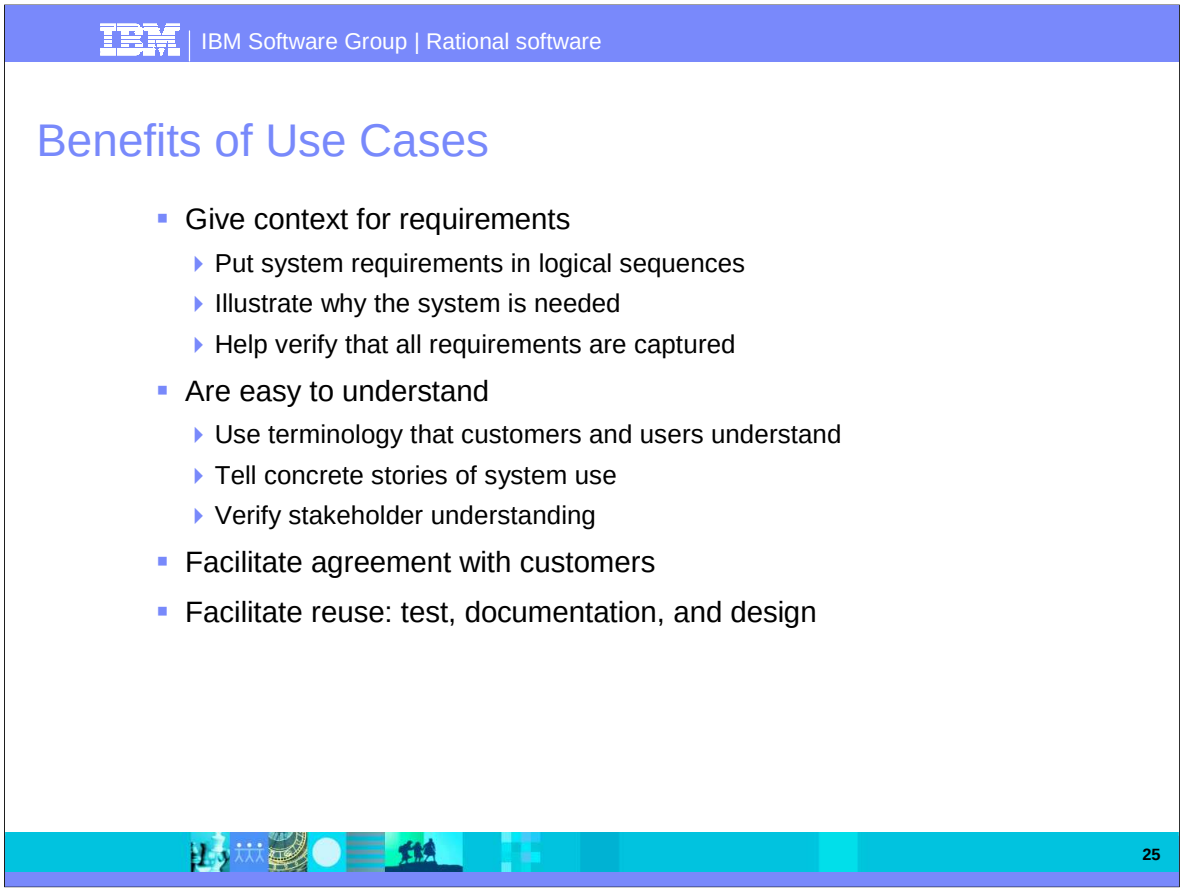
Use Cases Contain Software Requirements

- Each use case:
  - Describes actions the system takes to deliver something of value to an actor
  - Shows the system functionality an actor uses
  - Models a dialog between the system and actors
  - Is a complete and meaningful flow of events from the perspective of a particular actor



Use cases contain the detailed functional software requirements. Every time a use case says, “The system …,” is a detailed requirement of what the system must do.

Also, remember the definition: “A sequence of actions *performed by a system* that yields a measurable result of value for a particular actor.”



Use cases are a way to organize requirements **from a user perspective**. All the requirements for a user to accomplish a particular task are gathered together in a single use case. The use-case model is the collection of all the individual use cases.

Advantages of use-case modeling include:

- Use cases show why the system is needed. Use cases show what goals the users can accomplish by using the system.
- System requirements are placed in context. Each requirement is part of a logical sequence of actions to accomplish a goal of a user.
- Use cases are easy to understand. The model expresses requirements from a user perspective and in the user's own vocabulary. The model shows how users think of the system and what the system should do. Traditional forms of requirements capture need some translation to make them useable to different stakeholder types. When you translate something, information is often lost or misinterpreted. Use cases require no such translation and therefore provide a more consistent and reliable form of requirement capture.
- The model is a means to communicate requirements between customers and developers to make sure that the system we build is what the customer wants.

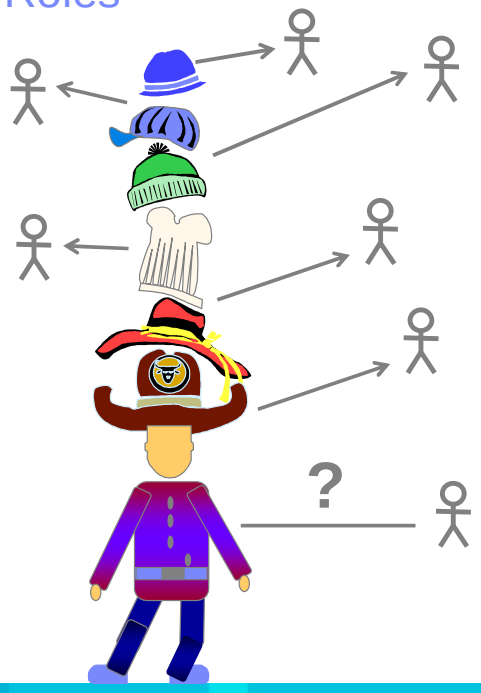
Use-case modeling is the best tool (so far) to capture requirements and put them in context.

Where Are We?

- Introduction to Use-Case Modeling
- ⇨ Find Actors and Use Cases
- Use-Case Specifications

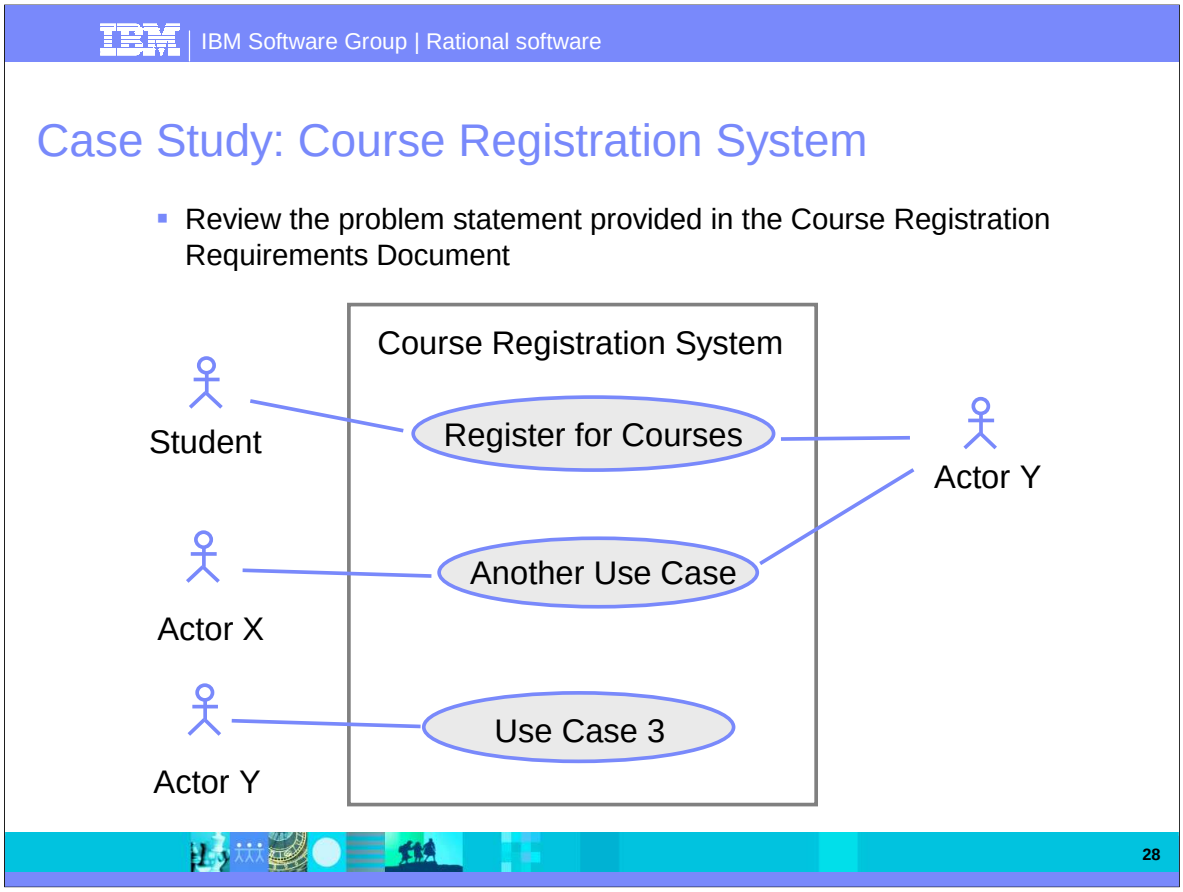
Define Actors: Focus on the Roles

- An actor represents a role that a human, hardware device, or another system can play in relation to the system
- Actor names should clearly denote the actor's role



The difference between an actor and an individual system user is that an actor represents a particular class of user rather than an actual user. Several users can play the same role, meaning they can be the same actor. In that case, each user constitutes an instance of the actor.

However, in some situations, only one person plays the role modeled by an actor. For example, there may be only one individual playing the role of system administrator for a rather small system.

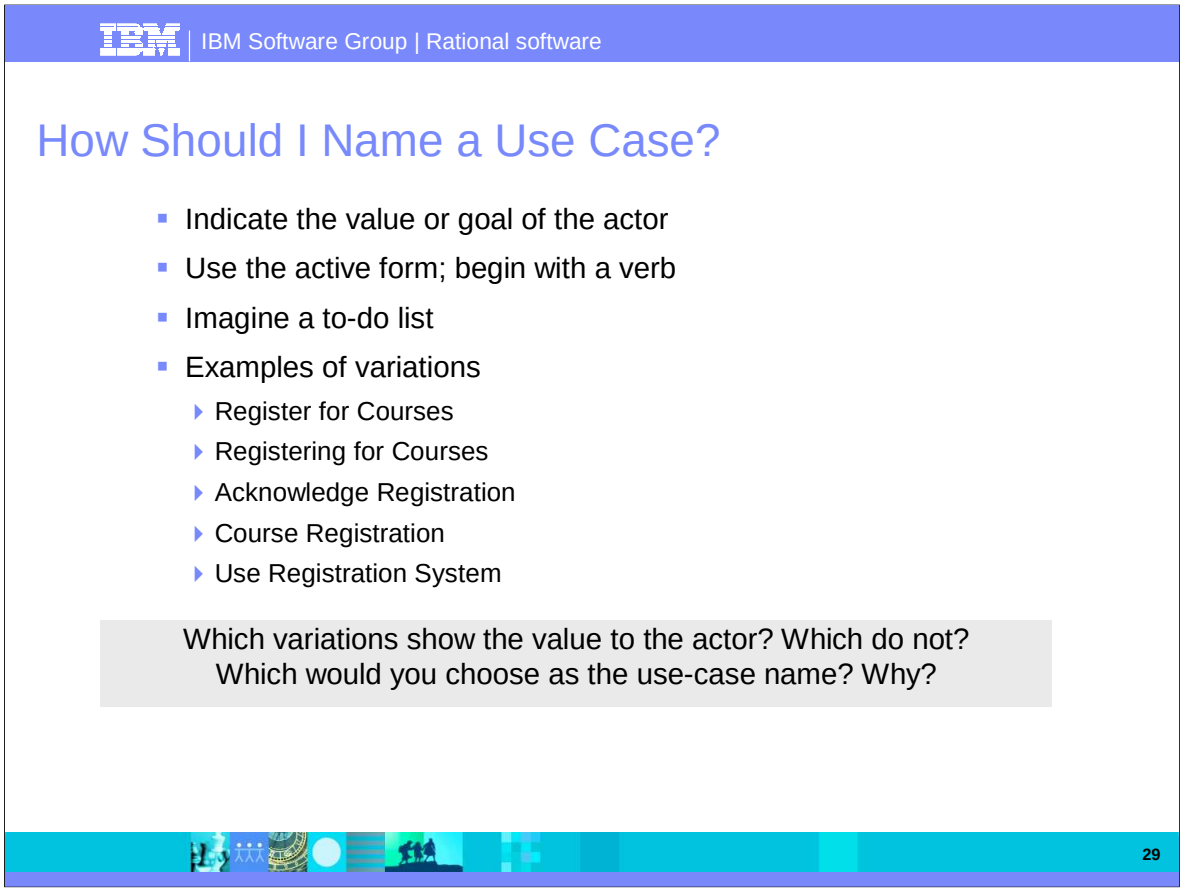


The use-case diagram shown here is a graphic description of the use-case model for an online Course Registration System. It shows two of the actors (Student and Course Catalog System) and one use case (Register for Courses) that they participate in.

The diagram is incomplete. You use the online course registration system as a case study in this module. You develop the use-case model for this example as we go through the module.

Take some time to look at the use-case model for an online Course Registration System in the Student Workbook. This example gives you an idea of what a use-case model looks like before you begin to develop one.

The use-case model for an online Course Registration System in the Student Workbook is incomplete. It contains only enough artifacts for the purpose of this module: Introduction to Use-Case Modeling. A larger, much more fully developed case study is presented later in the course.

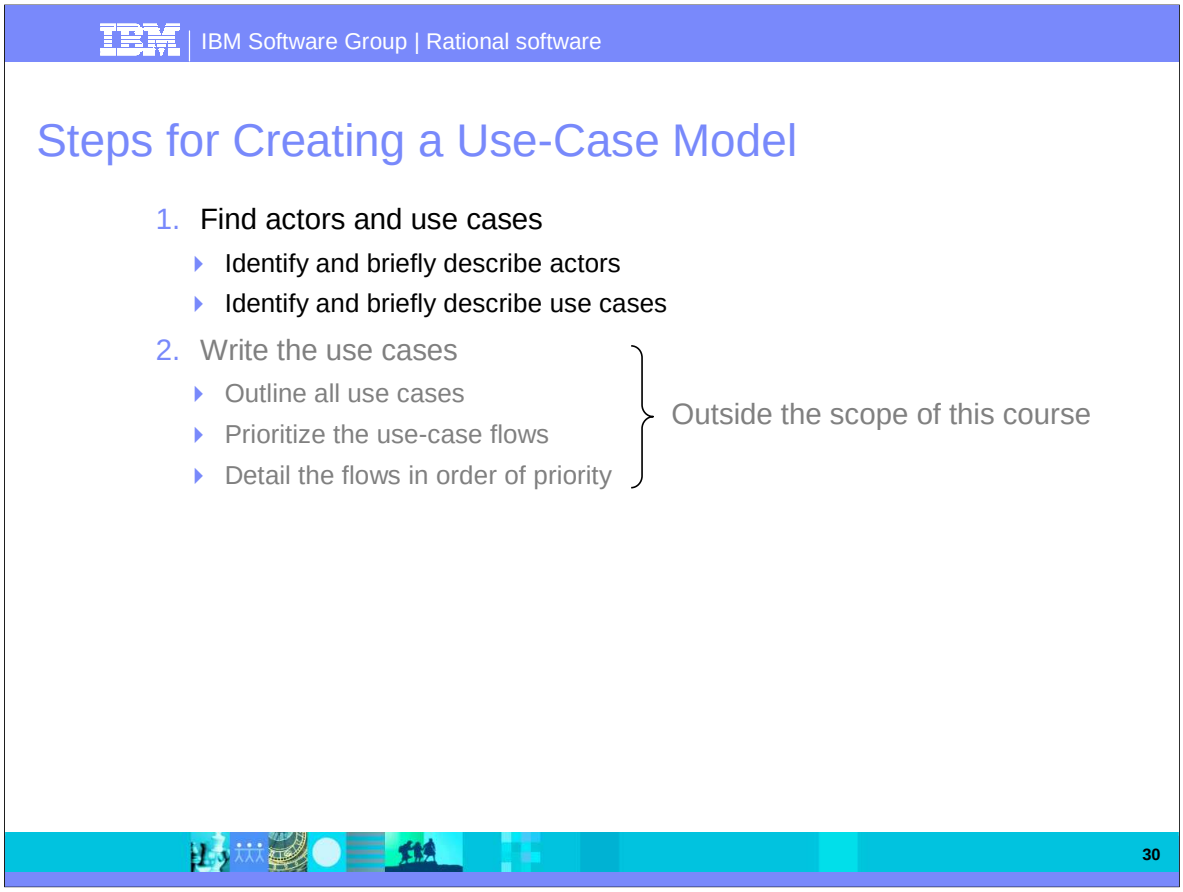


Each use case should have a name that indicates what is achieved by its interactions with the actor(s).

A good rule of thumb (but not dictated by any standard) is to attach the actor's name (of the primary actor) at the beginning of the use-case name to see if it makes a meaningful sentence. For example, does it make sense to say "The student registers for a course?" Does it make sense to say, "The student takes a course?"

Another approach is to ask, "Why does the actor want to use the system?"

Be focused on identifying the goal that is trying to be attained by using the system.



Creating a use-case model involves putting the pieces you have learned together. First, the actors and the use cases are found by using the requirements of customers and potential users as vital information. As they are discovered, the use cases and the actors are identified and illustrated in a use-case diagram. Next, the steps in a use case are outlined to get a sketch of the flow.

The actor's name must clearly denote the actor's role. Make sure there is little risk at a future stage of confusing one actor's name with another.

Define each actor by writing a brief description that includes the actor's area of responsibility and what the actor needs the system for. Because actors represent things outside the system, you need not describe them in detail.

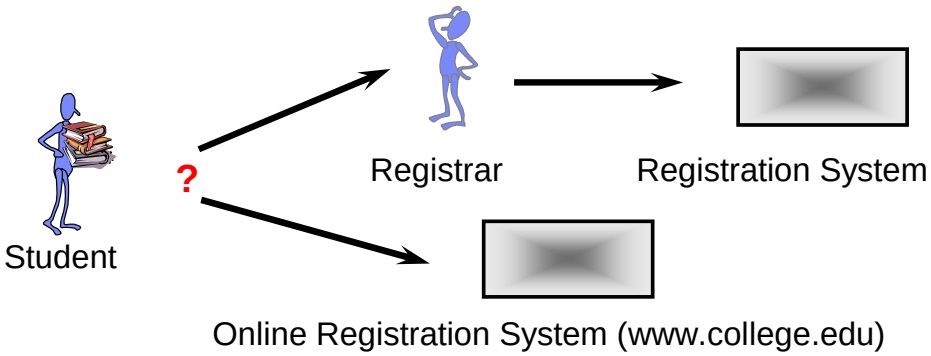
Each use case should have a name that indicates what is achieved by its interactions with the actor(s). The name may have to be several words to be understood. No two use cases can have the same name.

Define each use case by writing a brief description of it. As you write the description, refer to the glossary and, if you need to, define new terms.

When the actors and use cases have been found, each use case is described in detail. These descriptions show how the system interacts with the actors and what the system does in each individual case. In an iterative development environment, you select a set of use case flows to be detailed in each iteration. These are prioritized in such a way that technical risk is mitigated early and the customer gets important functionality before less important functionality.

Find Actors

- Who is pressing the keys (interacting with the system)?



- Who/what gets information from this system?
- Who/what provides information to the system?
- Who/what supports and maintains the system?

To identify the actors for a system, the simplest question to ask is: “Who is doing the actual interaction?” The actor is the one who is interacting with the system.

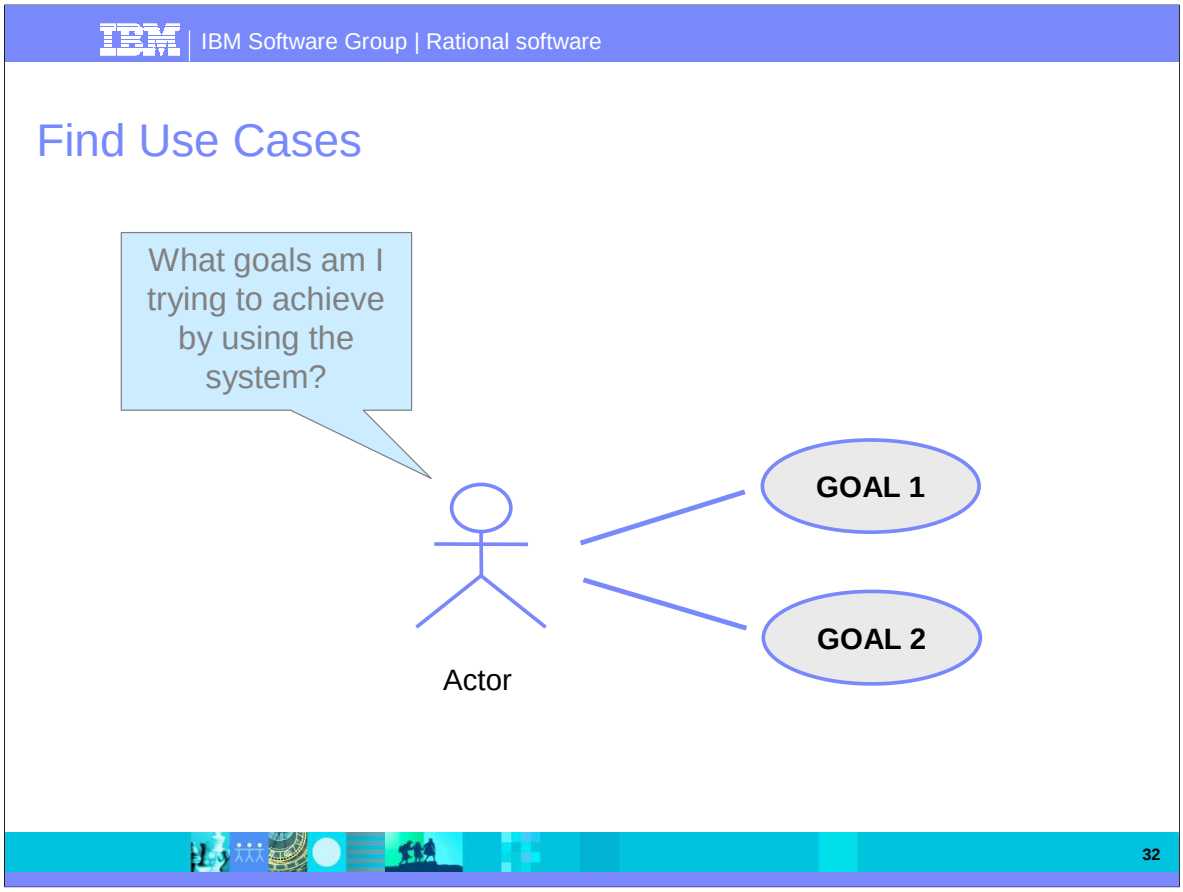
What if a person is using a speech recognition system? Then the actor is the one talking with the system.

An actor is any person or any thing that is outside the system and that exchanges data with the system. An actor can either give information, receive information, or give and receive information.

Actor is a role, not a particular person or thing. The name of the actor should represent, as clearly as possible, the actor’s role.

Make sure that there is little risk of confusing one actor’s name with another at a future stage of the work. Also, try to avoid an actor called “User,” rather try to figure out the role of that particular user.

In most instances, some person or some other system does something to trigger the start of the use case. If a use case in your system is initiated at a certain time, for example, at the end of the day or the end of the month, this can be modeled with a special actor, such as the “scheduler” or “time.” “Scheduler,” as opposed to “time,” is a useful name for such an actor because scheduler may be human or non-human. “Time” leaves an element of design in your use case model.



Identifying the use cases is the next step in developing your use-case model, once the actors have been identified. Use cases describe what an actor wants a system to do that provides some value to the actor. Use this process to identify the use cases for each actor.

The best way to find use cases is to consider what each actor requires of the system. Go through all the actors and identify the particular needs of each actor.

When you have made your first list of use cases, verify that all required functionality has been treated. Do not forget special use cases for system startup, termination, or maintenance. Also, do not forget to include use cases for automatically scheduled events. For example, a time-initiated job may run the payroll at midnight on the last day of each month. Use cases that concern automatically scheduled events are usually initiated by a special actor: the scheduler.

Try to keep all use cases at approximately the same level of importance. The notion of use-case levels as popularized by some authors is not recommended. It can lead to a functionally decomposed system.

Is the use case too complex? If it is, you may want to split it (a use-case report significantly longer than 10 pages may be a candidate).

Give each use case a name that indicates what an instance of the use case does. The name may have to consist of several words to be clearly understood.

Group Exercise

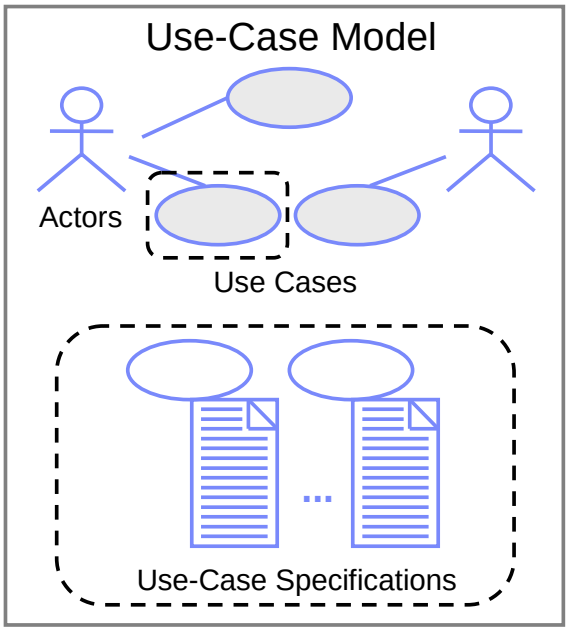
- Identify the actors who interact with the Course Registration System
- Identify use cases for the system
- Sketch a use-case diagram

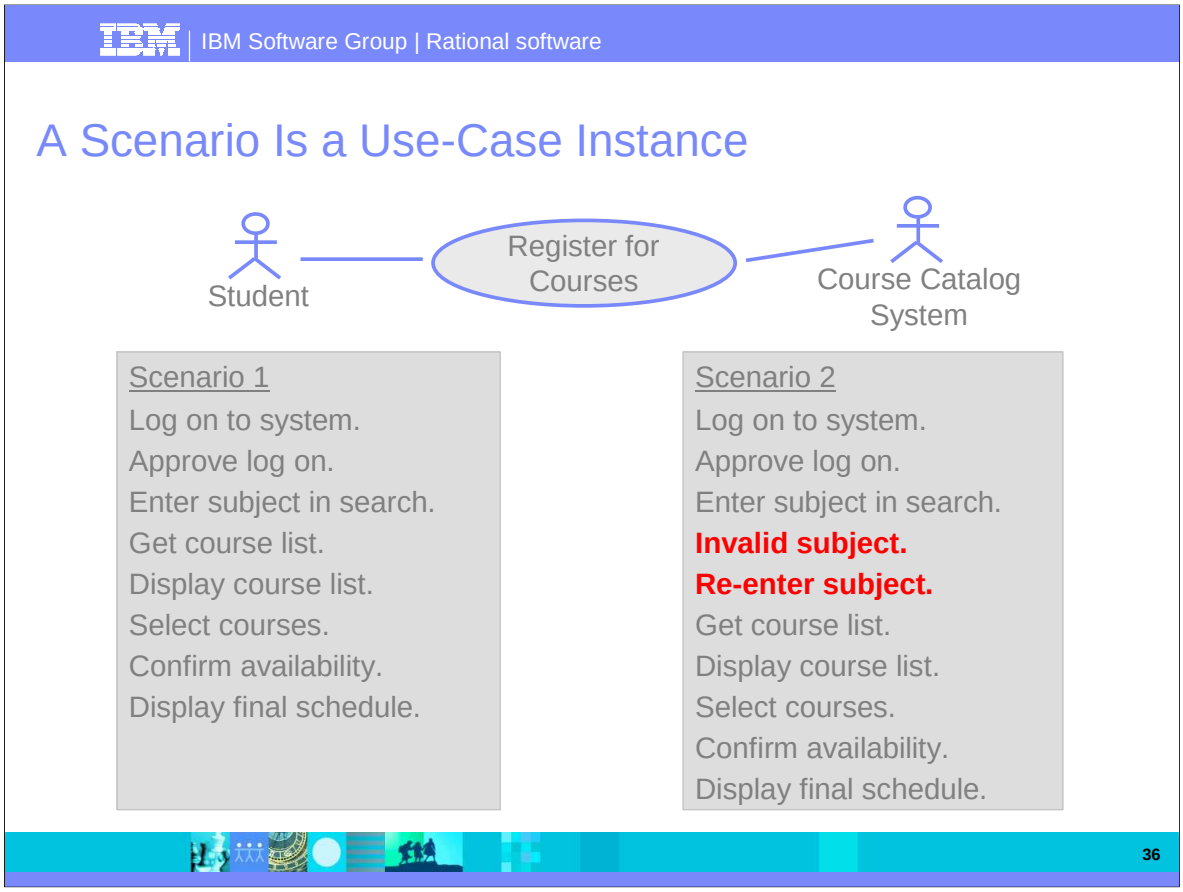
Where Are We?

- Introduction to Use-Case Modeling
- Find Actors and Use Cases
- ⇨ Use-Case Specifications

Use-Case Specifications

- Name
- Brief description
- Flow of Events
  - Basic flow
  - Alternative flows (regular variants, odd cases, error conditions)
- Special requirements
- Pre-conditions
- Post-conditions
- Etc.





A use-case instance describes one particular path through the flows of events described in a use case. It is a specific sequence of actions that illustrates the behavior of the system. This is also called a **scenario**. In the example here, the sketch of Scenario 1 for the Register for Courses use case shows a Student interacting with the Course Registration System and successfully enrolling in a course the first time. Scenario 2 shows a Student interacting with the Course Registration System and entering an invalid subject; that student must re-enter the subject before successfully getting the course list.

A use case defines a set of related scenarios. A use case represents all the possible sequences that might happen until the resulting value is achieved or until the system terminates the attempt. The Register for Courses use case represents both of these sequences, and all the other possible sequences of interactions that may occur when a Student tries to enroll in a course.

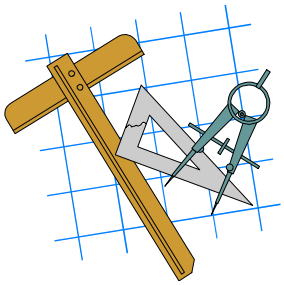
Users and stakeholders can often identify the sequence of actions they want to perform to obtain a result. Asking stakeholders for the sequence of actions they perform is a good way to start identifying the steps in a use case.

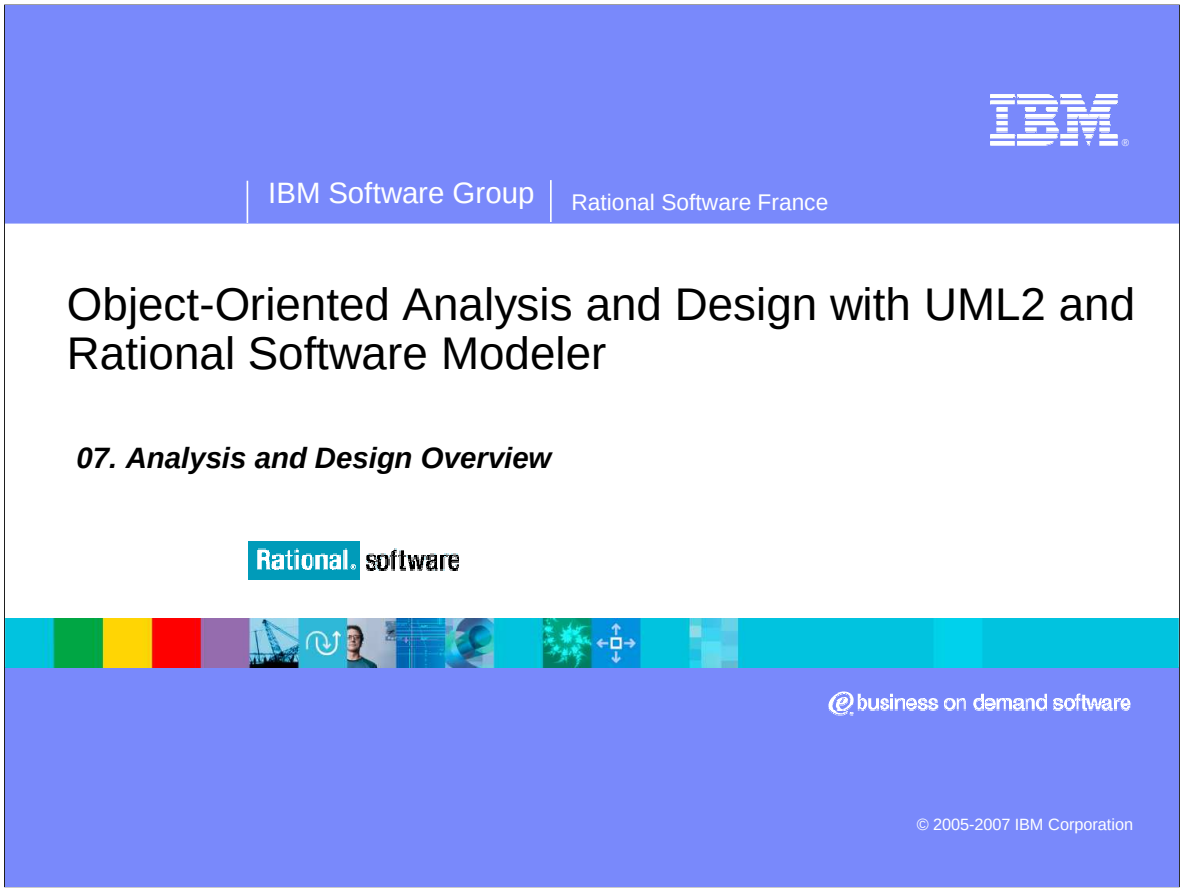
Other Requirements Management Artifacts

- Glossary
  - Defines important terms used in the project
- Supplementary Specification
  - Contains those requirements that do not map to a specific use case
    - Functional requirements that are general to many use cases
    - Non-functional requirements such as ease-of-use requirements, specific response times

Exercise

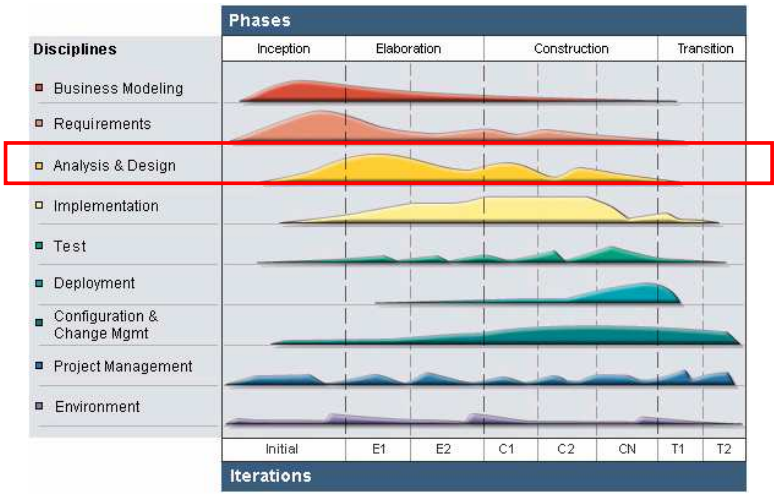
- Perform the exercise provided by the instructor (lab 3)

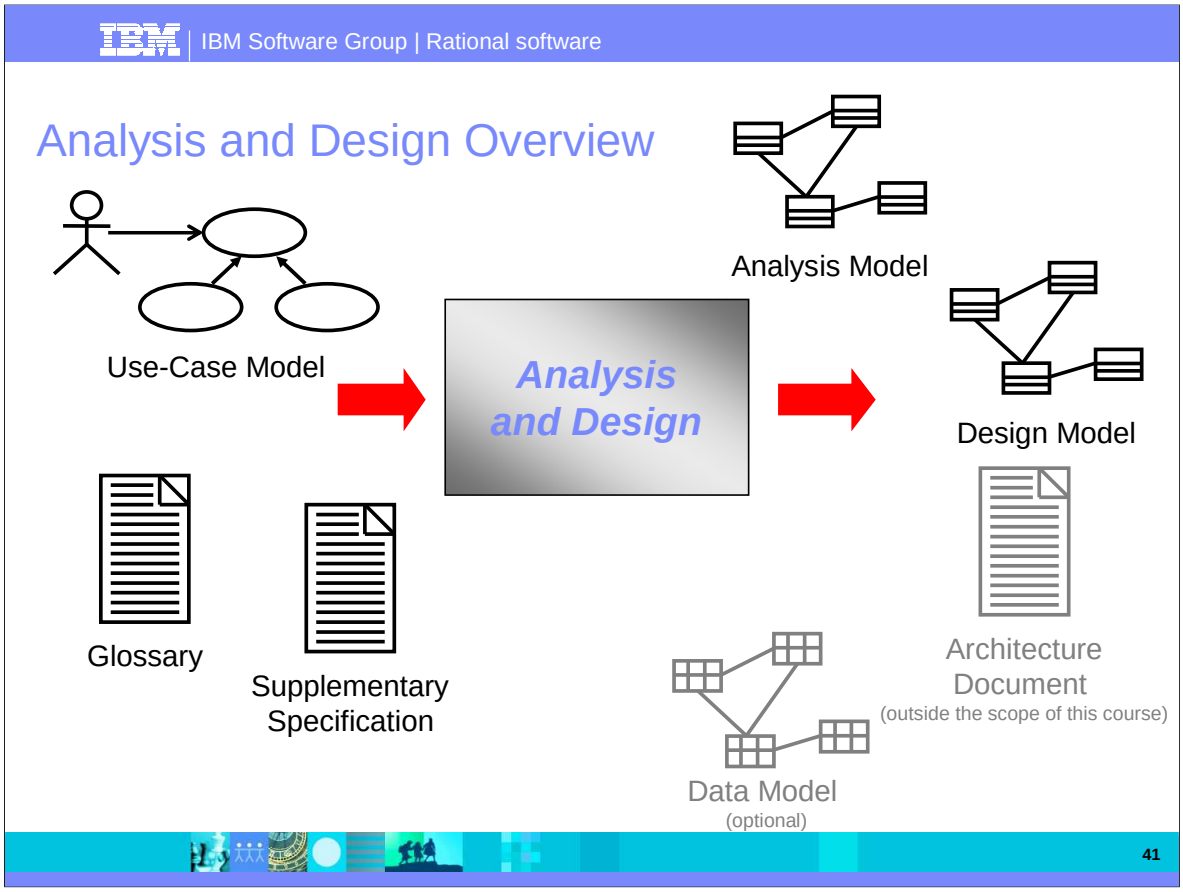




Analysis and Design in Context

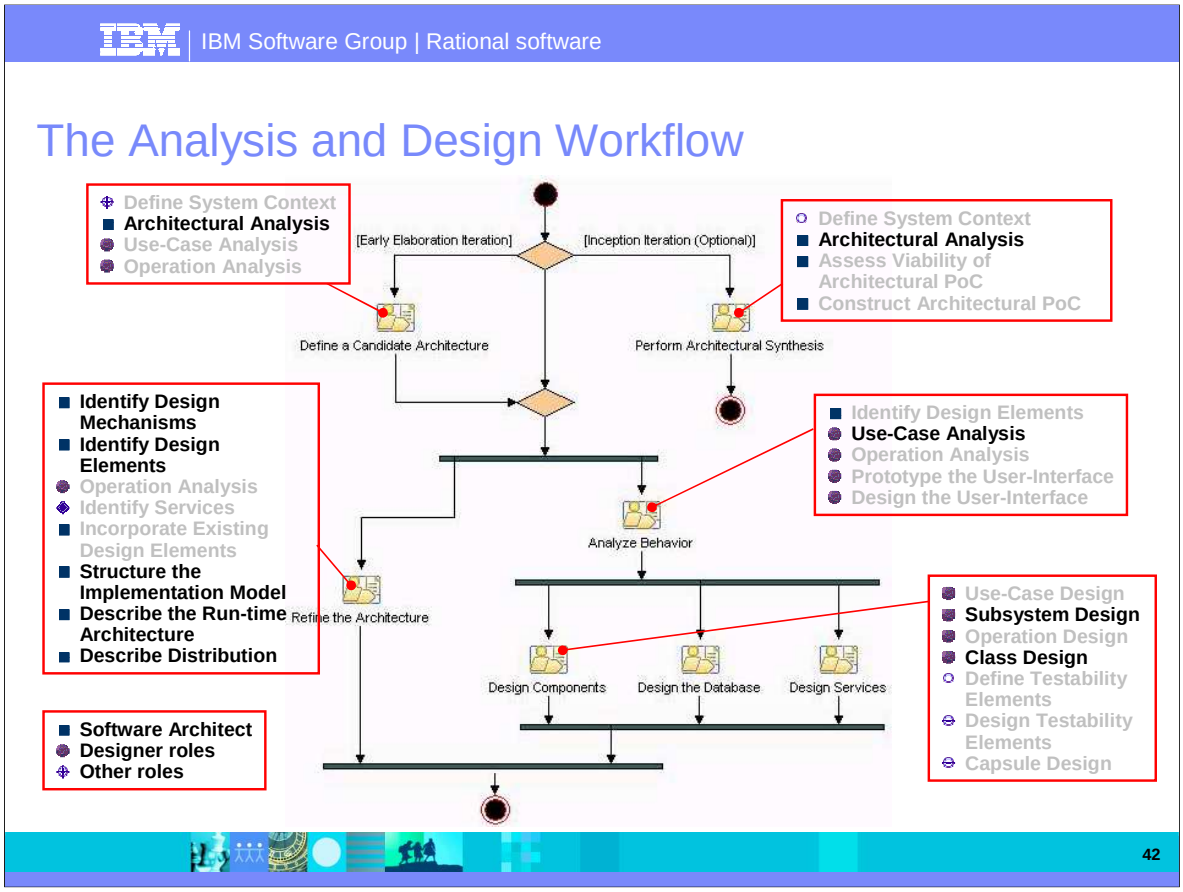
- The purposes of Analysis and Design are to:
  - Transform the requirements into a design of the system-to-be
  - Evolve a robust architecture for the system
  - Adapt the design to match the implementation environment, designing it for performance





Outputs:

- The **analysis model** contains the analysis classes and any associated work products. The analysis model may be a temporary work product, as it is in the case where it evolves into a design model, or it may continue to live throughout some or all of the project, and perhaps beyond, serving as a conceptual overview of the system.
- The **design model** is an abstraction of the implementation of the system. It is used to conceive as well as document the design of the software system. It is a comprehensive, composite work product encompassing all design classes, subsystems, packages, collaborations, and the relationships between them.
- The **software architecture document** provides a comprehensive overview of the architecture of the software system. It serves as a communication medium between the software architect and other project team members regarding architecturally significant decisions which have been made on the project.
- The **data model** is used to describe the logical and physical structure of the persistent information managed by the system. The data model may be initially created through reverse engineering of existing persistent data stores (databases) or may be initially created from a set of persistent design classes in the Design Model.



Perform Architectural Synthesis constructs and assesses an Architectural Proof-of-Concept to demonstrate that the system, as envisioned, is feasible.

Define a Candidate Architecture creates an initial sketch of the software architecture.

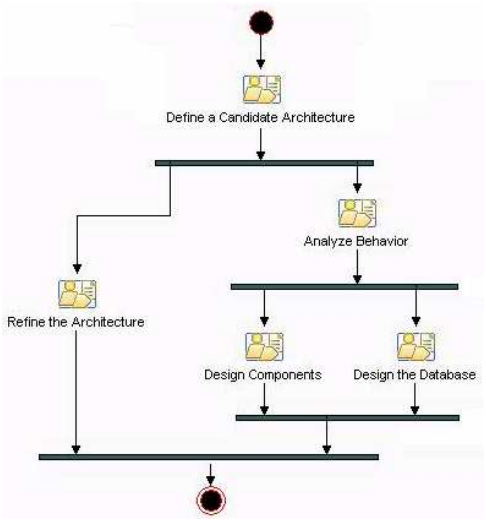
Refine the Architecture completes the architecture for an iteration.

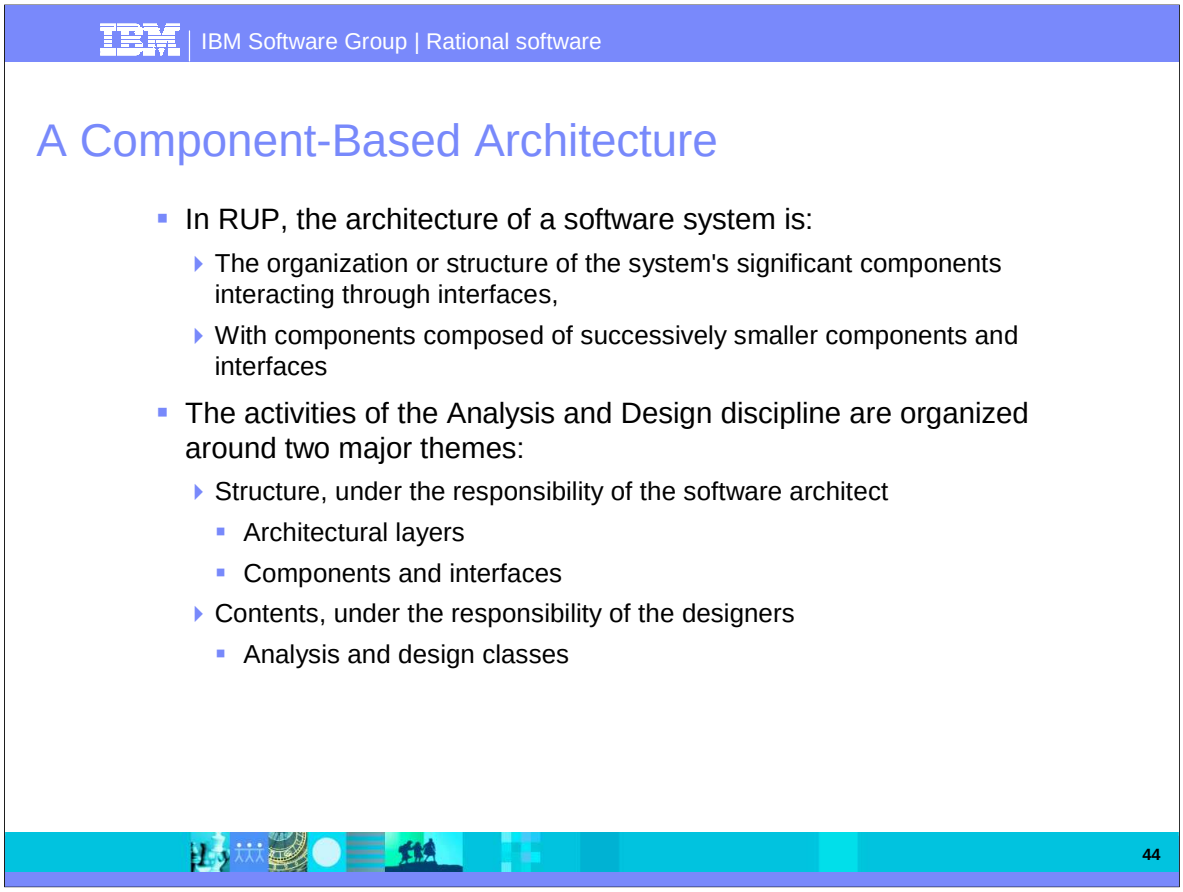
Analyze Behavior transforms the behavioral descriptions provided by the requirements into a set of elements upon which the design can be based.

Design Components refines the design of the system.

Simplified Workflow for the OOAD Course

- Early Elaboration iteration
- Goal of Elaboration:
  - To build a robust *architecture* that will support the requirements of the system at a reasonable cost and in a reasonable time
- To achieve this goal, we need:
  - To produce an evolutionary executable of production-quality components that will address all architecturally significant risks of the project
    - Addressing architecturally significant risks means selecting for the iteration the use-case scenarios that expose those risks





The **software architect role** leads and coordinates technical activities and artifacts throughout the project. The software architect establishes the overall structure for each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between these major groupings. Therefore, in contrast to the other roles, the software architect's view is one of breadth as opposed to one of depth.

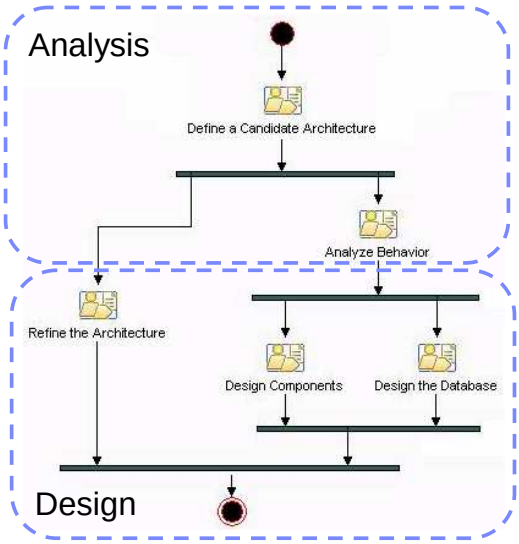
The software architect must be well-rounded and possess maturity, vision, and a depth of experience that allows for grasping issues quickly and making educated, critical judgment in the absence of complete information.

The **designer role** defines the responsibilities, operations, attributes, and relationships of one or several classes, and determines how they are adjusted to the implementation environment. In addition, the designer role may have responsibility for one or more classes, including analysis, design, subsystems, or testability.

The designer must have a solid working knowledge of use-case modeling techniques, system requirements, software design techniques (including UML and OOAD), and technologies with which the system will be implemented.

Roadmap for the OOAD Course

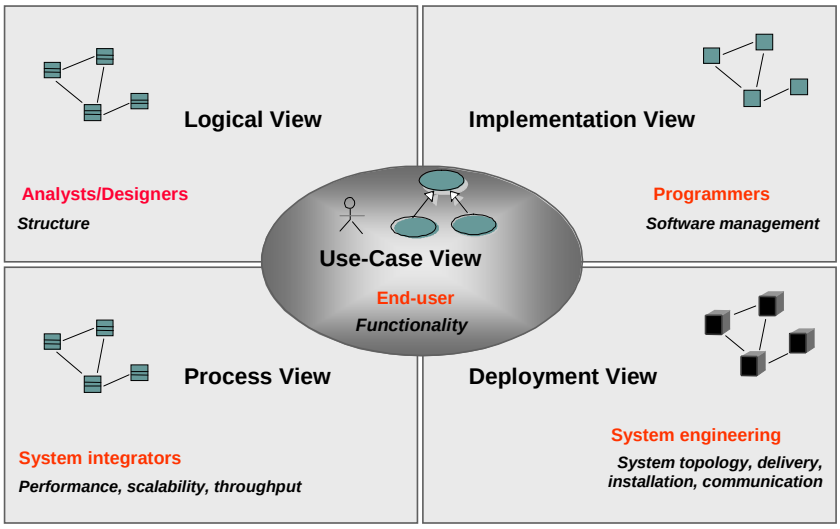
- Analysis
  - Architectural Analysis
    (Define a Candidate Architecture)
  - Use-Case Analysis
    (Analyze Behavior)
- Design
  - Identify Design Elements
    (Refine the Architecture)
  - Identify Design Mechanisms
    (Refine the Architecture)
  - Class Design
    (Design Components)
  - Subsystem Design
    (Design Components)
  - Describe the Run-time Architecture and Distribution
    (Refine the Architecture)
  - Design the Database



Analysis Versus Design

Analysis	Design
Focus on understanding the problem	Focus on understanding the solution
Idealized design	Operations and attributes
Behavior	Performance
System structure	Close to real code
Functional requirements	Object lifecycles
A small model	Nonfunctional requirements
	A large model

Architectural Views: The 4+1 View Model



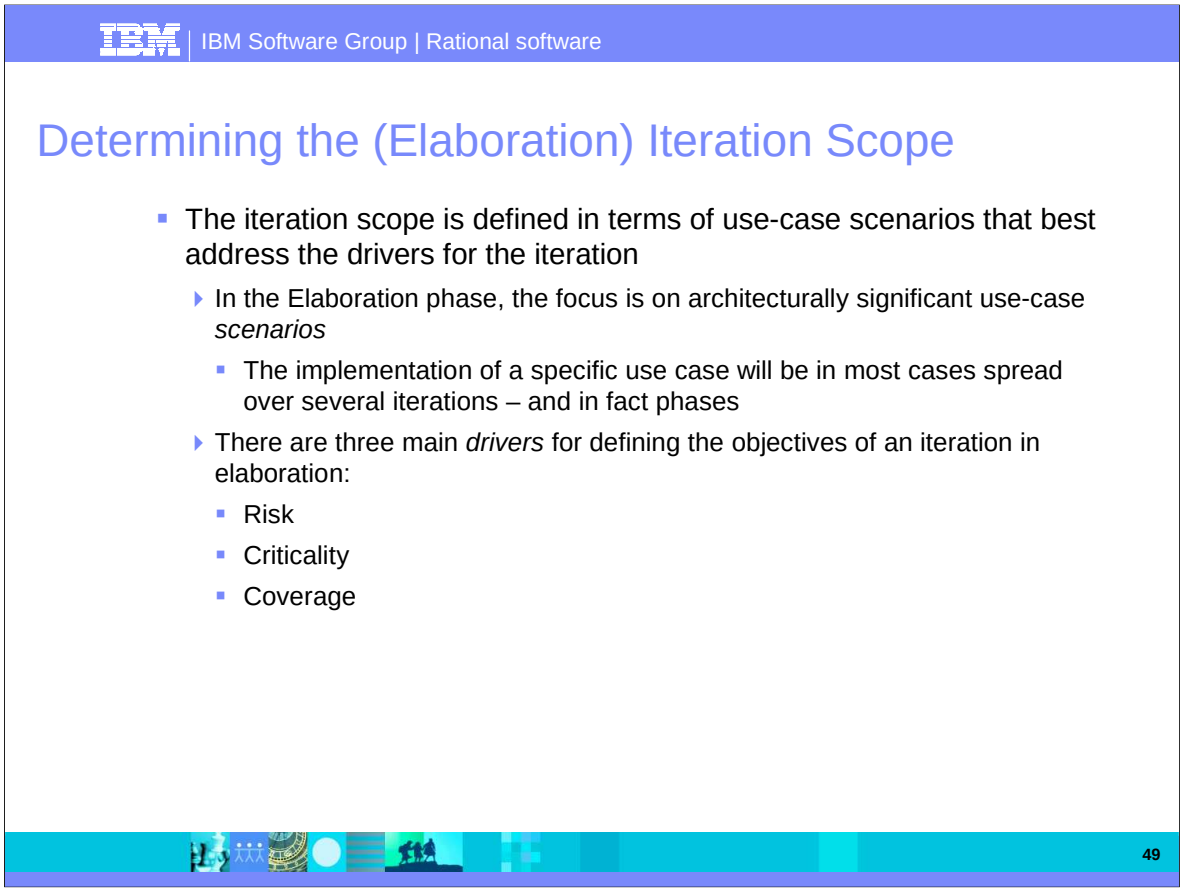
Architecture is represented by a number of different architectural views. In RUP, you start from a typical set of views, called the “4+1 view model” (Philippe Kruchten 1995, "The 4+1 view model of architecture," *IEEE Software.* 12(6), November 1995). It is composed of:

- The **Use-Case View**, which contains use cases and scenarios that encompasses architecturally significant behavior, classes, or technical risks. It is a subset of the Use-Case Model.
- The **Logical View**, which contains the most important design classes and their organization into packages and subsystems, and the organization of these packages and subsystems into layers. It contains also some use case realizations. It is a subset of the Design Model.
- The **Implementation View**, which contains an overview of the Implementation Model and its organization in terms of modules into packages and layers. The allocation of packages and classes (from the Logical View) to the packages and modules of the Implementation View is also described. It is a subset of the Implementation Model.
- The **Process View**, which contains the description of the tasks (process and threads) involved, their interactions and configurations, and the allocation of design objects and classes to tasks. This view need only be used if the system has a significant degree of Concurrency. In RUP, it is a subset of the Design Model.
- The **Deployment View**, which contains the description of the various physical nodes for the most typical platform configurations, and the allocation of tasks (from the Process View) to the physical nodes. This view need only be used if the system is distributed. It is a subset of the Deployment Model.

The architectural views are documented in a Software Architecture Document. You can envision additional views to express different special concerns: user-interface view, security view, data view, and so on. For simple systems, you may omit some of the views contained in the 4+1 view model.

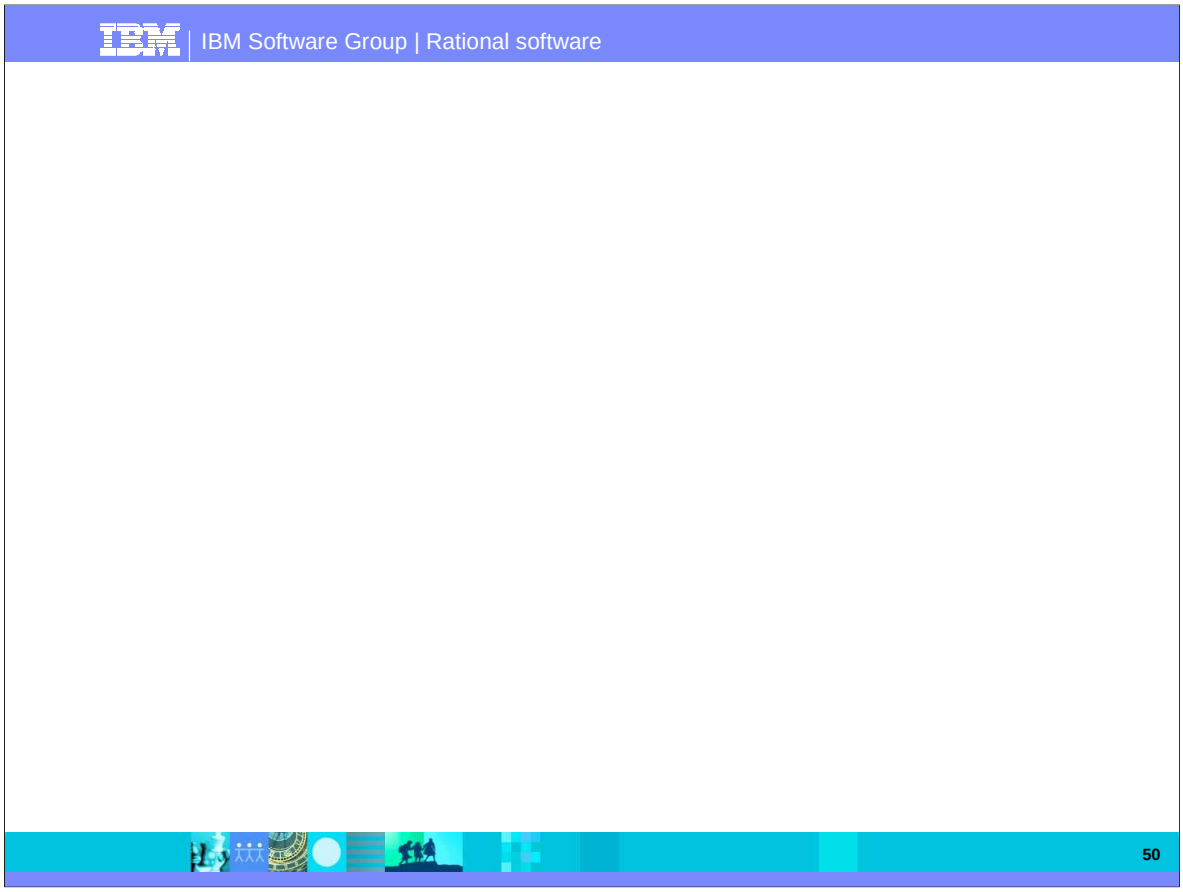
Organizing Models in RSA/RSM

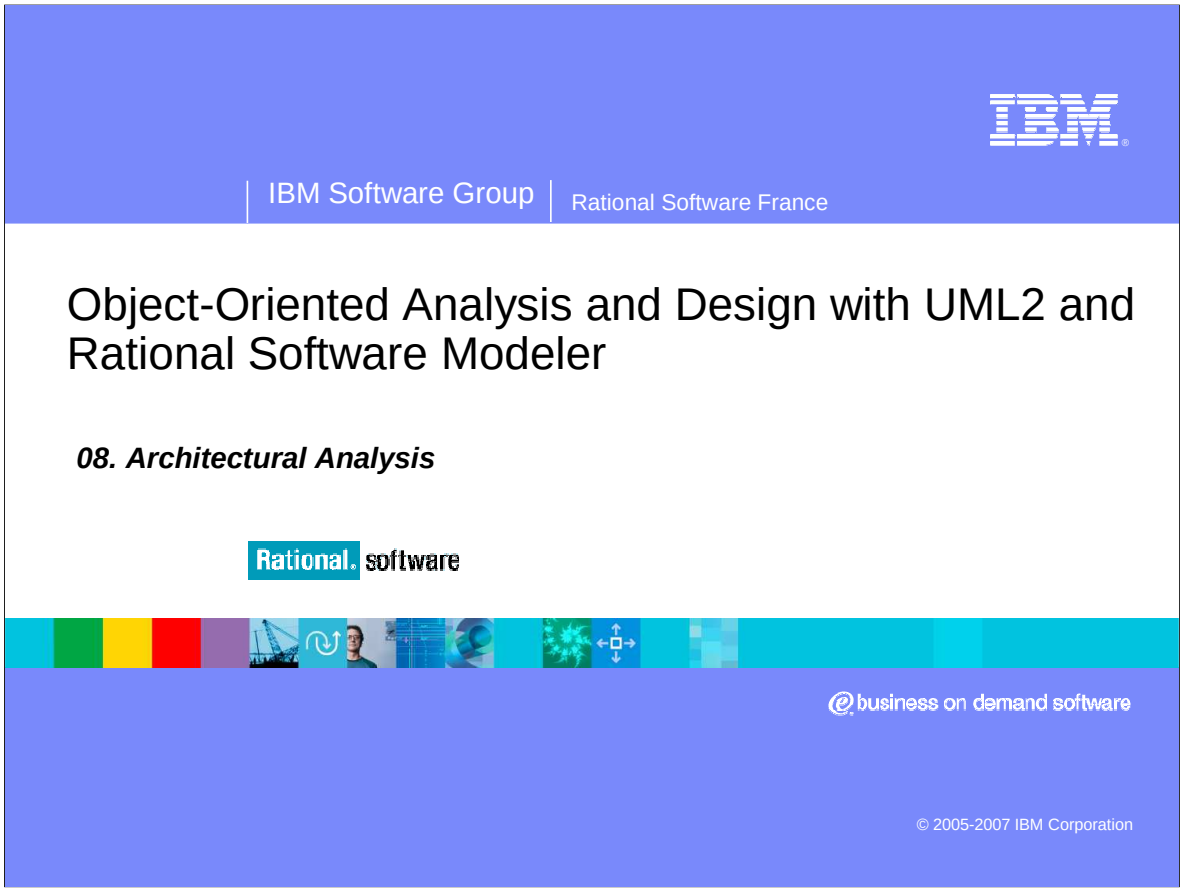
- Need for well-defined guidelines to represent the architectural views in your modeling and development environment
- Whitepaper: “Model Structure Guidelines For Rational Software Modeler And Rational Software Architect (2004 Release)”
  - Available on IBM developerWorks (http://www-128.ibm.com/developerworks/)
- Models and packages can contain any number of diagrams
  - One diagram is the “default” diagram, i.e. the diagram that will display when the owning model or package is opened
  - The default diagram should contain all the necessary information to navigate in the package, for instance:
    - Owned packages (double-click opens the package)
    - Other major owned elements, e.g. public classes and interfaces
    - Shortcuts to other diagrams (created by drag-and-drop)
    - Explanatory free text and/or notes
- Other guidelines will be introduced as we go along



The main driver to define iteration objectives are **risks**. You need to mitigate or retire your risks as early as you can. This is mostly the case in the elaboration phase, where most of your risks should be mitigated, but this can continue to be a key elements in construction as some risks remain high, or new risks are discovered. But since the goal of the elaboration phase is to baseline an architecture, some other considerations have to come into play, such as making sure that the architecture addresses all aspects of the software to be developed (**coverage**). This is important since the architecture will be used for further planning: organization of the team, estimation of code to be developed, etc.

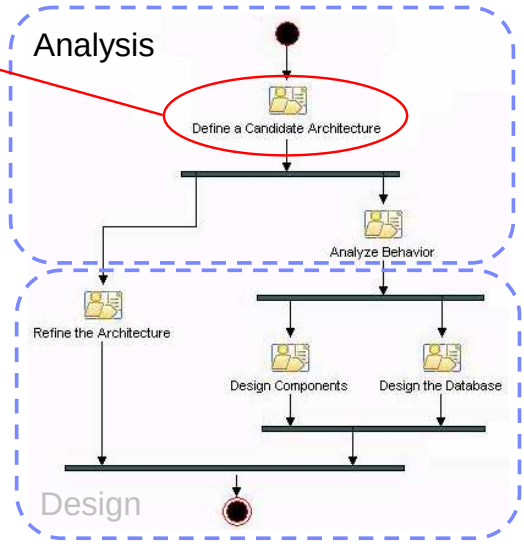
Finally, while focusing on risks is important, one should keep in mind what are the primary missions of the system; solving all the hard issues is good, but this must not be done in detriment of the core functionality: make sure that the critical functions or services of the system are indeed covered (**criticality**), even if there is no perceived risk associated with them.

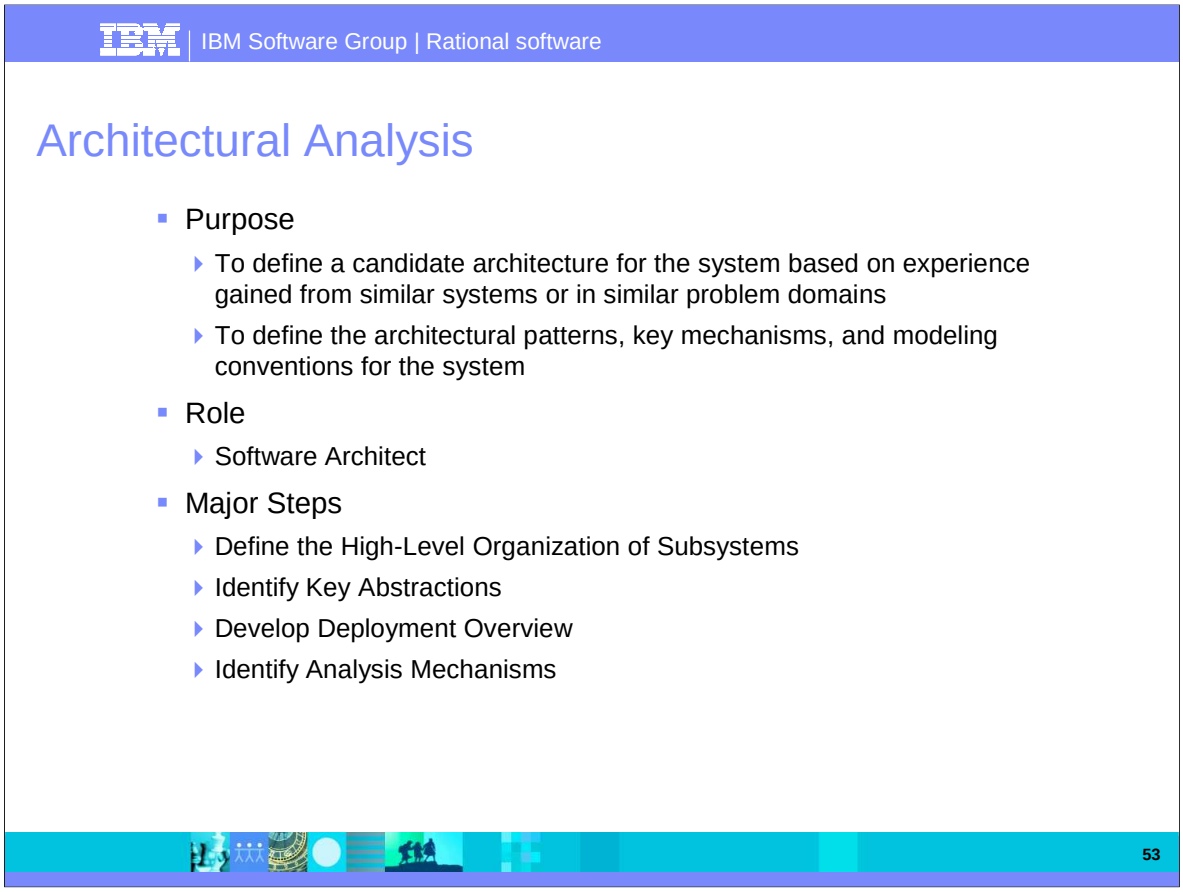




Roadmap for the OOAD Course

- Analysis
  - Architectural Analysis
    (Define a Candidate Architecture)
  - Use-Case Analysis
    (Analyze Behavior)
- Design
  - Identify Design Elements
    (Refine the Architecture)
  - Identify Design Mechanisms
    (Refine the Architecture)
  - Class Design
    (Design Components)
  - Subsystem Design
    (Design Components)
  - Describe the Run-time Architecture and Distribution
    (Refine the Architecture)
  - Design the Database





Architectural analysis focuses on defining a candidate architecture and constraining the architectural techniques to be used in the system. It relies on gathering experience gained in similar systems or problem domains to constrain and focus the architecture so that effort is not wasted in architectural rediscovery. In systems where there is already a well-defined architecture, architectural analysis might be omitted; architectural analysis is primarily beneficial when developing new and unprecedented systems.

Where Are We?

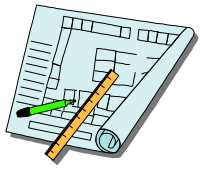
- Define the High-Level Organization of Subsystems
  - Identify Key Abstractions
  - Develop Deployment Overview
  - Identify Analysis Mechanisms

Define the High-Level Organization of Subsystems

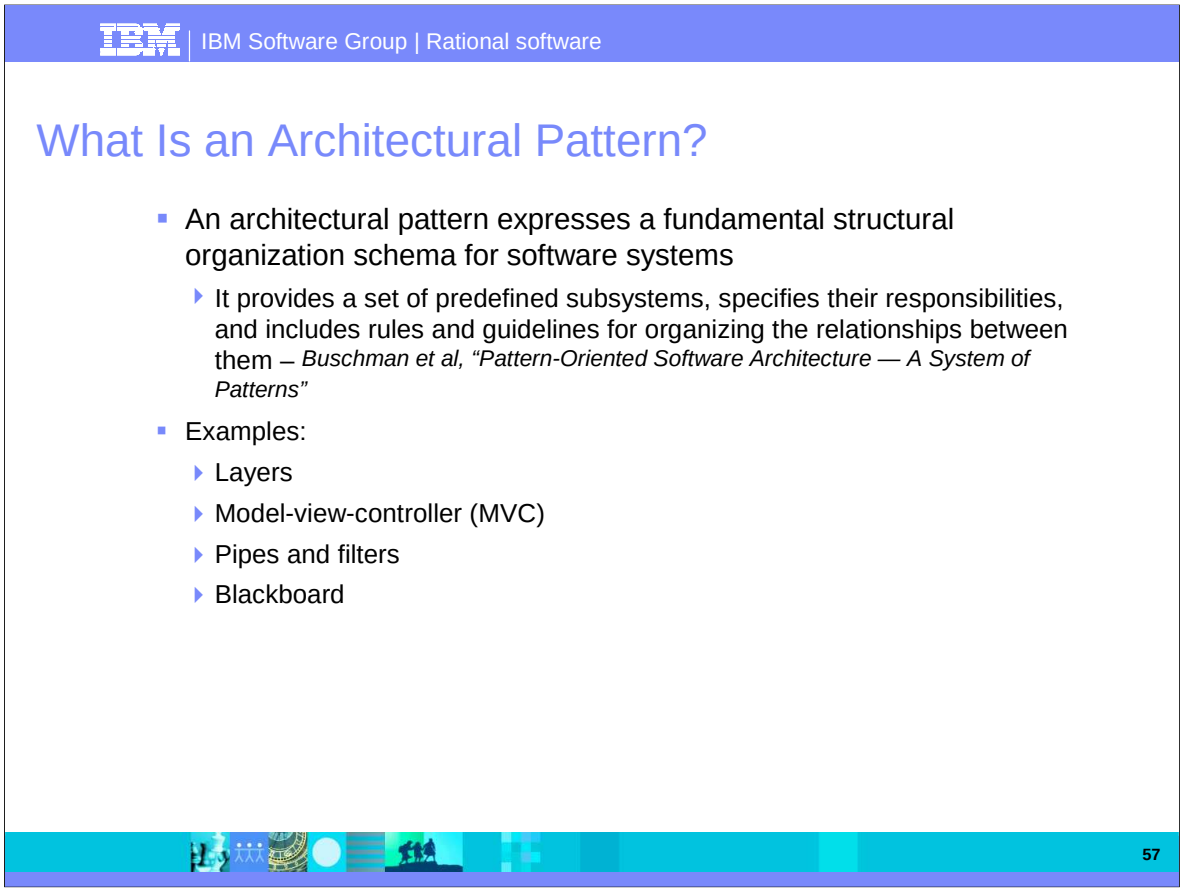
- Purpose
  - To create an initial structure for the Design Model
- Normally the design model is organized in layers – a common **architectural pattern** for moderate to large-sized systems
- During architectural analysis, you usually focus on the two high-level layers, that is, the **application** and **business-specific** layers
  - This is what is mean by the *high-level organization of subsystems*

Patterns and Frameworks

- Pattern
  - Provides a common solution to a common problem in a context
- Analysis/Design pattern
  - Provides a solution to a narrowly-scoped technical problem
  - Provides a fragment of a solution, or a piece of the puzzle
- Framework
  - Defines the general approach to solving the problem
  - Provides a skeletal solution, whose details may be Analysis/Design patterns



Design patterns are studied in the Design part of the course.



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.

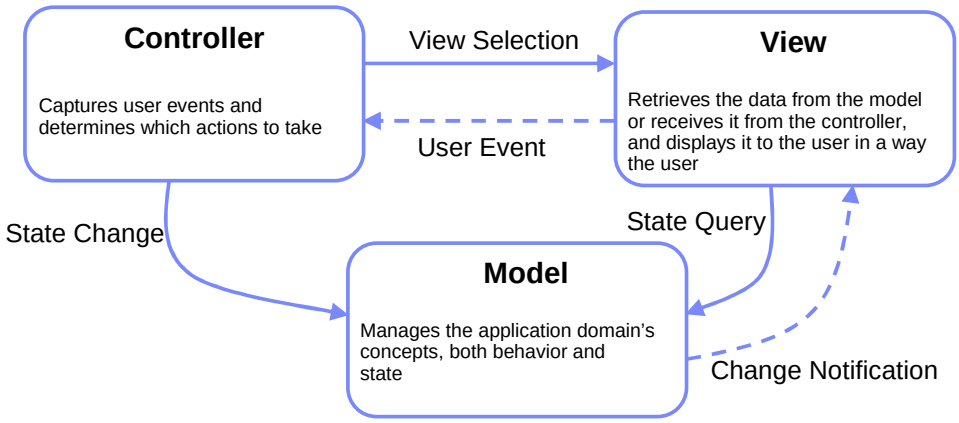
Blackboard: The Blackboard pattern is where independent, specialized applications collaborate to derive a solution, working on a common data structure.

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

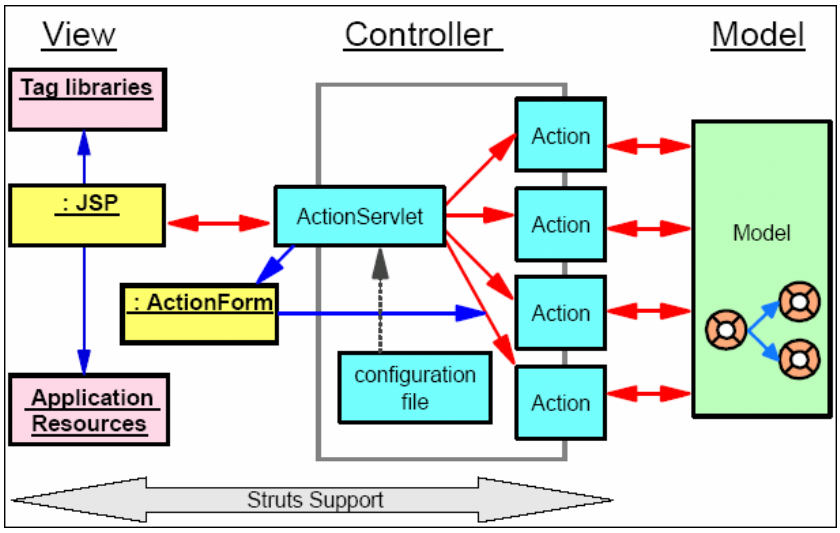
The architectural patterns listed above imply certain system characteristics, performance characteristics, and process and distribution architectures. Each solves certain problems but also poses unique challenges. For this course, you will concentrate on the Layers architectural pattern.

The Model-View-Controller (MVC) Architecture

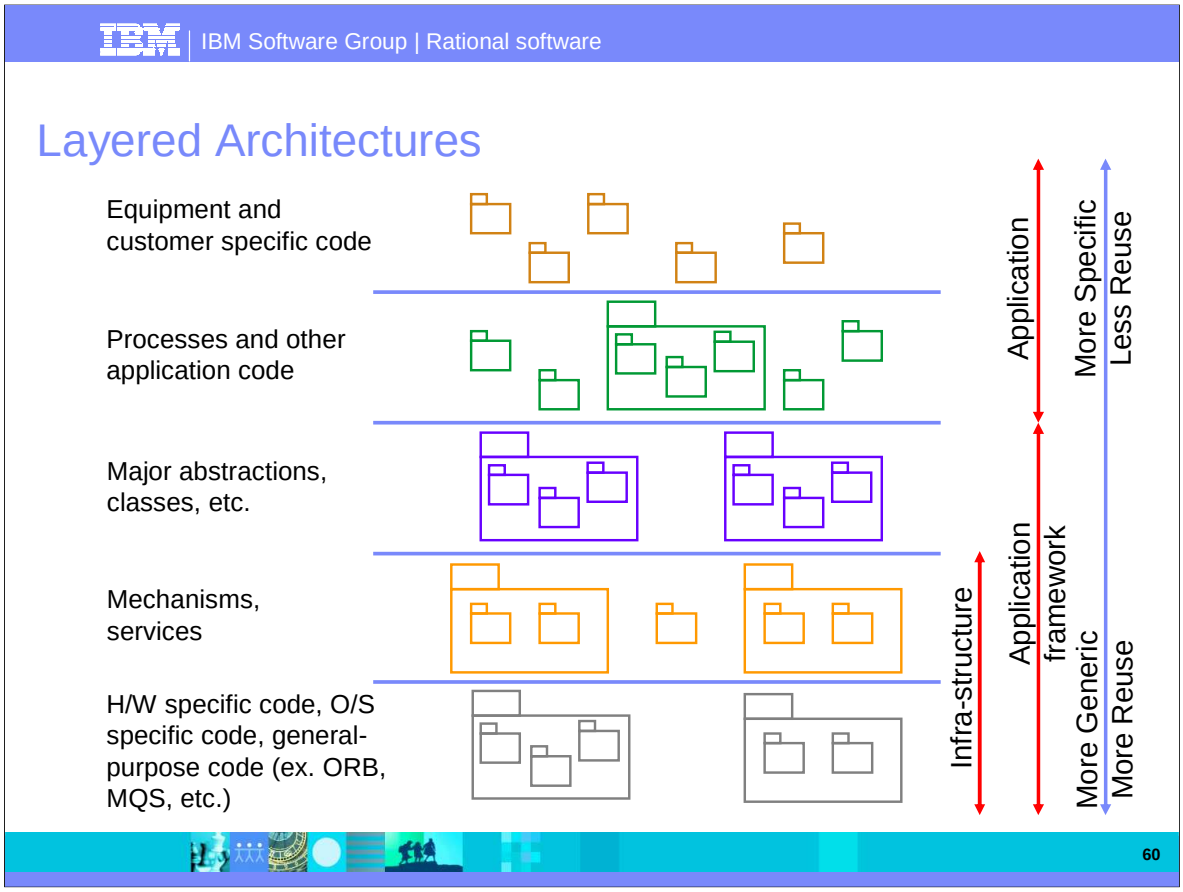
- Conceived in the mid-1980's
- Extensively applied in most object-oriented user interfaces
- Adapted to respond to specific platform requirements, such as J2EE



An MVC Example: Struts Components



(From the IBM Redbook, Rational Application Developer V6 Programming Guide, June 2005)



Context

A large system that requires decomposition.

Problem

A system which must handle issues at different levels of abstraction. For example: hardware control issues, common services issues and domain-specific issues. It would be extremely undesirable to write vertical components that handle issues at all levels. The same issue would have to be handled (possibly inconsistently) multiple times in different components.

Forces

- Parts of the system should be replaceable.
- Changes in components should not ripple.
- Similar responsibilities should be grouped together.
- Size of components-complex components may have to be decomposed.

Solution

Structure the systems into groups of components that form layers on top of each other. Make upper layers use services of the layers below only (never above). Try not to use services other than those of the layer directly below (don't skip layers unless intermediate layers would only add pass-through components).

Layering Considerations

- 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

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

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

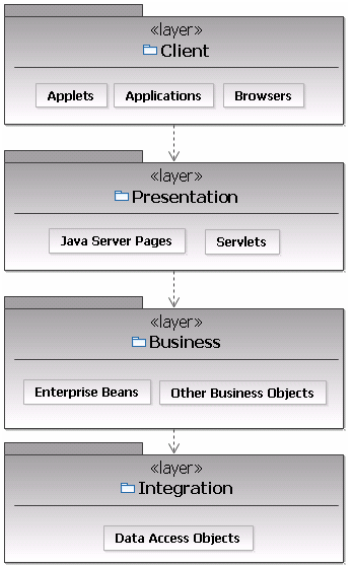
There is generally only a single application layer. On the other hand, the number of domain layers is dependent upon the complexity of both the problem and the solution spaces.

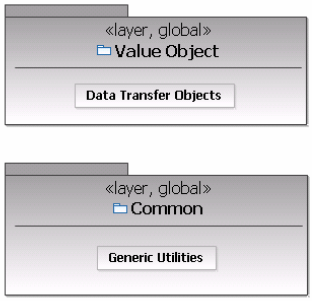
When a domain has existing systems, complex systems composed of inter-operating systems, and/or systems where there is a strong need to share information between design teams, the Business-Specific layer may be structured into several layers for clarity.

Modeling Architectural Layers

- Architectural layers can be modeled using packages stereotyped <<layer>>

Software Layers for a Generic J2EE Application

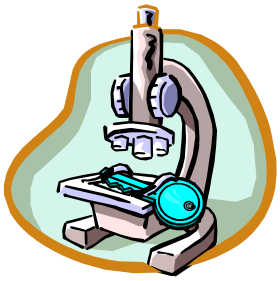




Note: <<global>> is a mere convention used here to indicate layers that can be used by all others

Where Are We?

- Define the High-Level Organization of Subsystems
- **Identify Key Abstractions**
- Develop Deployment Overview
- Identify Analysis Mechanisms



What Are Key Abstractions?

- A key abstraction is a concept, normally uncovered in Requirements, that the system must be able to handle
- Sources for key abstractions
  - Domain knowledge
  - Requirements
  - Glossary
  - Domain Model, or the Business Model (if one exists)

Describing Key Abstractions

- Key abstractions are modeled as analysis classes
- For each class, provide
  - A short description of the class
  - Its main attributes
  - Its relationships with other classes
- Don't spend too much time describing classes in detail at this initial stage
  - The purpose is not to identify classes that will survive throughout design
  - You will probably identify classes and relationships not actually needed by the use cases
  - This initial set of classes is useful to "jump-start" the Use-Case Analysis task

Do not turn the next page before being told to do so!

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. It is not the objective 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. This will help to reduce any duplicate effort that may result when different teams analyze the individual use cases.

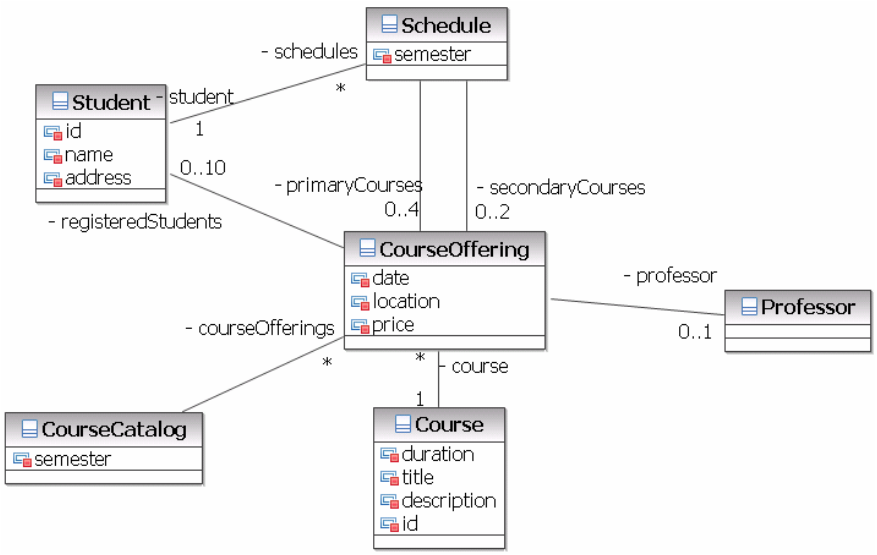
Relationships defined at this point reflect the semantic connections between the defined abstractions, not the relationships necessary to support the implementation or the required communication among abstractions.

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. Do not spend much time describing analysis classes in detail at this initial stage, because there is a risk that you might 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.

Group Exercise

- Identify the key abstractions for the Course Registration System

Key Abstractions for the Course Registration System

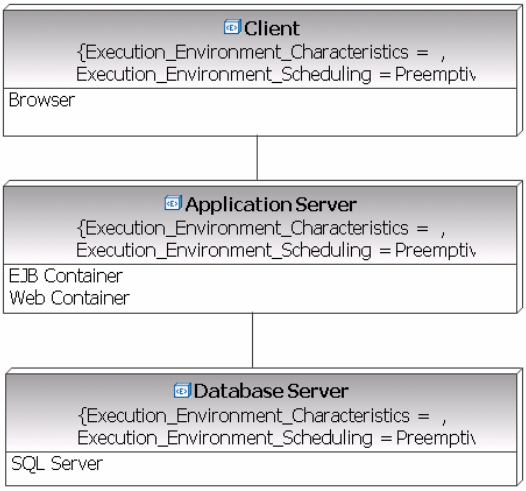


Where Are We?

- Define the High-Level Organization of Subsystems
- Identify Key Abstractions
- ⇨ Develop Deployment Overview
- Identify Analysis Mechanisms

Develop Deployment Overview

- Purpose:
  - To gain an understanding of the geographical distribution and operational complexity of the system
- Develop the high level overview of how the software is deployed to show:
  - Remote access
  - Distribution across multiple nodes
  - Existing hardware and software components



Where Are We?

- Define the High-Level Organization of Subsystems
- Identify Key Abstractions
- Develop Deployment Overview
- ⇨ Identify Analysis Mechanisms

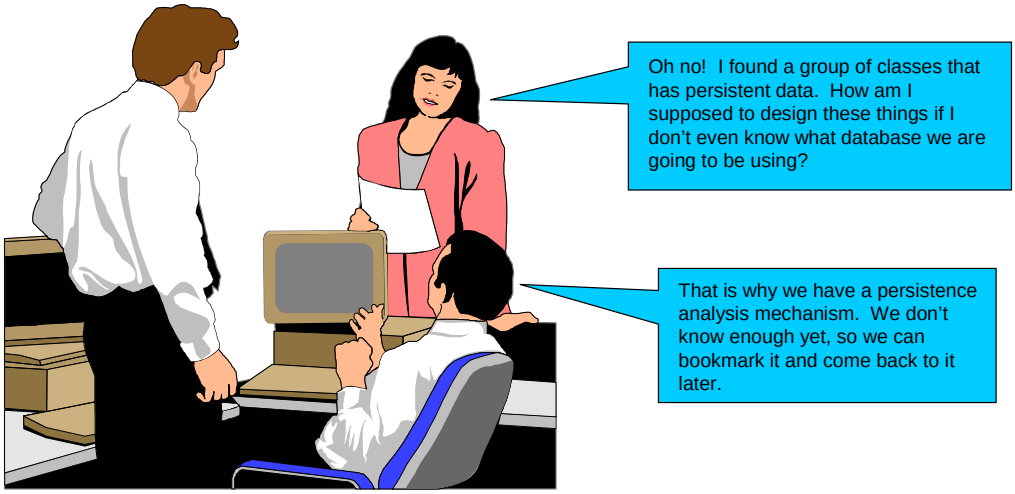
What Are Analysis Mechanisms?

- Analysis mechanisms are architectural mechanisms* used early in the Analysis and Design process:
  - Capture the key aspects of a solution in a way that is implementation independent
  - Are “computer science” concepts, usually unrelated to the problem domain
  - Provide specific behaviors to a domain-related class or component
- Examples:
  - Persistence
  - Inter-process communication
  - Error or fault handling
  - Notification
  - Messaging
  - Etc.

* Architectural mechanisms = Common concrete solutions to frequently encountered problems

Why Use Analysis Mechanisms?

- Analysis mechanisms are used during analysis to reduce the complexity of analysis and to improve its consistency by providing designers with a shorthand representation for complex behavior



Analysis mechanisms are primarily used as "placeholders" for complex technology in the middle and lower layers of the architecture. When mechanisms are used as "placeholders" in the architecture, the architecting effort is less likely to become distracted by the details of mechanism behavior.

Mechanisms allow the analysis effort to focus on translating the functional requirements into software concepts without bogging down in the specification of relatively complex behavior needed to support the functionality but which is not central to it. Analysis mechanisms often result from the instantiation of one or more architectural or analysis patterns**.**

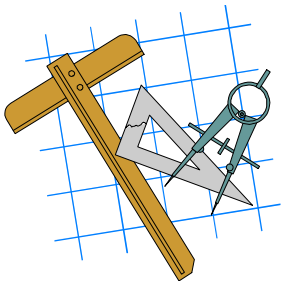
Persistence provides an example of analysis mechanisms. A persistent object is one that logically exists beyond the scope of the program that created it. The need to have object lifetimes that span use cases, process lifetimes, or system shutdown and startup, defines the need for object persistence. Persistence is a particularly complex mechanism. During analysis we do not want to be distracted by the details of how we are going to achieve persistence. This gives rise to a "persistence" analysis mechanism that allows us to speak of persistent objects and capture the requirements we will have on the persistence mechanism without worrying about what exactly the persistence mechanism will do or how it will work.

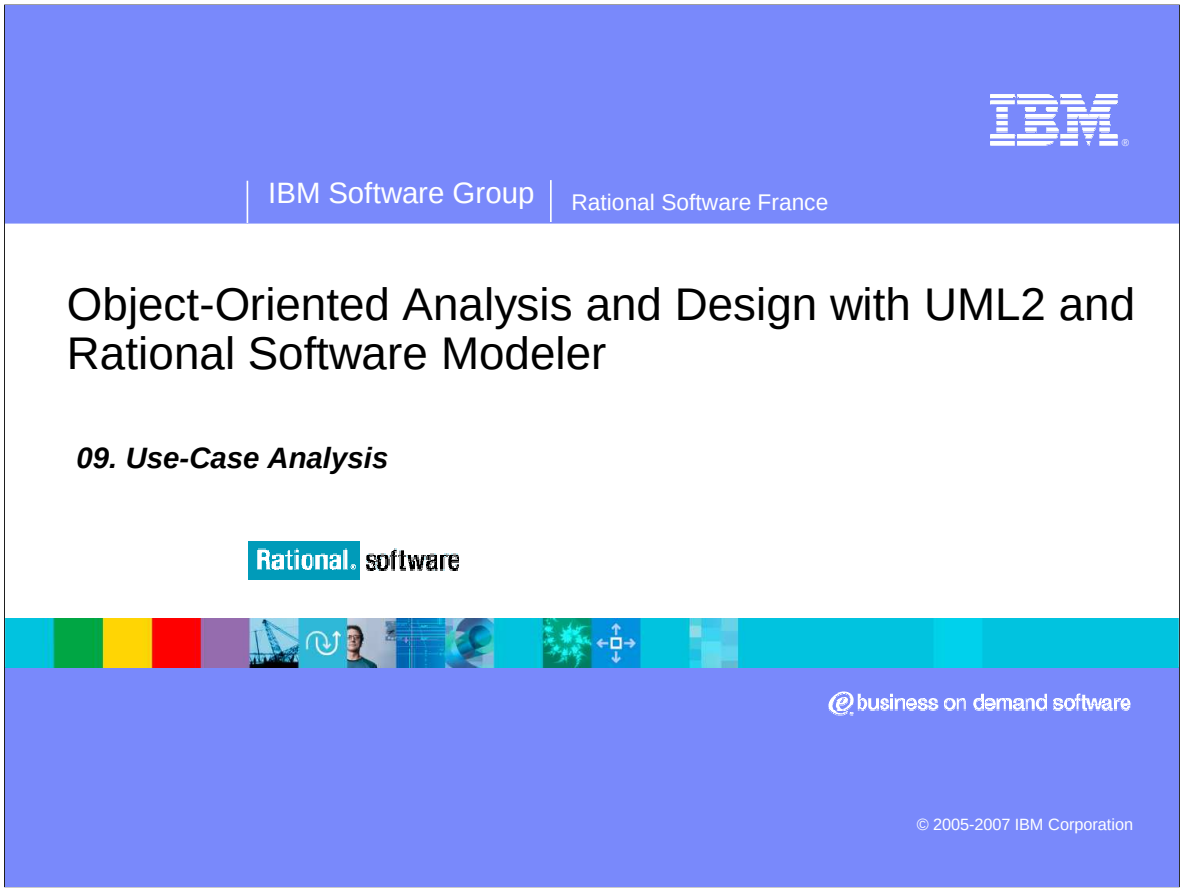
Identifying and Describing Analysis Mechanisms

- Analysis mechanisms can be identified top-down (a priori knowledge) or bottom-up (discovered as you go along)
  - Initially the name might be all that exists (for instance, persistence)
- As client classes get identify, it becomes necessary to qualify the use of each mechanism
  - For persistence, identify characteristics like granularity (size), volume (number), retrieval mechanism, update frequency, etc.
- Eventually, analysis mechanisms will be refined into design mechanisms
  - A design mechanism assumes some details of the implementation environment, but it is not tied to a specific implementation
  - Example: DBMS as the design mechanism for persistence
- And design mechanisms into actual implementation mechanisms
  - Example: Oracle

Exercise

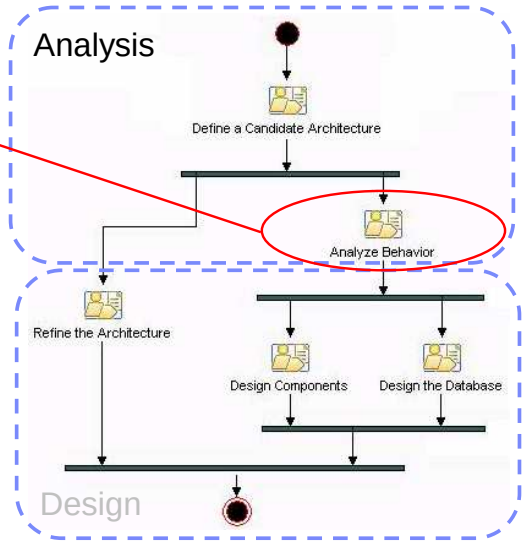
- Perform the exercise provided by the instructor (lab 4)





Roadmap for the OOAD Course

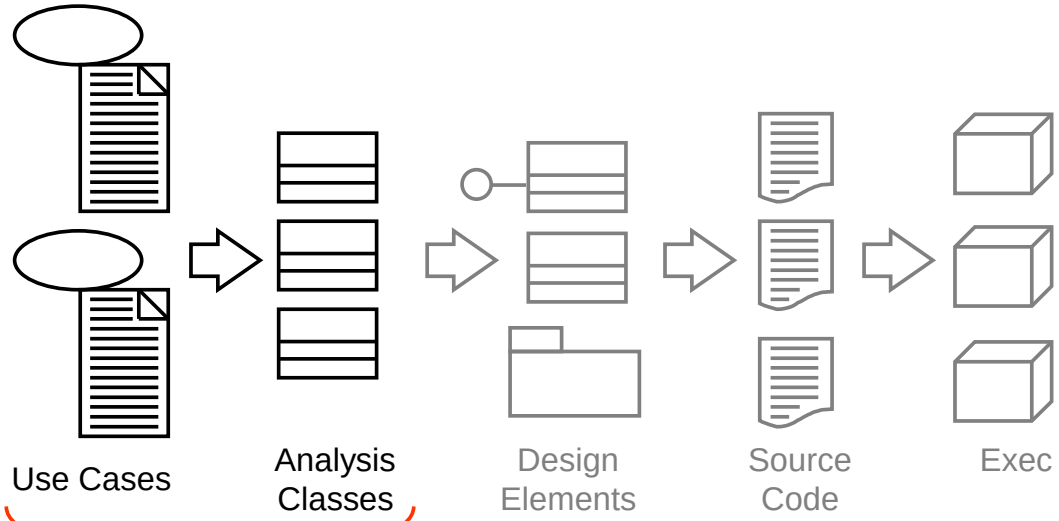
- Analysis
  - Architectural Analysis
    (Define a Candidate Architecture)
  - Use-Case Analysis
    (Analyze Behavior)
- Design
  - Identify Design Elements
    (Refine the Architecture)
  - Identify Design Mechanisms
    (Refine the Architecture)
  - Class Design
    (Design Components)
  - Subsystem Design
    (Design Components)
  - Describe the Run-time Architecture and Distribution
    (Refine the Architecture)
  - Design the Database



Use-Case Analysis

- Purpose
  - To identify the analysis classes of our system, including:
    - their “responsibilities”, attributes and associations to other classes, and
    - usage of analysis mechanisms
- Role
  - Designer
- Major Steps
  - Create Analysis Use-Case Realization
  - Supplement the Use-Case Description
  - Model Use-Case Scenarios with Interaction Diagrams
  - Model Participating Classes in Class Diagrams
  - Reconcile the Analysis Use-Case Realizations
  - Qualify Analysis Mechanisms

Analysis Classes: A First Step Toward Executables



Where Are We?

- Create Analysis Use-Case Realization
  - Supplement the Use-Case Description
  - Model Use-Case Scenarios with Interaction Diagrams
  - Model Participating Classes in Class Diagrams
  - Reconcile the Analysis Use-Case Realizations
  - Qualify Analysis Mechanisms

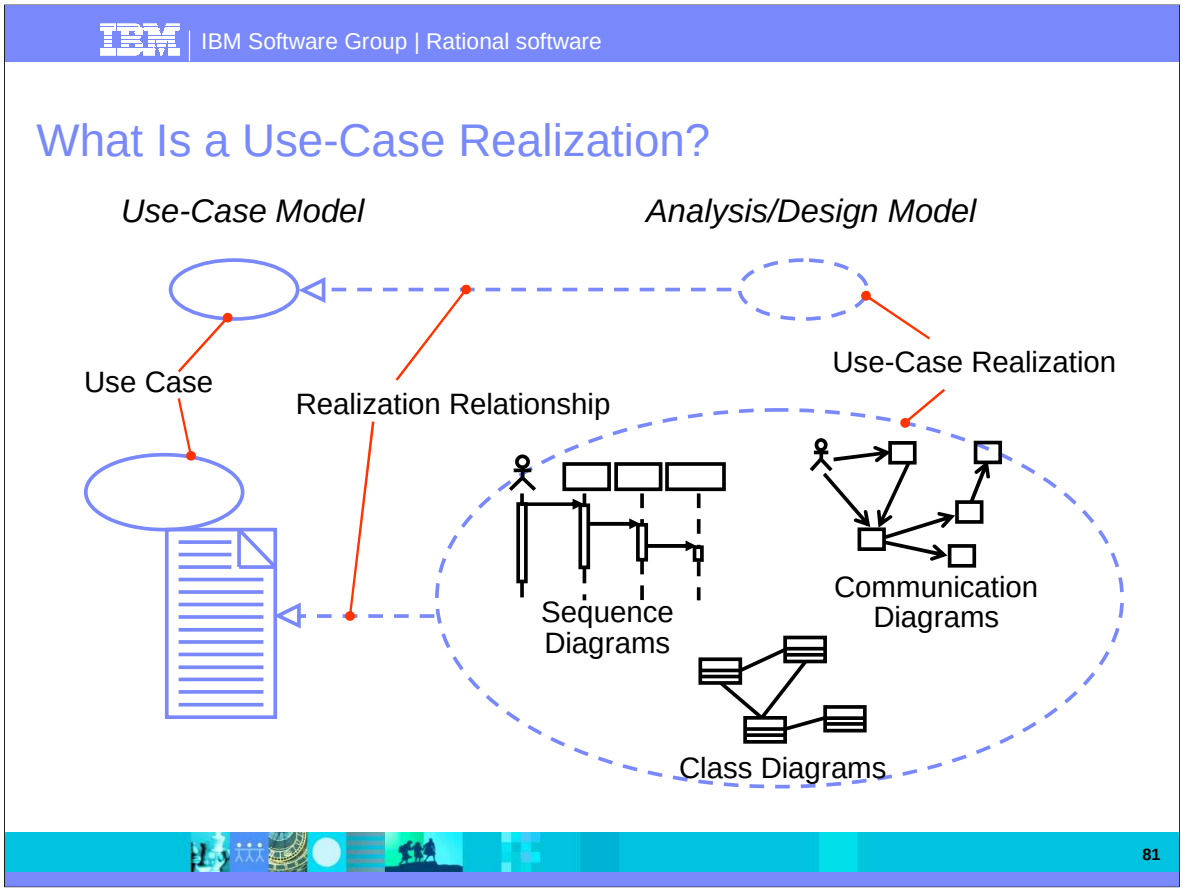
What Is a Use-Case Realization?

- The bridge between Requirements-centric tasks and Analysis/Design-centric tasks
- It provides:
  - A way to trace behavior in the Analysis and Design Models back to the Use-Case Model
  - A construct in the Analysis and Design Models, which organizes work products related to the use case but which belong to the design model
    - Typically consist of sequence and class diagrams
- Shown as a collaboration* stereotyped <<use-case realization>>

* UML Collaboration = structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality

A **use-case realization** describes how a particular use case is realized within the Analysis/Design Model, in terms of collaborating objects. A use-case realization ties together the use cases from the Use-Case Model with the classes and relationships of the Analysis/Design Model. A use-case realization specifies what classes must be built to implement each use case.

A use-case realization in the Analysis/Design Model can be traced to a use case in the Use-Case Model. A **realization relationship** is drawn from the use-case realization to the use case it realizes.



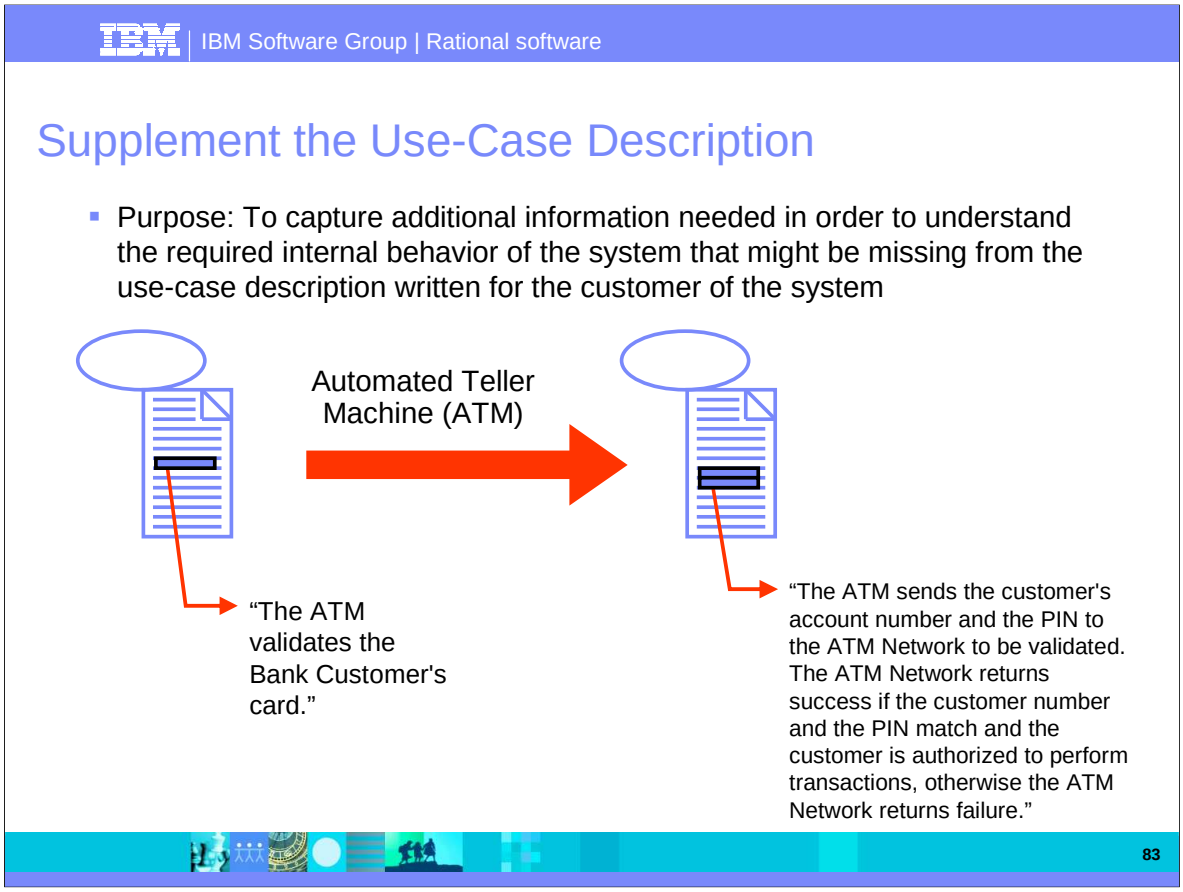
Use-case realizations are represented as stereotyped collaborations. The symbol for a collaboration is an ellipsis containing the name of the collaboration. The symbol for a use-case realization is a dotted line version of the collaboration symbol. In RSA 7.0, a UML Collaboration can only be shown as a rectangle.

A use-case realization consists of a set of **diagrams** that model the context of the collaboration (the classes/objects that implement the use case and their relationships — class diagrams), and the interactions of the collaborations (how these classes/objects interact to perform the use cases — communication and sequence diagrams).

The number and types of the diagrams that are used depend on what is needed to provide a complete picture of the collaboration and the guidelines developed for the project under development.

Where Are We?

- Create Analysis Use-Case Realization
- ⇨ Supplement the Use-Case Description
- Model Use-Case Scenarios with Interaction Diagrams
- Model Participating Classes in Class Diagrams
- Reconcile the Analysis Use-Case Realizations
- Qualify Analysis Mechanisms



Examine the use-case description to see if the internal behavior of the system is clearly defined. The internal behavior of the system should be **unambiguous**, so that it is clear what the system must do.

Sources of information for this detail include domain experts who can help define what the system needs to do. A good question to ask, when considering a particular behavior of the system, is "what does it mean for the system to do that thing?".

The following **alternatives exist for supplementing the description** of the Flow of Events:

- Do not describe it at all. This might be the case if you think the interaction diagrams are self-explanatory, or if the Flow of Events of the corresponding use case provides a sufficient description.
- Supplement the existing Flow of Event description. Add supplementary descriptions to the Flow of Events in areas where the existing text is unclear about the actions the system should take.
- Describe it as a complete textual flow, separate from the "external" Use Case Flow of Events description. This is appropriate in cases where the internal behavior of the system bears little resemblance to the external behavior of the system. In this case, a completely separate description, associated with the analysis use-case realization rather than the use case, is warranted.

Where Are We?

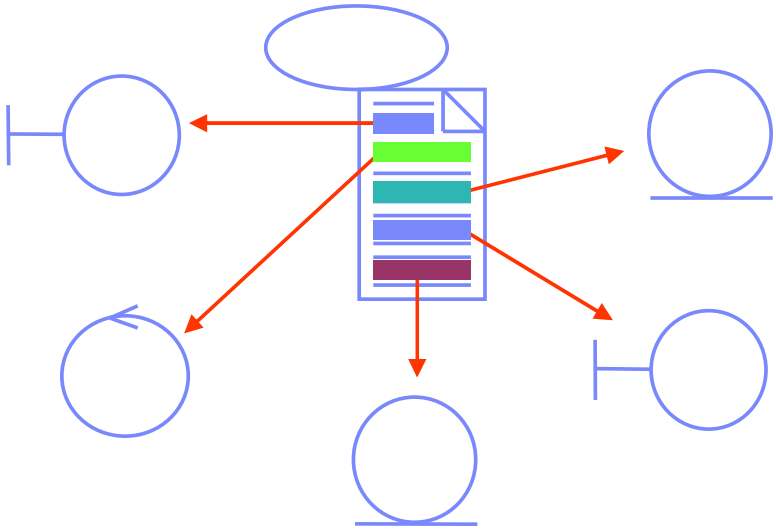
- Create Analysis Use-Case Realization
- Supplement the Use-Case Description
- ⇨ Model Use-Case Scenarios with Interaction Diagrams
- Model Participating Classes in Class Diagrams
- Reconcile the Analysis Use-Case Realizations
- Qualify Analysis Mechanisms

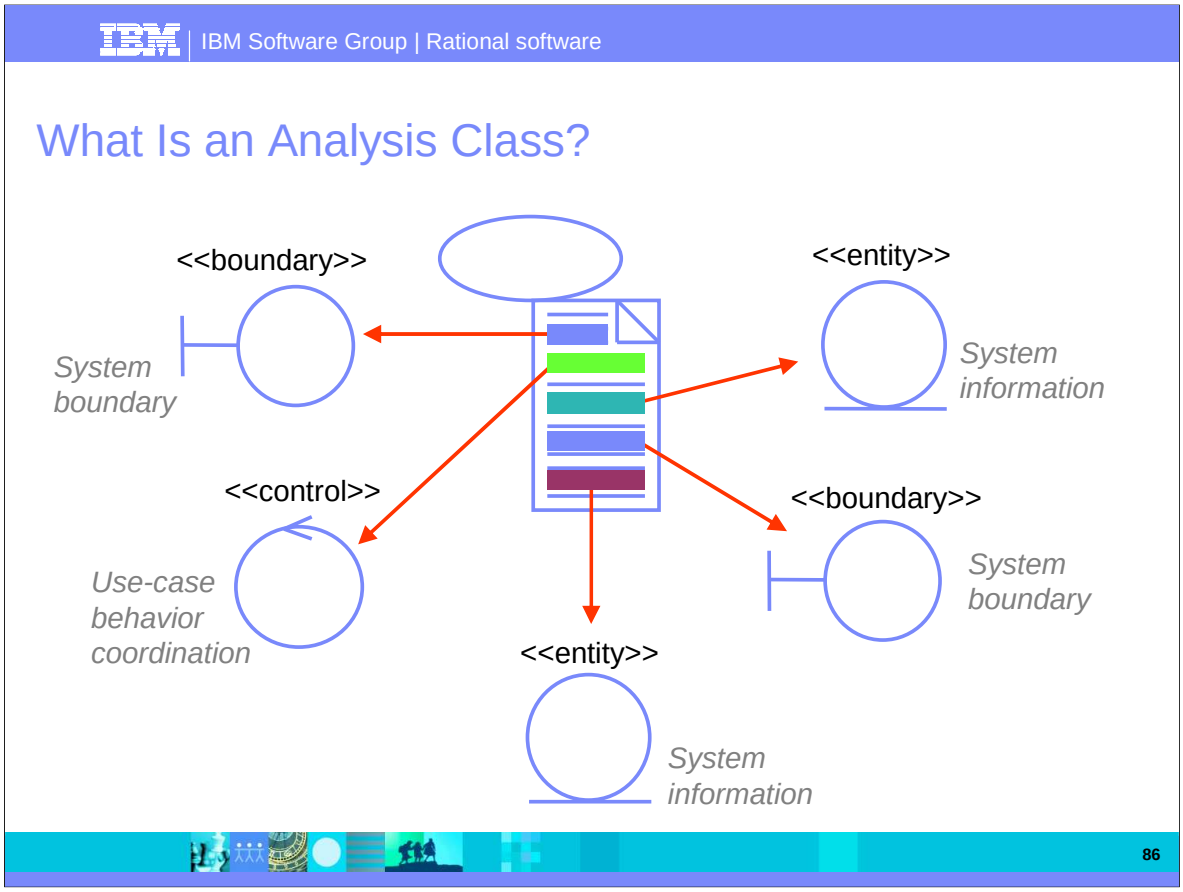
This section and the following cover the following steps in the RUP Use-Case Analysis task:

- Find Analysis Classes from Use-Case Behavior
- Distribute Behavior to Analysis Classes
- Describe Responsibilities
- Describe Attributes and Associations

Find Classes from Use-Case Behavior

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





Analysis classes represent an early conceptual model for “things in the system that have responsibilities and behavior”.

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

Three aspects of the system are likely to change independently from each other:

- The boundary between the system and its actors
- The information the system uses
- The control logic of the system

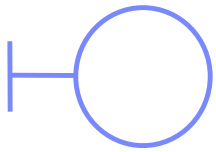
In an effort to isolate the parts of the system that will change more frequently, the following types of analysis classes are identified:

- Boundary
- Entity
- Control

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

What Is a Boundary Class?

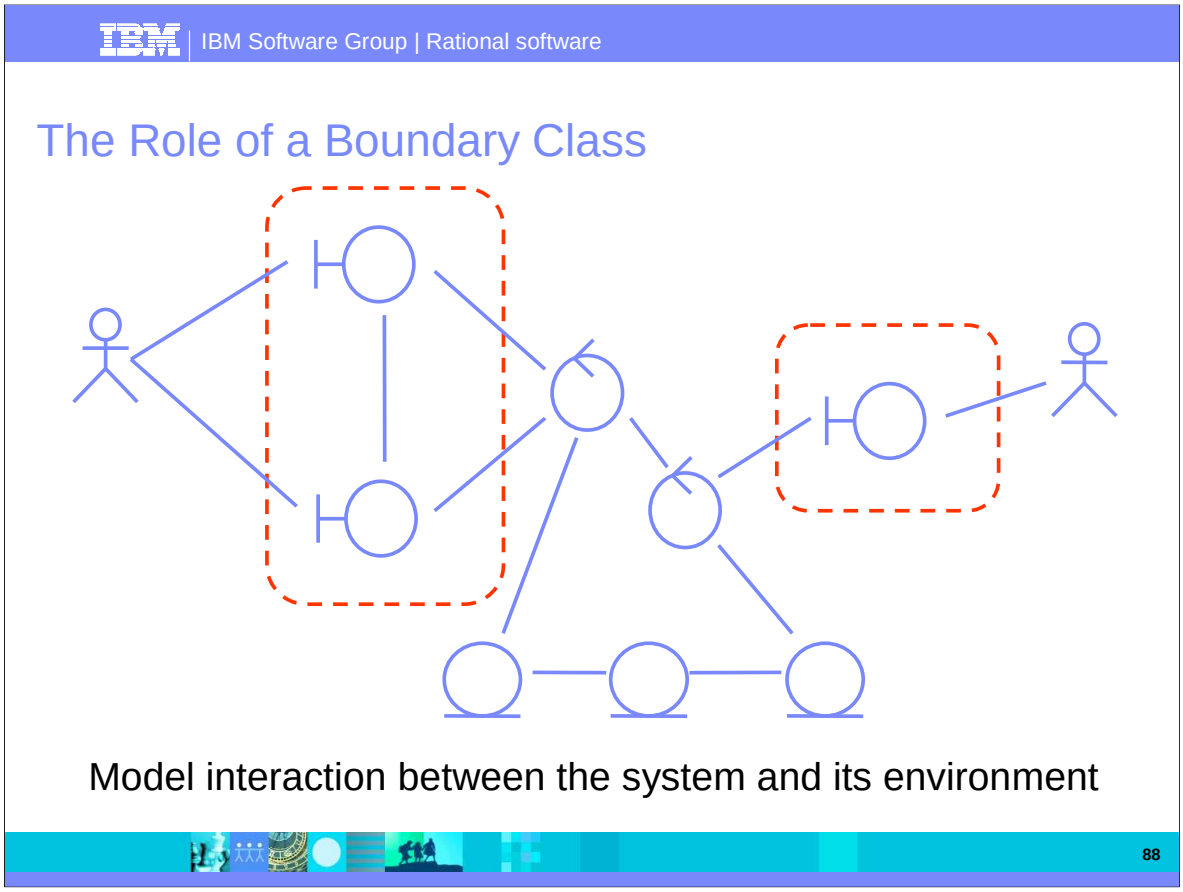
- Intermediates between the interface and something outside the system
- Several Types
  - User interface classes
  - System interface classes
  - Device interface classes
- Environment dependent
  - GUI
  - Communication protocols



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

A system can have several types of boundary classes:

- **User interface classes**: Classes that intermediate communication with human users of the system.
- **System interface classes**: Classes that intermediate communication with other systems. A boundary class that communicates with an external system is responsible for managing the dialog with the external system; it provides the interface to that system for the system being built.
- **Device interface classes**: Classes that provide the interface to devices which detect external events. These boundary classes capture the responsibilities of the device or sensor.



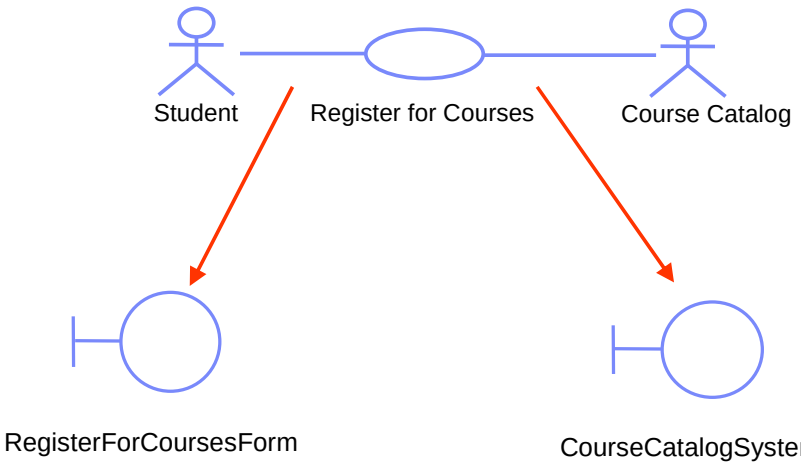
A boundary class is used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as the interface).

Because boundary classes are used between actors and the working of the internal system (actors can only communicate with boundary classes), they insulate external forces from internal mechanisms and vice versa. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes.

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.

Finding Boundary Classes

- One boundary class per actor/use case pair



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 box.

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 courses to be added to his or her Schedule.
- The CourseCatalogSystem interfaces with the legacy system that provides the unabridged catalog of all courses offered by the university.

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!

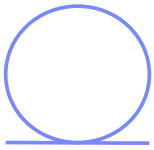
When identifying and describing analysis classes, be careful not to spend too much time on the details. Analysis classes are meant to be a first cut at the abstractions of the system. They help to clarify the understanding of the problem to be solved and represent an attempt at an idealized solution.

User Interface Classes: Boundary classes may be used as “holding places” for GUI classes. The objective is not to do GUI design in this 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. Sketches or screen captures from a user-interface prototype may have been used during the Requirements discipline 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.

What Is an Entity Class?

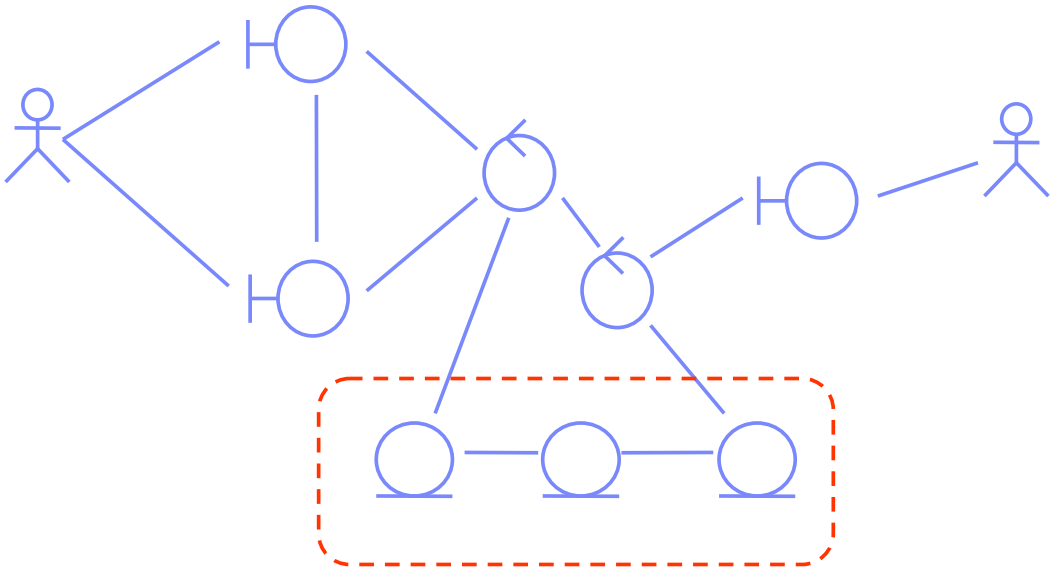
- Represents the key concepts of the system
- Models information that must be stored
  - Usually persistent
- Environment independent
- Not specific to one use case



Entity classes represent stores of information in the system. They are typically used to represent the key concepts that the system manages. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the lifetime of the system.

An entity object is usually not specific to one Use-Case Realization and sometimes it 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).

The Role of an Entity Class



Store and manage information in the system

Finding Entity Classes

- Key abstractions usually become entity classes
- Entity classes can also be found in:
  - Use-case flow of events (developed during requirements)
  - Glossary (developed during requirements)
  - Business-Domain Model (if business modeling has been performed)
- Look for system information that must be stored:
  - Nouns or nominal sentences that identify persistent data are candidates to become:
    - Attributes of an entity class, or
    - Entity classes on their own

Example: Course Registration System

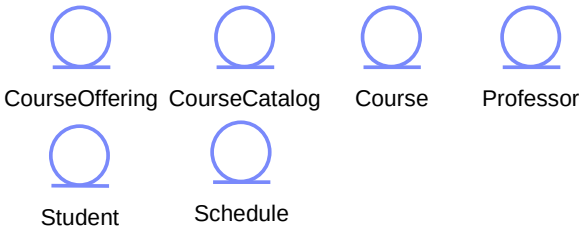
- Basic flow of events for the Submit Grades use case:

This use case starts when a Professor wishes to submit student grades for one or more classes completed in the previous semester:

1. The system displays a list of course offerings the Professor taught in the previous semester.
2. The Professor selects a course offering.
3. The system retrieves a list of all students who were registered for the course offering. The system displays each student and any grade that was previously assigned for the offering.
4. For each student on the list, the Professor enters a grade: A, B, C, D, F, or I. The system records the student's grade for the course offering. If the Professor wishes to skip a particular student, the grade information can be left blank and filled in at a later time. The Professor may also change the grade for a student by entering a new grade.

Example: Course Registration System

- Key abstractions (previously identified)

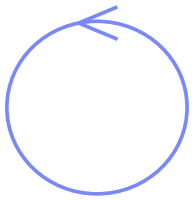


- Newly identified classes

- How would you characterize the new class?

What Is a Control Class?

- Use-case behavior coordinator
  - Complex use cases may need more than one control class
- Delegates responsibility to other classes
- Use-case dependent but environment independent

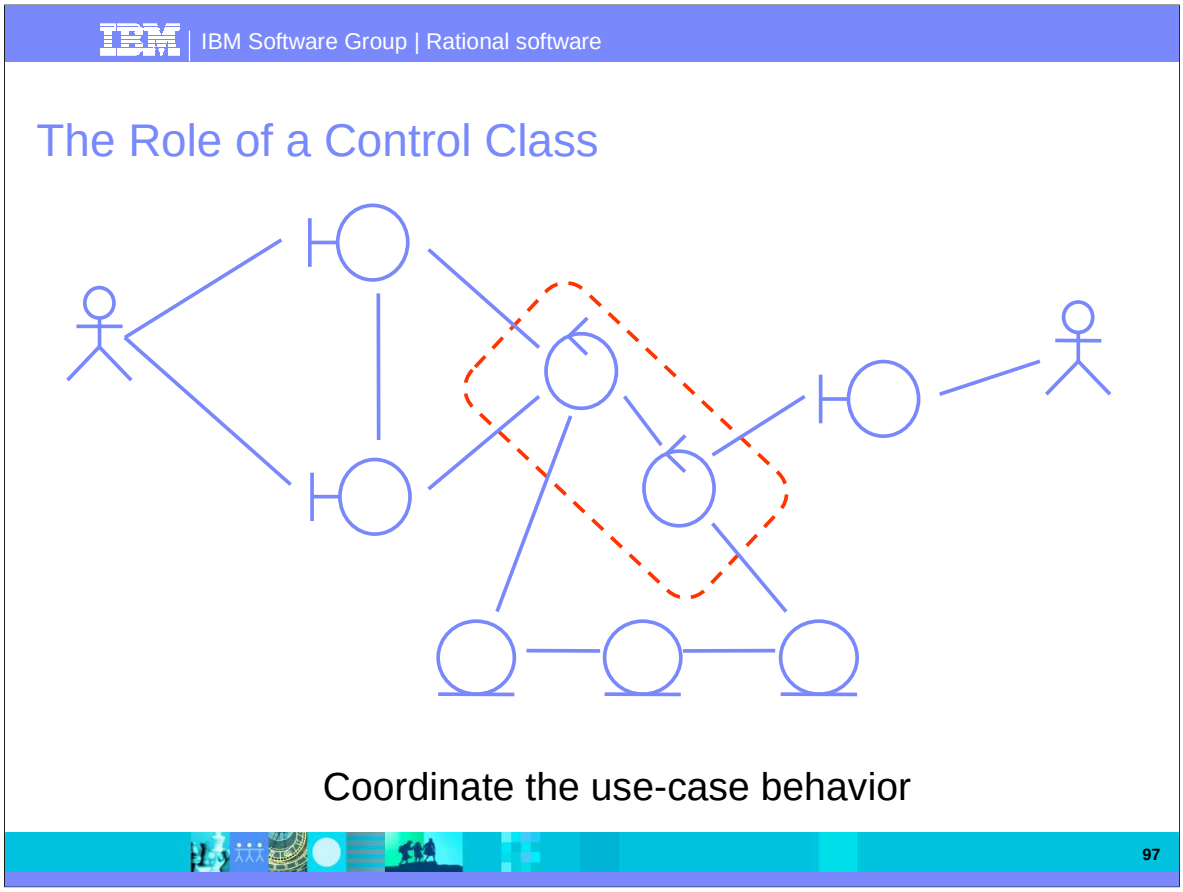


Control classes provide coordinating behavior in the system. The system can perform some use cases without control classes by using just entity and boundary classes. This is particularly true for use cases that involve only the simple manipulation of stored information. More complex use cases generally 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).



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.

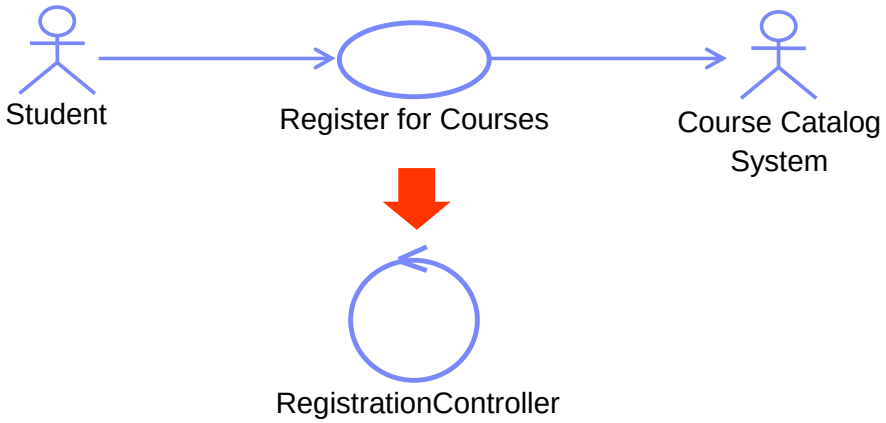
The behavior of a control object is closely related to the realization of a specific use case. In many scenarios, you might even say that the control objects "run" the Use-Case Realizations. However, some control objects can participate in more than one Use-Case Realization if the use-case tasks are 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.

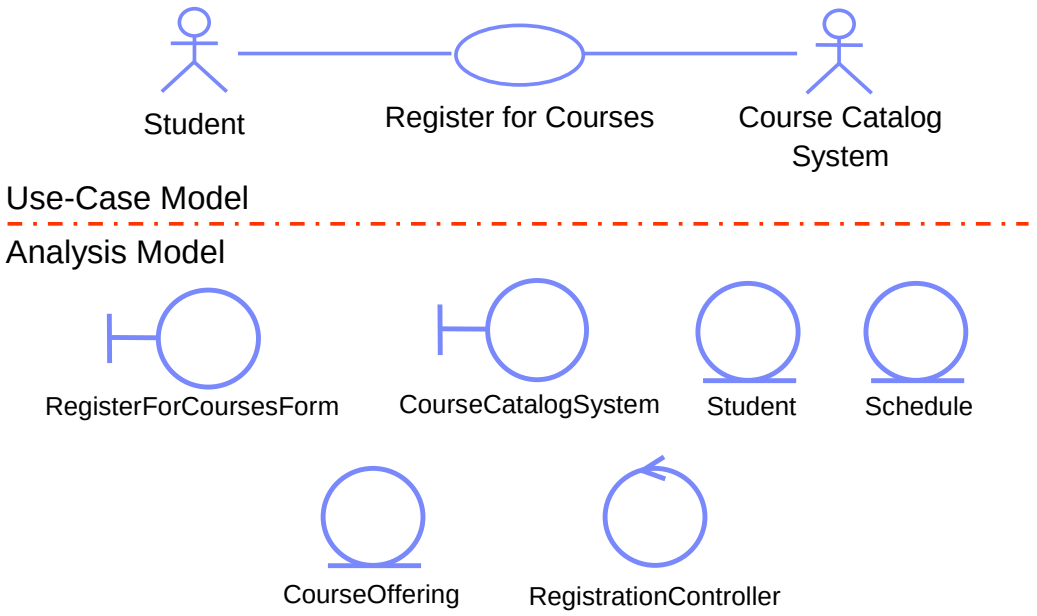
When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

Finding Control Classes

- In general, identify one control class per use case
  - As analysis continues, a complex use case's control class may evolve into more than one class

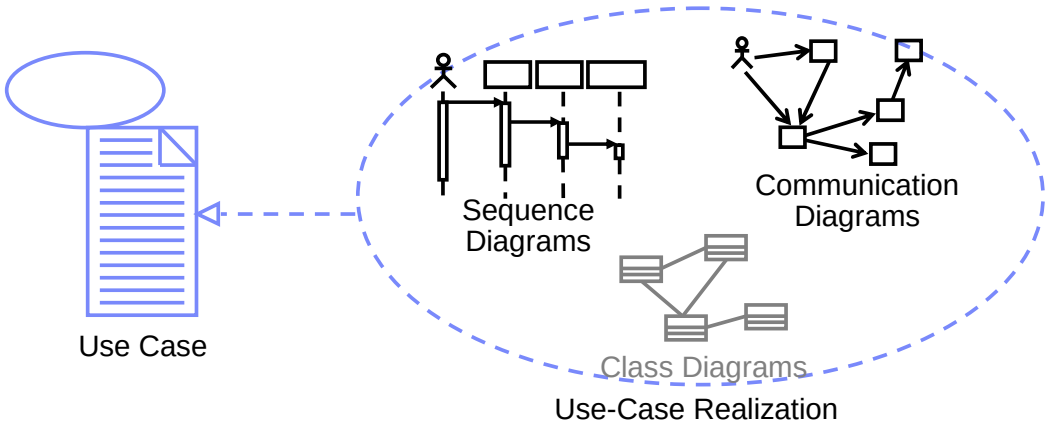


Summary: Course Registration System Example



Distribute Use-Case Behavior

- For each use-case flow of events:
  - Create one or more interaction diagrams (sequence diagrams recommended)
  - Identify analysis objects responsible for the required use-case behavior
  - Allocate use-case responsibilities to analysis classes



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 is 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 that 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 that you are developing (for example, 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 are described on the next slide.

Guidelines: Interaction Diagrams and Use Cases

- Each initial interaction diagram describes one use-case scenario
  - Diagrams should be named after the use-case scenarios
  - The interaction should begin with an actor, since an actor always invokes the use case
- One diagram is not enough
  - At least one diagram for the main flow of events
  - Plus at least one diagram for each non-trivial alternative or exceptional flow
  - Separate diagrams for complex flows

Guidelines: Creating Objects and Classes

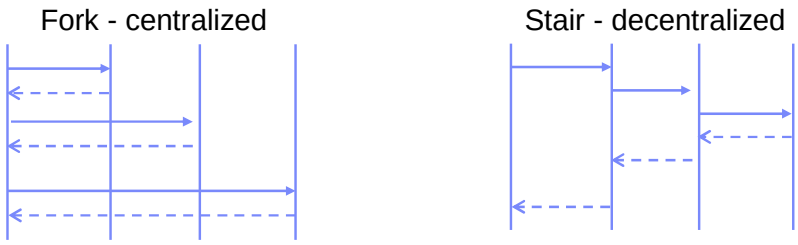
- Before you start analyzing the use case, put in place:
  - The Actor object that initiates the use case (as previously indicated)
  - The corresponding boundary and control objects
  - Other objects that may exist before the use case starts
    - Example: if there is a pre-condition for a student to be logged in, it is likely the system has already retrieved the corresponding Student object
- Assign each object to an existing class or to a new class
  - When creating new classes, capture their semantics immediately
    - Analysis stereotype
    - Description, attributes, relationships
  - Note: Ultimately, classes will be organized into packages and layers but this is NOT the focus of Use-Case Analysis

Guidelines: Allocating Responsibilities

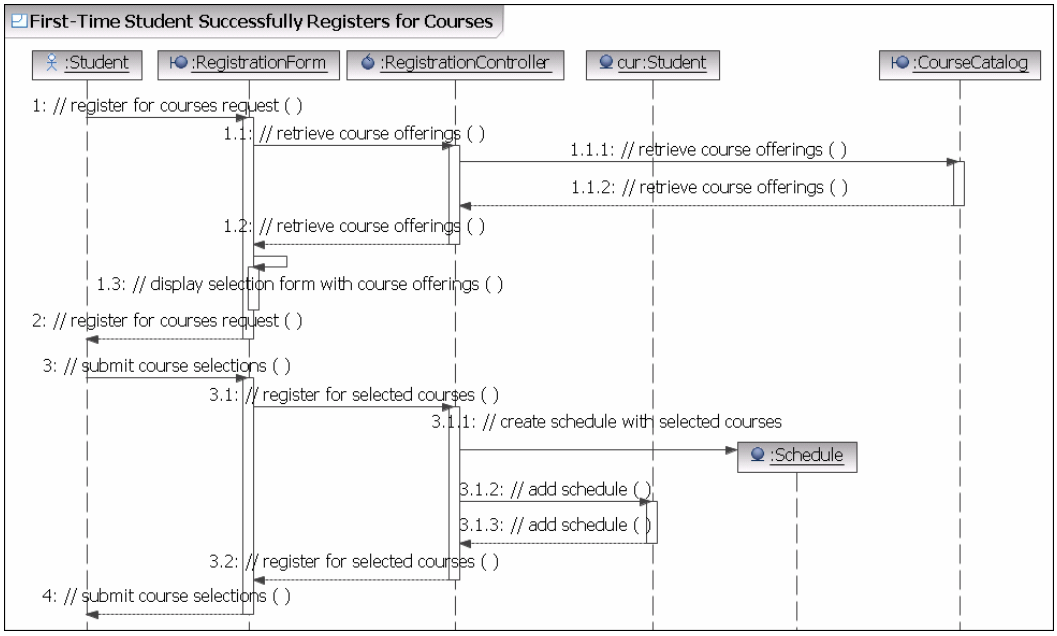
- Use-Case behavior is materialized by objects exchanging messages
  - When creating a message, create the corresponding class operation
    - Convention: start the operation name with "//"
      - It identifies the class operation as an analysis responsibility
      - Example: *// retrieve course offerings for the current semester*
      - During design, responsibilities will be refined into "real" operations
  - Who has the data needed to perform the responsibility?
    - If one class has the data, assign the responsibility to that class
    - If multiple classes have the data, you can
      - Put the responsibility with one class and add a relationship to the other
      - Put the responsibility on a "third party" (new class or existing control class for instance) and add relationships to classes needed to perform the responsibility

Guidelines: Centralized Vs. Decentralized Control

- Centralized control
  - An object controls the others
  - Interaction between the other objects is minimal or non-existent
- Decentralized control
  - No central object
  - Each object only knows a few of the other objects
  - Looks more “OO” but consider for instance the impact if the ordering of operations changes
- The two strategies are often combiner

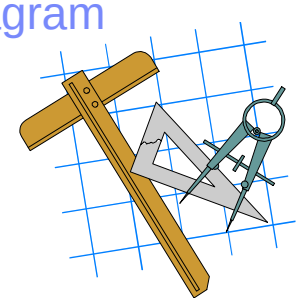


Example: Course Registration System



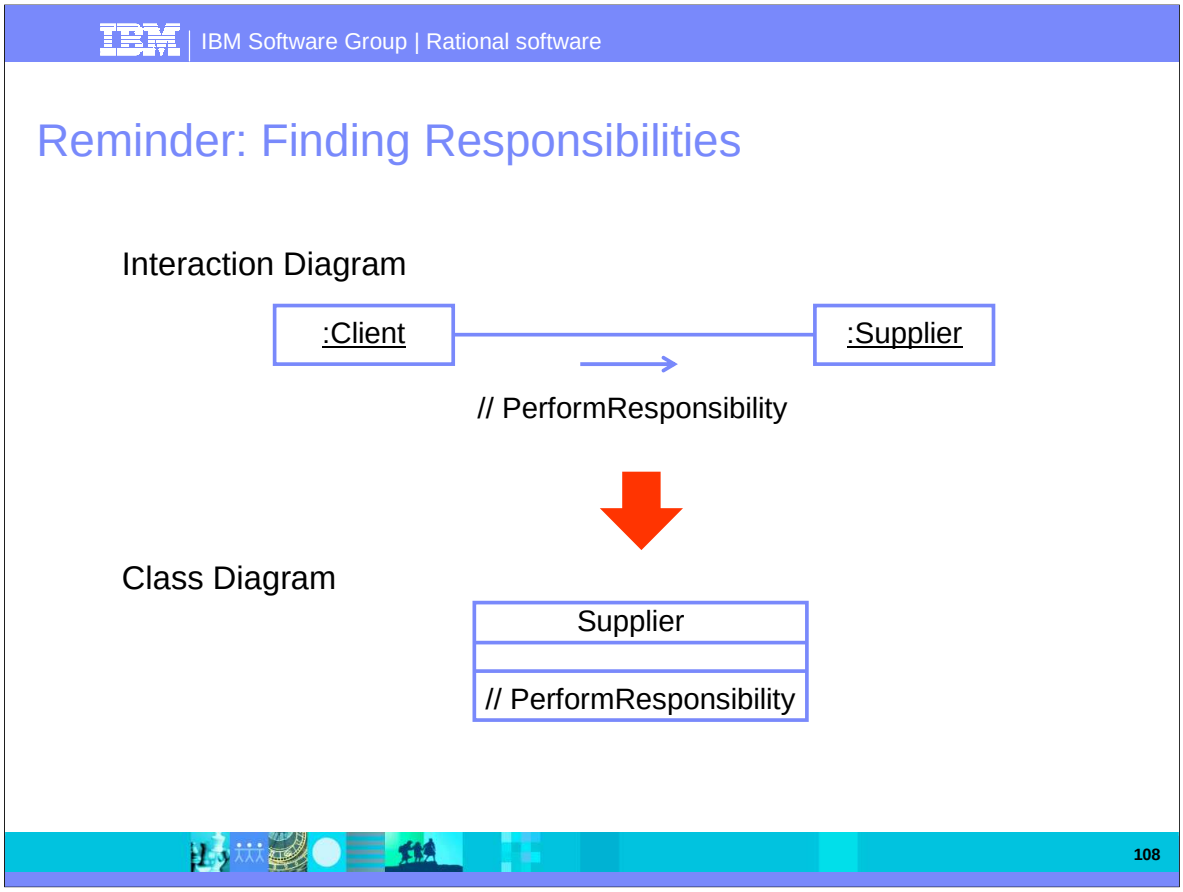
Exercise 1: Create A Sequence Diagram

- Perform the exercise provided by the instructor (lab 5 – tasks 5.1 and 5.2)



Where Are We?

- Create Analysis Use-Case Realization
- Supplement the Use-Case Description
- Model Use-Case Scenarios with Interaction Diagrams
- Model Participating Classes in Class Diagrams
- Reconcile the Analysis Use-Case Realizations
- Qualify Analysis Mechanisms

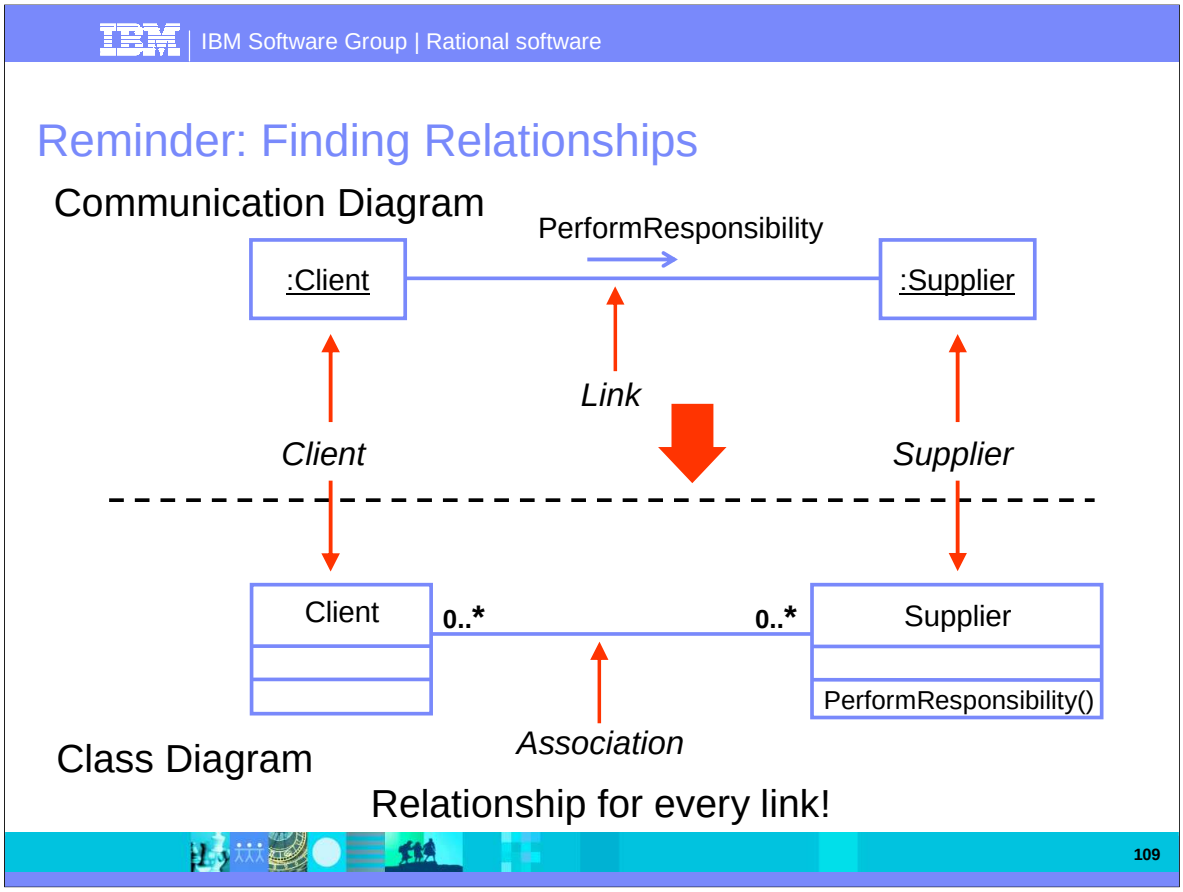


A responsibility is a statement of something an object can be asked to provide. Responsibilities evolve into one (or more) operations on classes in design; they can be characterized as:

- 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.

We have chosen to document analysis class responsibilities as “analysis” operations by adopting the naming convention to precede the “analysis” operation name by “//”. This naming convention indicates that the operation is being used to describe the responsibilities of the analysis class and that they WILL PROBABLY change/evolve in design.



A link between two objects (explicit in the Communication diagrams, implicit in the Sequence diagrams) indicate that there must be some form of relationship between the corresponding classes. This relationship can be an association, an aggregation, a dependency, etc. A link is an instance of a relationship.

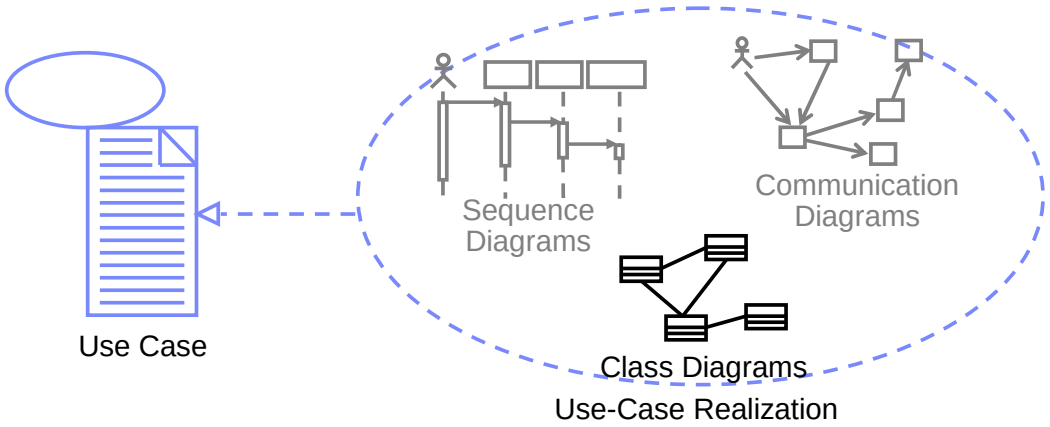
Reflexive links do not need to be instances of reflexive relationships; an object can send messages to itself. A reflexive relationship is needed when two different objects of the same class need to communicate.

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.

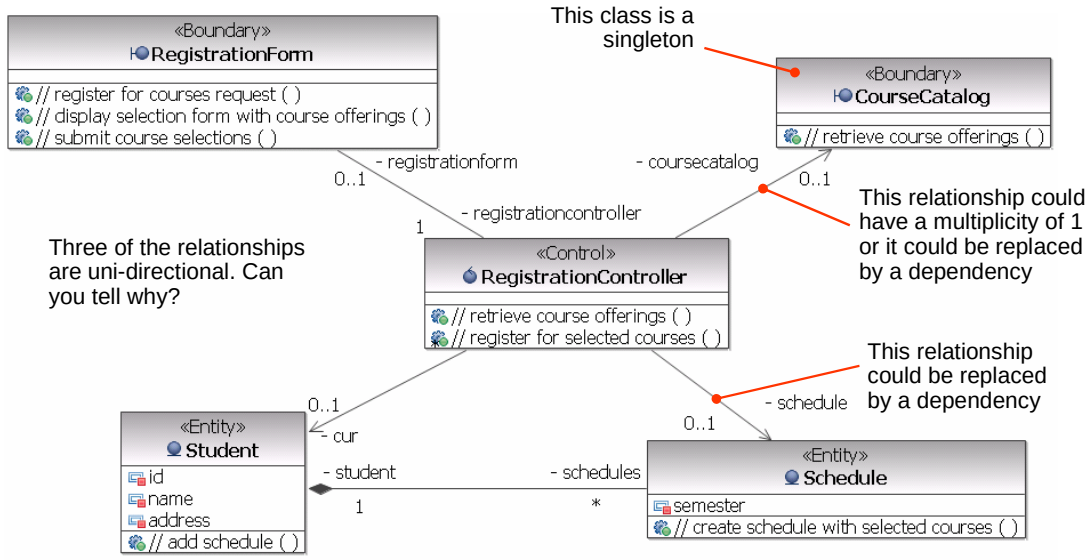
Remember to give the associations role names and multiplicities. You can also specify navigability, although this will be refined in Class Design.

Creating a VOPC

- 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

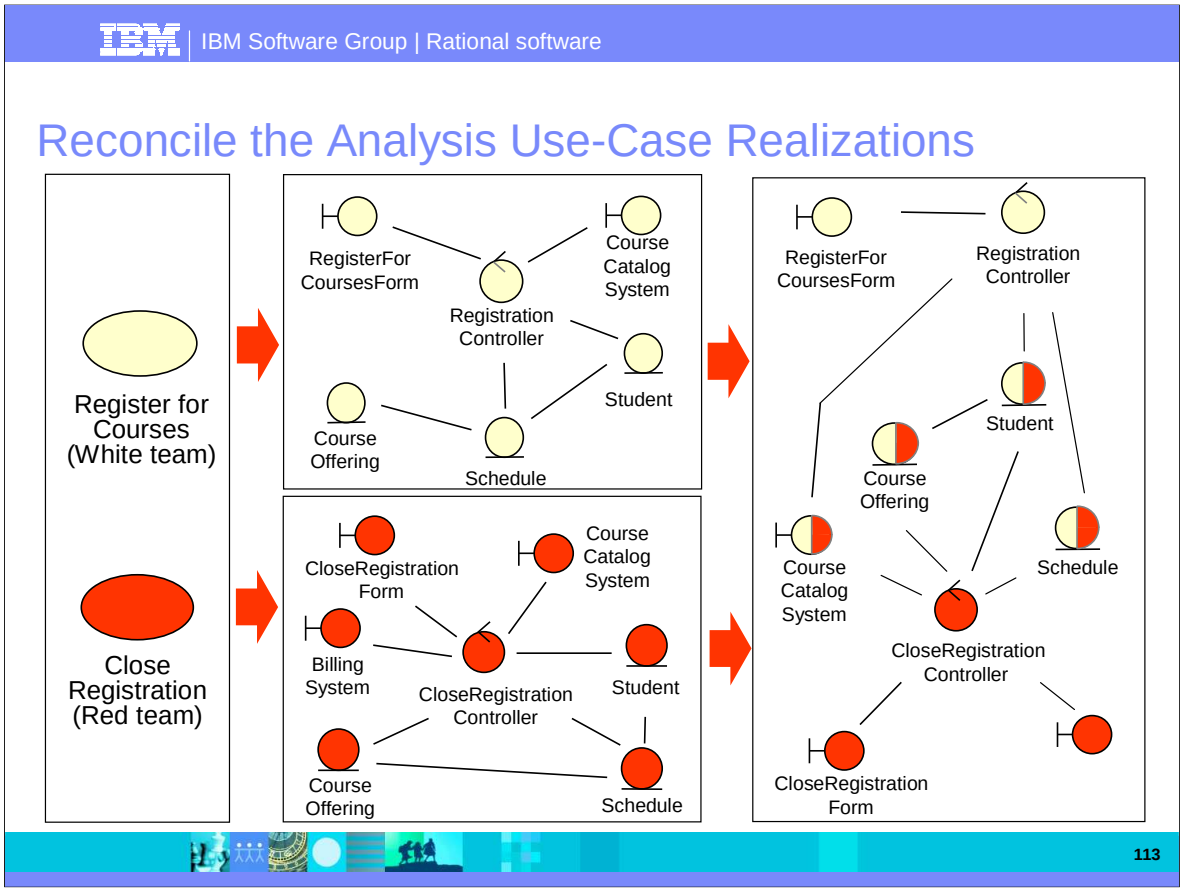


Example: Course Registration System



Where Are We?

- Create Analysis Use-Case Realization
- Supplement the Use-Case Description
- Model Use-Case Scenarios with Interaction Diagrams
- Model Participating Classes in Class Diagrams
- ⇨ Reconcile the Analysis Use-Case Realizations
- Qualify Analysis Mechanisms



Different use cases will contribute to the same classes. In the example above, the classes CourseCatalogSystem, CourseOffering, Schedule and Student participate in both the Register for Courses and Close Registration use cases.

A class can participate in any number of use cases. It is therefore important to examine each class for consistency across the whole system.

Merge classes that define similar behaviors or that represent the same phenomenon.

Merge entity classes that define the same attributes, even if their defined behavior is different; aggregate the behaviors of the merged classes.

Where Are We?

- Create Analysis Use-Case Realization
- Supplement the Use-Case Description
- Model Use-Case Scenarios with Interaction Diagrams
- Model Participating Classes in Class Diagrams
- Reconcile the Analysis Use-Case Realizations
- ⇨ Qualify Analysis Mechanisms

Describing Analysis Mechanisms

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

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.

Example: Course Registration System

- 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

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 (that is, 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 later during Design. For now, just take it as an architectural given that all control classes are distributed for the OOAD course example and exercise.

Exercise 2: Create A VOPC Diagram

- Perform the exercise provided by the instructor (lab 5 – tasks 5.3 and 5.4)