IBM Software Group | Rational Software France

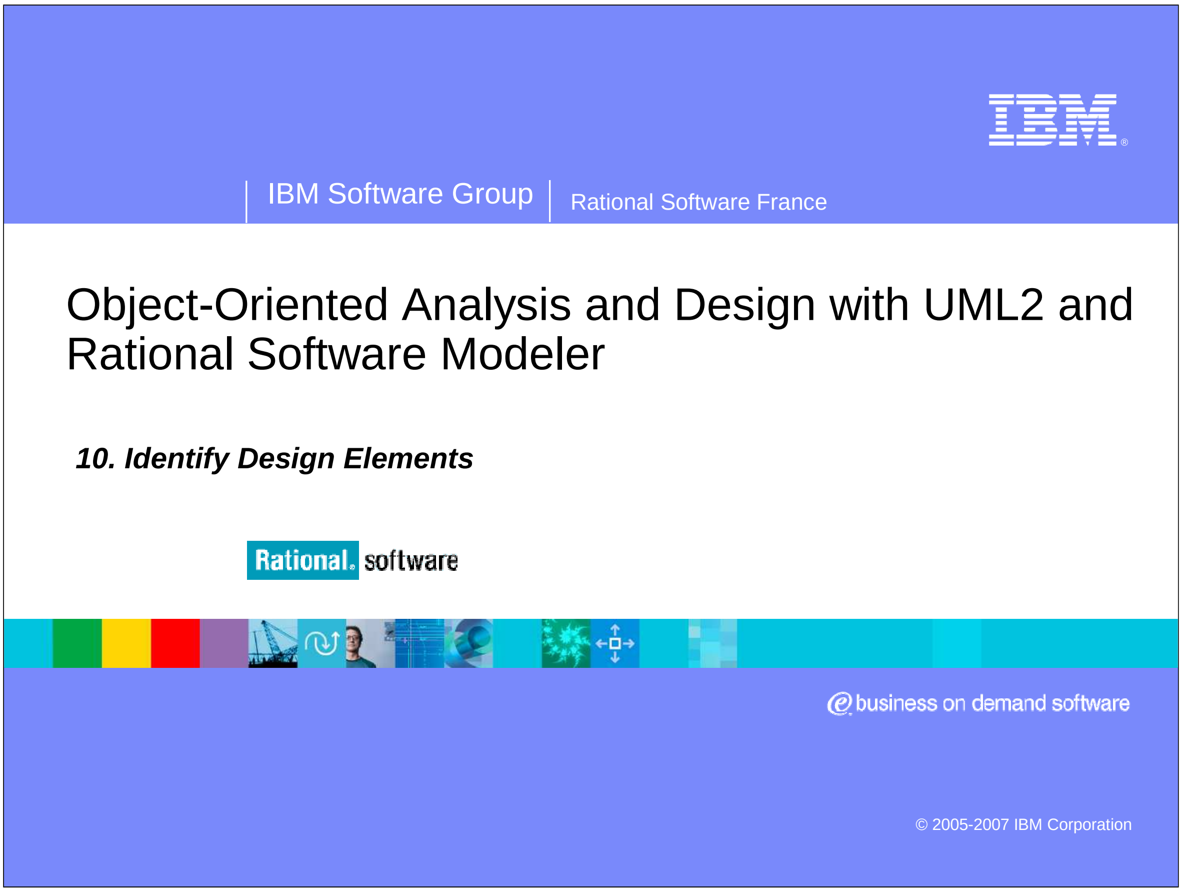
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

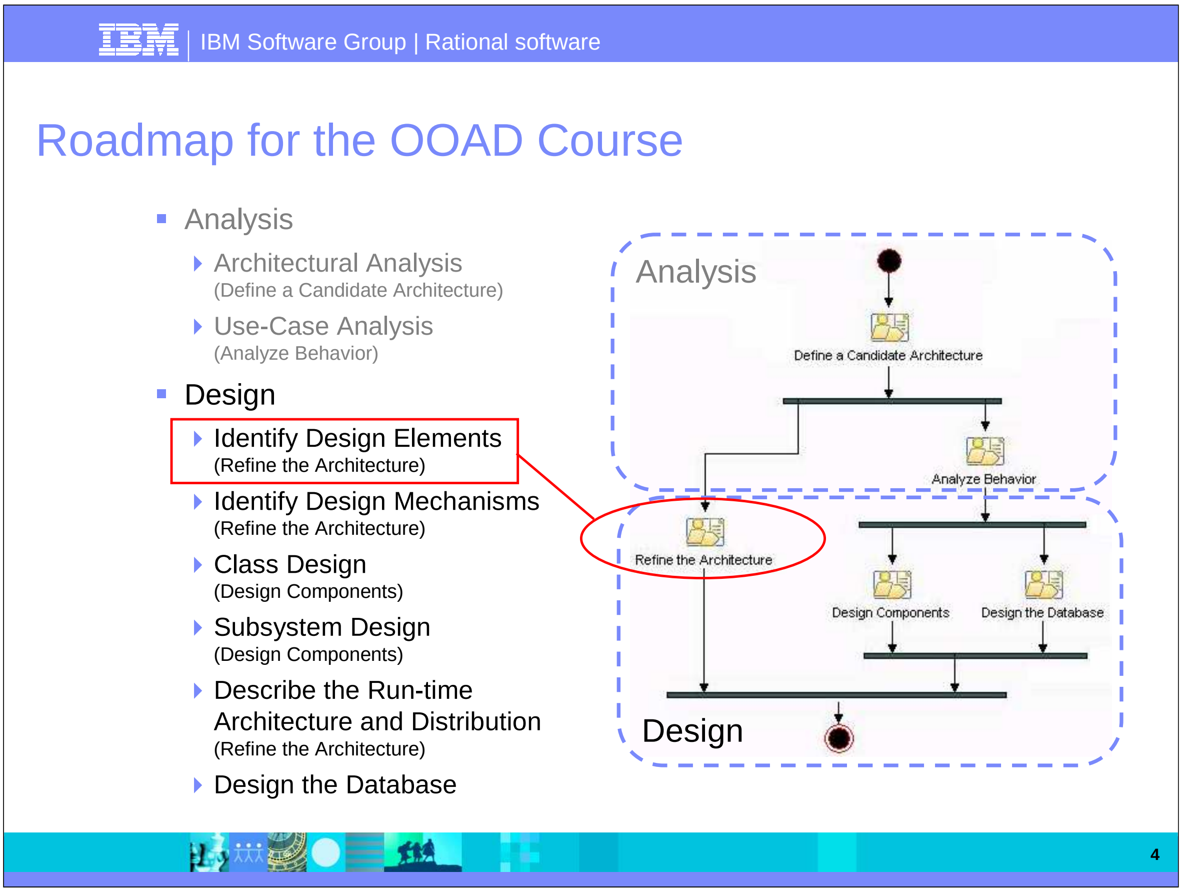
PART III – Object-Oriented Design

Rational. software

@business on demand software

Table of Contents


IBM
IBM Software Group
Rational Software France
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler
10. Identify Design Elements
Rational. software
business on demand software
© 2005-2007 IBM Corporation



In **Architectural Analysis**, an initial attempt was made to define the layers of our system, concentrating on the upper layers. In **Use-Case Analysis**, you analyzed your requirements and allocated the responsibilities to analysis classes.

In **Identify Design Elements**, the analysis classes are refined into design elements (design classes and subsystems).

In Use-Case Analysis, you were concerned with the "what." In the architecture activities, you are concerned with the "how". Architecture is about making choices.

IBM Software Group | Rational software

Identify Design Elements

- Purpose
  - To analyze interactions of analysis classes to identify design model elements
- Role
  - Software Architect
- Major Steps
  - Map Analysis Classes to Design Elements
  - Identify Subsystems and Subsystem Interfaces
  - Update the Organization of the Model
- Note:
  - The objective is to identify design elements, NOT to refine the design, which is covered in Design Components

5

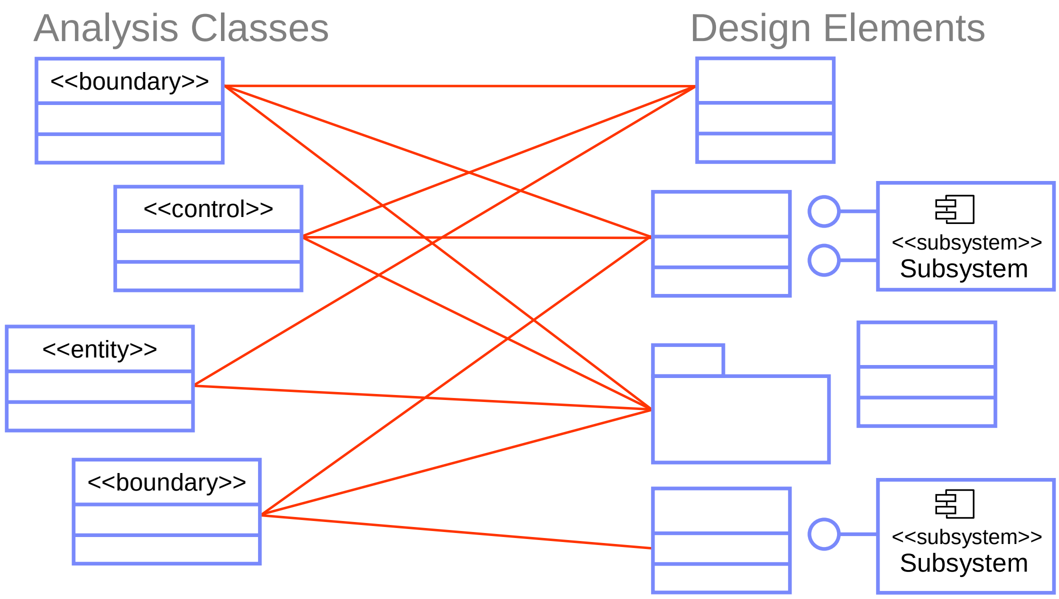
IBM | IBM Software Group | Rational software

Where Are We?

- ➡ Map Analysis Classes to Design Elements
  - Identify Subsystems and Subsystem Interfaces
  - Update the Organization of the Model

6

From Analysis Classes to Design Elements



Many-to-Many Mapping

IBM Software Group | Rational software

Analysis Classes vs. Design Elements

- Analysis classes:
  - Handle primarily functional requirements
  - Model objects from the “problem” domain
- Design elements:
  - Must also handle nonfunctional requirements
  - Model objects from the “solution” domain

8

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

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

Drivers When Identifying Design Elements

- Non-functional requirements, for instance consider:
  - Application to be distributed across multiple servers
  - Real-time system vs. e-Commerce application
  - Application must support different persistent storage implementations
- Architectural choices
  - For instance, .NET vs. Java Platform
- Technological choices
  - For instance, Enterprise Java Beans can handle persistence
- Design principles (identified early in the project's life cycle)
  - Use of patterns (discussed in detail in the Identify Design Mechanisms module)
  - Best practices (industry, corporate, project)
  - Reuse strategy

Mapping the Design Model to Other Models

- Maintaining a separate analysis model
  - Every Analysis Class in the Analysis Model should be associated with at least one design class in the Design Model

- Mapping design to implementation
  - The decision to map design to implementation should be made before design starts
  - May vary based on how you map the design elements to implementation classes, files, packages and subsystems in the implementation model but should be consistent
- Impact of using an MDD/MDA approach

IBM Software Group | Rational software

Identifying Design Classes

- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
    - Typically, entity classes survive relatively intact into Design
- A more complex analysis class may:
  - Be split into multiple classes
  - Become a part of another class
  - Become a package
  - Become a subsystem (discussed later)
  - Become a relationship
  - Be partially realized by hardware
  - Not be modeled at all
  - Any combination …

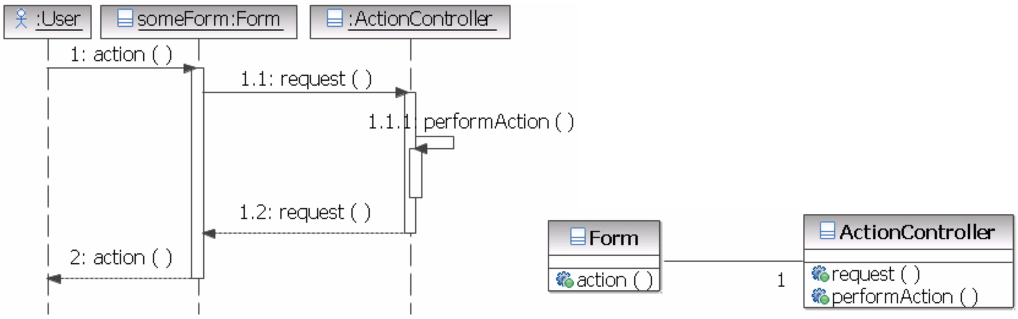
11

Some examples:

- A single boundary class representing a user interface may result in multiple classes, one per window.
- A control class may become a design class directly, or become a method within a design class.
- A single entity class may become multiple classes (for example, an aggregate with contained classes, or a class with associated database mapping or proxy classes, etc.).

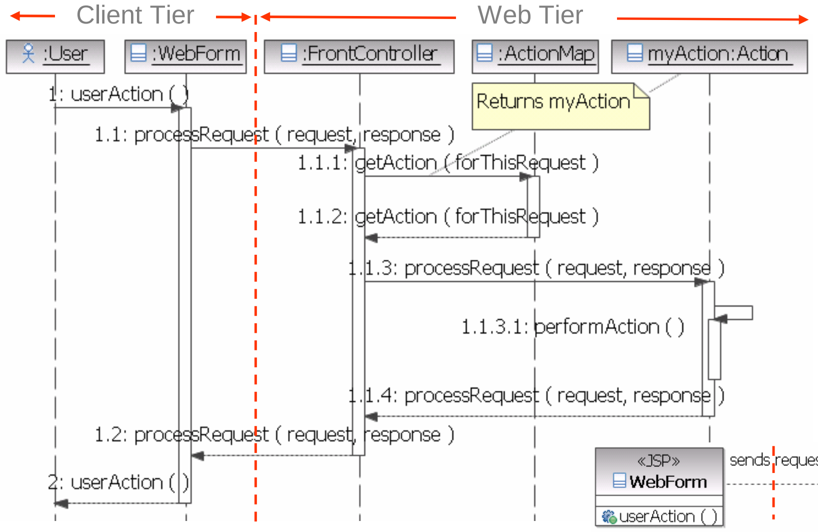
Example: Analysis

- At the end of Analysis, let's assume we ended up with the (very simple and yet generic) model below
  - Our requirements stipulate that this is a typical J2EE Web application, with a thin client and a Web server…



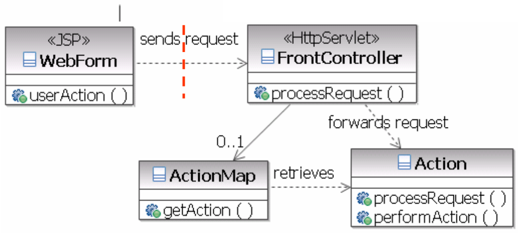
IBM Software Group | Rational software

Example: Design



In our example, the form becomes a JSP and the controller is split in 2 classes: a *FrontController* servlet (a J2EE best practice and pattern) and an *Action* class that does the actual work (*performAction*)

Patterns are discussed in detail in the next module



In fact our architect has decided to use the Struts framework, which will among other things handle the *FrontController* and *ActionMap* parts …

13

The purpose of this slide is not to describe a complete solution. In fact there are many possible variants depending on many factors. And this is what we need to have a generic solution (the FrontController and Action scheme here) for a common problem (user actions in web pages). The next module (Identify Design Mechanism) discusses this topic in more detail.

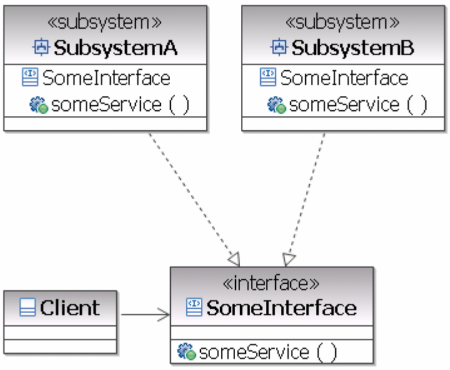
Where Are We?

- Map Analysis Classes to Design Elements
- Identify Subsystems and Subsystem Interfaces
- Update the Organization of the Model

IBM Software Group | Rational software

Subsystems As Replaceable Design Elements

- Subsystems are components that provide services to their clients only through public interfaces
  - Any two subsystems that realize the same interfaces are interchangeable
  - Subsystems support multiple implementation variants
- Subsystems can be used to partition the system into units which:
  - Can be independently changed without breaking other parts of the systems
  - Can be independently developed (as long as the interfaces remain unchanged)
  - Can be independently ordered, configured, or delivered



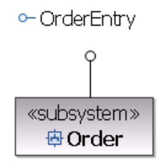
Subsystems are ideal for modeling components - the replaceable units of assembly in component-based development

15

IBM | IBM Software Group | Rational software

Candidate Subsystems

- Analysis Classes providing complex services and/or utilities
  - For example, security authorization services
- Boundary classes
  - User interfaces
  - Access to external systems and/or devices
- Classes providing optional behavior or different levels of the same services
- Highly coupled elements
- Existing products that export interfaces (communication software, database access support, etc.)



16

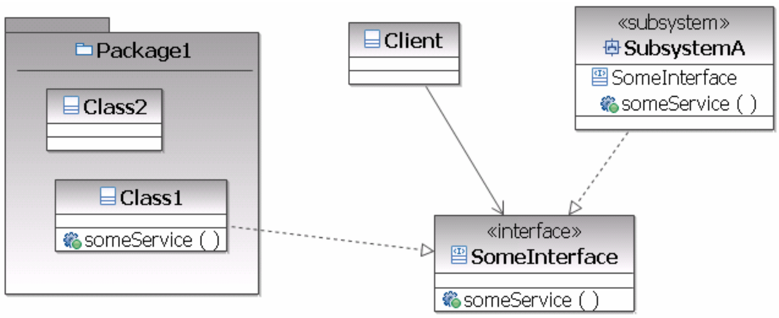
A complex analysis class is mapped to a design subsystem if it appears to embody behavior that cannot be the responsibility of a single design class acting alone. A complex design class may also become a subsystem, if it is likely to be implemented as a set of collaborating classes.

The design subsystem is used to encapsulate these collaborations in such a way that clients of the subsystem can be completely unaware of the internal design of the subsystem, even as they use the services provided by the subsystem. If the participating classes/subsystems in a collaboration interact only with each other to produce a well-defined set of results, the collaboration and its collaborating design elements should be encapsulated within a subsystem.

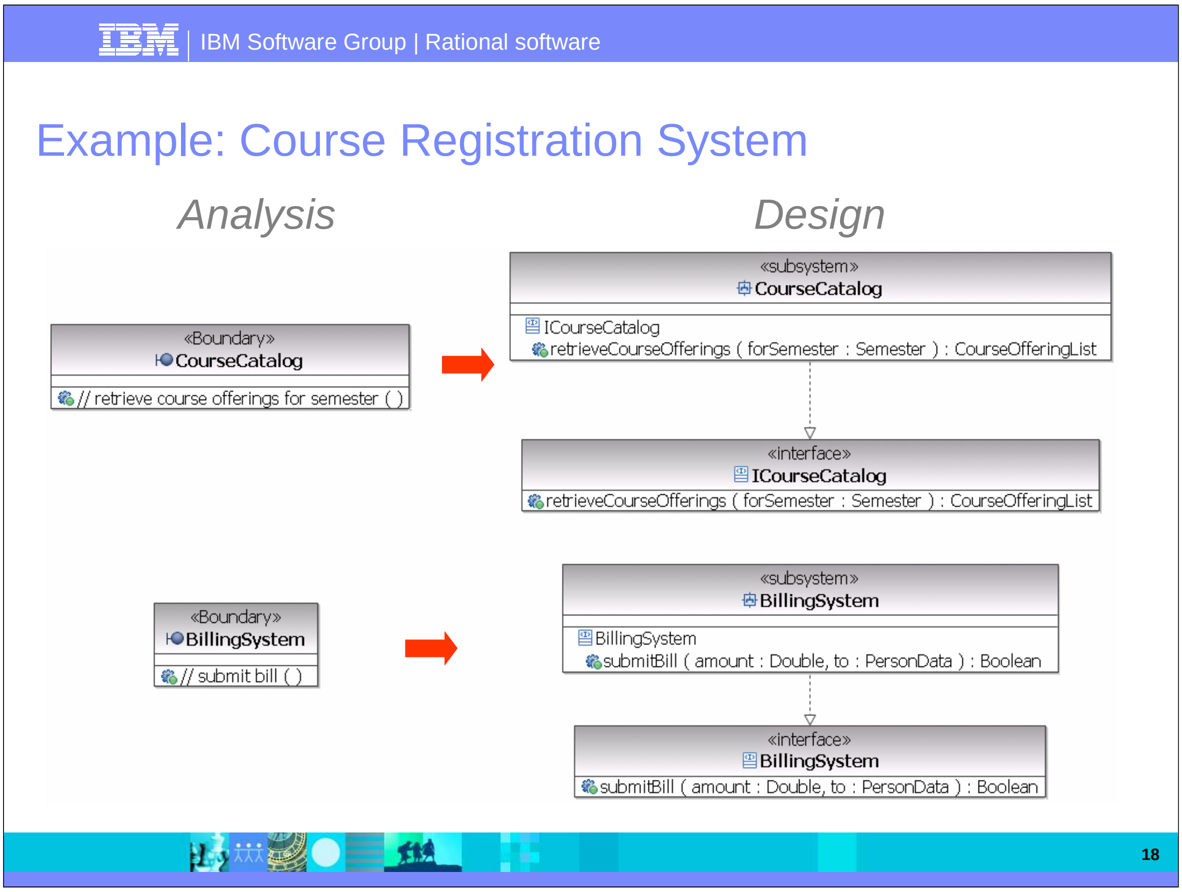
This rule can be applied to subsets of collaborations as well. Anywhere part or all of a collaboration can be encapsulated and simplified, doing so will make the design easier to understand.

Packages and Subsystems

- Packages and subsystems both provide structure
  - In fact in UML 1.x, subsystems were a cross between packages (providing structure) and classes (providing behavior)
- Both packages and subsystems can be used to achieve the desired effect (see diagram)
  - Subsystems should be preferred in most cases, as they provide better encapsulation, better de-coupling and are more easily replaceable



Collections of types and data structures (e.g. stacks, lists, queues) may be better represented as packages, because they reveal more than behavior, and it is the particular contents of the package that are important and useful (and not the package itself, which is simply a container).



During Use-Case Analysis, we modeled two boundary classes, the BillingSystem and the CourseCatalog, whose responsibilities were to cover the details of the interfaces to the external systems. It was decided by the architects of the Course Registration System that the interactions to support external system access will be more complex than can be implemented in a single class. Thus, subsystems were identified to encapsulate these responsibilities and provide interfaces that give the external systems access.

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

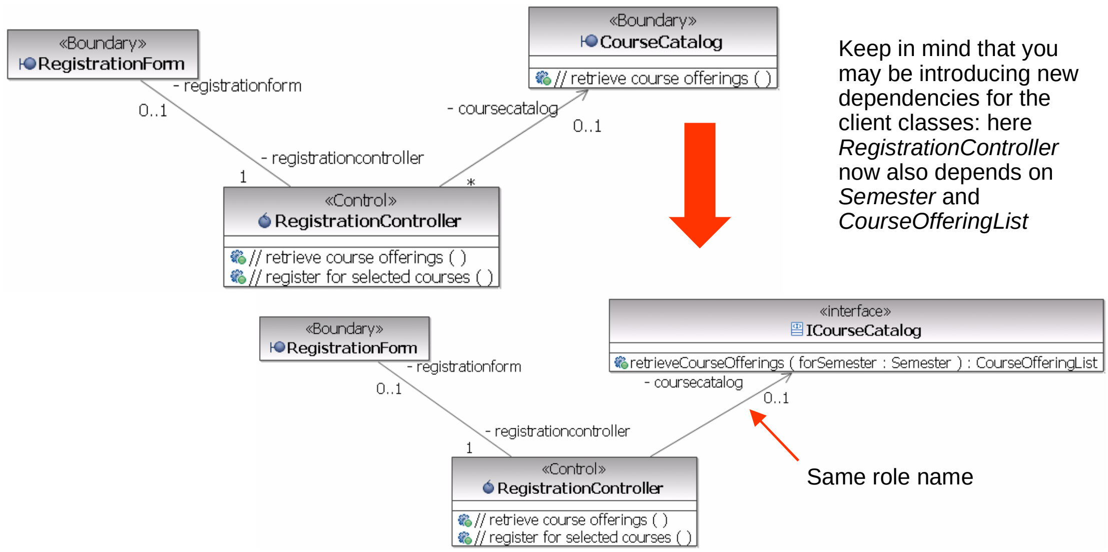
The CourseCatalog subsystem encapsulates all the work involved for communicating to the legacy Course Catalog System. The system provides access to the unabridged catalog of all courses and course offerings provided by the university, including those from previous semesters.

These are subsystems rather than packages because a simple interface to their complex internal behaviors can be created. Also, by using a subsystem with an explicit and stable interface, the particulars of the external systems to be used (in this case, the Billing System and the legacy Course Catalog) could be changed at a later date with no impact on the rest of the system.

IBM Software Group | Rational software

Incorporating Interfaces in Class Diagrams

- Every relationship to the initial analysis class must be replaced by an equivalent relationship to the subsystem interface

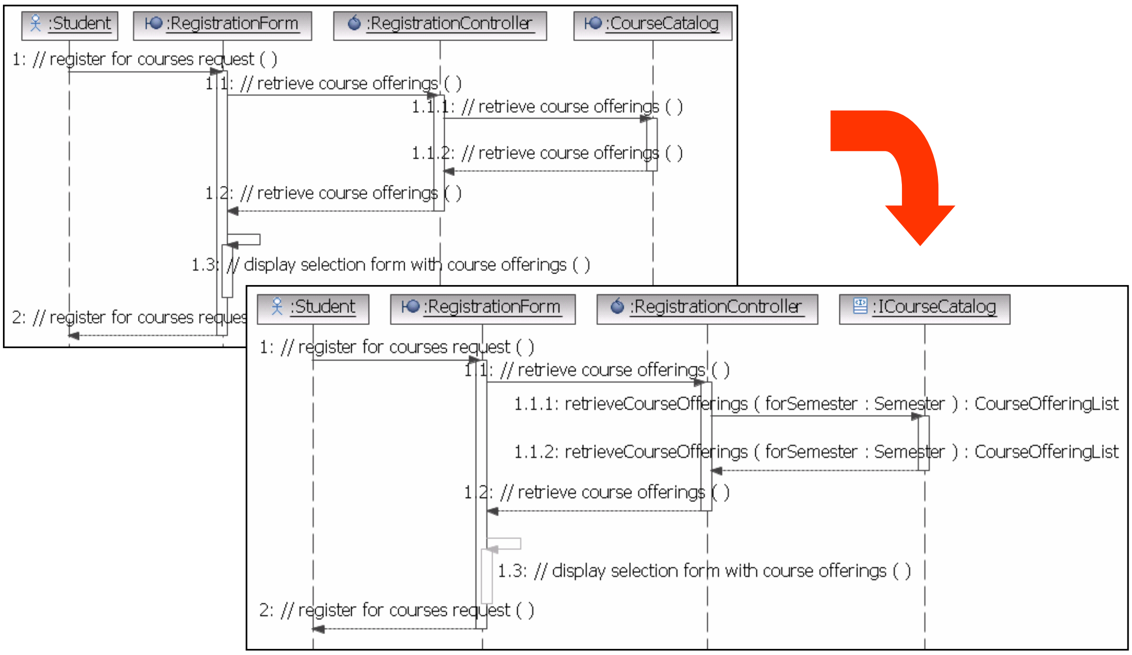


19

In RSA/RSM, these changes have to be performed manually:

- Retrieve the interface to use and drag it to the diagram
- Select the relationship and move the target end from the analysis class to the interface
- Delete the analysis class from the diagram
- Delete the analysis class from the design model after all changes have been made

Incorporating Interfaces in Sequence Diagrams



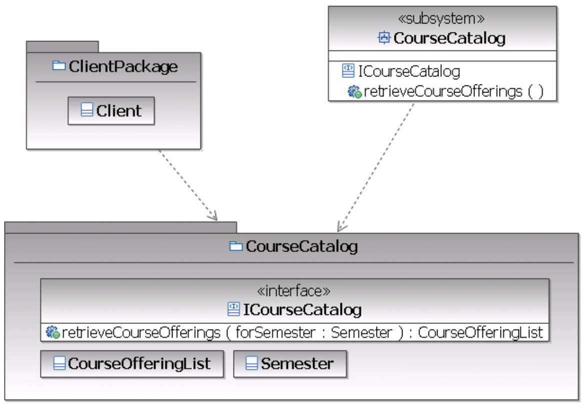
20

In RSA/RSM, simply drag the interface over the analysis object and update the message.

IBM | IBM Software Group | Rational software

Subsystem Dependencies

- Keep in mind: The interfaces provided (and/or required) by a subsystem are **outside** the subsystem
- Often the services described by an interface will involve non-standard types, e.g. *Semester* and *CourseOfferingList*
  - You can group the interfaces and types in a single package
  - Both the client packages and the realizing subsystem have dependencies on this package



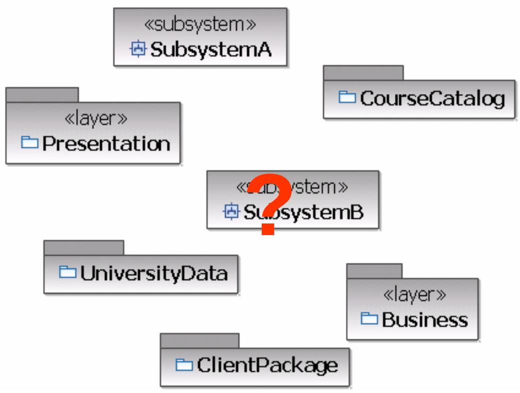
21

Where Are We?

- Map Analysis Classes to Design Elements
- Identify Subsystems and Subsystem Interfaces
- Update the Organization of the Model

The Building Blocks of our Architecture

- Keep in mind: we are building a component-based architecture
  - The building blocks of our architecture are the packages, subsystems, and other components of our system
- The building blocks are “layered” in order to achieve a number of goals like application availability, security, performance, user-friendliness, reuse, …, and of course functionality to end-users
- To achieve our goals, we need to control how our building blocks are packaged and assigned across layers

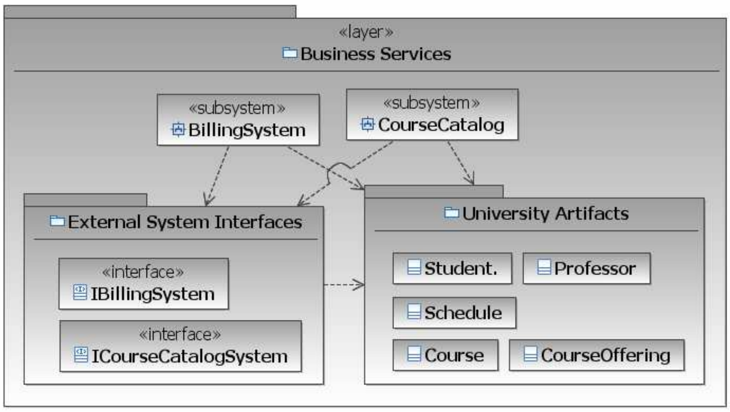


Design Packages

- Design packages are used to group related design elements together
- Design packages and subsystems are the building blocks of our architecture
  - They should be organized to achieve the goals of this architecture
  - Simply grouping logically related classes is not enough
  - Apply the basic object-oriented principles:
    - Encapsulation
    - Separation of interface and implementation
    - Loose coupling with the “outside”
- Design packages are also often used as configuration units and to organize the allocation of work across development teams
- Remember: If one element of package A has a relationship with at least one element of package B, then package A depends on package B

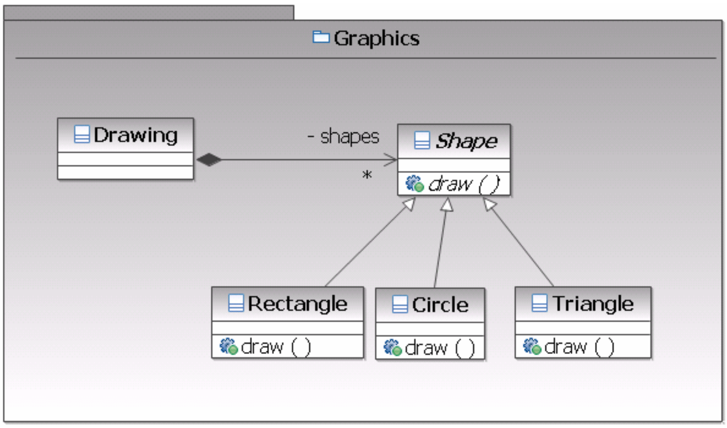
Example 1: What Is Wrong With This Picture?

- Can you point out the weaknesses of this model organization?
- What changes would you suggest?



Example 2: Improve This Model

- How would you improve this model?
- Could you use an interface instead of an abstract class?



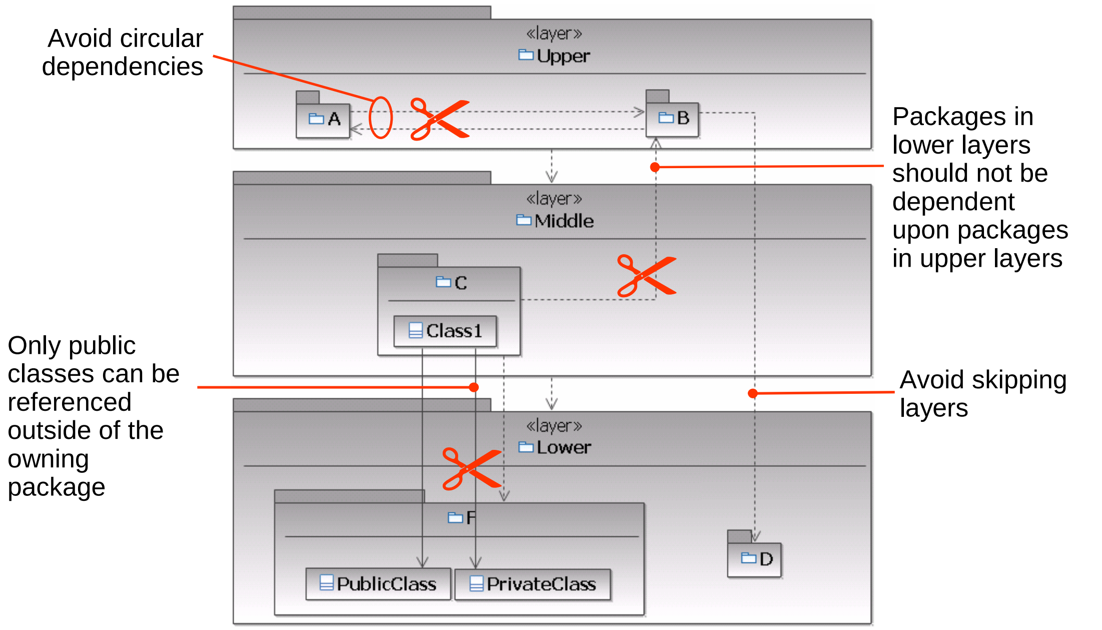
IBM Software Group | Rational software

Packaging Tips

- Consider grouping two design elements in the same package if:
  - They are dependent on each other (relationships)
  - Their instances interact with a large number of messages (to avoid having a complicated intercommunication)
  - They interact with, or affected by changes in, the same actor
- If an element is related to an optional service, group it with its collaborators in a separate subsystem
- Consider moving two design elements in different packages if:
  - One is optional and the other mandatory
  - They are related to different actors
- Think of the dependencies that co-located elements may have on your element
- Consider how stable your design element is:
  - Try to move stable elements down the layer hierarchy, unstable elements up

27

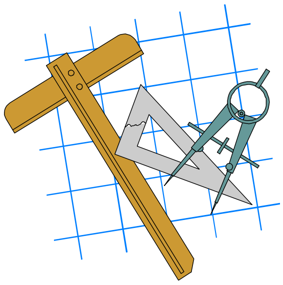
Evaluating Package Coupling



IBM | IBM Software Group | Rational software

Exercise

- Perform the exercise provided by the instructor (lab 6)



29

IBM Software Group | Rational software

30

IBM Software Group | Rational Software France

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

11. Identify Design Mechanisms

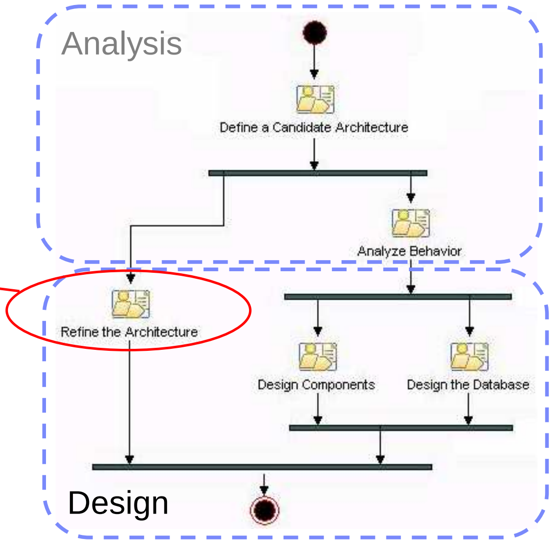
Rational. software

business on demand software

IBM Software Group | Rational software

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



32

IBM Software Group | Rational software

Identify Design Mechanisms

- Purpose
  - To analyze interactions of analysis classes to identify design model elements
- Role
  - Software Architect
- Major Steps
  - Identify Design and Implementation Mechanisms
  - Document Architectural Mechanisms

33

Where Are We?

- Introduction to Design Patterns
- Identify Design and Implementation Mechanisms
- Document Architectural Mechanisms

IBM Software Group | Rational software

What Is a Design Pattern?

- A design pattern describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context
- Popularized by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (the “Gang of Four”) in Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1994
- Deep, really useful patterns are typically ancient; you see one and will often remark, “Hey, I’ve done that before.”
  (Grady Booch, Foreword in Core J2EE Patterns, Deepak Alur, John Crupi & Dan Malks, Prentice Hall, 2003)
- Patterns are “half baked,” meaning that you always have to finish them off in the oven of your own project
  (Martin Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2003)

35

Design patterns are medium-to-small-scale patterns, smaller in scale than architectural patterns but typically independent of programming language. When a design pattern is bound, it forms a portion of a concrete design model (perhaps a portion of a design mechanism). Design patterns tend, because of their level, to be applicable across domains.

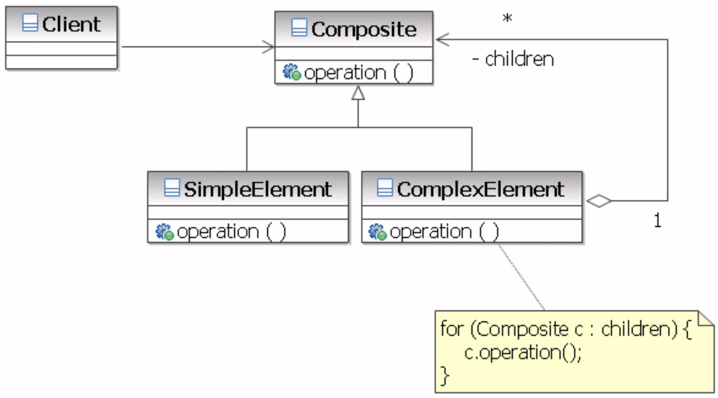
We will introduce several patterns in this module and the remaining design modules.

Some of the GoF Patterns

Pattern	Example
Command (behavioral pattern)	Issue a request to an object without knowing anything about the operation requested or the receiver of the request: for example, the response to a menu item, an undo request, the processing of a time-out
Abstract factory (creational pattern)	Create GUI objects (buttons, scrollbars, windows, etc.) independent of the underlying OS: the application can be easily ported to different environments
Proxy (structural pattern)	Handle distributed objects in a way that is transparent to the client objects (*remote proxy*) Load a large graphical object or any entity object “costly” to create/initialize only when needed (*on demand*) and in a transparent way (*virtual proxy*)
Observer (behavioral pattern)	When the state of an object changes, the dependent objects are notified. The changed object is independent of the observers.

IBM Software Group | Rational software

Example of a Structural Pattern: Composite (GoF)



- Examples:
  - File system composed of files and directories
  - Graphic composed of elementary shapes and assemblages of shapes

37

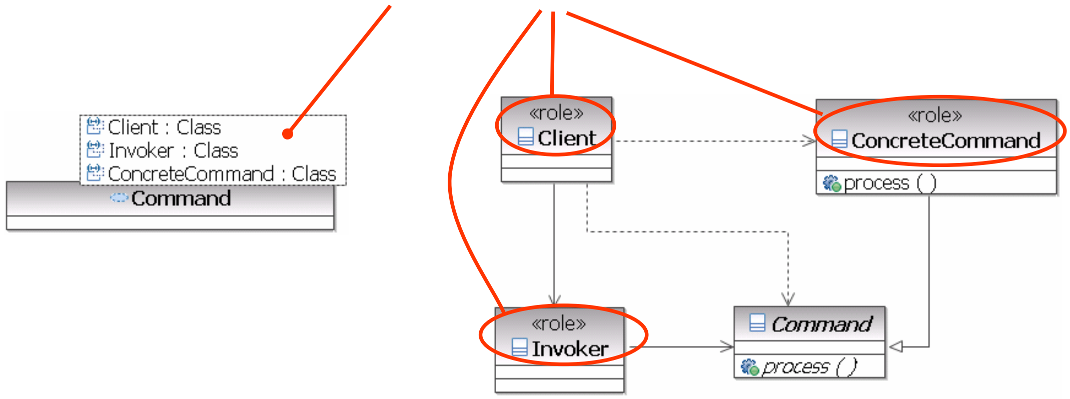
Case Study: Building a Generic GUI Component

- The problem
  - Imagine we want to build a reusable GUI component
  - To keep it simple, we will limit ourselves to the implementation of generic menus in a windowing system (in such a way that it will be possible to add new menus without having to modify the GUI component)
- The solution
  - Is based on the Command pattern
  - Will now be exposed by your instructor

Representing Patterns in UML

- A design pattern is a parameterized collaboration
  - Note: <<role>> is not a standard stereotype

Parameters of the collaboration

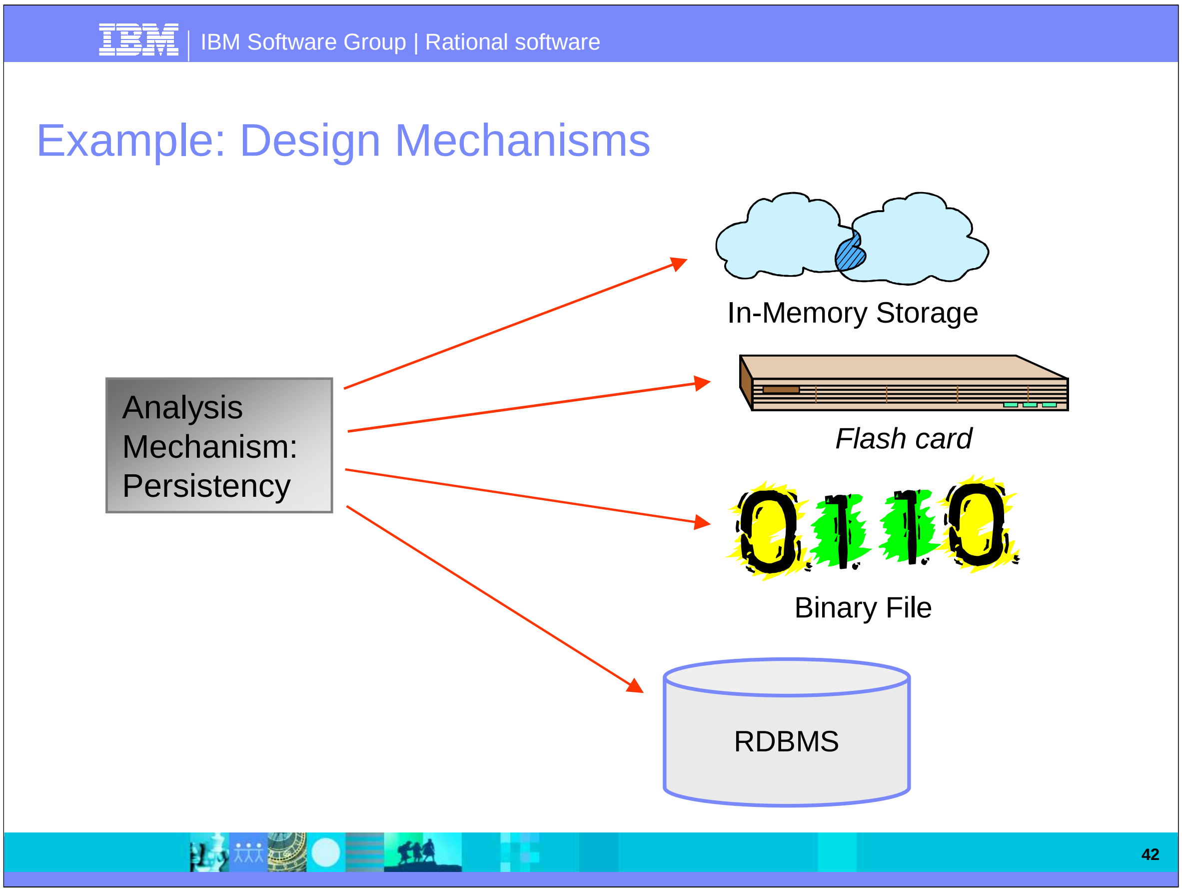


Where Are We?

- Introduction to Design Patterns
- ➡ Identify Design and Implementation Mechanisms
- Document Architectural Mechanisms

Design Mechanisms

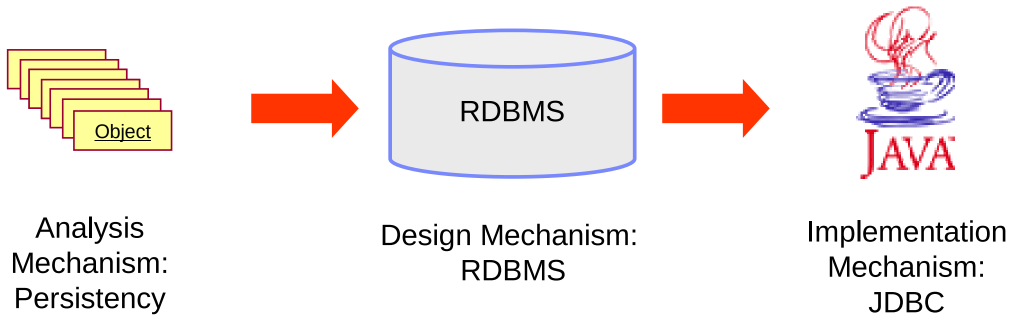
- A design mechanism is a refinement of a corresponding analysis mechanism
  - It adds concrete detail to the conceptual analysis mechanism, but stops short of requiring particular technology - for example, a particular vendor's implementation of a RDBMS
  - It may instantiate one or more patterns (architectural or design patterns)
- To identify design mechanisms from analysis mechanisms:
  - Identify the clients of each analysis mechanism
  - Identify characteristic profiles for each analysis mechanism
  - Group clients according to their use of characteristic profiles
  - Proceed bottom up and make an inventory of the design mechanisms that you have at your disposal


IBM Software Group | Rational software
Example: Design Mechanisms
Analysis Mechanism: Persistency
In-Memory Storage
Flash card
0110
Binary File
RDBMS
42

IBM Software Group | Rational software

Implementation Mechanisms

- An implementation mechanism is a refinement of a corresponding design mechanism
  - It may use, for example, a particular programming language and other implementation technology
  - It may instantiate one or more idioms or implementation patterns



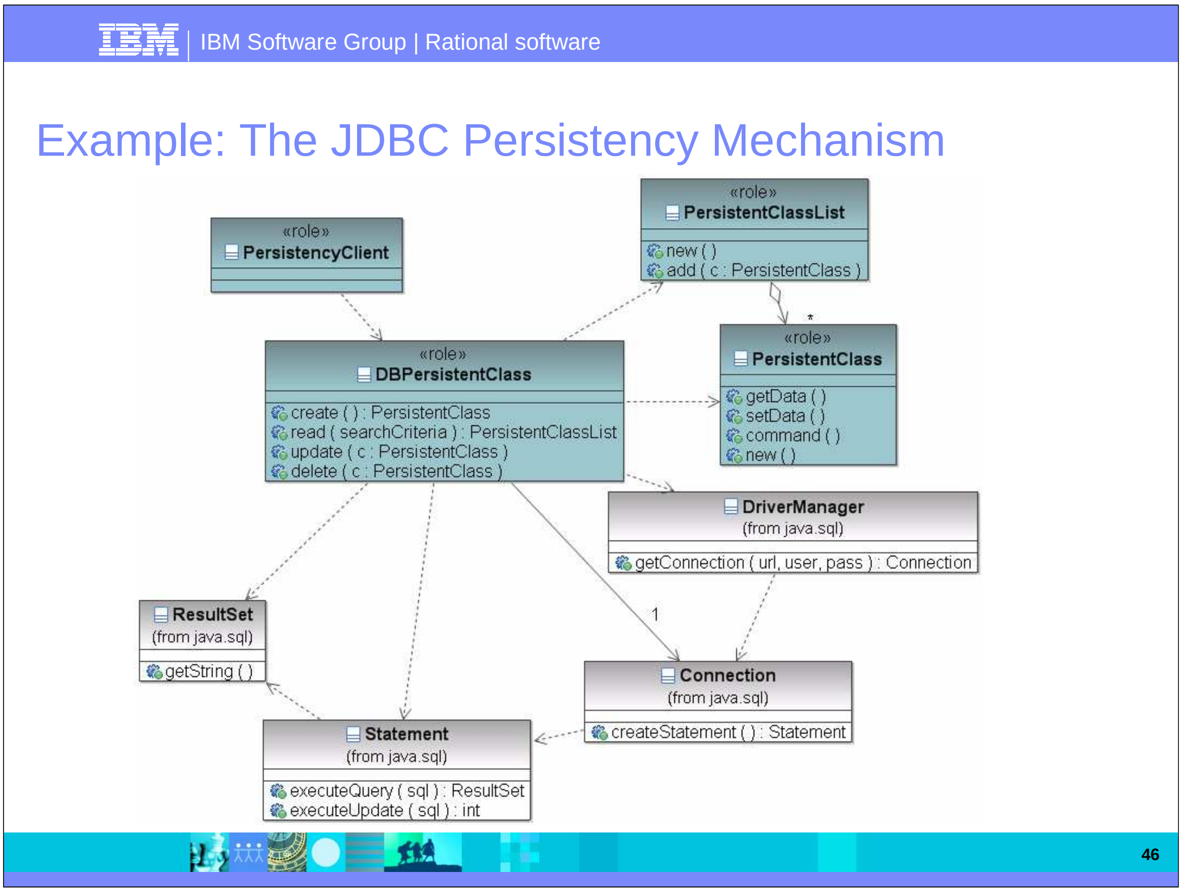
43

Where Are We?

- Introduction to Design Patterns
- Identify Design and Implementation Mechanisms
- ➡ Document Architectural Mechanisms

Document Architectural Mechanisms

- A mechanism represents a pattern that constitutes a *common solution* to a *common problem*
  - Our ultimate goal is to ensure consistency in the implementation of our system, while improving productivity
- Having defined *what* implementation mechanism should be used by all client classes with the same characteristics profile, the software architect also defines *how* to use it
  - The end result is a collaboration that will be documented like any other collaboration: using sequence diagrams and diagrams of participating classes



The next few slides demonstrate the JDBC mechanism chosen for our persistent classes in our example.

For JDBC, a client works with a DBPersistentClass to read and write persistent data. The DBPersistentClass is responsible for accessing the JDBC database using the DriverManager Java class. Once a database Connection is opened, the DBPersistentClass can then create SQL statements that will be sent to the underlying RDBMS and executed using the Statement class. The Statement is what “talks” to the database. The result of the SQL query is returned in a ResultSet object.

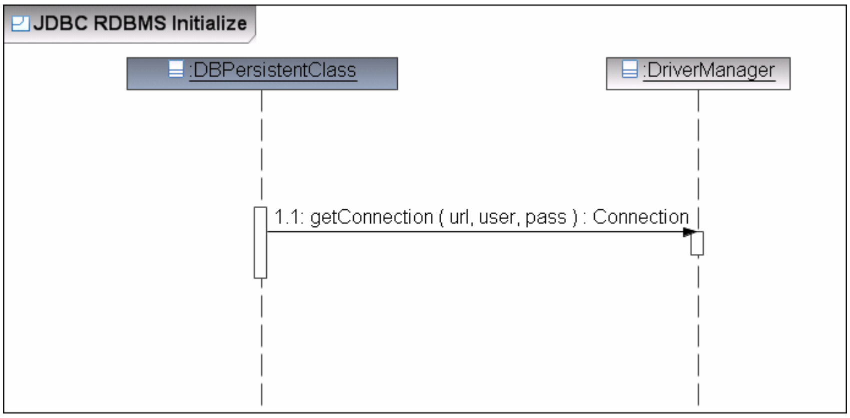
DBPersistentClass understands the OO-to-RDBMS mapping and has the ability to interface with the RDBMS. It flattens the object, writes it to the RDBMS, reads the object data from the RDBMS, and builds the object. Every class that is persistent has a corresponding DBPersistentClass.

The PersistentClassList is used to return a set of persistent objects as a result of a database query (for example, DBClass.read()).

The <<role>> stereotype was used for anything that should be regarded as a placeholder for the actual design element to be supplied by the developer. This convention makes it easier to apply the mechanism, because it is easier to recognize what the designer must supply.

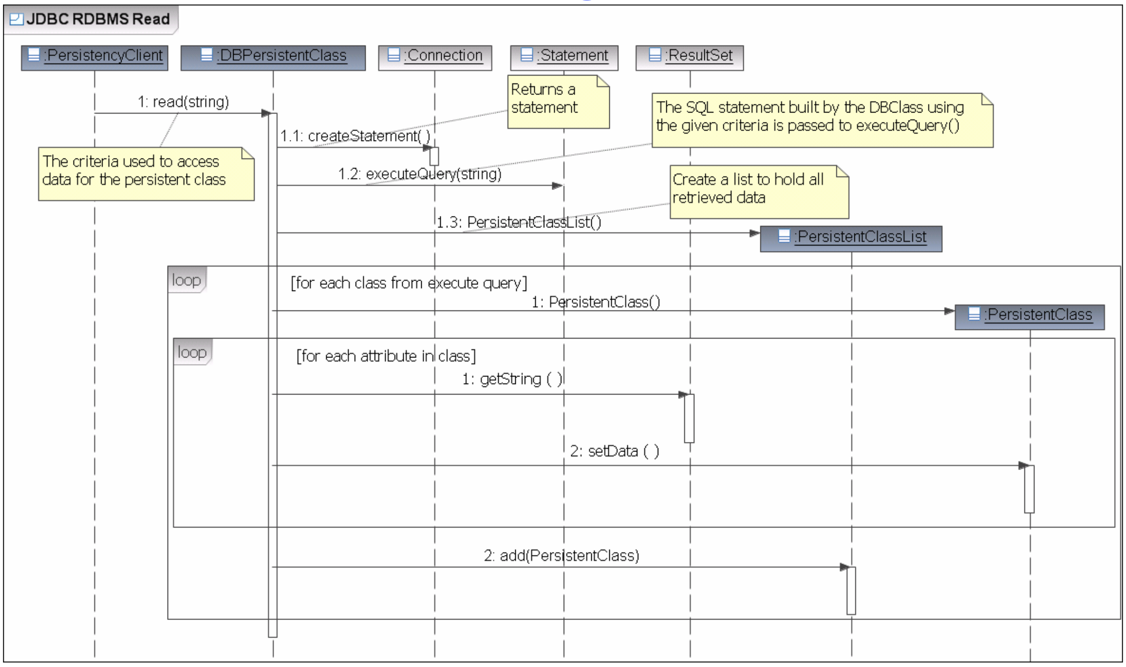
JDBC Mechanism: Initializing the Connection

- Initialization must occur before any persistent class can be accessed
  - getConnection() returns a Connection object for the specified url



IBM Software Group | Rational software

JDBC Mechanism: Retrieving Data

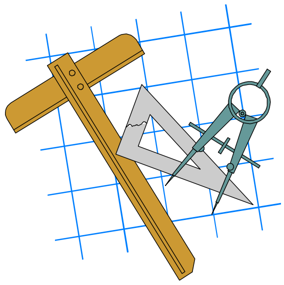


48

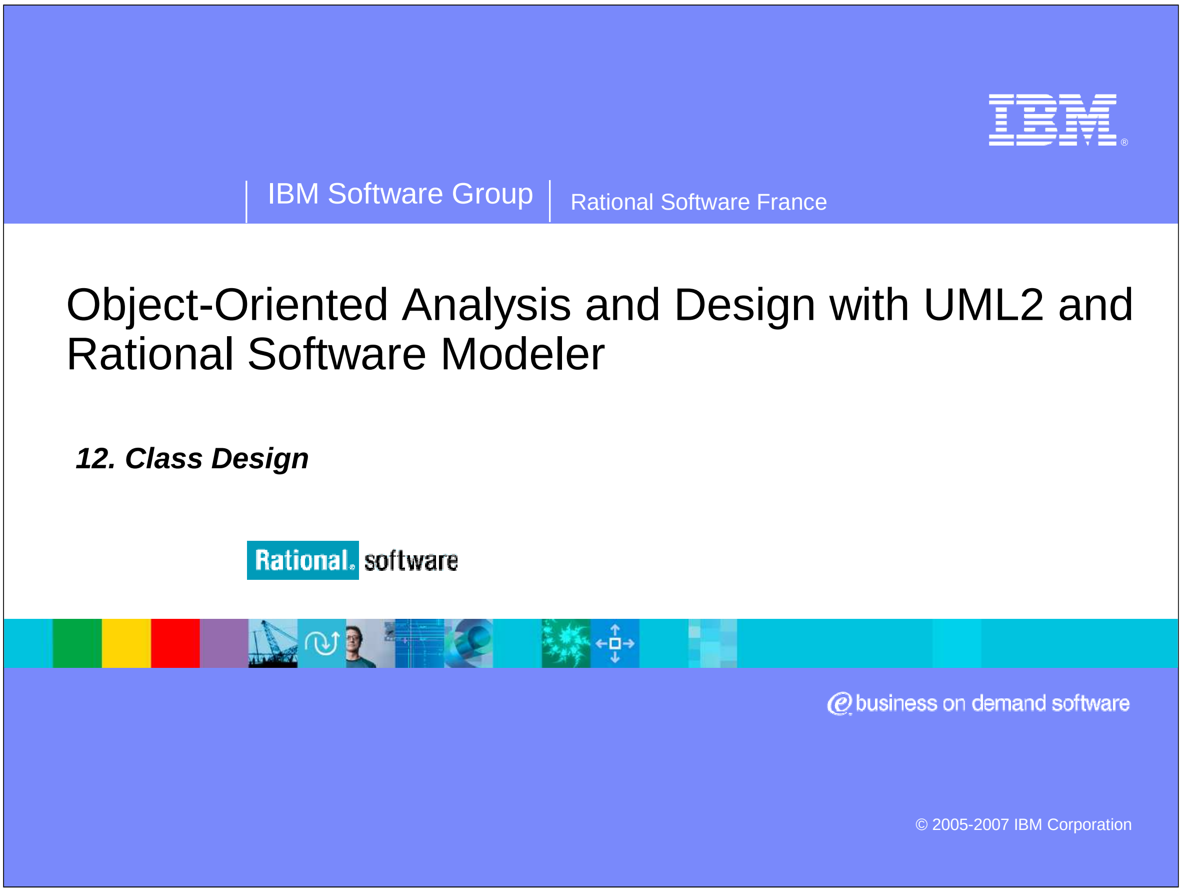
IBM Software Group | Rational software

Exercise

- There is no exercise in this module

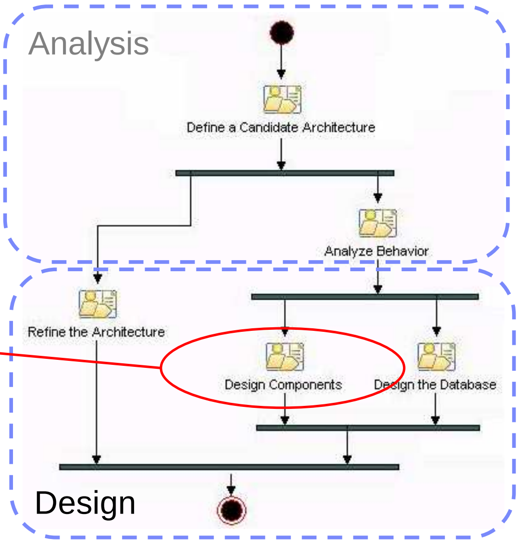


49


IBM Software Group
Rational Software France
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler
12. Class Design
Rational. software
business on demand software
© 2005-2007 IBM Corporation

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

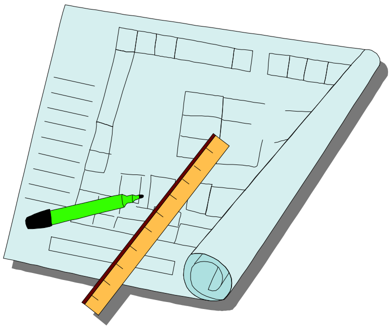


Class Design

- Purpose
  - To ensure that the class provides the behavior the use-case realizations require
  - To ensure that sufficient information is provided to unambiguously implement the class
  - To handle nonfunctional requirements related to the class
  - To incorporate the design mechanisms used by the class
- Role
  - Designer
- Major Steps
  - Create Initial Design Classes
  - Refine Design Classes

Where Are We?

- ➡ Create Initial Design Classes
- Refine Design Classes



Class Design Considerations

- Specific strategies can be used to design a class, depending on its original analysis stereotype (boundary, control, entity)
  - Analysis stereotypes not maintained in Design
- Consider how design patterns can be used to help solve implementation issues
- Consider how the architectural mechanisms will be realized in terms of the defined design classes

(…) I argue that the goal of a model is to capture design decisions as directly as possible, and the best way to do this is to evolve the model by adding elements rather than by replacing them.
(Jim Rumbaugh, p.1 in OMT Insights, Prentice Hall, 1996)

Specific strategies can be used to design a class, depending on its original analysis stereotype (boundary, control, and entity). These stereotypes are most useful during Use-Case Analysis when identifying classes and allocating responsibility. At this point in design, you really no longer need to make the distinction — the purpose of the distinction was to get you to think about the roles objects play, and make sure that you separate behavior according to the forces that cause objects to change. Once you have considered these forces and have a good class decomposition, the distinction is no longer really useful.

How Many Classes Are Needed?

- Many, simple classes means that each class:
  - Encapsulates less of the overall system intelligence
  - Is more reusable
  - Is easier to implement
- A few, complex classes means that each class:
  - Encapsulates a large portion of the overall system intelligence
  - Is less likely to be reusable
  - Is more difficult to implement

A class should have a single well-focused purpose.
A class should do one thing and do it well!

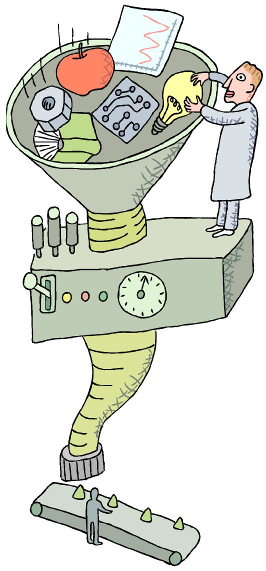
Strategies for Designing Analysis Classes

- Boundary Classes
  - Consider the use of subsystems (see module 10, Identify Design Elements)
  - Many patterns available for Web Browser-based User Interfaces
    - See for instance Core J2EE Patterns, Deepak Alur, John Crupi & Dan Malks, Prentice Hall, 2003
- Control Classes
  - Control classes are directly impacted by issues of concurrency and distribution: see module 14, Describe the Run-time Architecture and Distribution
- Entity Classes
  - Entity classes are usually persistent: see module 15, Design the Database

Remember: the software architect is responsible for the overall design of the architecture and the designer for the actual contents – there is sometimes a fine line between the two. In the remainder of this module we discuss issues related to the detailed design of our classes.

Where Are We?

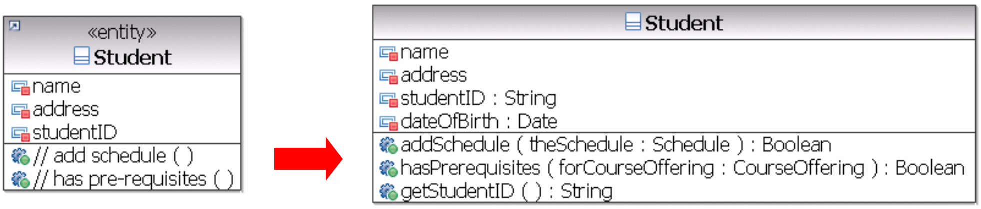
- Create Initial Design Classes
- ➡ Refine Design Classes



IBM Software Group | Rational software

Define Design Operations

- Design operations are directly derived from analysis responsibilities
  - Specify operation name and full operation signature (parameters and return type)
- Additional operations
  - Operations not explicitly defined in analysis (e.g. getters/setters)
  - Manager functions (like constructors, destructors)
  - Functions for copying objects, to test for equality, to test for optional relationships (e.g. isProfessorAssigned() for a CourseOffering class), etc.
  - Helper functions (often private or protected)



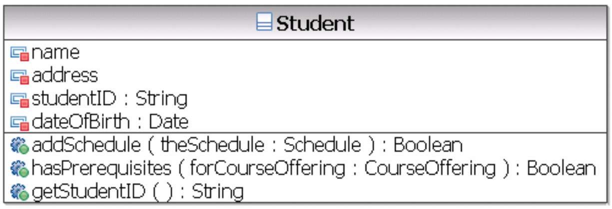
59

IBM | IBM Software Group | Rational software

Define Design Attributes

- Design attributes are derived from analysis attributes
  - Specify name, type and optional default value
  - Private visibility in most cases
- Type can be a built-in data type (UML2 or other), user-defined data type, or user-defined class
  - Consider not using data types from the implementation language

address could be typed as a String or as a new class Address

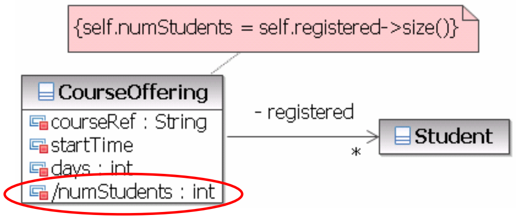


60

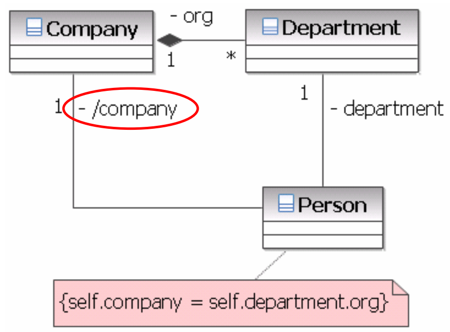
IBM | IBM Software Group | Rational software

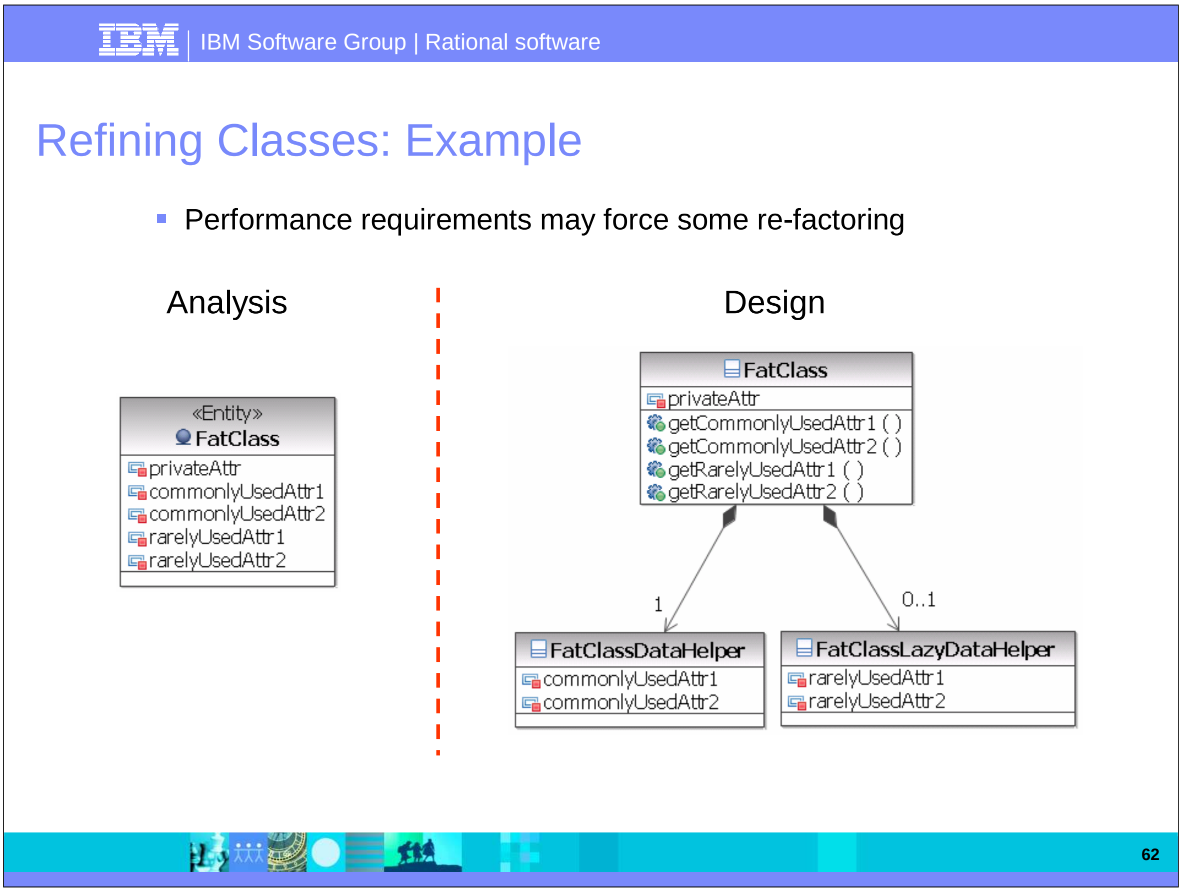
Derived Attributes

- Attributes whose value may be calculated based on the value of other attributes, typically introduced for performance reason
  - But avoid optimizing before you know you really need it!
- Identified by a “/”



Also applicable to roles





During Analysis, entity classes may have been identified and associated with the analysis mechanism for persistence, representing manipulated units of information. Performance considerations may force some re-factoring of persistent classes, causing changes to the Design Model that are discussed jointly between the database designer and the designer responsible for the class. The details of a database-based persistence mechanism are designed during Database Design*,* which is beyond the scope of this course.

Here we have a persistent class with five attributes. One attribute is not really persistent; it is used at runtime for bookkeeping. From examining the use cases, we know that two of the attributes are used frequently. Two other attributes are used less frequently. During Design, we decide that we'd like to retrieve the commonly used attributes right away, but retrieve the rarely used ones only if some client asks for them. We do not want to make a complex design for the client, so, from a data standpoint, we will consider the FatClass to be a proxy in front of two real persistent data classes. It will retrieve the FatClassDataHelper from the database when it is first retrieved. It will only retrieve the FatClassLazyDataHelper from the database in the rare occasion that a client asks for one of the rarely used attributes.

Such behind-the-scenes implementation is an important part of tuning the system from a data-oriented perspective while retaining a logical object-oriented view for clients to use.

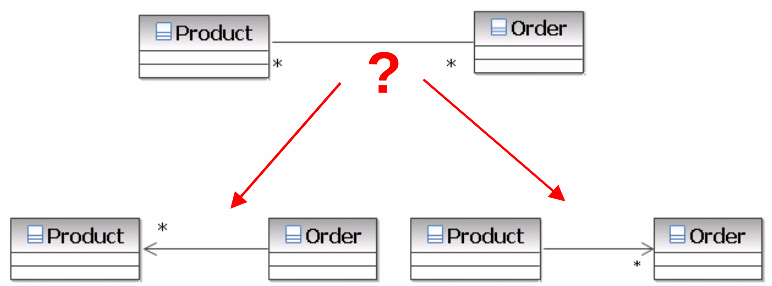
Refine Relationships

- Navigability
- Multiplicity
- Generalization vs. aggregation
- Factoring and delegation
- Refactoring

IBM | IBM Software Group | Rational software

Navigability: Which Directions Are Really Needed?

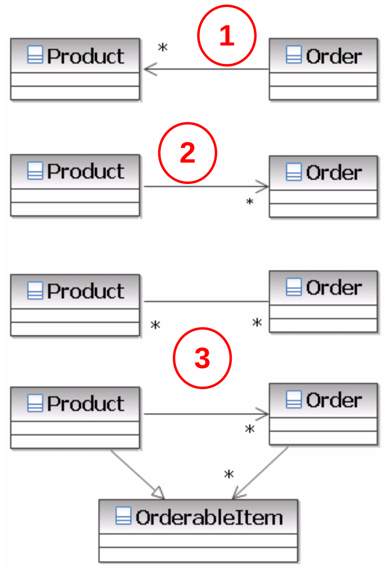
- Restricting navigability reduces dependencies and increases reuse



64

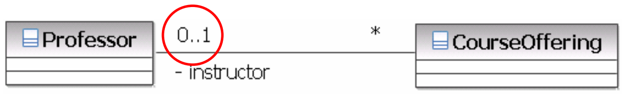
Navigability: Alternatives

1. The total number of orders is small, or we rarely need a list of orders that reference a given product
2. The total number of products is small, or we rarely need a list of products included in a given order
3. The numbers of products and orders are not small and one must be able to navigate in both directions



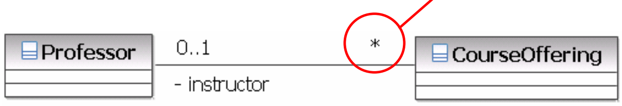
Multiplicity Design

- Multiplicity = 1, or Multiplicity = 0..1
  - May be implemented directly as a simple value or pointer
  - No further "design" is required



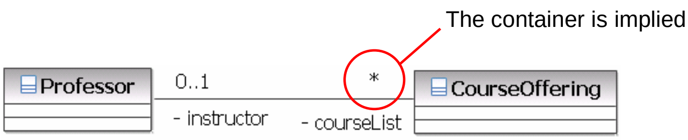
- Multiplicity > 1
  - Cannot use a simple value or pointer
  - Further "design" may be required

Needs a container for CourseOffering objects

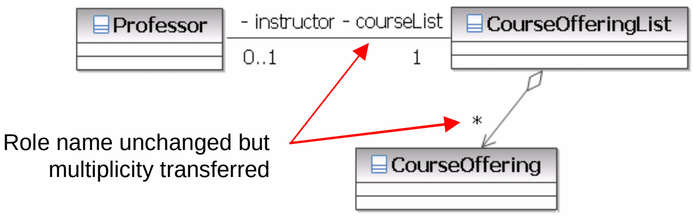


Modeling a Container Class

- The container class may be implied by the multiplicity (n > 1) or it may be explicitly modeled

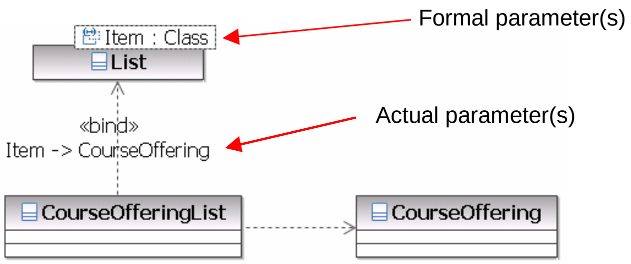


OR



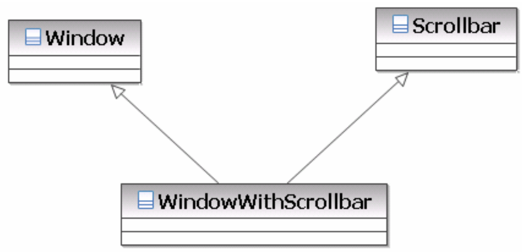
Parameterized Class

- A class definition that defines other classes
- In UML, known as “templates”
- Often used for container classes
  - Sets, lists, dictionaries, stacks, queues
- C++, Java 5



Generalization vs. Aggregation

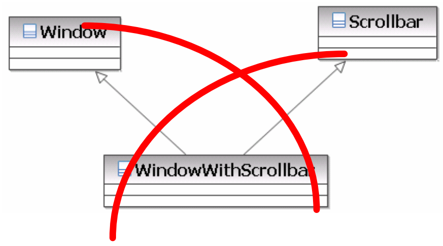
- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship



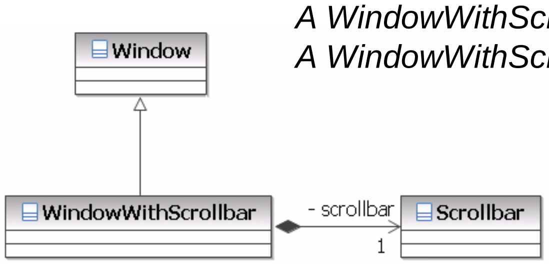
Is this correct?

IBM | IBM Software Group | Rational software

Generalization vs. Aggregation (cont.)



A WindowWithScrollbar "is a" Window
A WindowWithScrollbar "contains a" Scrollbar

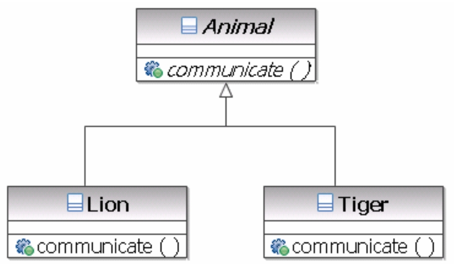


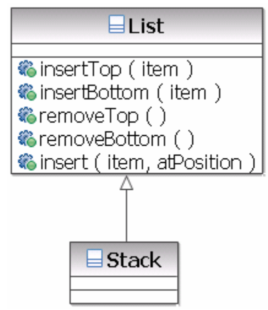
70

IBM Software Group | Rational software

Generalization: Substitution Principle

- Follows the “is a” style of programming
- Liskov Substitution Principle: It should be possible to replace an object of type *T* by any instance of a subtype of *T*





Do these classes follow the “is a” style of programming?

71

A subtype is a type of relationship expressed with inheritance. A subtype specifies that the descendent is a type of the ancestor and must follow the rules of the “is a” style of programming.

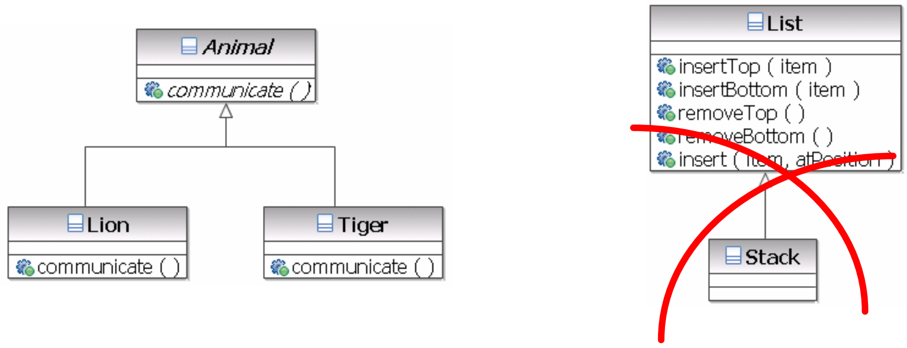
The “is a” style of programming states that the descendent "is a" type of the ancestor and can fill in for all its ancestors in any situation.

The “is a” style of programming passes the Liskov Substitution Principle, which states: “If for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.”

IBM | IBM Software Group | Rational software

Generalization: Substitution Principle (cont.)

- Classes *Lion* and *Tiger* follow the Substitution Principle, not *Stack*



72

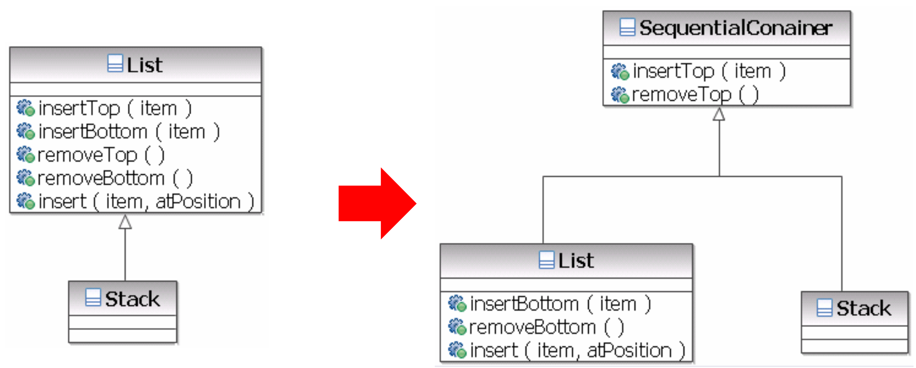
The classes on the left-hand side of the diagram do follow the "is a" style of programming: a Lion is an Animal and a Tiger is an animal.

The classes on the right side of the diagram do *not* follow the “is a” style of programming: a Stack is not a List. Stack needs some of the behavior of a List but not all of the behavior. If a method expects a List, then the operation insert(position) should be successful. If the method is passed a Stack, then the insert (position) will fail.

IBM | IBM Software Group | Rational software

Sharing Implementation: Factoring

- Supports the reuse of the implementation of another class
- Cannot be used if the class you want to "reuse" cannot be changed

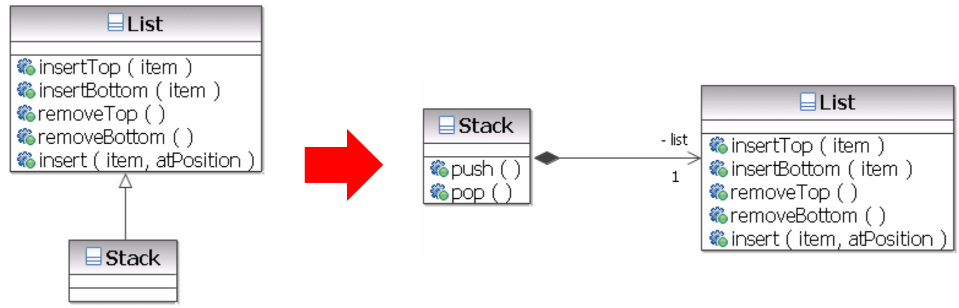


73

IBM | IBM Software Group | Rational software

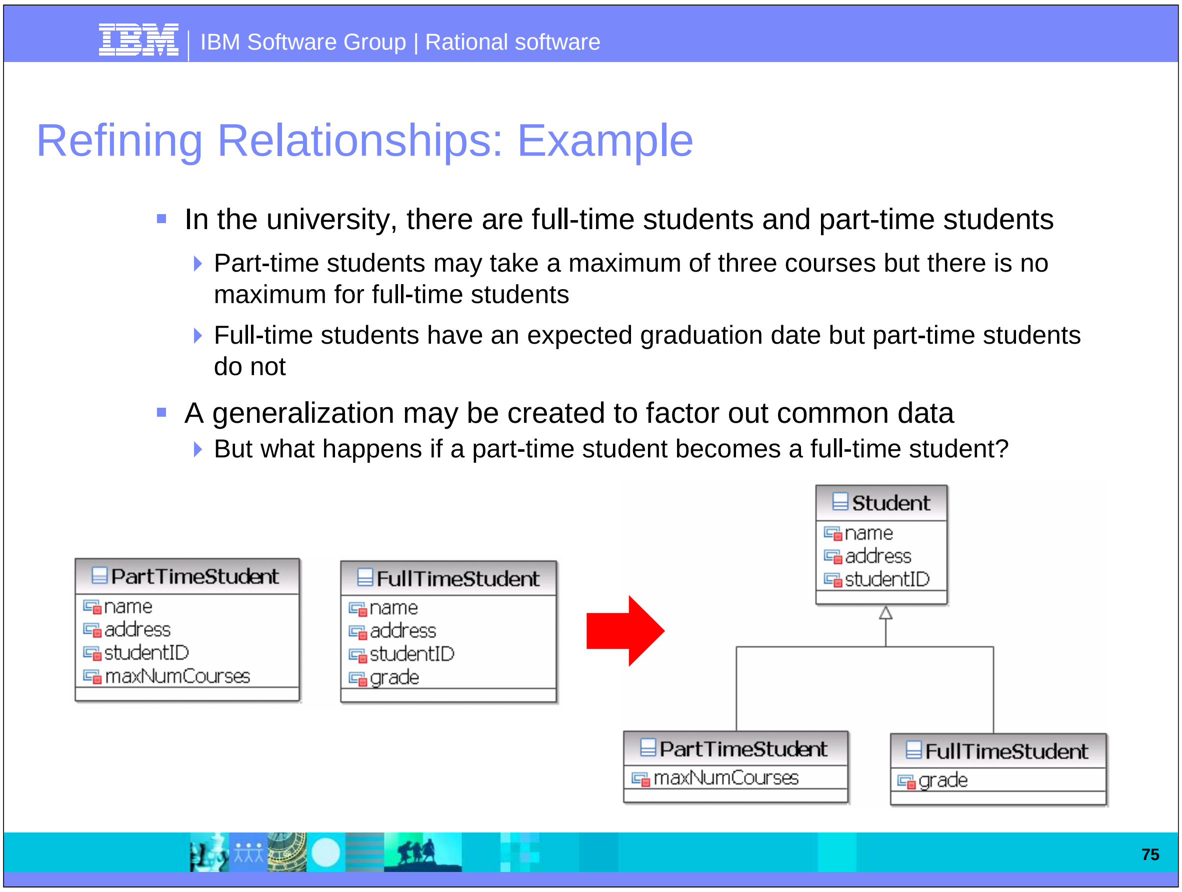
Sharing Implementation: Delegation

- Supports the reuse of the implementation of another class
- Can be used even if the class you want to "reuse" cannot be changed



74

With delegation, you use a composition relationship to "reuse" the desired functionality. All operations that require the "reused" service are "passed through" to the contained class instance.



Changing a student from part-time to full-time involves a non-trivial sequence of steps:

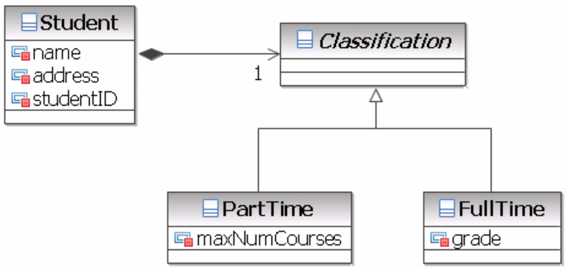
- Creation of an object *FullTimeStudent*.
- Copy of the shared data from *PartTimeStudent* to *FullTimeStudent*.
- Notification to all clients of *PartTimeStudent*.
- Destruction of the *PartTimeStudent* object.

And what happens if in addition there is a requirement to maintain a history of the student.

IBM Software Group | Rational software

Refining Relationships: Example (cont.)

- The solution makes the change from *PartTime* to *FullTime* simple and efficient (no data copy or notifications)
- Now possible to maintain a history by simply changing the composition multiplicity to 1..*
- Added flexibility: e.g. if a student lives on the campus, we could add additional data, such as the room location, in a *ResidentInfo* class with a 0..1 composition from *Student* to *ResidentInfo*



76

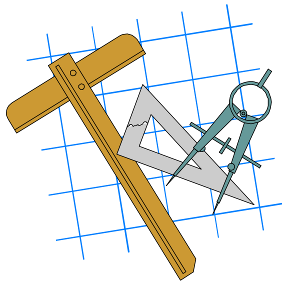
The solution makes the change from *PartTime* to *FullTime* simple and efficient. The data copy and the notifications to clients of *PartTime* are no longer required. It is now possible to maintain a history by simply changing the composition multiplicity to 1..*. A *dateOfChange* attribute can then be added to *Classification* and the history list can be ordered by date.

What’s more, this structure adds to the flexibility of the model: imagine for instance that the student lives on the campus. In this case, we could add additional data, such as the room location, in a *ResidentInfo* class with a 0..1 composition from *Student* to *ResidentInfo.*

<u>Note</u>: The *State* pattern uses this structure in which a class *State* is introduced instead of *Classification*. The aggregate (the equivalent of *Student* in our diagram) can then invoke operations without having to know the current state. When there is a change of state, the aggregate receives a new *State* object, an instance of a subclass of *State*. When a request is received, the aggregate simply invokes the correct operation of *State*, as it is implemented in the subclass.

Exercise (Optional)

- Perform the exercise provided by the instructor (lab 7)



IBM Software Group | Rational Software France

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

13. Subsystem Design

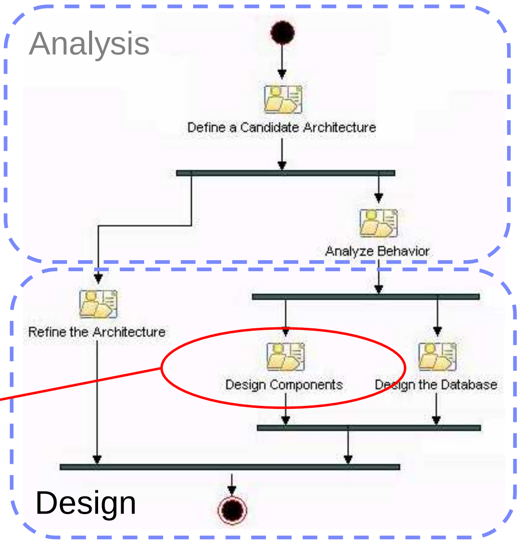
Rational. software

business on demand software

IBM | IBM Software Group | Rational software

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



80

Subsystem Design

- Purpose
  - To incorporate the subsystems in the Design model and document their behavior
- Role
  - Designer
- Major Steps
  - Incorporate the subsystems into the Design model
  - Specify the internal behavior of the subsystems

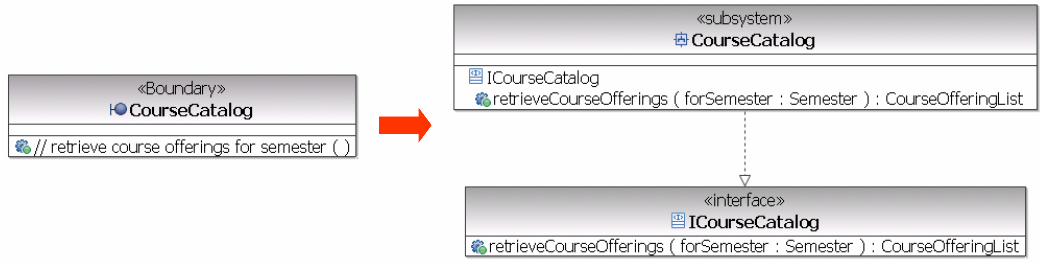
Where Are We?

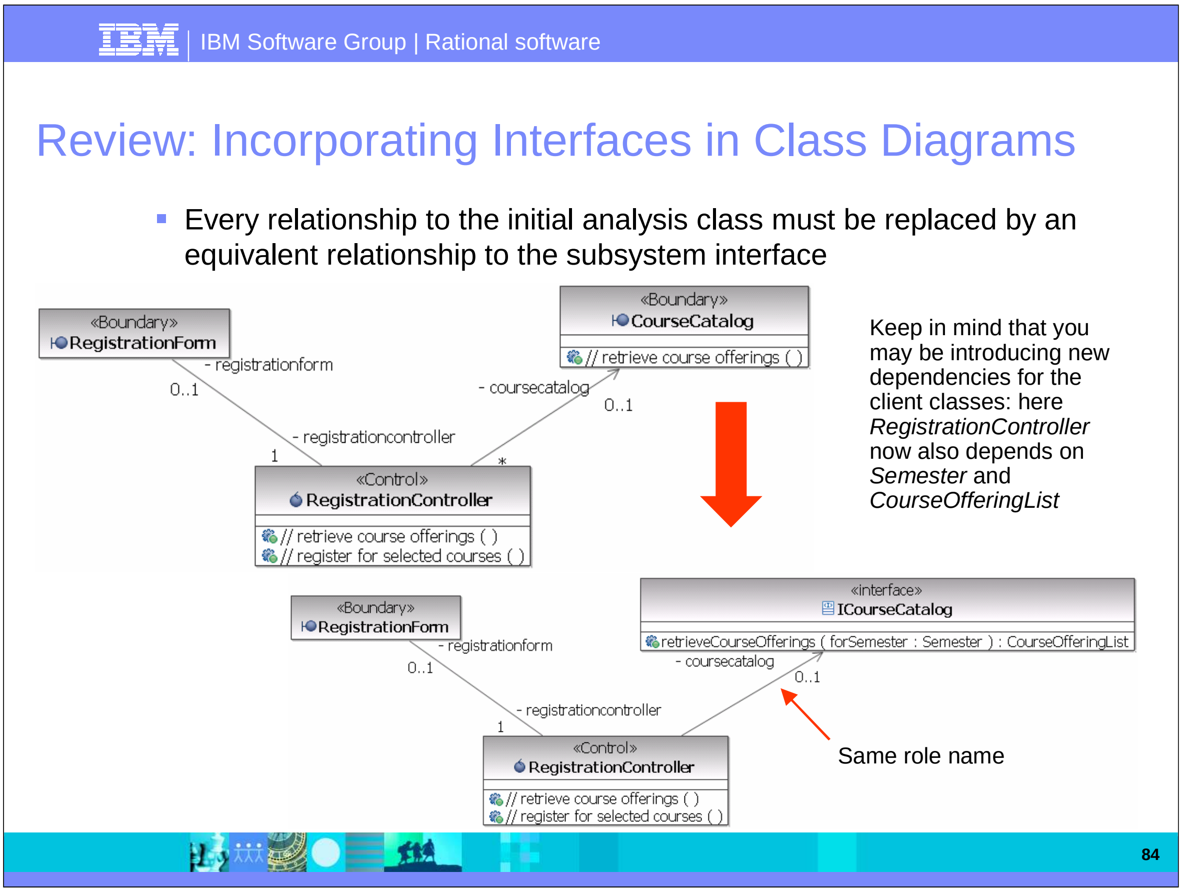
- ➡ Incorporate the subsystems into the Design model
- Specify the internal behavior of the subsystems

Review: Subsystem and Subsystem Interfaces

- Subsystems are components that provide services to their clients only through public interfaces
  - Any two subsystems that realize the same interfaces are interchangeable
  - Subsystems and subsystem interfaces were identified in the *Identify Design Elements* task (module 10)

Analysis *Design*

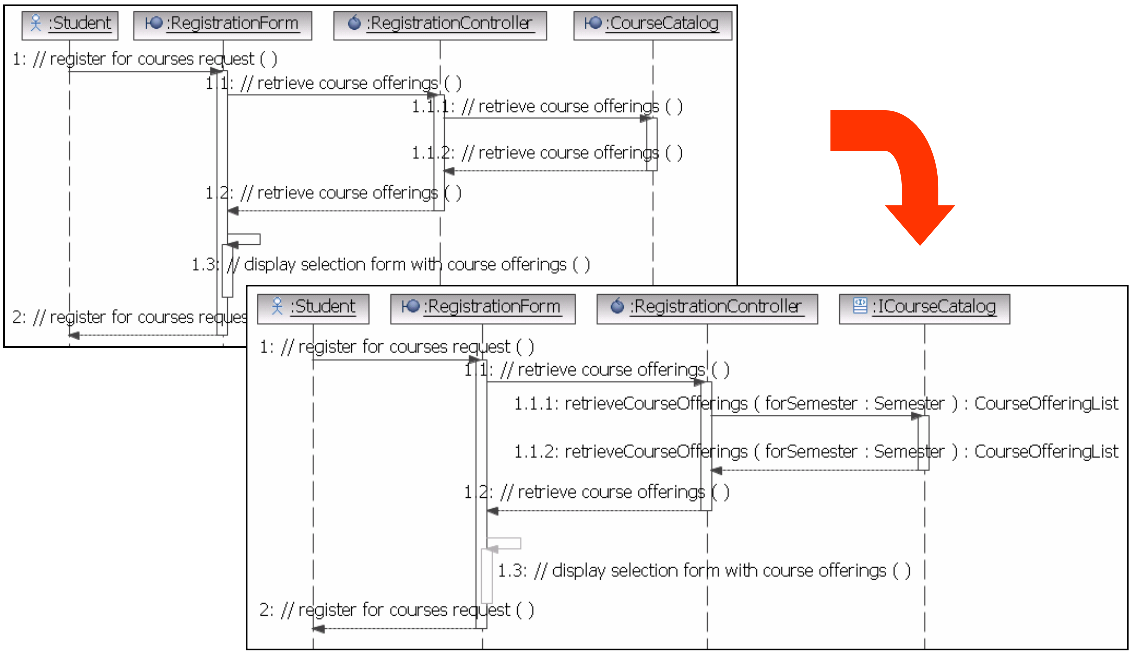




In RSA/RSM, these changes have to be performed manually:

- Retrieve the interface to use and drag it to the diagram
- Select the relationship and move the target end from the analysis class to the interface
- Delete the analysis class from the diagram
- Delete the analysis class from the design model after all changes have been made

Review: Incorporating Interfaces in Sequence Diagrams

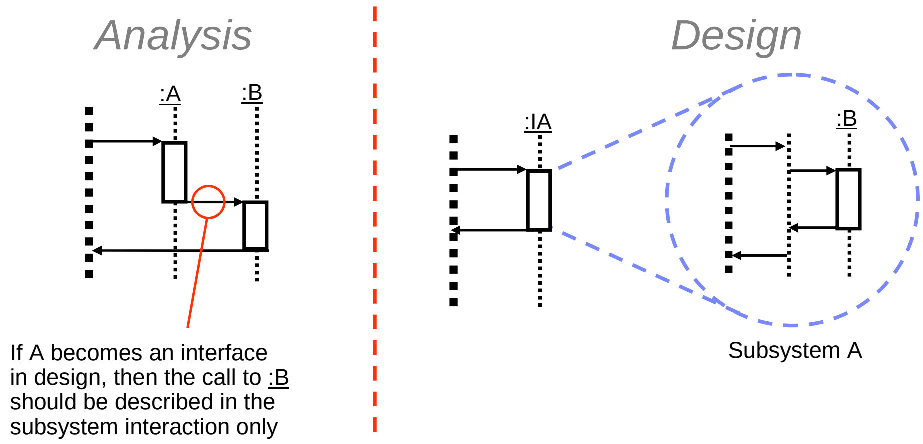


In RSA/RSM, simply drag the interface over the analysis object and update the message.

IBM Software Group | Rational software

Encapsulating Subsystem Interactions

- Subsystem interactions must be described in their own interaction diagrams (next topic)
- Imagine we have an analysis interaction involving an analysis class (A) that is converted to an interface IA



IBM | IBM Software Group | Rational software

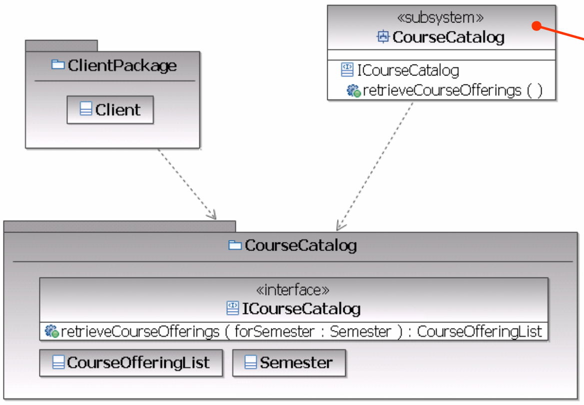
Where Are We?

- Incorporate the subsystems into the Design model
- Specify the internal behavior of the subsystems

87

Internal Behavior of Subsystems

- So far, we have only reasoned in terms of the outside view of the subsystems (the interfaces)
- We are now looking at the internal behavior
  - Keep in mind, encapsulation is the key: the client is completely independent of the subsystems that provide the implementation



Subsystems are similar to packages in the sense they contain other design elements:
- At least one of those design elements will "realize" the interface(s)
- Design elements inside a subsystem are never public

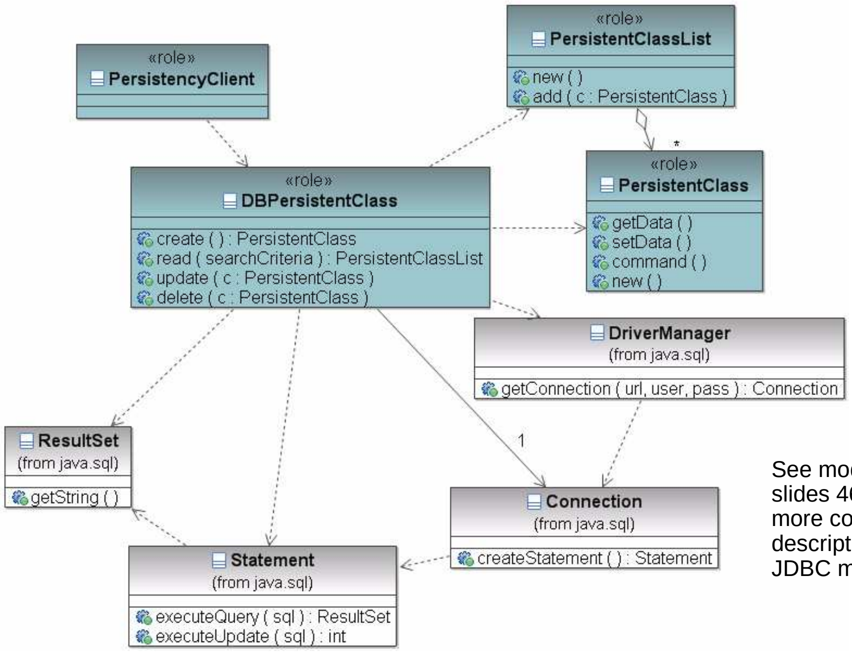
IBM | IBM Software Group | Rational software

Internal Behavior of Subsystems (cont.)

- Similar to any collaboration
  - One (or more) interaction diagram(s) for *each* service provided by the subsystem
  - One (or more) class diagram(s) showing the classes involved in the implementation of the services
  - To illustrate this discussion, we will use the *CourseCatalog* subsystem
    - In *Identify Design Mechanisms*, we described a JDBC Mechanism to apply to persistent classes

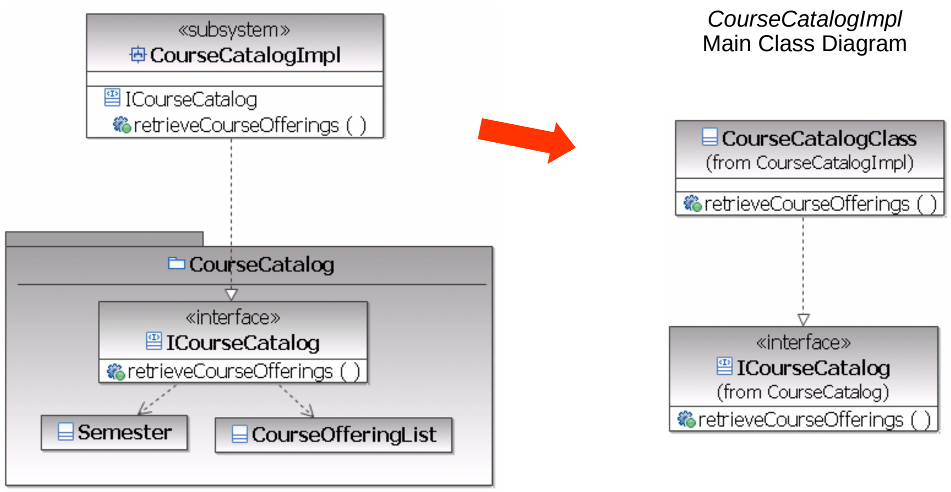
89

Review: The JDBC Persistency Mechanism



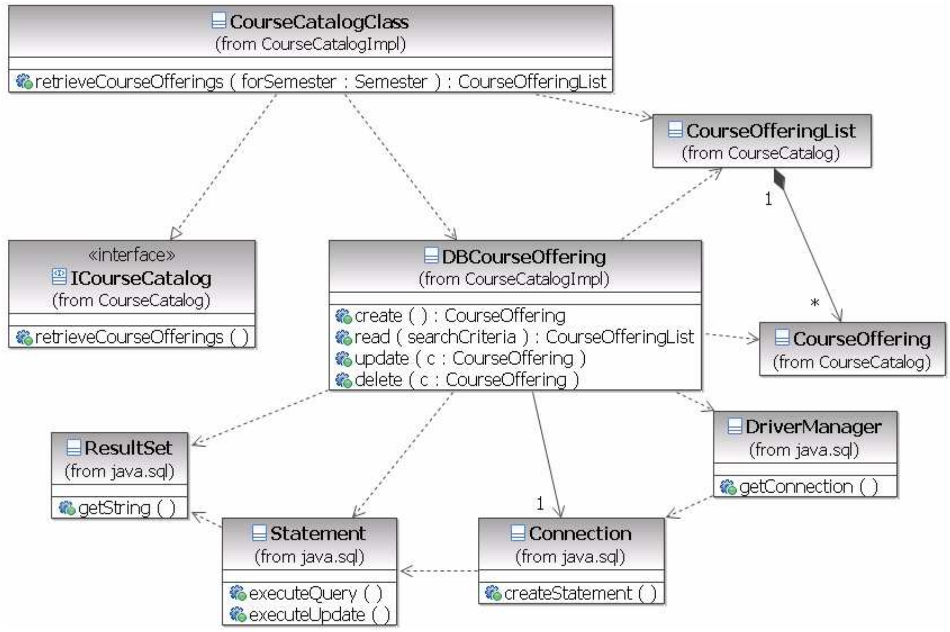
See module 11, slides 46 to 48 for a more complete description of the JDBC mechanism

Step 1: Create the Class That Realizes the Interface

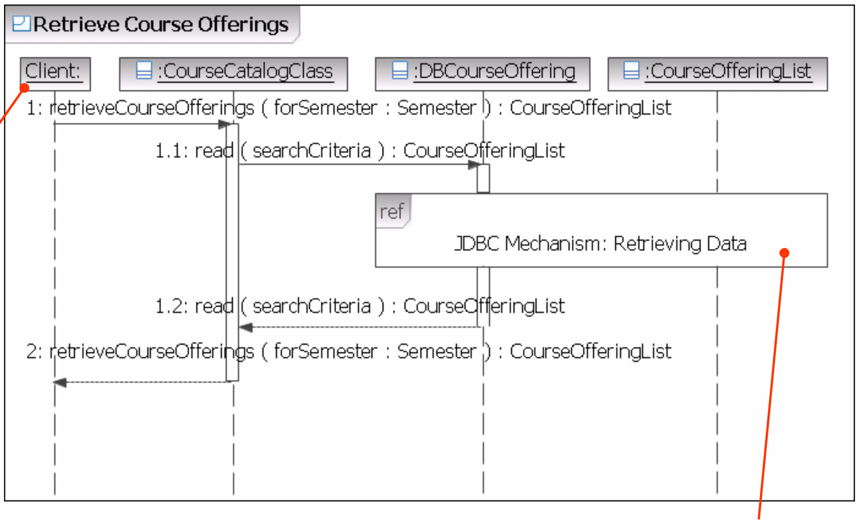


91

Step 2: Incorporate the JDBC Mechanism



Retrieve Course Offerings Interaction Diagram



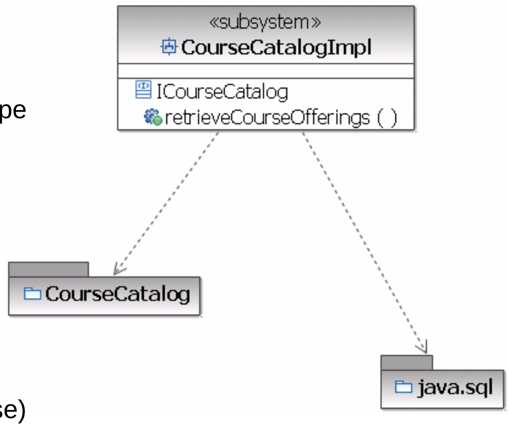
Note the use of an anonymous object to represent the client

Rather than duplicating the information, we make use of the Interaction Use UML2 construct

IBM Software Group | Rational software

Controlling Dependencies

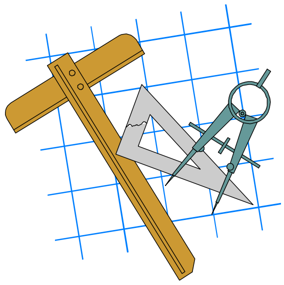
- Controlling dependencies is critical for the architecture (see slides in module 8 about component-based architectures and in module 9 about layered architectures)
- Dependencies result from:
  - Relationships from one element to another
  - References to another element in an operation parameters and/or return type
  - References to another element in an attribute type
- In the case of our subsystem, it was very easy to determine the dependencies from our subsystem to other components
  - For a complete system, you need to automate this processing (see exercise)



94

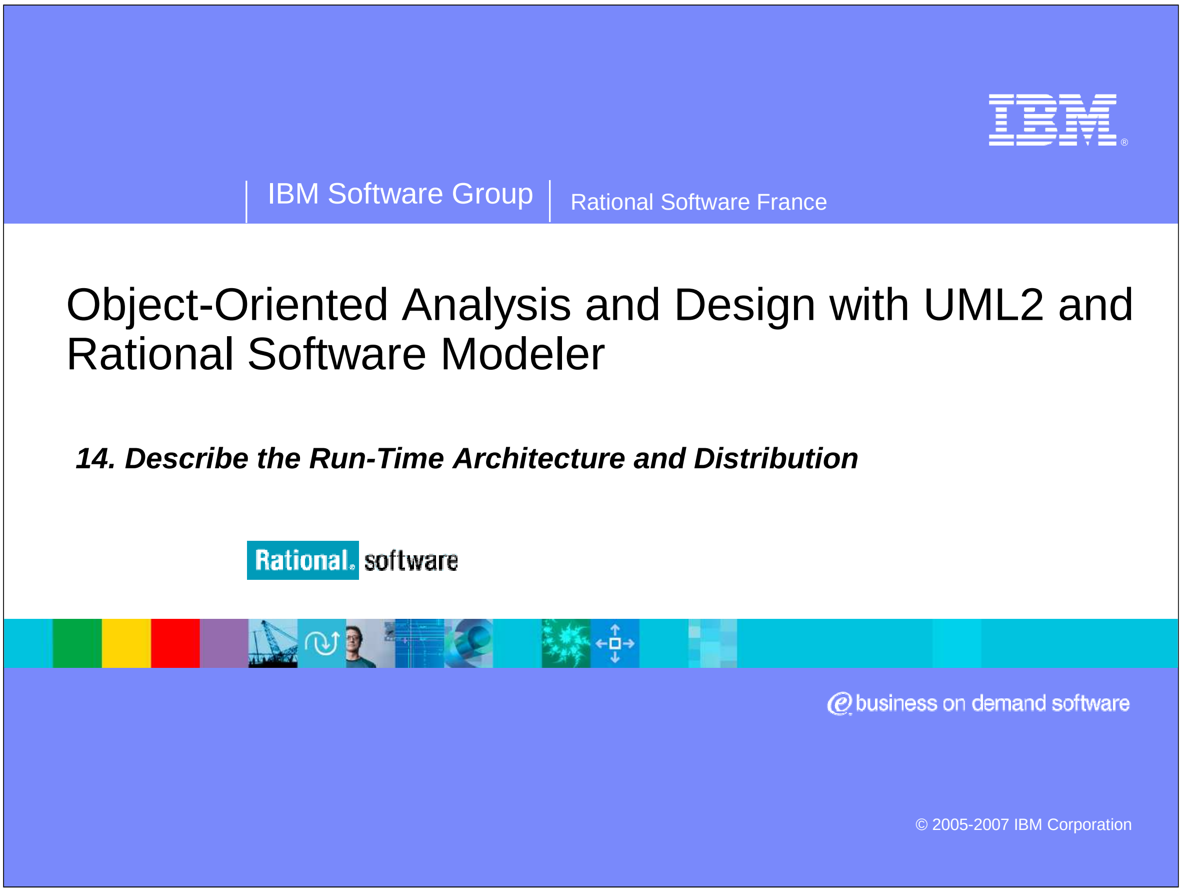
Exercise

- Perform the exercise provided by the instructor (lab 8)



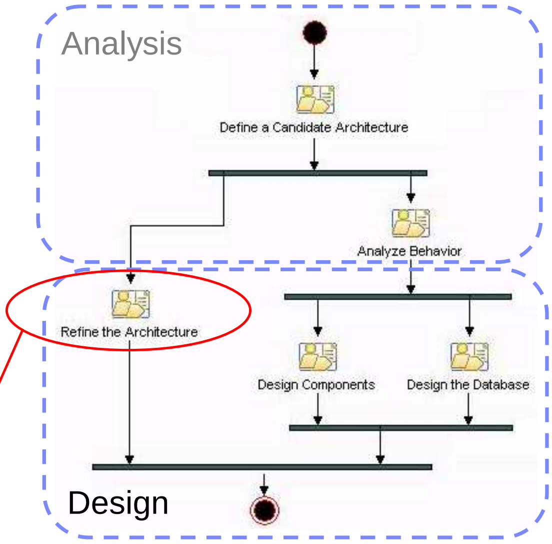
IBM Software Group | Rational software

96


IBM Software Group
Rational Software France
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler
14. Describe the Run-Time Architecture and Distribution
Rational. software
business on demand software
© 2005-2007 IBM Corporation

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



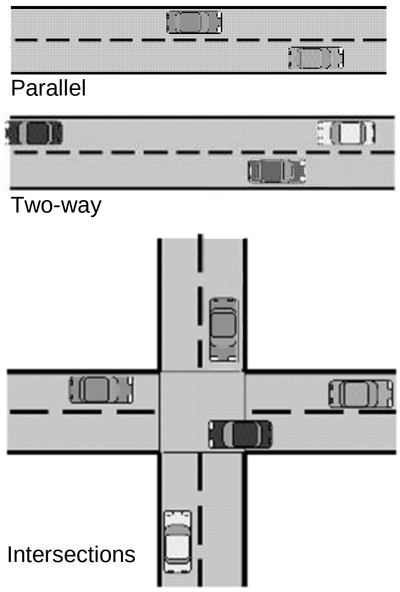
Where Are We?

- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Concurrency Control
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations

IBM Software Group | Rational software

What Is Concurrency?

- The performance of two or more activities during the same time interval
- Example of concurrency at work:
  - Parallel roads require little coordination
  - Two-way roads require some coordination for safe interaction
  - Intersections require careful coordination



100

Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon, of course. In the real world, at any given time many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency.

When dealing with concurrency issues in software systems, you must consider two important aspects:

- Being able to detect and respond to external events occurring in a random order.
- Ensuring that these events are responded to in some minimum required interval.

IBM Software Group | Rational software

Why Do We Need Concurrency?

- Some reasons for concurrency
  - Reactive software systems:
    - Many systems must respond to externally generated events which may occur at somewhat random times, in some-what random order, or both
  - Optimized processing time
    - Executing tasks in parallel
    - Preventing one activity from blocking another while waiting for I/O
  - Controllability of the system
    - Ability to start, stop, or otherwise influence in mid-stream a system function
- Concurrent software permits a “separation of concerns” among concurrent activities
- But, when concurrent activities interact or share the same resources, concurrency issues will arise
  - Lost updates, race conditions, deadlocks, etc.

101

Some of the driving forces behind finding ways to manage concurrency are external. That is, they are imposed by the demands of the environment. In real-world systems, many things are happening simultaneously and must be addressed “in real-time” by software. To do so, many real time software systems must be “reactive.” They must respond to externally generated events that might occur at somewhat random times, in somewhat random order, or both.

There also can be internally inspired reasons for concurrency. For example, performing tasks in parallel can substantially speed up the computational work of a system if multiple CPUs are available. Even within a single processor, multitasking can dramatically speed things up by preventing one activity from blocking another while waiting for I/O. A common situation in which this occurs is during the startup of a system. There are often many components, each of which requires time to be made ready for operation. Performing these operations sequentially can be painfully slow.

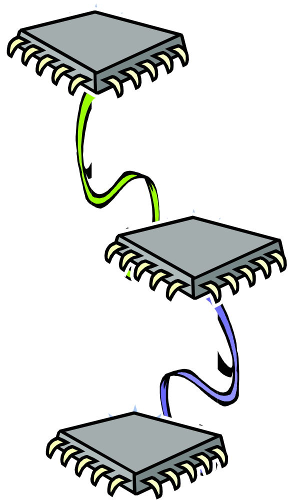
Controllability of the system can also be enhanced by concurrency. For example, one function can be started, stopped, or otherwise influenced in midstream by other concurrent functions — something extremely difficult to accomplish without concurrent components.

If each concurrent activity evolved independently, in a truly parallel fashion, managing them would be relatively simple: we could just create separate programs to deal with each activity. However, this is not the case. The challenges of designing concurrent systems arise mainly because of the interactions that happen between concurrent activities. When concurrent activities interact, some sort of coordination is required.

IBM Software Group | Rational software

Realizing Concurrency: Concurrency Mechanisms

- To support concurrency, a system must provide for multiple threads of control
- Common concurrency mechanisms
  - Multitasking
    - The operating systems simulate concurrency on a single CPU by interleaving the execution of different tasks
  - Multiprocessing
    - Multiple CPUs execute concurrently
  - Application-based solutions
    - The application software takes responsibility for switching between different branches of code at appropriate times



102

Of course, multiple processors offer the opportunity for truly concurrent execution. Most commonly, each task is permanently assigned to a process in a particular processor, but under some circumstances tasks can be dynamically assigned to the next available processor. Perhaps the most accessible way of doing this is by using a “symmetric multiprocessor.” In such a hardware configuration, multiple CPUs can access memory through a common bus.

Operating systems that support symmetric multiprocessors can dynamically assign threads to any available CPU. Examples of operating systems that support symmetric multiprocessors are SUN’s Solaris and Microsoft’s Windows NT.

IBM Software Group | Rational software

Concurrency Requirements

- Concurrency requirements are driven by:
  - The degree to which the system must be distributed
  - The degree to which the system is event-driven
  - The computation intensity of key algorithms
  - The degree of parallel execution supported by the environment
- Concurrency requirements are ranked in terms of importance to resolve conflicts

103

Concurrency requirements define the extent to which parallel execution of tasks is required for the system. These requirements help shape the architecture.

A system whose behavior must be distributed across processors or nodes virtually requires a multi-process architecture. A system that uses some sort of Database Management System or Transaction Manager also must consider the processes that those major subsystems introduce.

If dedicated processors are available to handle events, a multi-process architecture is probably best. On the other hand, to ensure that events are handled, a uni-process architecture may be needed to circumvent the “fairness” resource-sharing algorithm of the operating system: It may be necessary for the application to monopolize resources by creating a single large process, using threads to control execution within that process.

In order to provide good response times, it might be necessary to place computationally intensive activities in a process or thread of their own so that the system still is able to respond to user inputs while computation takes place, albeit with fewer resources. If the operating system or environment does not support threads (lightweight processes), there is little point in considering their impact on the system architecture.

The above requirements are mutually exclusive and might conflict with one another. Ranking requirements in terms of importance will help resolve the conflict.

Example: Course Registration System

- In the Course Registration System, the concurrency requirements come from the requirements and the architecture:
  - Multiple users must be able to perform their work concurrently
  - If a course offering becomes full while a student is building a schedule including that offering, the student must be notified
  - Risk-based prototypes have found that the legacy course catalog database cannot meet our performance needs without some creative use of mid-tier processing power

The above concurrency requirements were documented in the Course Registration System Supplemental Specification.

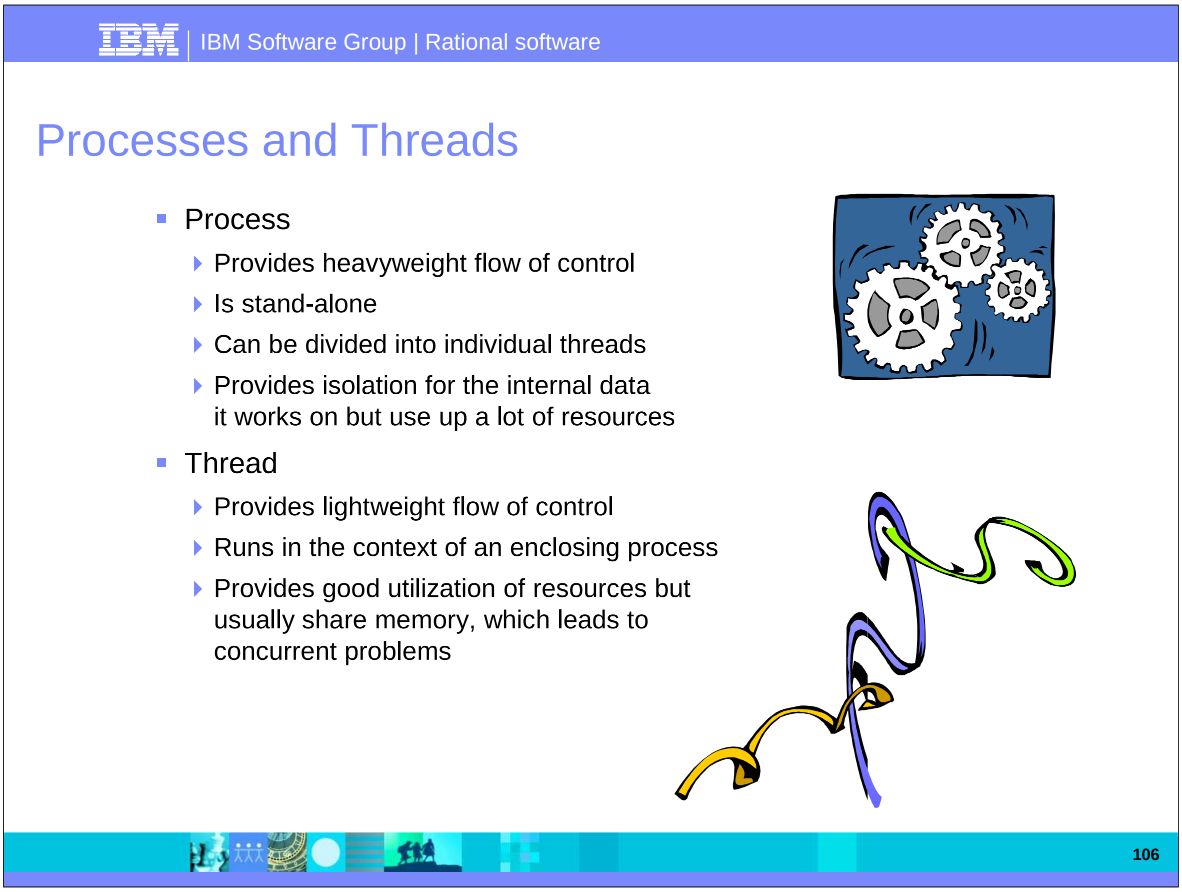
The first requirement is typical of any system, but the multi-tier aspects of our planned architecture will require some extra thought for this requirement.

The second requirement demonstrates the need for a shared, independent process that manages access to the course offerings.

The third issue leads us to use some sort of mid-tier caching or preemptive retrieval strategy.

Where Are We?

- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Dealing With Concurrency Problems
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations



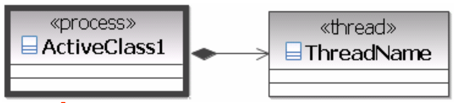
When the operating system provides multitasking, a common unit of concurrency is the process. A process is an entity provided, supported, and managed by the operating system whose sole purpose is to provide an environment in which to execute a program. The process provides a memory space for the exclusive use of its application program, a thread of execution for executing it, and perhaps some means for sending messages to and receiving them from other processes. In effect, the process is a virtual CPU for executing a concurrent piece of an application.

Many operating systems, particularly those used for real-time applications, offer a “lighter weight” alternative to processes, called “threads” or “lightweight threads.”

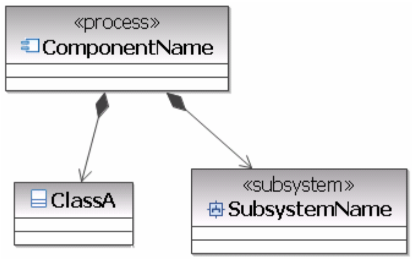
Threads are a way of achieving a slightly finer granularity of concurrency within a process. Each thread belongs to a single process, and all the threads in a process share the single memory space and other resources controlled by that process.

Modeling Processes

- Processes can be modeled using
  - Active classes (Class Diagrams) and Objects (Interaction Diagrams)
  - Components (Component Diagrams)
  - Stereotype <<process>>
- Process relationships can be modeled as dependencies
- Threads can be modeled using
  - Regular classes
  - Stereotype <<thread>>
- Process to thread and process/thread to class/subsystem modeled as compositions



An active class is a class that "owns" its own thread of execution
(this is not the standard UML2 representation)



You can use "active" classes to model processes and threads. An active class is a class that "owns" its own thread of execution and can initiate control activity, contrasted with passive classes that can only be acted upon. Active classes can execute in parallel (that is, concurrently) with other active classes.

The model elements can be stereotyped to indicate whether they are processes (<<process>> stereotype) or threads (<<thread>> stereotype).

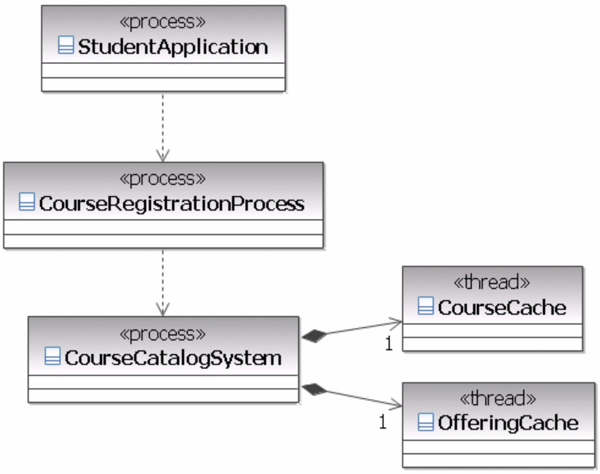
Note: Even though you use "active" classes to model processes and threads, they are classes only in the meta-modeling sense. They aren't the same kind of model elements as classes. They are only meta-modeling elements used to provide an address space and a run-time environment in which other class instances execute, as well as to document the process structure. If you try to take them further than that, confusion may result.

Process communication is modeled using dependency relationship whether you use classes or components to represent your processes.

In cases where the application has only one process, the processes may never be explicitly modeled. As more processes or threads are added, modeling them becomes important.

Example: Course Registration System

- *StudentApplication*
  - One instance for each student who is currently registering for courses
- *CourseRegistrationProcess*
  - One instance for each student who is currently registering for courses
- *CourseCatalogSystem*
  - Separate process that can be shared by multiple users registering for courses
    - The threads are used to asynchronously retrieve items from the legacy system



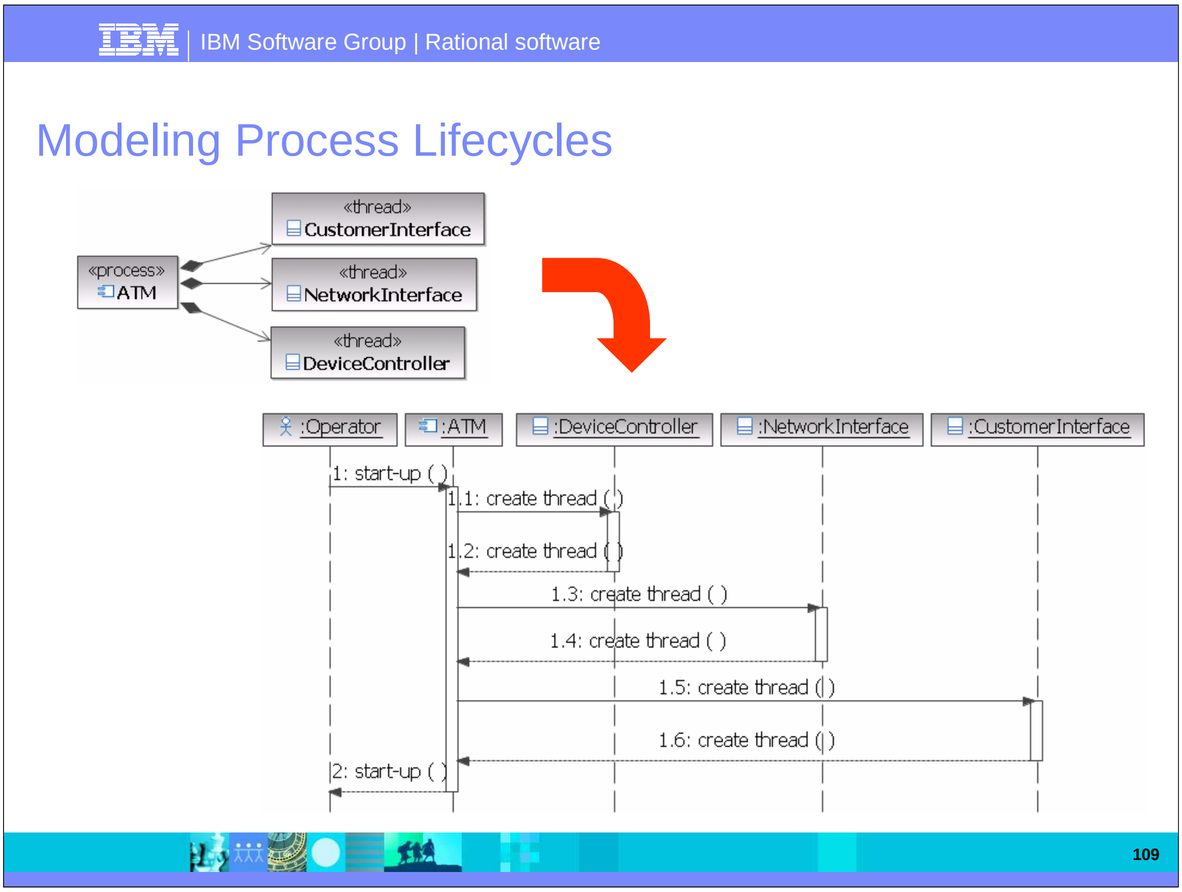
The above example demonstrates how processes and threads are modeled. Processes and threads are represented as stereotyped classes. Separate processes have dependencies among them. When there are threads within a process composition is used. The composition relationship indicates that the threads are contained within the process (that is, cannot exist outside of the process).

The StudentApplication process manages the student functionality, including user interface processing and coordination with the business processes. There is one instance of this process for each student who is currently registering for courses.

The CourseRegistrationProcess encapsulates the course registration processing. There is one instance of this process for each student who is currently registering for courses.

The CourseRegistrationProcess talks to the separate CourseCatalogSystemAccess process, which manages access to the legacy system. CourseCatalogSystemAccess is a separate process that can be shared by multiple users registering for courses. This allows for a cache of recently retrieved courses and offerings to improve performance.

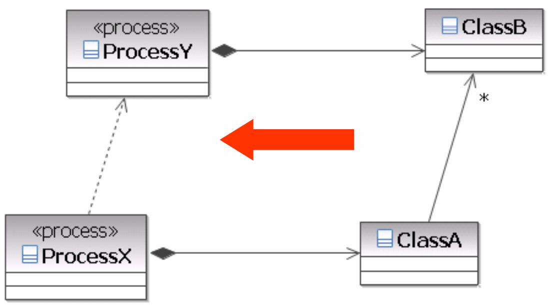
The separate threads within the CourseCatalogSystemAccess process, CourseCache, and OfferingCache are used to asynchronously retrieve items from the legacy system. This improves response time.



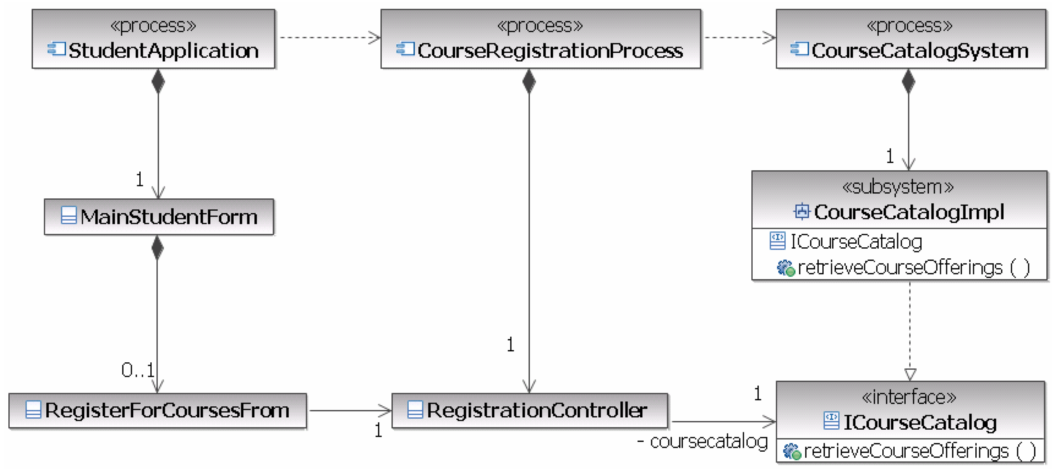
In the Automated Teller Machine, asynchronous events must be handled coming from three different sources: the user of the system, the ATM devices (in the case of a jam in the cash dispenser, for example), or the ATM Network (in the case of a shutdown directive from the network). To handle these asynchronous events, we can define three separate threads of execution within the ATM itself, as shown below using active classes in UML.

Modeling Process Relationships

- Process relationships can be modeled as dependencies
- Process relationships must support design element relationships



Example: Course Registration System



Where Are We?

- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Concurrency Control
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations

Dealing With Concurrency Problems

- About concurrency problems:
  - Difficult to enumerate the possible scenarios
  - Hard to test for
  - Difficult to reproduce
- Two main situations
  - Loss of data during the execution of database transactions
    - Example: two sessions S1 and S2 read the same record holding a value "X", S1 appends a "Y" to the data and commits the result ("XY"), S2 appends a "Z" to the data and commits the result ("XZ") overwriting S1's update (lost update)
  - Incorrect results generated during the concurrent execution of multiple interacting computational tasks (concurrent computing)
    - Example: if two threads T1 and T2, which increment the value of a global integer by one, run simultaneously without locking or synchronization, the result can be 1 or 2 (race condition)

Software flaws in Life-critical systems can be disastrous. Race conditions were among the flaws in the Therac-25 radiation therapy machine, which led to the death of five patients and injuries to several more. Another example is the Energy Management System provided by GE Energy and used by Ohio-based FirstEnergy Corp. (and by many other power facilities as well). A race condition existed in the alarm subsystem; when three sagging power lines were tripped simultaneously, the condition prevented alerts from being raised to the monitoring technicians, delaying their awareness of the problem. This software flaw eventually led to the North American Blackout of 2003. (GE Energy later developed a software patch to correct the previously undiscovered error.)

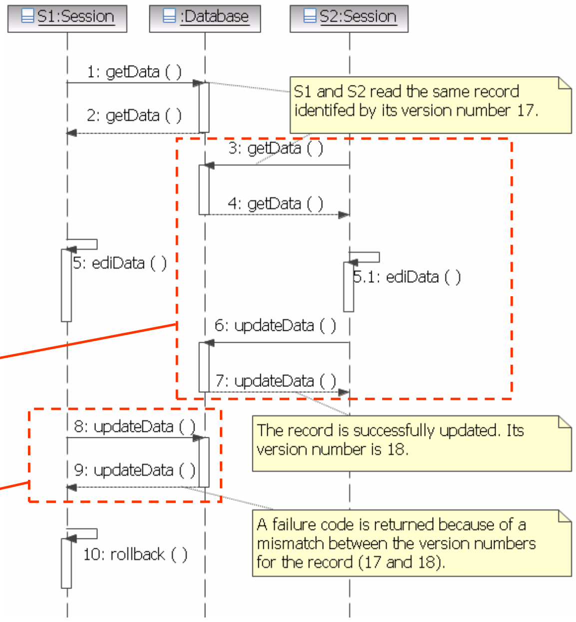
(Source: Wikipedia 2007)

Concurrency Control (in the Field of Databases)

- Purpose
  - To ensure that database transactions are executed in a safe manner
- Two main forms of concurrency control:
  - Optimistic lock
    - Conflict detection scheme
  - Pessimistic lock
    - Conflict prevention scheme
    - Can lead to deadlock situations

Optimistic Lock Pattern

- Source: Martin Fowler, Optimistic Offline Lock in Patterns of Enterprise Application Architecture, Addison Wesley, 2003



Business Transaction Boundary

System Transaction Boundary

Where Are We?

- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Concurrency Control
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations

Client/server is a conceptual way of breaking up the application into service requestors (clients) and service providers (servers).

A client often services a single user and often handles end-user presentation services (GUIs). A system can consist of several different types of clients, examples of which include user workstations and network computers.

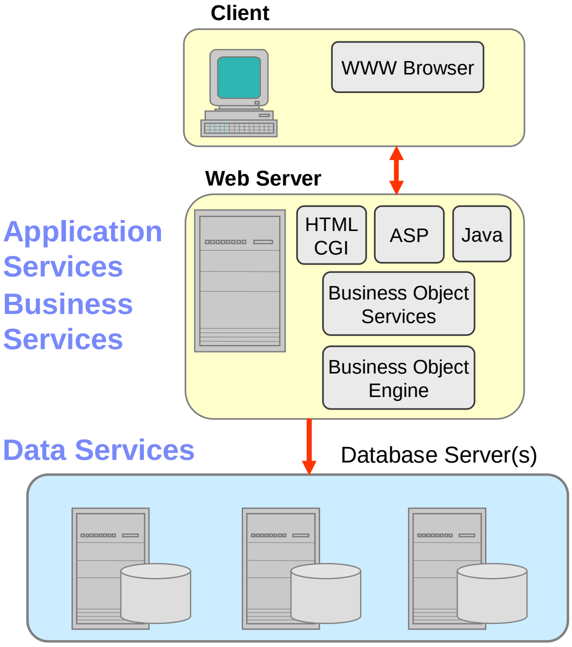
The server usually provides services to several clients simultaneously. These services are typically database, security, or print services. A system can consist of several different types of servers. For example: *database servers*, handling database machines such as Oracle, DB2; *print servers*, handling the driver logic, such as queuing for a specific printer; *communication servers* (TCP/IP, ISDN, X.25); *window manager servers* (X); and *file servers* (NFS under UNIX).

The application and business logic is distributed among both the client and the server (application partitioning).

IBM Software Group | Rational software

Client/Server Architectures

- Typical applications include
  - Application Services
  - Business Services
  - Data Services
- Different types of architectures based on how these services are allocated to processing nodes, for instance:
  - Two-Tier “Fat Client” Architecture
  - Three-Tier Architecture
  - Web Application Architecture



117

Fat client distribution pattern: Much of the functionality in the system runs on the client.

Three-tier architecture: The system is divided into three logical partitions: application services, business services, and data services. The “logical partitions” may in fact map to three or more physical nodes.

Application services, primarily dealing with GUI presentation issues, tend to execute on a dedicated desktop workstation with a graphical, windowing operating environment.

Data services tend to be implemented using database server technology, which normally executes on one or more high-performance, high-bandwidth nodes that serve hundreds or thousands of users, connected over a network.

Business services are typically used by many users in common, so they tend to be located on specialized servers as well, although they may reside on the same nodes as the data services.

Partitioning functionality along these lines provides a relatively reliable pattern for scalability: by adding servers and rebalancing processing across data and business servers, a greater degree of scalability is achieved.

At the other end of the spectrum from the fat client is the typical **Web Application** (which might be characterized as fat server or “anorexic client”). Since the client is simply a Web browser running a set of HTML pages and Java applets, Java Beans, or ActiveX components, there is very little application there at all. Nearly all work takes place on one or more Web servers and data servers.

Web applications are easy to distribute and easy to change. They are relatively inexpensive to develop and support (since much of the application infrastructure is provided by the browser and the web server). However, they might not provide the desired degree of control over the application, and they tend to saturate the network quickly if not well-designed (and sometimes despite being well-designed).

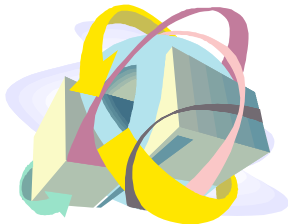
Where Are We?

- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Concurrency Control
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations

IBM Software Group | Rational software

Process-to-Node Allocation Considerations

- Client/Server architecture
- Response time and system throughput
- Minimization of cross-network traffic
- Node capacity
- Communication medium bandwidth
- Availability of hardware and communication links
- Rerouting requirements



119

Processes must be assigned to a hardware device for execution in order to distribute the workload of the system.

Those processes with fast response time requirements should be assigned to the fastest processors.

Processes should be allocated to nodes so as to minimize the amount of cross-network traffic. Network traffic, in most cases, is quite expensive. It is an order of magnitude or two slower than inter-process communication. Processes that interact to a great degree should be co-located on the same node. Processes that interact less frequently can reside on different nodes. The crucial decision, and one that sometimes requires iteration, is where to draw the line.

Additional considerations:

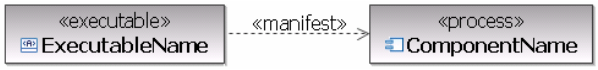
- Node capacity (in terms of memory and processing power)
- Communication medium bandwidth (bus, LANs, WANs)
- Availability of hardware and communication links
- Rerouting requirements for redundancy and fault-tolerance

IBM Software Group | Rational software

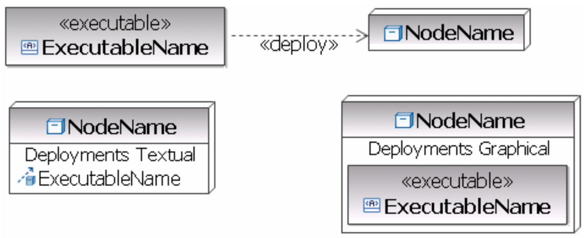
Modeling the Allocation of Processes to Nodes

- Processes are typically represented as components stereotyped <<process>>

- Processes will be rendered in the physical world as executables
  - An executable will be represented as an artifact stereotyped <<executable>>



- Executables will be deployed to processing nodes



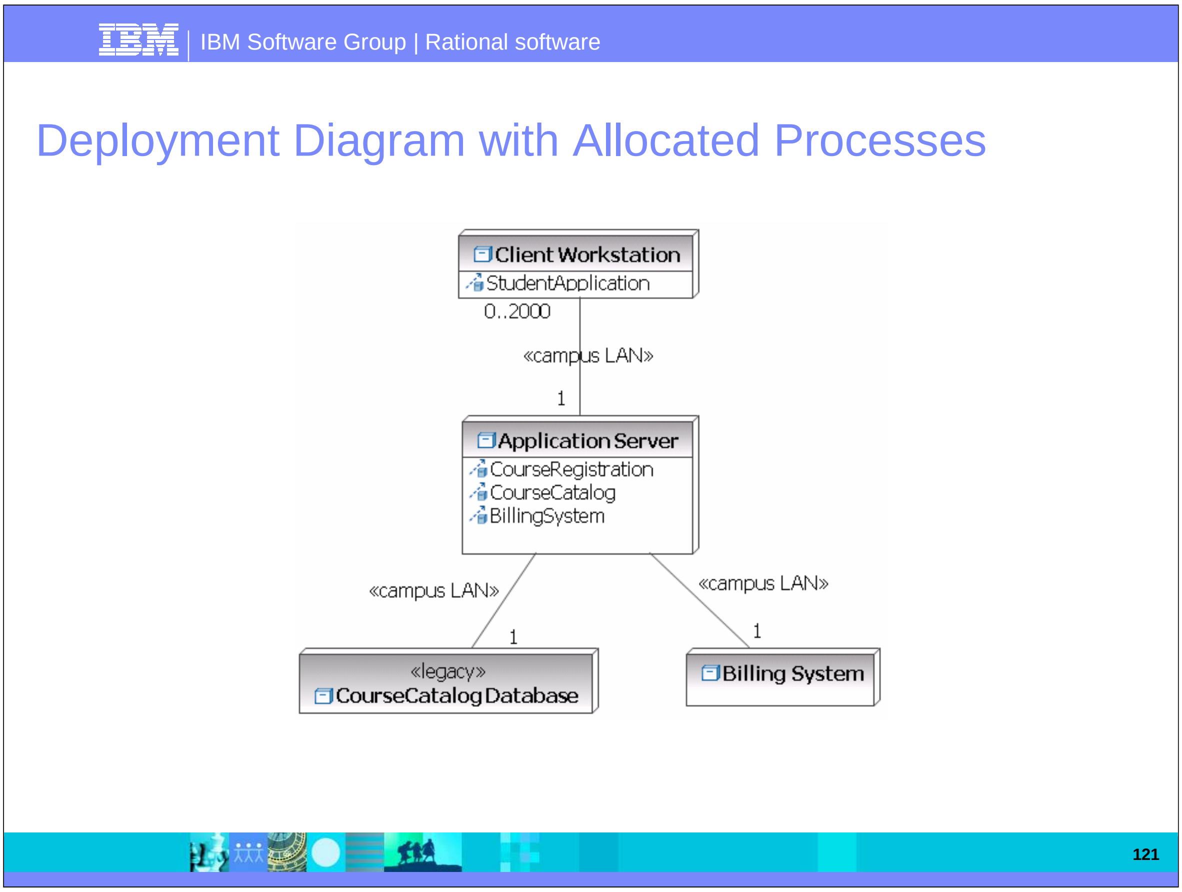
(the three representations above are equivalent)

120

Deployment diagrams allow you to capture the topology of the system nodes, including the assignment of run-time elements to them.

A deployment diagram contains nodes connected by associations. The associations indicate a communication path between the nodes.

Nodes may contain artifacts which indicates that the artifact lives on or runs on the node. An example of a run-time object is a process.



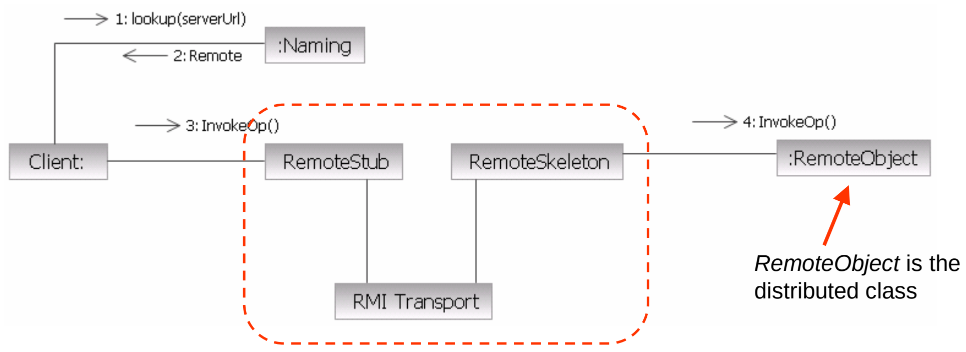
The above diagram once again illustrates the Deployment View for the Course Registration System. Note: No threads are shown in the above diagram, because threads always run in the context of a process.

Where Are We?

- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Concurrency Control
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations

Example of a Distribution Mechanism: Java RMI

- RMI = Remote Method Invocation



Remote Method Invocation (RMI) is a Java-specific mechanism that allows client objects to invoke operations on server objects as if they were local. The only catch is that, with basic RMI, you must know where the server object resides.

The mechanisms of invoking an operation on a remote object are implemented using “proxies” on the client and server, as well as a service that resides on both that handles the communication.

The client establishes the link with the remote object via the *Naming* utility that is delivered with RMI. There is a single instance of the *Naming* class on every node. The *Naming* instances communicate with one another to locate remote objects. Once the connection is established (via *lookup()*), it may be reused any time the client needs to access the remote object.

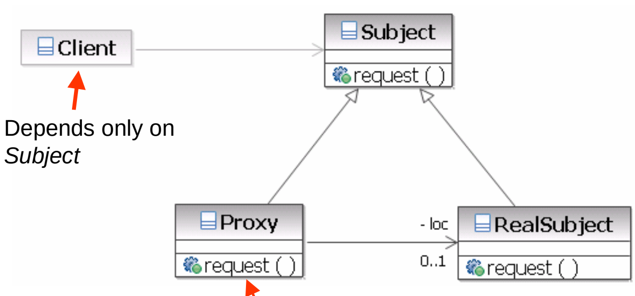
RemoteStub and *RemoteSkeleton* are automatically generated. To get them, you run the compiled distributed class through the *rmic* compiler to generate the stubs and skeletons. You then must add the code to look up the object on the server. The lookup returns a reference to the auto-generated *RemoteStub*.

For example, say we had a class, *ClassA*, that is distributed through RMI. Once *ClassA* is created, it is run through the *rmic* compiler, which generates the stub and skeleton. When you do the lookup, the *Naming* object returns a reference to a *ClassA*, but it is really a *ClassA* stub. Thus, no client adjusting needs to happen. Once a class is run through *rmic*, you can access it as if it were a local class, the client does not know the difference.

IBM | IBM Software Group | Rational software

Using the Proxy Design Pattern

- A proxy is a placeholder for another object to control access to it
- Applicability
  - Remote proxy (our example)
  - Virtual proxy (creates “expensive” objects on demand)
  - Etc.

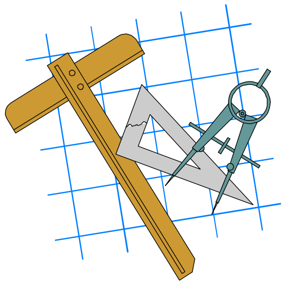



```
if (loc == null) {
   // retrieve copy of the RealSubject instance
   // from a remote site
   loc = …;
}
loc.request();
```

124

Exercise

- Perform the exercise provided by the instructor (lab 9)



IBM Software Group | Rational software

IBM Software Group | Rational Software France

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

15. Design the Database

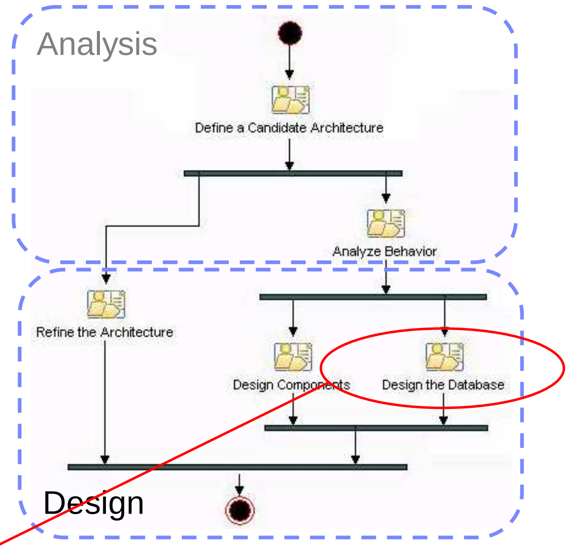
Rational. software

business on demand software

IBM | IBM Software Group | Rational software

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



128

Where Are We?

- ➡ Relational Databases and Object Orientation
- Mapping Objects to Tables
- Strategies for Implementing Persistence

IBM | IBM Software Group | Rational software

The “Object/Relational Impedance Mismatch”

- RDBMS and Object Orientation are not entirely compatible
  - RDBMS
    - Focus is on data
    - Better suited for ad-hoc relationships and reporting application
    - Expose data (column values)
  - Object Oriented system
    - Focus is on behavior
    - Better suited to handle state-specific behavior where data is secondary
    - Hide data (encapsulation)

130

Relational databases and object orientation are not entirely compatible. They represent two different views of the world: In an RDBMS, all you see is data; in an object-oriented system, all you see is behavior. The object-oriented model tends to work well for systems with complex behavior and state-specific behavior in which data is secondary, or systems in which data is accessed navigationally in a natural hierarchy (for example, bills of materials). The RDBMS model is well suited to reporting applications and systems in which the relationships are dynamic or ad hoc.

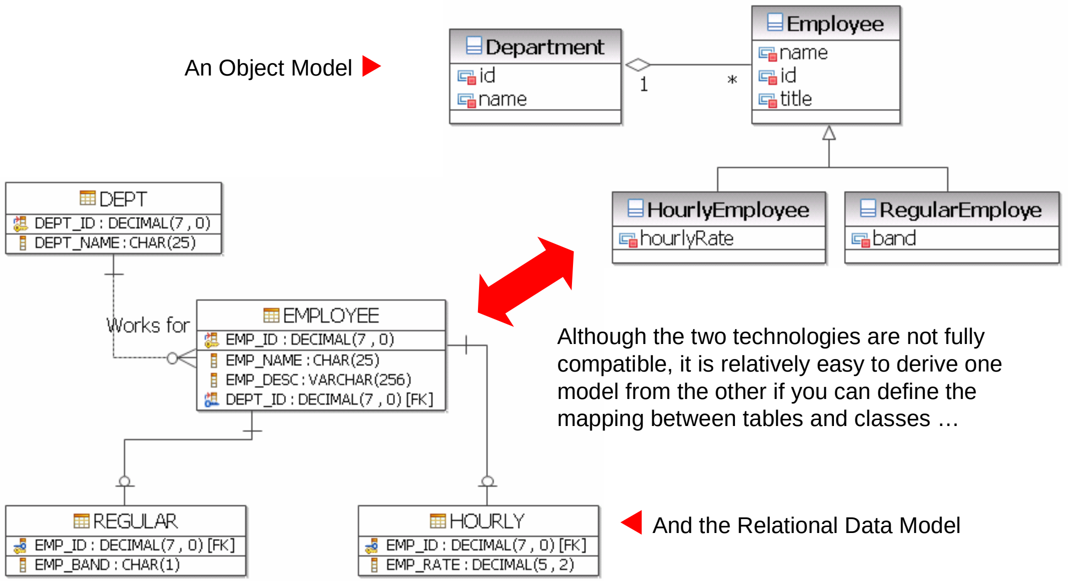
The real fact of the matter is that a lot of information is stored in relational databases, and if object-oriented applications want access to that data, they need to be able to read and write to an RDBMS. In addition, object-oriented systems often need to share data with non-object-oriented systems. It is natural, therefore, to use an RDBMS as the sharing mechanism.

While object-oriented and relational design share some common characteristics (an object’s attributes are conceptually similar to an entity’s columns), fundamental differences make seamless integration a challenge. The fundamental difference is that data models expose data (through column values) while object models hide data (encapsulating it behind its public interfaces).

IBM | IBM Software Group | Rational software

The Data and Object Models

An Object Model



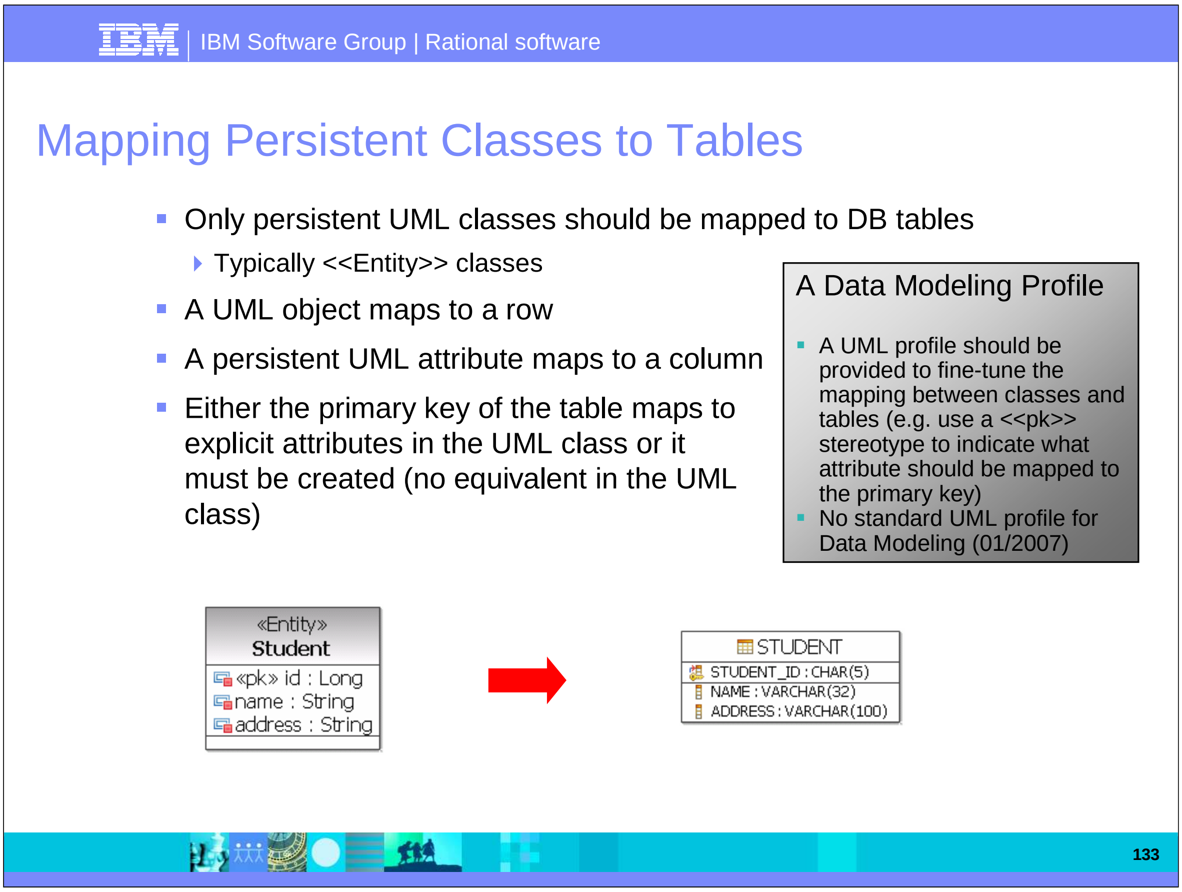
Although the two technologies are not fully compatible, it is relatively easy to derive one model from the other if you can define the mapping between tables and classes …

And the Relational Data Model

131

Where Are We?

- Relational Databases and Object Orientation
- ➡ Mapping Objects to Tables
- Strategies for Implementing Persistence



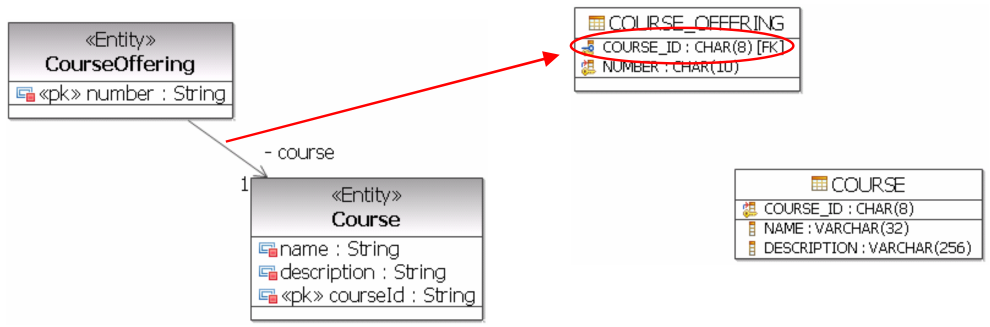
The persistent classes in the Design Model represent the information the system must store. Conceptually, these classes might resemble a relational design (for example, the classes in the Design Model might be reflected in some fashion as entities in the relational schema). As we move from elaboration into construction, however, the goals of the Design Model and the Relational Data Model diverge. The objective of relational database development is to normalize data, whereas the goal of the Design Model is to encapsulate increasingly complex behavior. The divergence of these two perspectives — data and behavior — leads to the need for mapping between related elements in the two models.

In a relational database written in third normal form, every row in the tables — every “tuple" — is regarded as an object. A column in a table is equivalent to a persistent attribute of a class (keep in mind that a persistent class may have transient attributes). So, in the simple case where we have no associations to other classes, the mapping between the two worlds is simple. The data type of the attribute corresponds to one of the allowable data types for columns.

IBM Software Group | Rational software

Mapping Associations Between Persistent Classes

- Associations between two persistent objects are realized as foreign keys to the associated objects
  - A foreign key is a column in one table that contains the primary key value of associated object



134

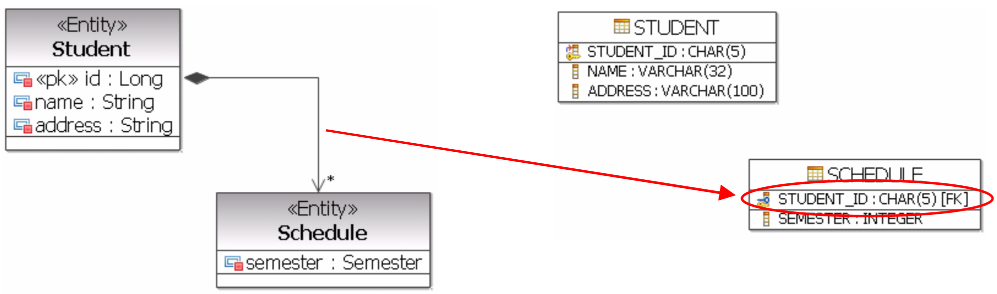
Associations between two persistent objects are realized as foreign keys to the associated objects. A foreign key is a column in one table that contains the primary key value of the associated object.

Assume we have the above association between Course and CourseOffering. When we map this into relational tables, we get a Course table and a Course Offering table. The Course Offering table has columns for attributes listed, plus an additional COURSE_ID column that contains foreign-key references to the primary key of associated rows in the Course table. For a given Course Offering, the COURSE_ID column contains the code of the Course with which the Course Offering is associated. Foreign keys allow the RDBMS to join related information together.

IBM Software Group | Rational software

Mapping Aggregation to the Data Model

- Aggregation is also modeled using foreign key relationships



135

Aggregation is also modeled using foreign key relationships.

Assume we have the above aggregation between Student and Schedule. (Note: This is modeled as a composition, but remember that composition is a nonshared aggregation).

When we map this into relational tables, we get a Student table and a Schedule table. The Schedule table has columns for attributes listed, plus an additional column for Student_ID that contains foreign-key references to associated rows in the Student table. For a given Schedule, the Student_ID column contains the Student_ID of the Student that the Schedule is associated with. Foreign keys allow the RDBMS to join related information together.

In addition, to provide referential integrity in the Data Model, we would also want to implement a cascading delete constraint, so that whenever the Student is deleted, all of its Schedules are deleted as well.

IBM Software Group | Rational software

Other Relationships

- Modeling many-to-many relationships
  - Creation of an associative table holding the foreign keys to the other two tables
- Modeling Inheritance in the Data Model
  - A Data Model does not support modeling inheritance in a direct way
  - Three options:
    - Map the entire class hierarchy to a single table
    - Map each *concrete* class to its own table
    - Map each class to its own table

136

The standard relational Data Model does not support modeling inheritance associations in a direct way. A number of strategies can be used to model inheritance:

- Use separate tables to represent the super-class and subclass. Have, in the subclass table, a foreign key reference to the super-class table. In order to “instantiate” a subclass object, the two tables would have to be joined together. This approach is conceptually easier and makes changes to the model easier, but it often performs poorly due to the extra work.
- Duplicate all inherited attributes and associations as separate columns in the subclass table. This is similar to de-normalization in the standard relational Data Model.

Where Are We?

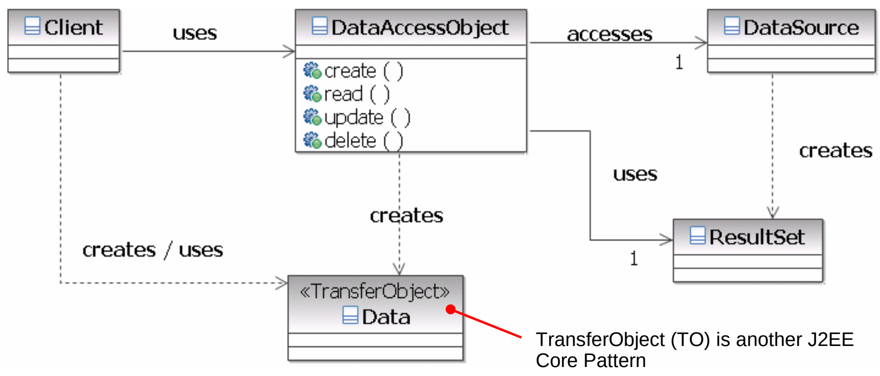
- Relational Databases and Object Orientation
- Mapping an Object Model to a Data Model
- ⇨ Strategies for Implementing Persistence

Strategies for Implementing Persistence

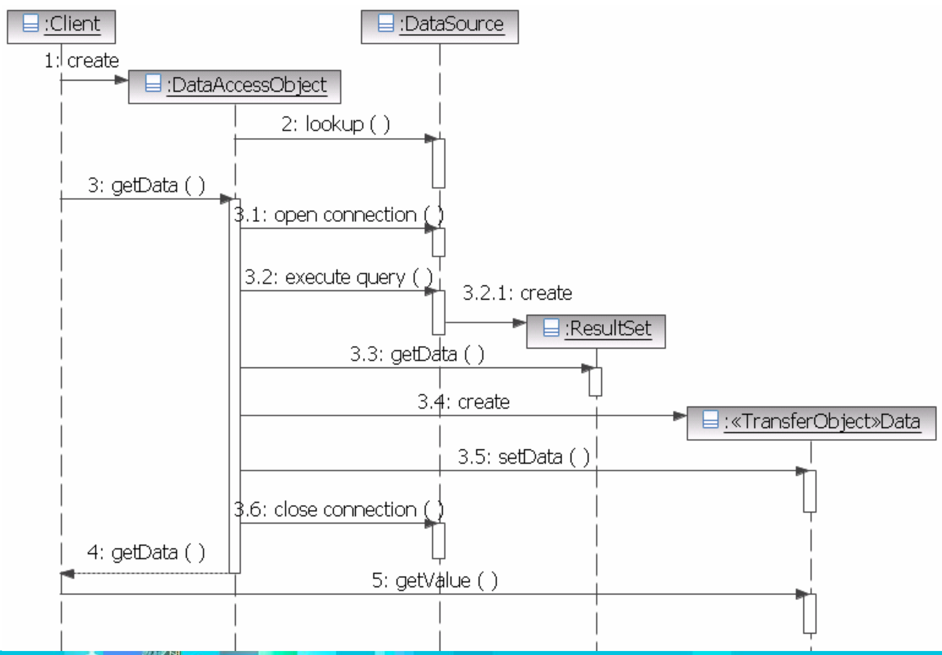
- Business objects access data sources directly
  - In Java applications, this is typically done using JDBC
  - Simple but business objects directly coupled to the database
- Data access objects (DAOs)
  - DAOs encapsulate the database access logic
  - Isolate business objects from the data sources
- Persistence frameworks
  - Database access code automatically generated by the persistence framework
  - Overall performance usually better
  - Examples: Enterprise JavaBeans (EJB), Hibernate, OJB (ObJectRelationalBridge)
- Any combination of the above

The Data Access Object (DAO) Pattern

- Source: Core J2EE Patterns, Deepak Alur, John Crupi & Dan Malks, Prentice Hall, 2003
- A Data Access Object encapsulates all access to the persistent store:
  - The DAO manages the connection with the data source to store and obtain data

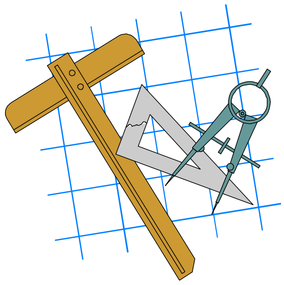


The Data Access Object (DAO) Pattern (cont.)



Exercise

- There is no exercise in this module



IBM Software Group | Rational software

142