► ► ► **Module 5**
Architectural Analysis

**Mastering Object-Oriented Analysis
and Design with UML**
Module 5: Architectural Analysis

Topics

Objectives: Architectural Analysis

Objectives: Architectural Analysis

- Explain the purpose of Architectural Analysis and where it is performed in the lifecycle.
- Describe a representative architectural pattern and set of analysis mechanisms, and how they affect the architecture.
- Describe the rationale and considerations that support the architectural decisions.
- Show how to read and interpret the results of Architectural Analysis:
  - Architectural layers and their relationships
  - Key abstractions
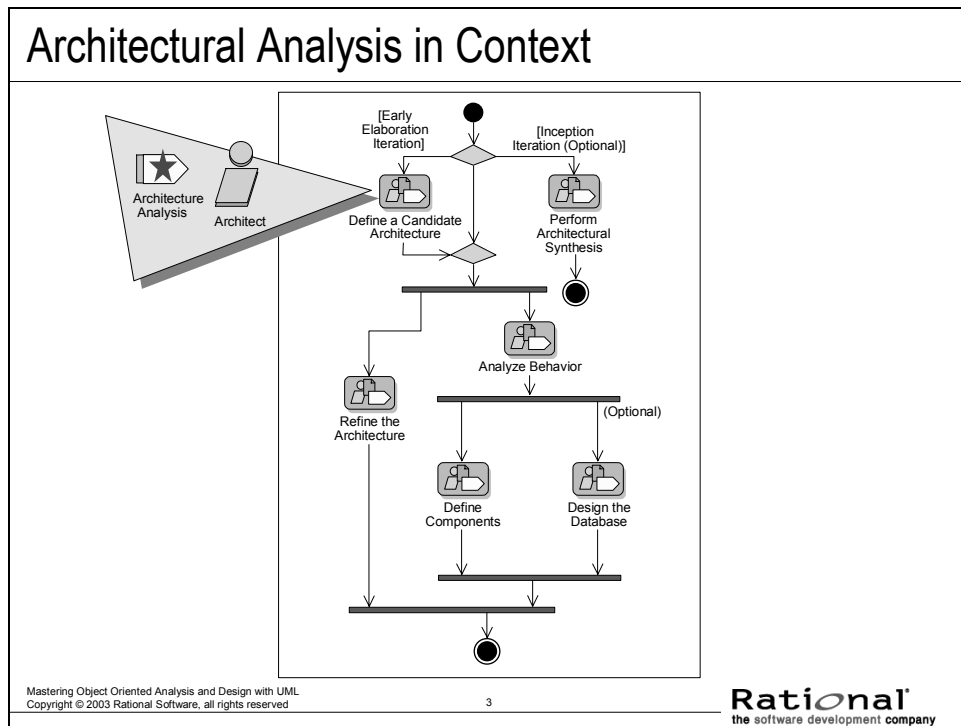 - Analysis mechanisms

2

Rational
the software development company

A focus on software architecture allows you to articulate the structure of the software system (the packages/components), and the ways in which they integrate (the fundamental mechanisms and patterns by which they interact).

Architectural Analysis is where we make an initial attempt at defining the pieces/parts of the system and their relationships, organizing these pieces/parts into well-defined layers with explicit dependencies, concentrating on the upper layers of the system. This will be refined, and the lower layers will be defined during Incorporate Existing Design Elements.

Architectural Analysis in Context

Architectural Analysis in Context

Architecture Analysis

Architect

[Early Elaboration Iteration]

[Inception Iteration (Optional)]

Define a Candidate Architecture

Perform Architectural Synthesis

Analyze Behavior

Refine the Architecture

(Optional)

Define Components

Design the Database

3

As you may recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process. **Architectural Analysis** is an activity in the Define a Candidate Architecture workflow detail.

Architectural Analysis is how the project team (or the architect) decides to define the project's high-level architecture. It is focused mostly on bounding the analysis effort in terms of agreed-upon architectural patterns and idioms, so that the "analysis" work is not working so much from "first principles." **Architectural Analysis** is very much a configuring of Use-Case Analysis.
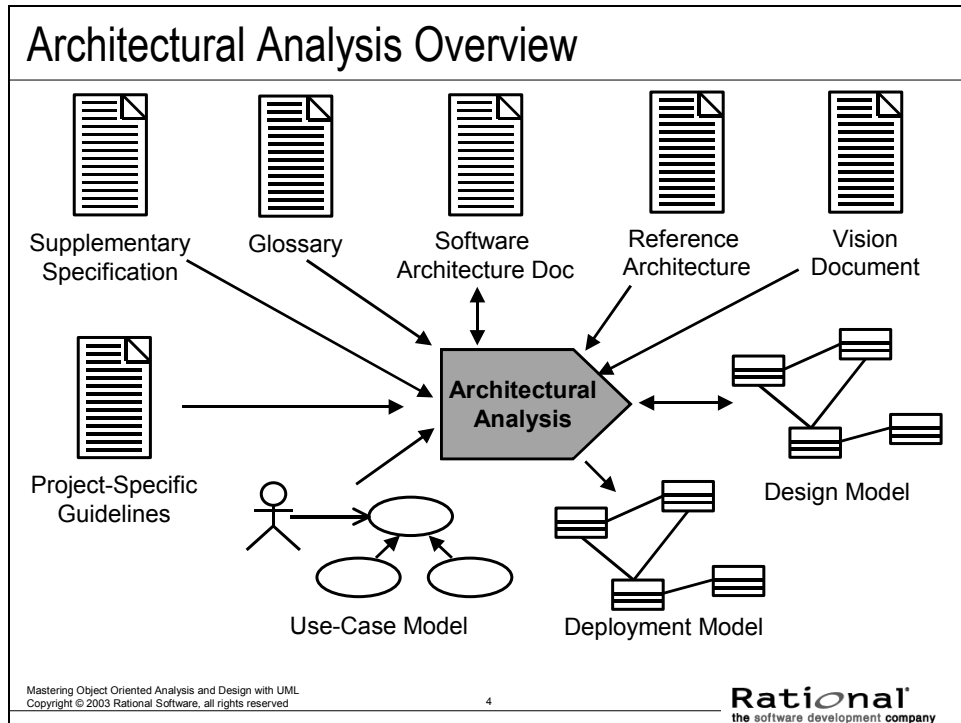
During **Architectural Analysis,** we concentrate on the upper layers of the system, making an initial attempt at defining the pieces/parts of the system and their relationships and organizing these pieces/parts into well-defined layers with explicit dependencies.

In Use-Case Analysis, we will expand on this architecture by identifying analysis classes from the requirements. Then, in Incorporate Existing Design Elements, the initial architecture is refined, and the lower architecture layers are defined, taking into account the implementation environment and any other implementation constraints.

Architectural Analysis is usually done once per project, early in the Elaboration phase. The activity is performed by the software architect or architecture team.

This activity can be skipped if the architectural risk is low.

Architectural Analysis Overview

Architectural Analysis Overview

Supplementary Specification · Glossary · Software Architecture Doc · Reference Architecture · Vision Document

Project-Specific Guidelines

Architectural Analysis

Use-Case Model

Deployment Model

Design Model

Purpose:
- To define a candidate architecture for the system based on experience gained from similar systems or in similar problem domains.
- To define the architectural patterns, key mechanisms, and modeling conventions for the system.
- To define the reuse strategy.
- To provide input to the planning process.

Input Artifacts:
- Use-Case Model
- Supplementary Specifications
- Glossary
- Design Model
- Reference Architecture
- Vision Document
- Project Specific Guidelines
- Software Architecture Document

Resulting Artifacts:
- Software Architecture Document
- Design Model
- Deployment Model

Key Concepts

Architectural Analysis Steps

☆ ◆ Key Concepts
  - ◆ Define the High-Level Organization of Subsystems
  - ◆ Identify Analysis mechanisms
  - ◆ Identify Key Abstractions
 - ◆ Create Use-Case Realizations
  - ◆ Checkpoints

5

Rational
the software development company

The above are the topics we will be discussing within the **Architectural Analysis** module. Unlike the designer activity modules, you will not be discussing each step of the activity, as the objective of this module is to understand the important **Architectural Analysis** concepts, not to learn HOW to create an architecture.

Before you discuss **Architectural Analysis** in any detail, it is important to review/define some key concepts.

Review: What Is Architecture: The "4+1 View" Model
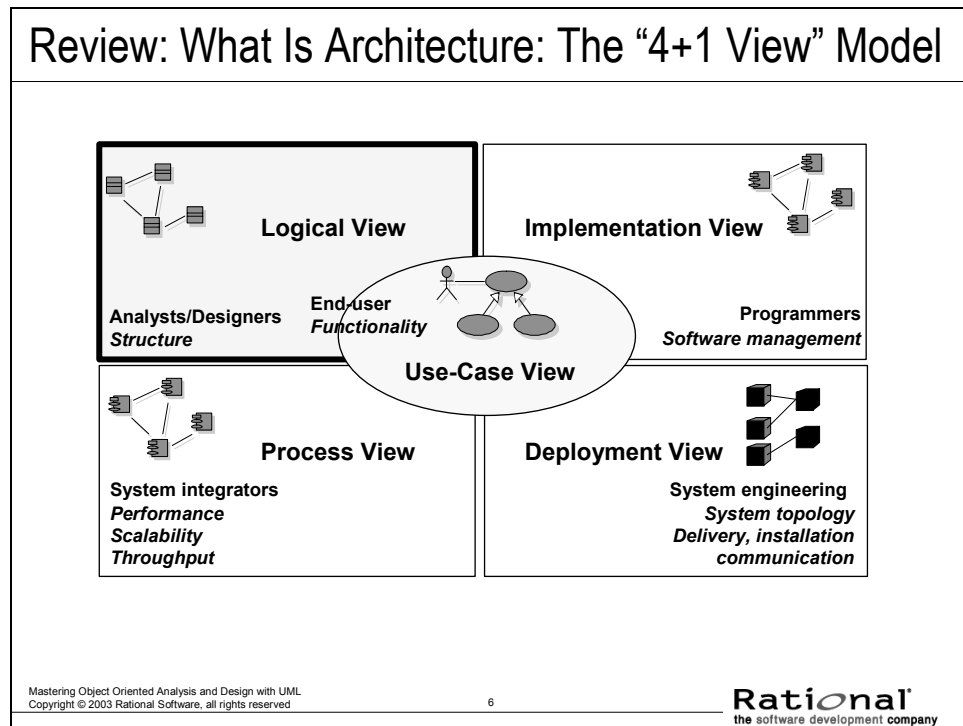
The above diagram describes the model Rational uses to describe the software architecture. This is the recommended way to represent a software architecture. There may be other "precursor" architectures that are not in this format. The goal is to mature those architectural representations into the 4+1 view representation.

In **Architectural Analysis,** you will concentrate on the Logical View. The other views will be addressed in later architecture modules:

- The Logical View will be refined in the Identify Design Mechanisms, Identify Design modules.

- The Process View will be discussed in the Describe Run-time Architecture module.

- The Deployment View will be discussed in the Describe Distribution module.

- The Implementation View is developed during Implementation and is thus considered out of scope for this Analysis and Design course.

Review: What Is a Package?

Review: What Is a Package?

- ◆ A package is a general-purpose mechanism for organizing elements into groups.
- ◆ It is a model element that can contain other model elements.

University
Artifacts
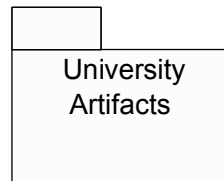
- ◆ A package can be used
  - ▪ To organize the model under development.
  - ▪ As a unit of configuration management.

7

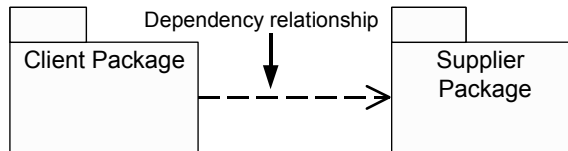Rati○nal®
the software development company

Packages were first introduced in the Introduction to Object Orientation module. The slide is repeated here for review purposes.

Packages can be used to group any model elements. However, in this module, we will be concentrating on how they are used within the Design Model.

Package Relationships: Dependency

Package Relationships: Dependency

- ◆ Packages can be related to one another using a dependency relationship.

```
                    Dependency relationship
┌──────┐                                    ┌──────┐
│ Client Package │         │                │ Supplier │
│              │          ▼                 │ Package  │
│              │  - - - - - - - - >│        │          │
└──────────────┘                            └──────────┘
```

- ◆ Dependency Implications
  - • Changes to the Supplier package may affect the Client package.
  - • The Client package cannot be reused independently because it depends on the Supplier package.

Rati⊘**nal**
the software development company

Elements in one package can import elements from another package. In the UML, this is represented as a dependency relationship.
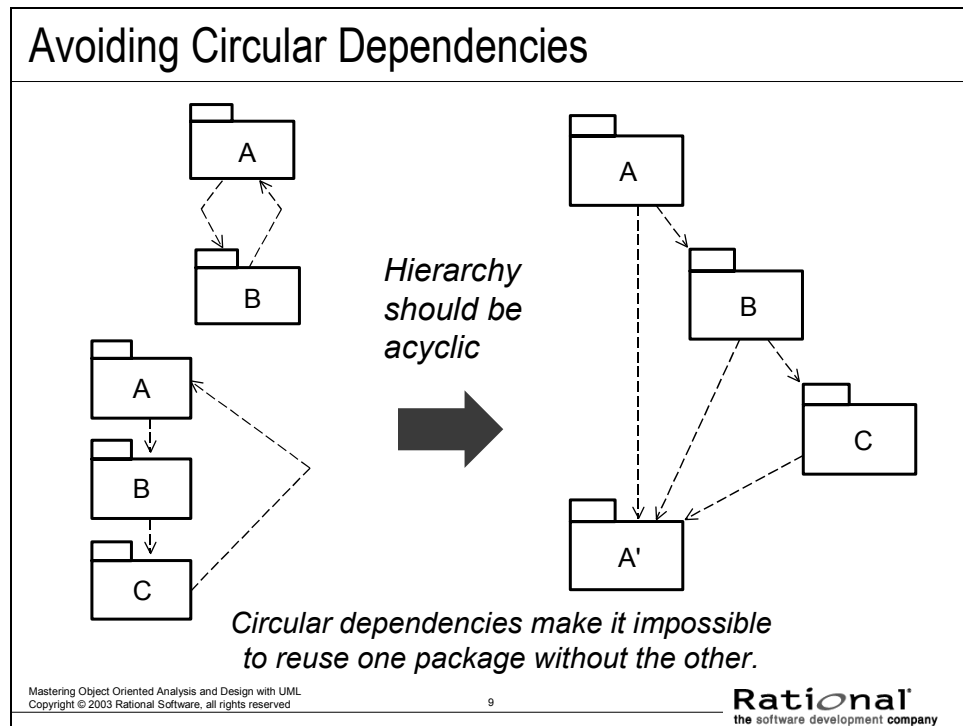
The relationships of the packages reflect the allowable relationships between the contained classes. A dependency relationship between packages indicates that the contents of the supplier packages may be referenced by the client. In the above example, if a dependency relationship exists between the Client package and the Supplier package, then classes in the Client package may access classes in the Supplier package.

Some of the implications of package dependency are:

- Whenever a change is made to the Supplier package, the Client package must, potentially, be recompiled and re-tested.

- The Client package cannot be reused independently because it depends on the Supplier package.

The grouping of classes into logical sets and the modeling of their relationships can occur anywhere in the process when a set of cohesive classes is identified.

Avoiding Circular Dependencies

Avoiding Circular Dependencies

Hierarchy should be acyclic

Circular dependencies make it impossible to reuse one package without the other.

9

Rational®
the software development company

It is desirable that the package hierarchy be acyclic. This means that the following situation should be avoided (if possible):

• Package A uses package B, which uses package A.

If a circular dependency exists between Package A and Package B, if you change Package A it might cause a change in Package B, which might cause a change in Package A, etc. A circular dependency between packages A and B means that they will effectively have to be treated as a single package.

Circles wider than two packages must also be avoided. For example, package A uses package B, which uses package C, which uses package A.

Circular dependencies may be able to be broken by splitting one of the packages into two smaller ones. In the above example, the elements in package A that were needed by the other packages were factored out into their own package, A', and the appropriate dependencies were added.

Define the High-Level Organization of Subsystems

Architectural Analysis Steps

- ◆ Key Concepts
- ☆ ◆ Define the High-Level Organization of Subsystems
- ◆ Identify Analysis mechanisms
- ◆ Identify Key Abstractions
- ◆ Create Use-Case Realizations
- ◆ Checkpoints
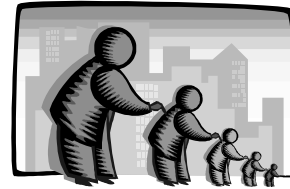
10

Rational
the software development company

Early in the software development lifecycle, it is important to define the modeling conventions that everyone on the project should use. The modeling conventions ensure that the representation of the architecture and design are consistent across teams and iterations.

Patterns and Frameworks

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

11

Rational
the software development company

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

A **pattern** codifies specific knowledge collected from experience. Patterns provide examples of how good modeling solves real problems, whether you come up with the pattern yourself or you reuse someone else's. Design patterns are discussed in more detail on the next slide.

Frameworks differ from Analysis and Design patterns in their scale and scope. Frameworks describe a skeletal solution to a particular problem that may lack many of the details, and that may be filled in by applying various Analysis and Design patterns.
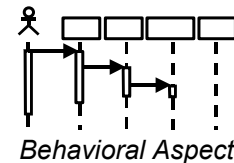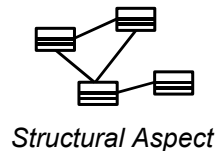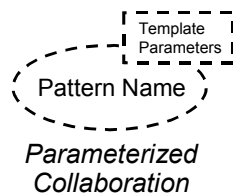
A framework is a micro-architecture that provides an incomplete template for applications within a specific domain. Architectural frameworks provide the context in which the components run. They provide the infrastructure (plumbing, if you will) that allows the components to co-exist and perform in predictable ways. These frameworks may provide communication mechanisms, distribution mechanisms, error processing capabilities, transaction support, and so forth.

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

What Is a Design Pattern?

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

Template Parameters

Pattern Name

Parameterized Collaboration　　*Structural Aspect*　　*Behavioral Aspect*

12

Rational®
the software development **company**

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

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

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

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

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

What Is an Architectural Pattern?

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

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

Model-View-Controller: The MVC pattern is where an application is divided into three partitions: The Model, which is the business rules and underlying data, the View, which is how information is displayed to the user, and the Controllers, which process the user input.
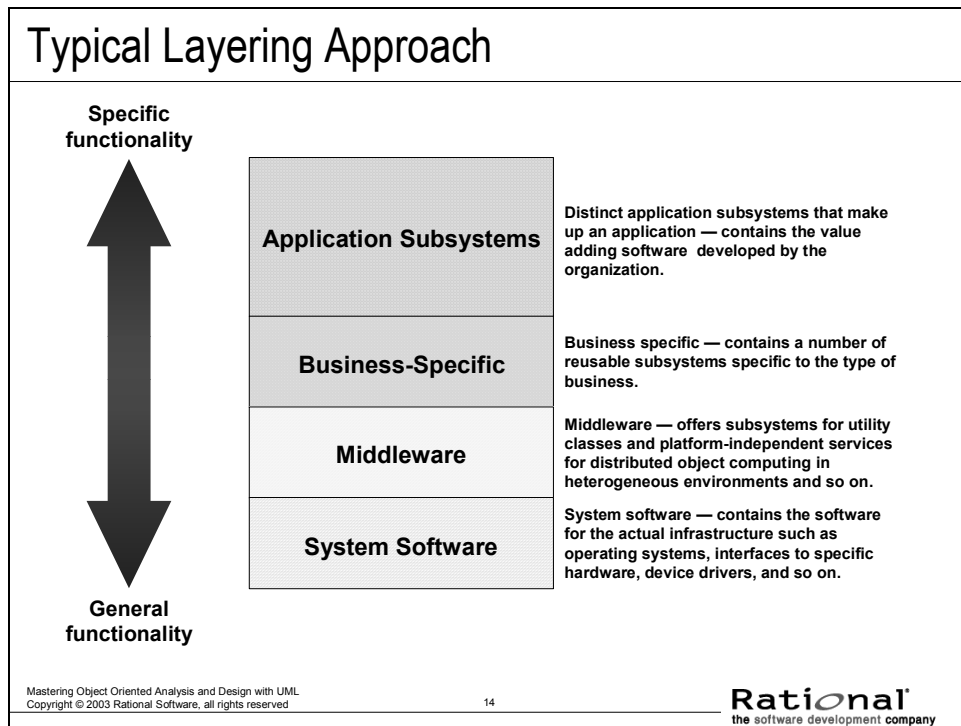
Pipes and Filters: In the Pipes and Filters pattern, data is processed in streams that flow through pipes from filter to filter. Each filter is a processing step.

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

Architectural patterns can work together. (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.

Typical Layering Approach

Typical Layering Approach

Specific functionality

Application Subsystems	Distinct application subsystems that make up an application — contains the value adding software developed by the organization.
Business-Specific	Business specific — contains a number of reusable subsystems specific to the type of business.
Middleware	Middleware — offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.
System Software	System software — contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers, and so on.

General functionality

Rati○nal
the software development company

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

The number and composition of layers is dependent upon the complexity of both the problem domain and the solution space:

- There is generally only a single application-specific layer.

- In domains with existing systems, or that have large systems composed of inter-operating smaller systems, the Business-Specific layer is likely to partially exist and may be structured into several layers for clarity.

- Solution spaces, which are well-supported by middleware products and in which complex system software plays a greater role, have well-developed lower layers, with perhaps several layers of middleware and system software.

This slide shows a sample architecture with four layers:

- The top layer, **Application layer**, contains the application-specific services.

- The next layer, **Business-Specific layer**, contains business-specific components used in several applications.

- The **Middleware layer** contains components such as GUI-builders, interfaces to database management systems, platform-independent operating system services, and OLE-components such as spreadsheets and diagram editors.

- The bottom layer, **System Software layer**, contains components such as operating systems, databases, interfaces to specific hardware, and so on.

Architectural Pattern: Layers
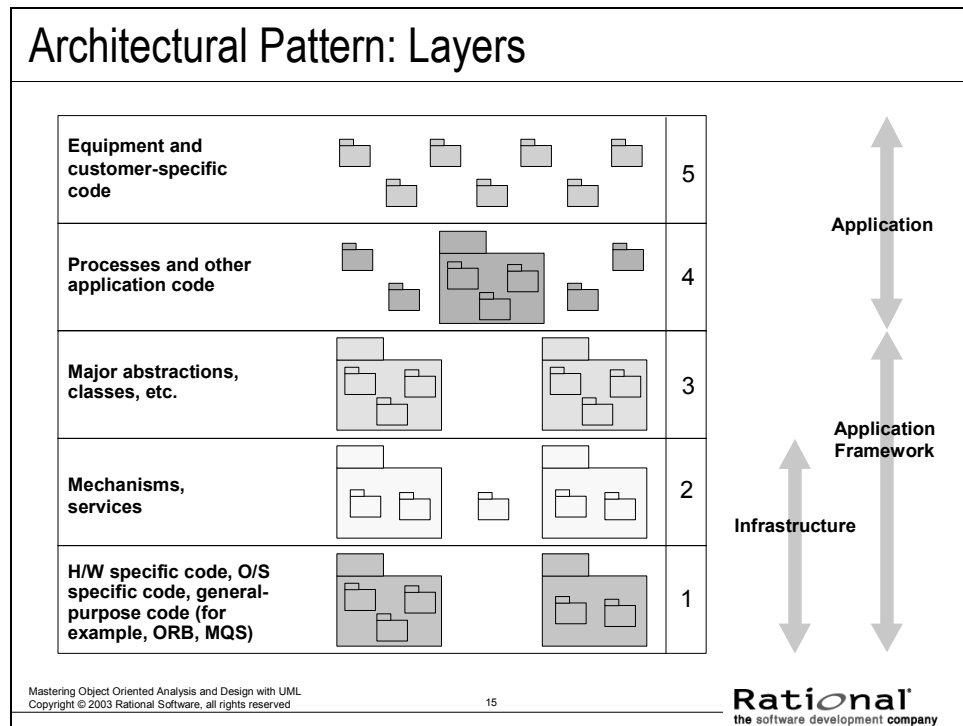
Context

A large system that requires decomposition.

Problem

A system that 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. (Do not skip layers unless
intermediate layers would only add pass-through components.)

A strict layered architecture states that design elements (classes, components,
packages, and subsystems) only utilize the services of the layer below them. Services
can include event-handling, error-handling, database access, and so forth. It contains
more palpable mechanisms, as opposed to the raw operating system level calls
documented in the bottom layer.

Layering Considerations

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

16

Rational
the software development company

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 Applicationlayer. 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.

In **Architectural Analysis**, we are concentrating on the upper-level layers (the Application and Business-Specific layers). The lower level layers (infrastructure and vendor-specific layers) will be defined in Incorporate Existing Design Elements.

Modeling Architectural Layers

Modeling Architectural Layers

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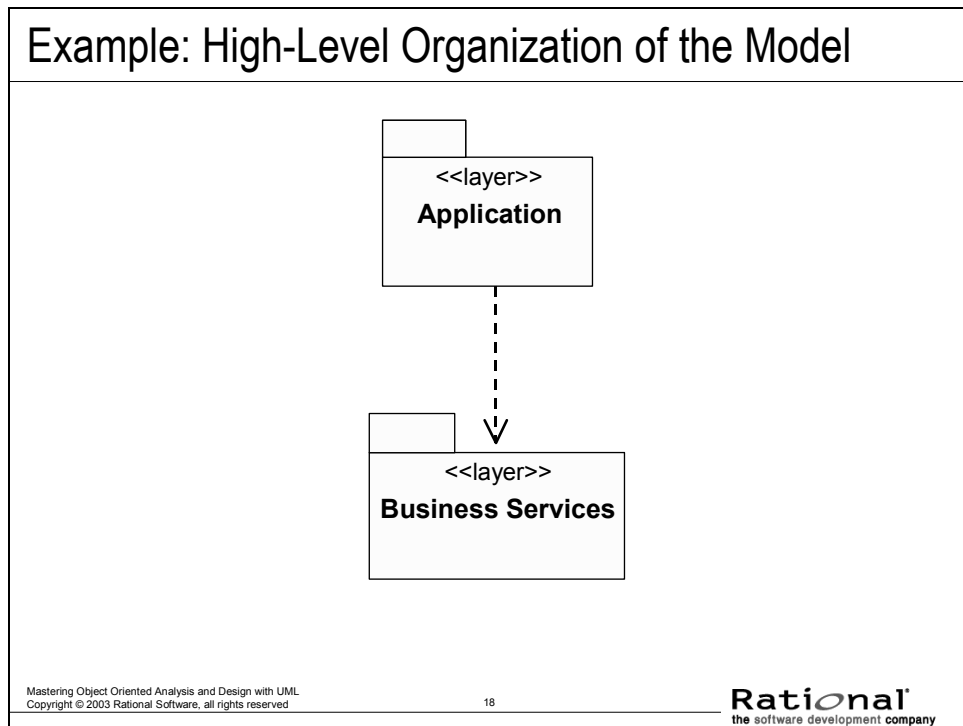- ◆ <<layer>> stereotype
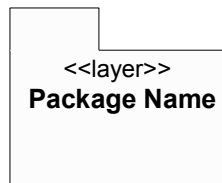
<<layer>>
Package Name

17

Rati**o**nal
the software development **company**

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

Example: High-Level Organization of the Model

Example: High-Level Organization of the Model

The above example includes the Application and Business-Specific layers for the Course Registration System.

The Application layer contains the design elements that are specific to the Course Registration application.

We expect that multiple applications will share some key abstractions and common services. These have been encapsulated in the Business Services layer, which is accessible to the Application layer. The Business Services layer contains business-specific elements that are used in several applications, not necessarily just this one.

Identify Analysis Mechanisms

Architectural Analysis Steps

- ◆ Key Concepts
- ◆ Define the High-Level Organization of Subsystems
- ☆ ◆ Identify Analysis mechanisms
- ◆ Identify Key Abstractions
- ◆ Create Use-Case Realizations
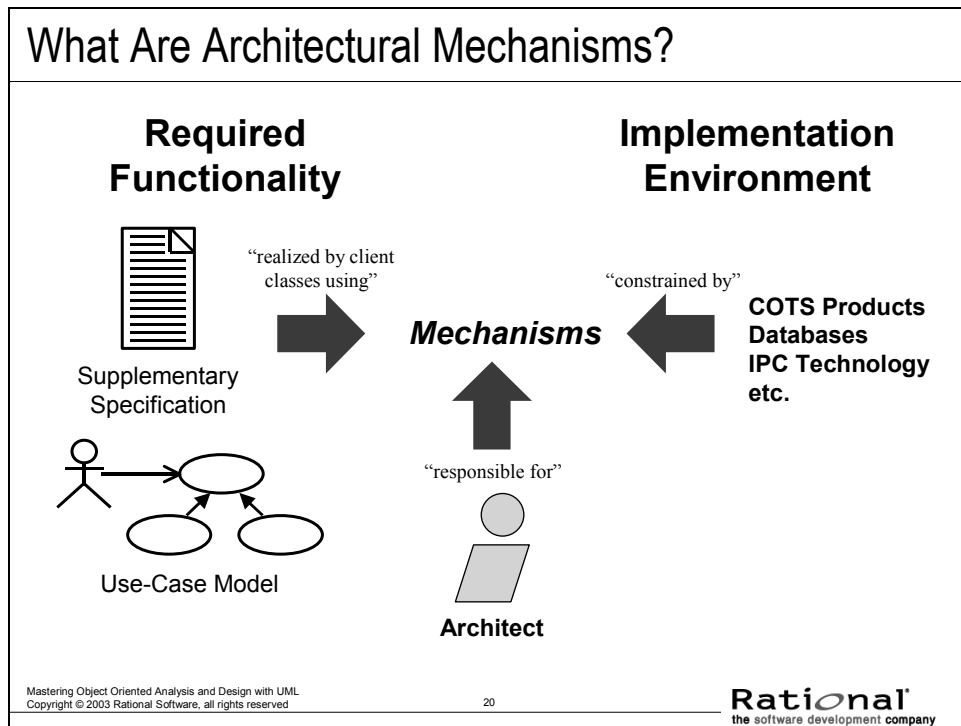- ◆ Checkpoints

19

Rational
the **software** development **company**

The architecture should be simple, but not simplistic. It should provide standard behavior through standard abstractions and mechanisms. Thus, a key aspect in designing a software architecture is the definition and the selection of the mechanisms that designers use to give "life" to their objects.

In **Architectural Analysis**, it is important to identify the analysis mechanisms for the software system being developed. Analysis mechanisms focus on and address the nonfunctional requirements of the system (that is, the need for persistence, reliability, and performance), and builds support for such non-functional requirements directly into the architecture.

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. Mechanisms allow the analysis effort to focus on translating the functional requirements into software abstractions without becoming bogged down in the specification of relatively complex behavior that is needed to support the functionality but which is not central to it.

What Are Architectural Mechanisms?

What Are Architectural Mechanisms?

Required Functionality

Implementation Environment

Supplementary Specification

"realized by client classes using"

Mechanisms

"constrained by"

COTS Products Databases IPC Technology etc.

Use-Case Model

"responsible for"

Architect

20

Rati⌀nal
the software development company

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

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

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

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

Architectural Mechanisms: Three Categories

There are three categories of architectural mechanisms. The only difference between them is one of refinement.

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

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

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

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

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

Why Use Analysis Mechanisms?

An analysis mechanism represents a pattern that constitutes a common solution to a common problem. These mechanisms may show patterns of structure, patterns of behavior, or both. They are used during analysis to reduce the complexity of the 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.

Analysis mechanisms are typically, but not necessarily, unrelated to the problem domain, but instead are "computer science" concepts. As a result, they typically occupy the middle and lower layers of the architecture. They provide specific behaviors to a domain-related class or component, or correspond to the implementation of cooperation between classes and/or components.

Sample Analysis Mechanisms

Sample Analysis Mechanisms

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

Rati☺nal
the software development company

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

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

Examples of communication mechanisms include inter-process communication (IPC) and inter-node communication (a.k.a. remote process communication or RPC). RPC has both a communication and a distribution aspect.

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

Examples of Analysis Mechanism Characteristics

Examples of Analysis Mechanism Characteristics

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

Rational
the software development company

Analysis mechanism characteristics capture some nonfunctional requirements of the system.

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

- **Granularity**: Range of size of the persistent objects
- **Volume**: Number of objects to keep persistent
- **Duration**: How long to keep persistent objects
- **Access mechanism**: How is a given object uniquely identified and retrieved?
- **Access frequency**: Are the objects more or less constant; are they permanently updated?
- **Reliability**: Shall the objects survive a crash of the process, the processor; the whole system?

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

- **Latency**: How fast must processes communicate with another?
- **Synchronicity**: Asynchronous communication
- **Size of message**: A spectrum might be more appropriate than a single number.
- **Protocol**, flow control, buffering, and so on.

Example of Analysis Mechanism Characteristics (cont.)

Example of Analysis Mechanism Characteristics (cont.)

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

25

Rational
the software development company

Security:
- **Data granularity**: Package-level, class-level, attribute level
- **User granularity**: Single users, roles/groups
- **Security Rules**: Based on value of data, on algorithm based on data, and on algorithm based on user and data
- **Privilege Types**: Read, write, create, delete, perform some other operation
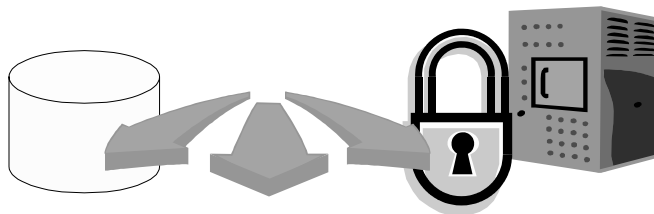
Describing Analysis Mechanisms
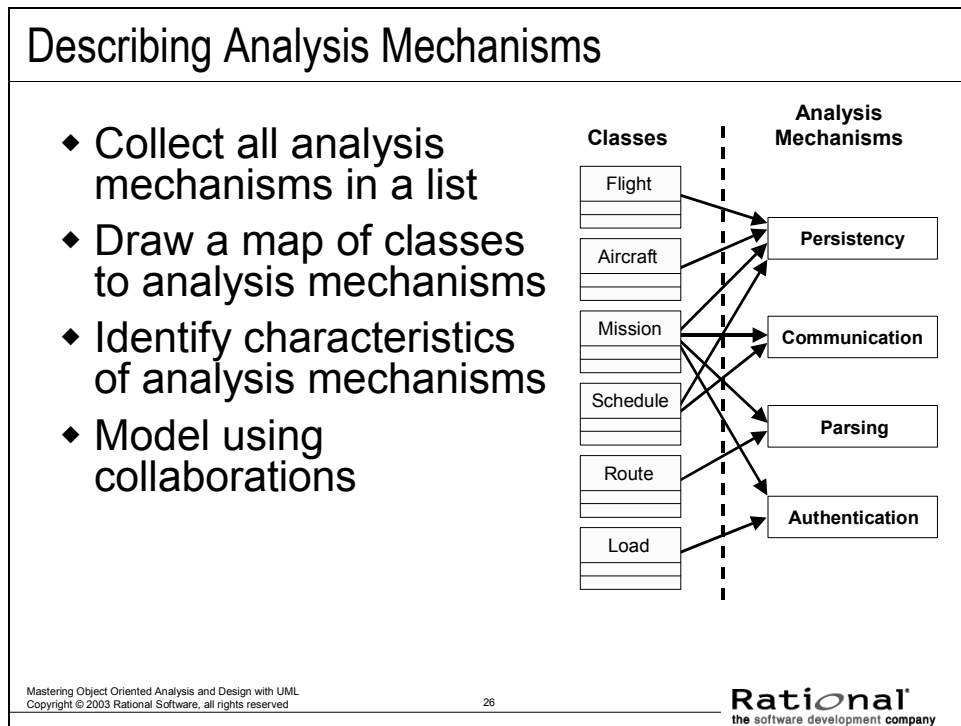
The process for describing analysis mechanisms is:

1. **Collect all analysis mechanisms in a list.** The same analysis mechanism may appear under several different names across different use-case realizations, or across different designers. For example, **storage**, **persistency**, **database**, and **repository** might all refer to a persistency mechanism. **Inter-process communication**, **message passing**, or **remote invocation** might all refer to an inter-process communication mechanism.

2. **Draw a map of the client classes to the analysis mechanisms** (see graphic on slide).

3. **Identify Characteristics of the analysis mechanisms.** To discriminate across a range of potential designs, identify the key characteristics used to qualify each analysis mechanism. These characteristics are part functionality, part size, and performance.

4. **Model Using Collaborations.** Once all of the analysis mechanisms are identified and named, they should be modeled through the collaboration of a "society of classes." Some these classes do not directly deliver application functionality, but exist only to support it. Very often, these "support classes" are located in the middle or lower layers of a layered architecture, thus providing a common support service to all application-level classes.

Example: Course Registration Analysis Mechanisms

Example: Course Registration Analysis Mechanisms

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

27

Rati⌀nal®
the software development company

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

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

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

Security: A means to control access to an element.

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

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

Identify Key Abstractions

Architectural Analysis Steps

- ◆ Key Concepts
- ◆ Define the High-Level Organization of Subsystems
- ◆ Identify Analysis mechanisms
- ☆ ◆ Identify Key Abstractions
- ◆ Create Use-Case Realizations
- ◆ Checkpoints

Rational
the software development company

This is where the key abstractions for the problem domain are defined. Furthermore, this is where the "vocabulary" of the software system is established.

The purpose of this step is to "prime the pump" for analysis by identifying and defining the key abstractions that the system must handle. These may have been initially identified during business modeling and requirement activities. However, during those activities, the focus was on the problem domain. During analysis and design, our focus is more on the solution domain.

What Are Key Abstractions?

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)

Mastering Object Oriented Analysis and Design with UML
Copyright © 2003 Rational Software, all rights reserved   29
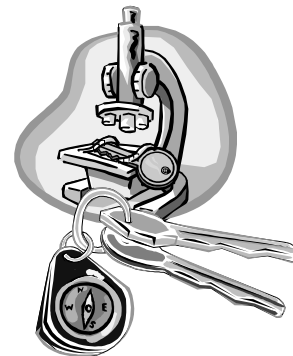
Rational
the software development company

Requirements and Business Modeling activities usually uncover key concepts that the system must be able to handle. These concepts manifest themselves as key design abstractions. Because of the work already done, there is no need to repeat the identification work again during Use-Case Analysis. To take advantage of existing knowledge, we identify preliminary entity analysis classes to represent these key abstractions on the basis of general knowledge of the system. Sources include the **Requirements**, the **Glossary**, and in particular, the **Domain Model**, or the **Business Object Model**, if you have one.

Defining Key Abstractions

Defining Key Abstractions

- ◆ Define analysis class relationships
- ◆ Model analysis classes and relationships on class diagrams
  - ▪ Include brief description of analysis class
- ◆ Map analysis classes to necessary analysis mechanisms
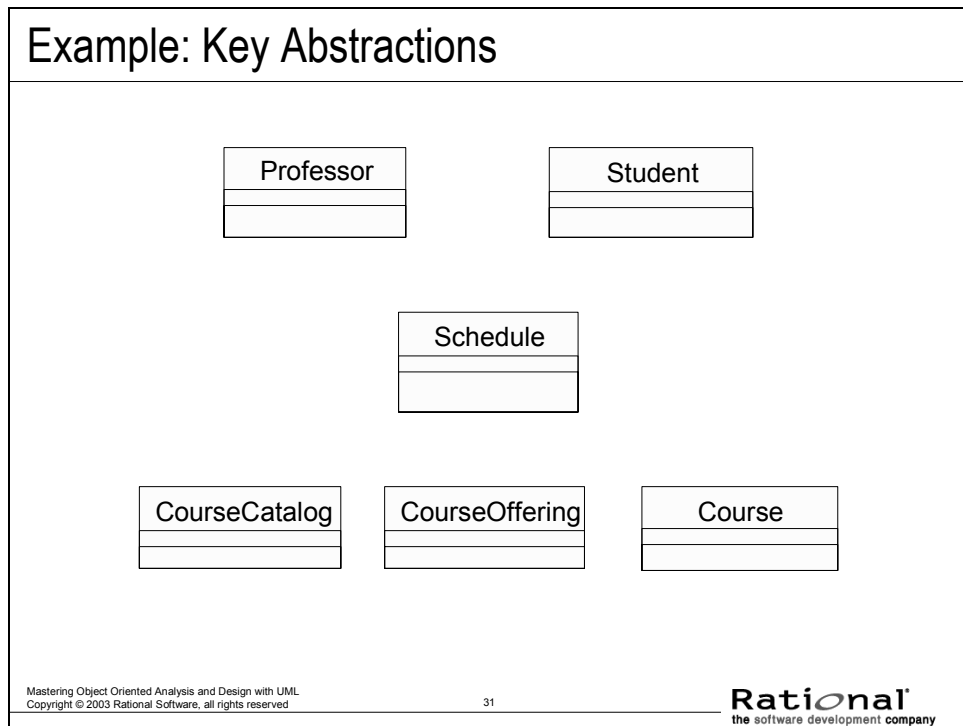
Rati⌀nal
the software development **company**

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.

Example: Key Abstractions

Example: Key Abstractions

Professor		Student	

Schedule

CourseCatalog	CourseOffering	Course

31

Rational
the software development company

Professor: A person teaching classes at the university.

Student: A person enrolled in classes at the university.

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

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

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

Course: A class offered by the university.

Create Use-Case Realizations

Architectural Analysis Steps

- ◆ Key Concepts
- ◆ Define the High-Level Organization of Subsystems
- ◆ Identify Analysis mechanisms
- ◆ Identify Key Abstractions
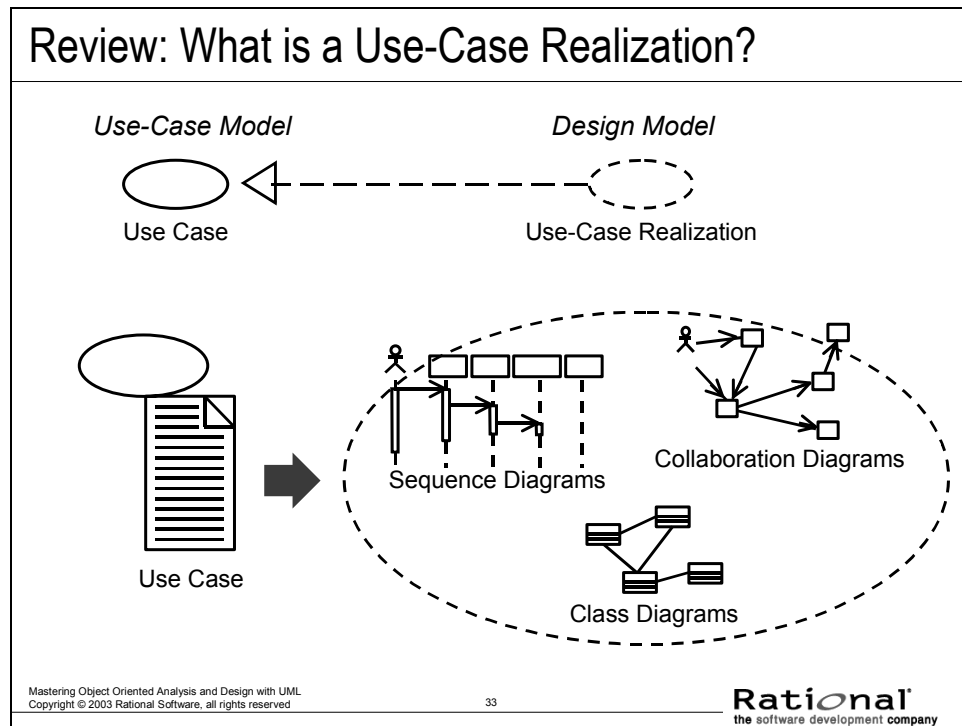- ☆ ◆ Create Use-Case Realizations
- ◆ Checkpoints

Rational
the software development company

A use-case realization represents the design perspective of a use case. It is an organization model element used to group a number of artifacts related to the use-case design. Use cases are separate from use-case realizations, so you can manage each individually and change the design of the use case without affecting the baseline use case. For each use case in the Use-Case Model, there is a use-case realization in the design model with a dependency (stereotyped «realize») to the use case.

Review: What is a Use-Case Realization?

Review: What is a Use-Case Realization?

Use-Case Model　　　　　　*Design Model*

Use Case　　　　　　Use-Case Realization

Sequence Diagrams

Collaboration Diagrams

Use Case

Class Diagrams

33

Rational
the software development company

A use-case realization is the expression of a particular use case within the Design Model. It describes the use case 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 Design Model. A use-case realization specifies what classes must be built to implement each use case.

In the UML, use-case realizations are stereotyped collaborations. The symbol for a collaboration is an oval containing the name of the collaboration. The symbol for a use-case realization is a "dotted line" version of the collaboration symbol.

A use-case realization in the 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.

Within the UML, a use-case realization can be represented using 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 — collaboration 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.

The Value of Use-Case Realizations
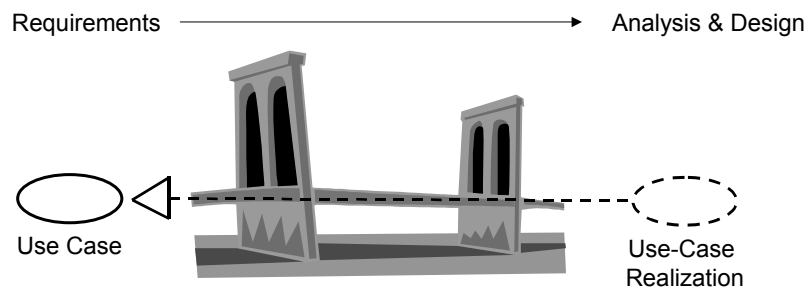
Use cases form the central focus of most of the early analysis and design work. To enable the transition between requirements-centric activities and design-centric activities, the use-case realization serves as a bridge, providing a way to trace behavior in the Design Model back to the Use-Case Model, as well as organizing collaborations in the Design Model around the Use-Case concept.

For each Use Case in the Use Case Model, create a use-case realization in the Design Model. The name for the use-case realization should be the same as the associated use case, and a trace dependency should be established from the use case realization to its associated use case.

Architectural Analysis Steps

Architectural Analysis Steps

- ◆ Key Concepts
- ◆ Define the High-Level Organization of Subsystems
- ◆ Identify Analysis mechanisms
- ◆ Identify Key Abstractions
- ◆ Create Use-Case Realizations
- ☆ ◆ Checkpoints

35

Rati**o**nal®
the software development company

This is where the quality of the architecture modeled up to this point is assessed against some very specific criteria.

In this module, we will concentrate on those checkpoints that the designer is most concerned with (that is, looks for). The architect should do a much more detailed review of the **Architectural Analysis** results and correct any problems before the project moves on to the next activity. We will not cover those checkpoints, as they are out of scope of this course. (Remember, this is not an architecture course.)

Checkpoints

<table>
<tr><td colspan="2">Checkpoints</td></tr>
<tr><td>

◆ **General**

  ▪ Is the package partitioning and layering done in a logically consistent way?

  ▪ Have the necessary analysis mechanisms been identified?

◆ **Packages**

  ▪ Have we provided a comprehensive picture of the services of the packages in upper-level layers?

</td><td></td></tr>
<tr><td colspan="2" align="right">*(continued)*</td></tr>
<tr><td colspan="2">

Mastering Object Oriented Analysis and Design with UML
Copyright © 2003 Rational Software, all rights reserved   36

Rati⌀nal
the software development **company**

</td></tr>
</table>

The next few slides contains the key things a designer would look for when assessing the results of **Architectural Analysis**. An architect would have a more detailed list.

A well-structured architecture encompasses a set of classes, typically organized into multiple hierarchies

Note: At this point, some of the packages/layers may not contain any classes, and that is okay. More classes will be identified over time, starting in the next activity, Use-Case Analysis.

Checkpoints (cont.)

Checkpoints (cont.)

- ◆ Classes
  - ▪ Have the key entity classes and their relationships been identified and accurately modeled?
  - ▪ Does the name of each class clearly reflect the role it plays?
  - ▪ Are the key abstractions/classes and their relationships consistent with the Business Model, Domain Model, Requirements, Glossary, etc.?

37

Rational
the software development company

A well-structured class provides a crisp abstraction of something drawn from the vocabulary of the problem domain or the solution domain.

Review

Review: Architectural Analysis

- What is the purpose of Architectural Analysis?
- What is a package?
- What are analysis mechanisms? Give examples.
- What key abstractions are identified during Architectural Analysis? Why are they identified here?
- What is a layered architecture? Give examples of typical layers.

38

Rational
the software development company

Exercise: Architectural Analysis

Exercise: Architectural Analysis

- ◆ Given the following:
  - ▪ Some results from the Requirements discipline:
    - • Problem statement
    - • Use-Case Model main diagram
    - • Glossary
  - ▪ Some architectural decisions:
    - • (textually) The upper-level architectural layers and their dependencies

(continued)

39

Rati⊘nal
the software development **company**

The goal of this exercise is to jump-start analysis.

References to givens:

- **Requirements Results**: *Payroll Requirements Document*

- **Architectural Decisions**: *Payroll Architecture Handbook*, Logical View, Architectural Analysis section.

Note: This exercise has been tightly scoped to emphasize the Analysis and Design modeling concepts and reduce the emphasis on architectural issues. Thus, much of the architecture has been provided to you, rather than asking you to provide it as part of the exercise. Remember, this is not an architecture course.

Exercise: Architectural Analysis (cont.)

♦ Identify the following:
  ▪ The key abstractions

(continued)

Rational
the software development **company**

To identify the key abstractions, you can probably concentrate on the Problem Statement and the Glossary.

Create a class to represent each key abstraction   Be sure to include a brief description for each class. You do not need to allocate the classes to packages. That will occur in the next module. You do not need to define relationships between the classes at this point. We will concentrate on class relationships in later modules.

The class diagrams of the upper-level layers and their dependencies should be drawn using the given textual descriptions.

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

Refer to the following slides if needed;

Exercise: Architectural Analysis (cont.)

Exercise: Architectural Analysis (cont.)

- ◆ Produce the following:
  - ▪ Class diagram containing the key abstractions
  - ▪ Class diagram containing the upper-level architectural layers and their dependencies
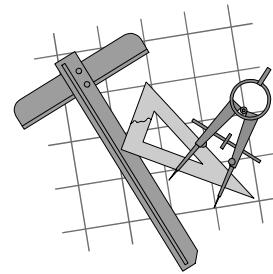
Ration**al**®
the software development company

You will need to create two different class diagrams in this solution: one showing the key abstractions and one showing the architectural layers and their dependencies.

Refer to the following slides if needed;

- Package Relationships: Dependency – p. 5-8

- Example: Key Abstractions – p. 5-30

- Example: High-level Organization of the Model – p. 5-18

Exercise: Review

Exercise: Review

- ◆ Compare your key abstractions with the rest of the class
  - ▪ Have the key concepts been identified?
  - ▪ Does the name of each class reflect the role it plays?
- ◆ Compare your class diagram showing the upper-level layers
  - ▪ Do the package relationships support the Payroll System architecture?

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

42

Rational
the software development company