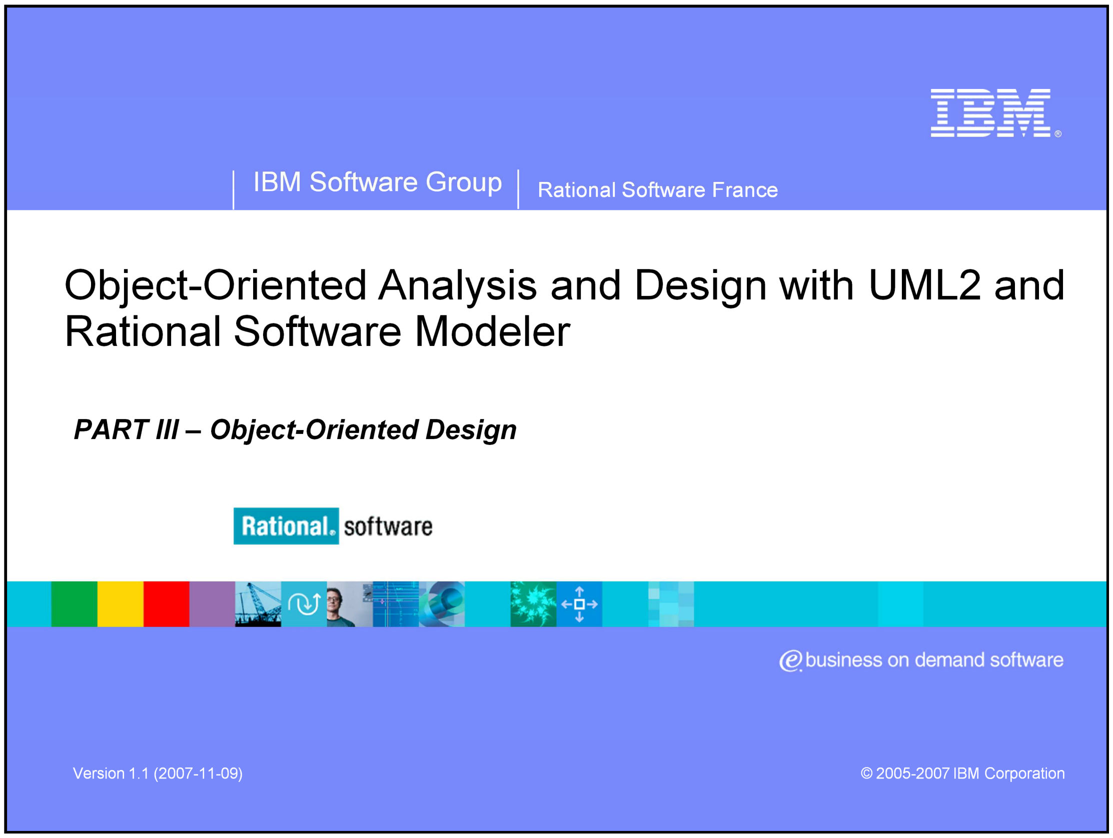


IBM Software Group | Rational Software France

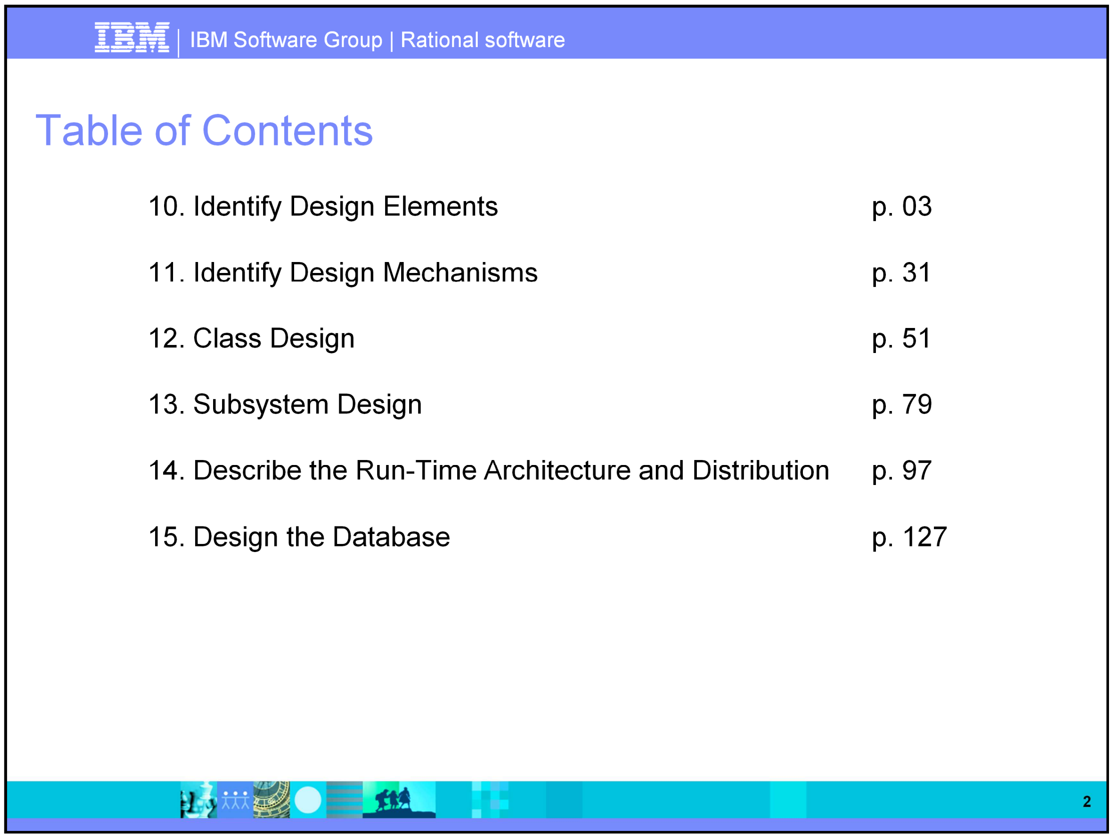
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

PART III – Object-Oriented Design

Rational. software

e business on demand software

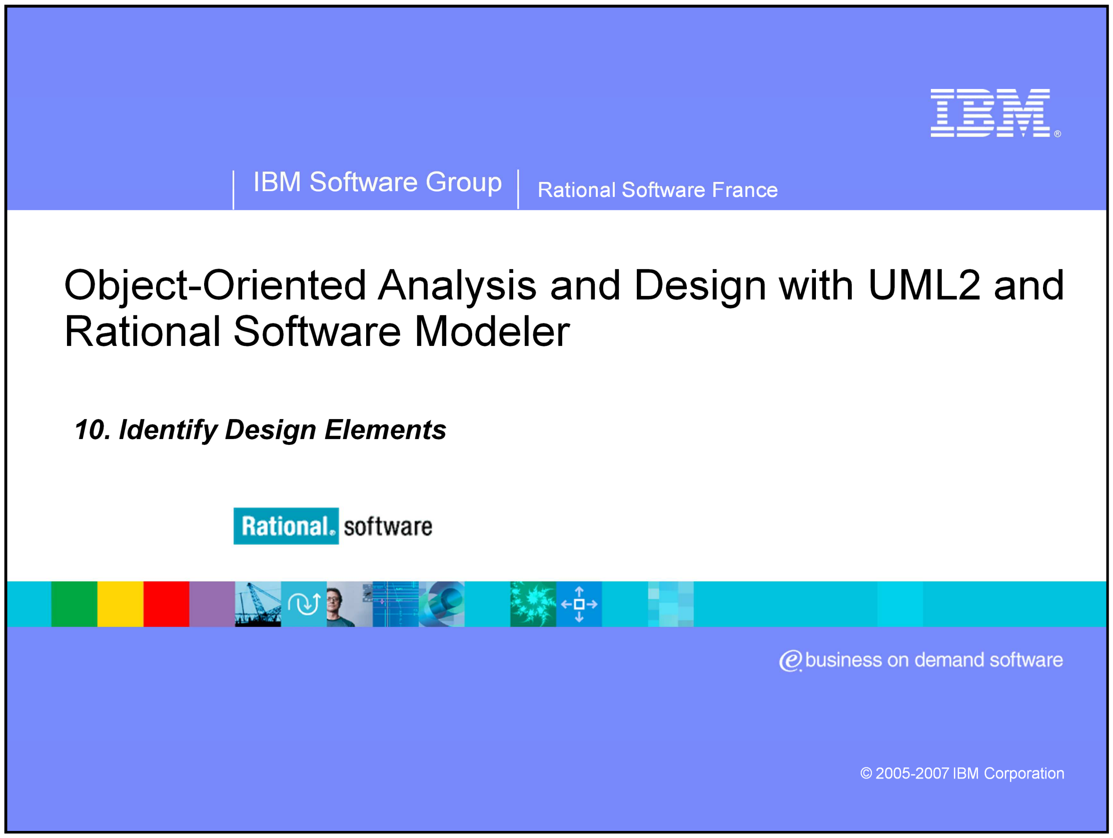
Version 1.1 (2007-11-09)



IBM Software Group | Rational software

Table of Contents

2



IBM Software Group | Rational Software France

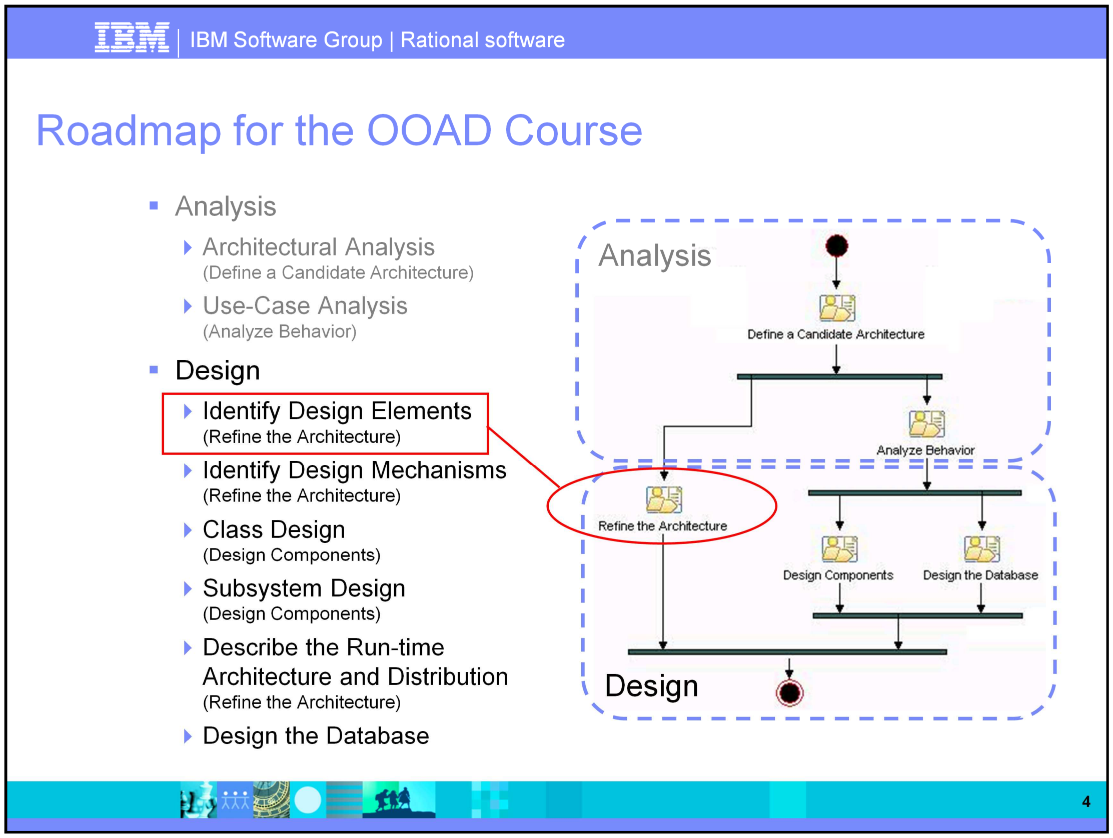
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

10. Identify Design Elements

Rational. software

e business on demand software

© 2005-2007 IBM Corporation



IBM | IBM Software Group | Rational software

Roadmap for the OOAD Course

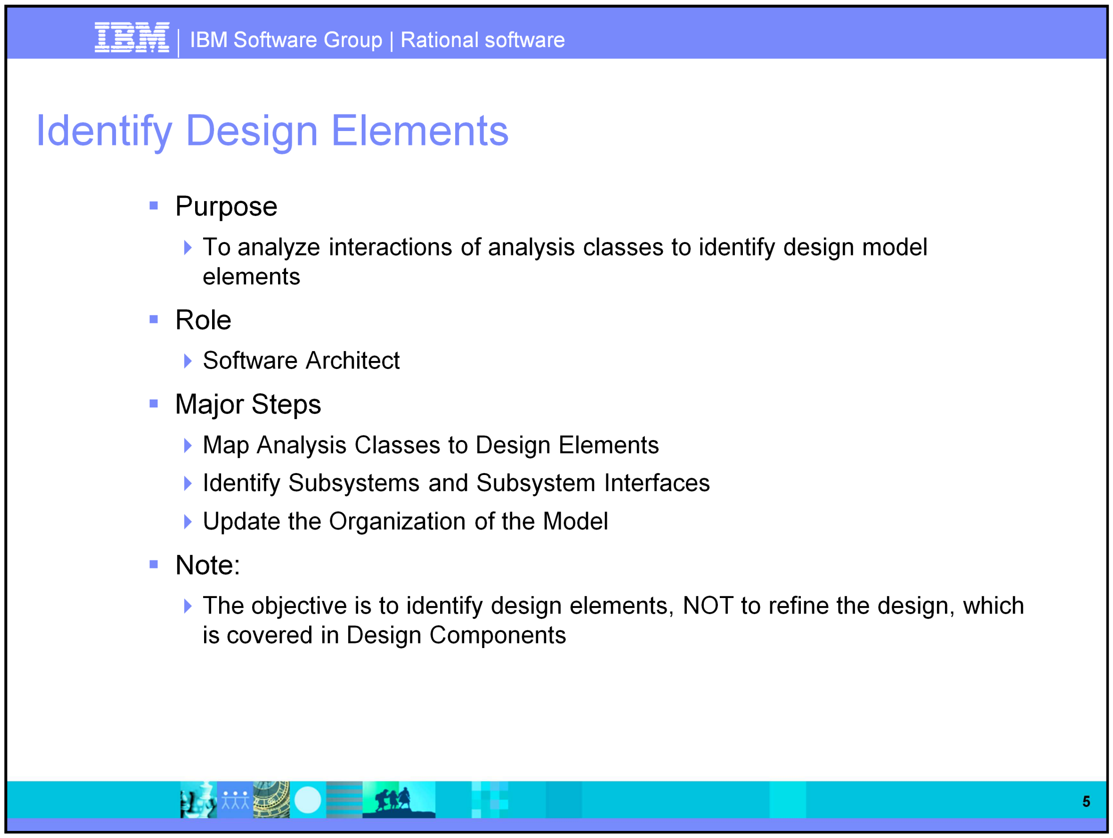
- Analysis
  - Architectural Analysis (Define a Candidate Architecture)
  - Use-Case Analysis (Analyze Behavior)
- Design
  - Identify Design Elements (Refine the Architecture)
  - Identify Design Mechanisms (Refine the Architecture)
  - Class Design (Design Components)
  - Subsystem Design (Design Components)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

4

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.

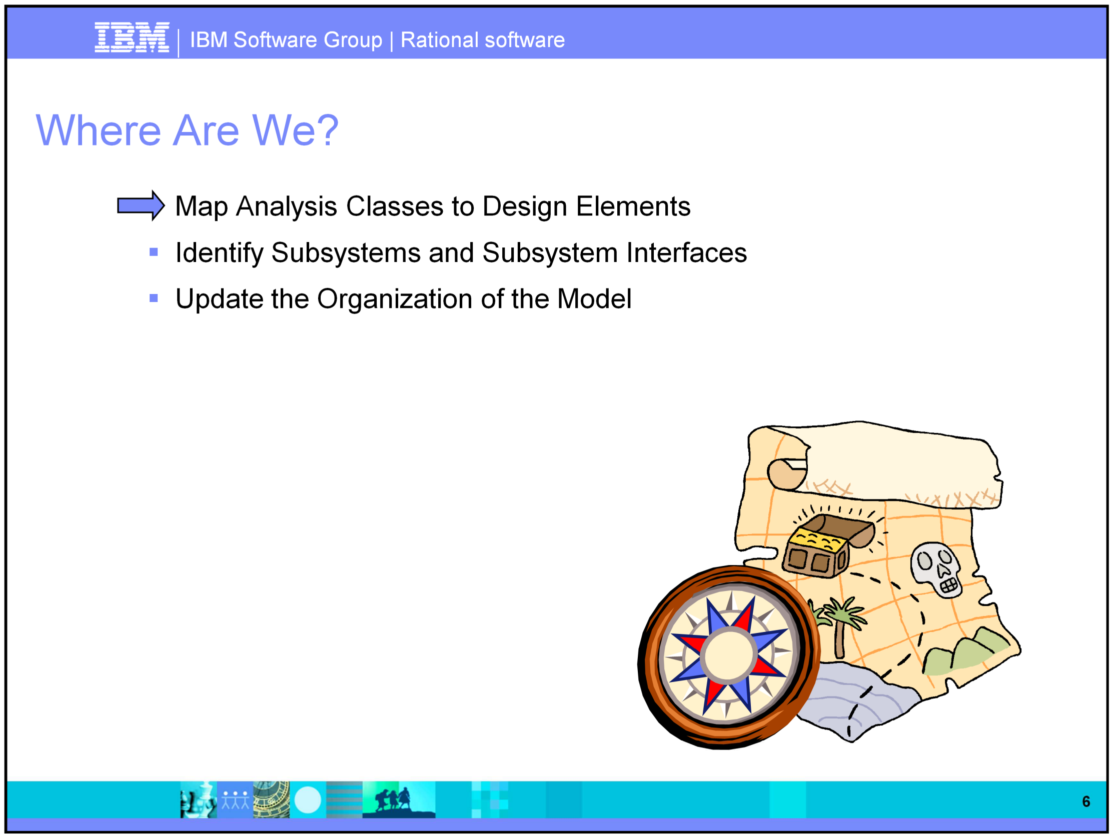
In **Identify Design Elements**, the analysis classes are refined into design elements (design classes and subsystems).

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.



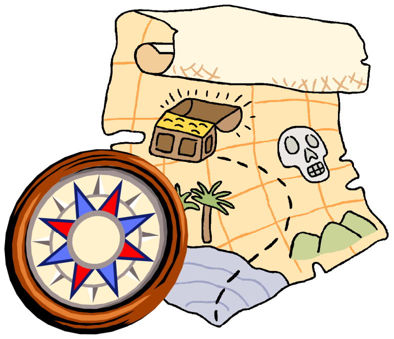
Identify Design Elements

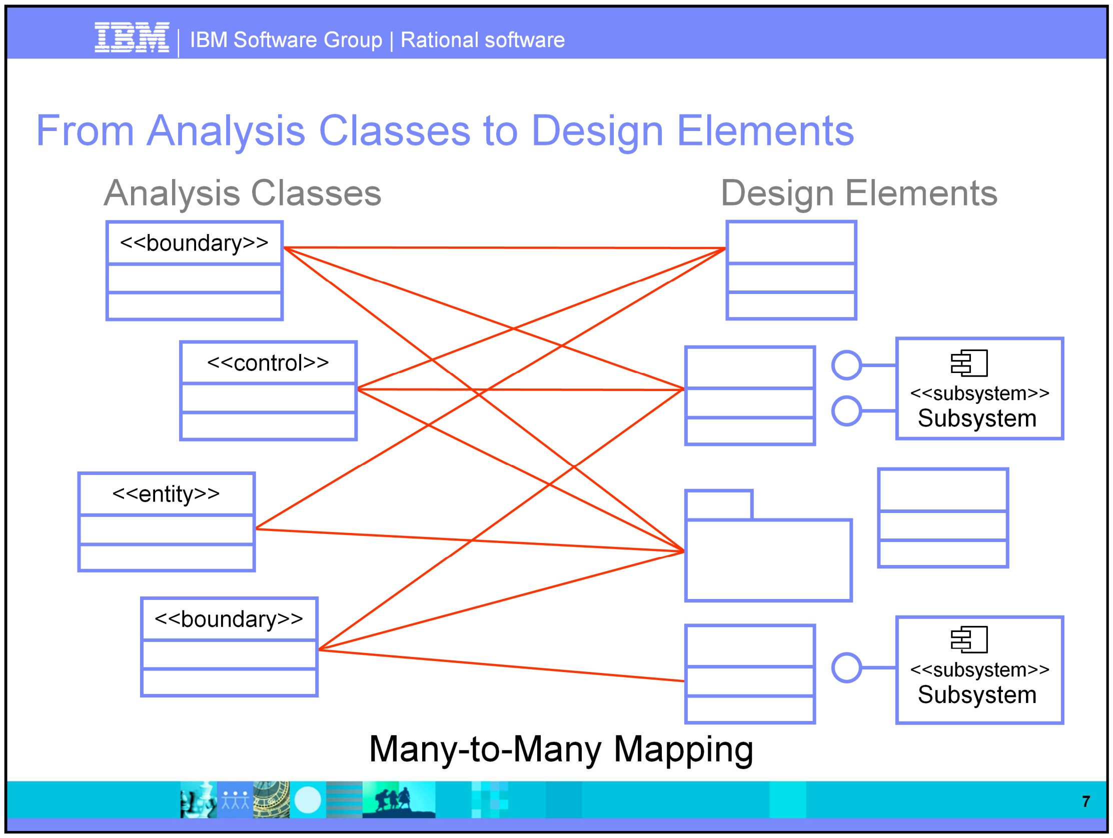
- Purpose
  - To analyze interactions of analysis classes to identify design model elements
- Role
  - Software Architect
- Major Steps
  - Map Analysis Classes to Design Elements
  - Identify Subsystems and Subsystem Interfaces
  - Update the Organization of the Model
- Note:
  - The objective is to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



Where Are We?

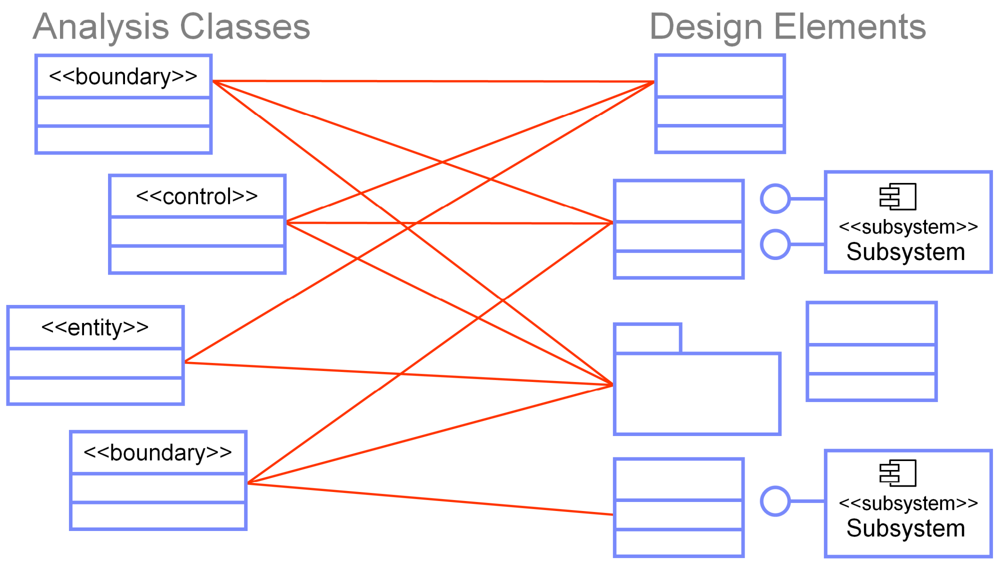
- Map Analysis Classes to Design Elements
- Identify Subsystems and Subsystem Interfaces
- Update the Organization of the Model





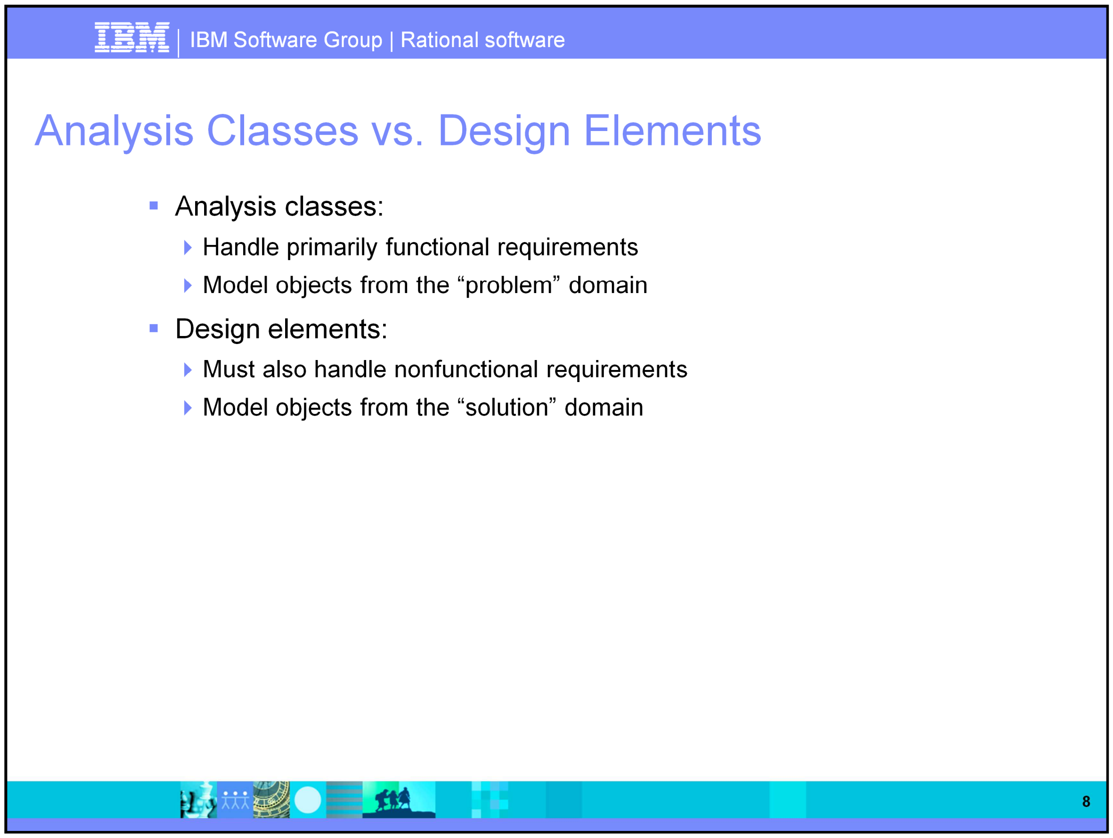
IBM Software Group | Rational software

From Analysis Classes to Design Elements



Many-to-Many Mapping

7



IBM Software Group | Rational software

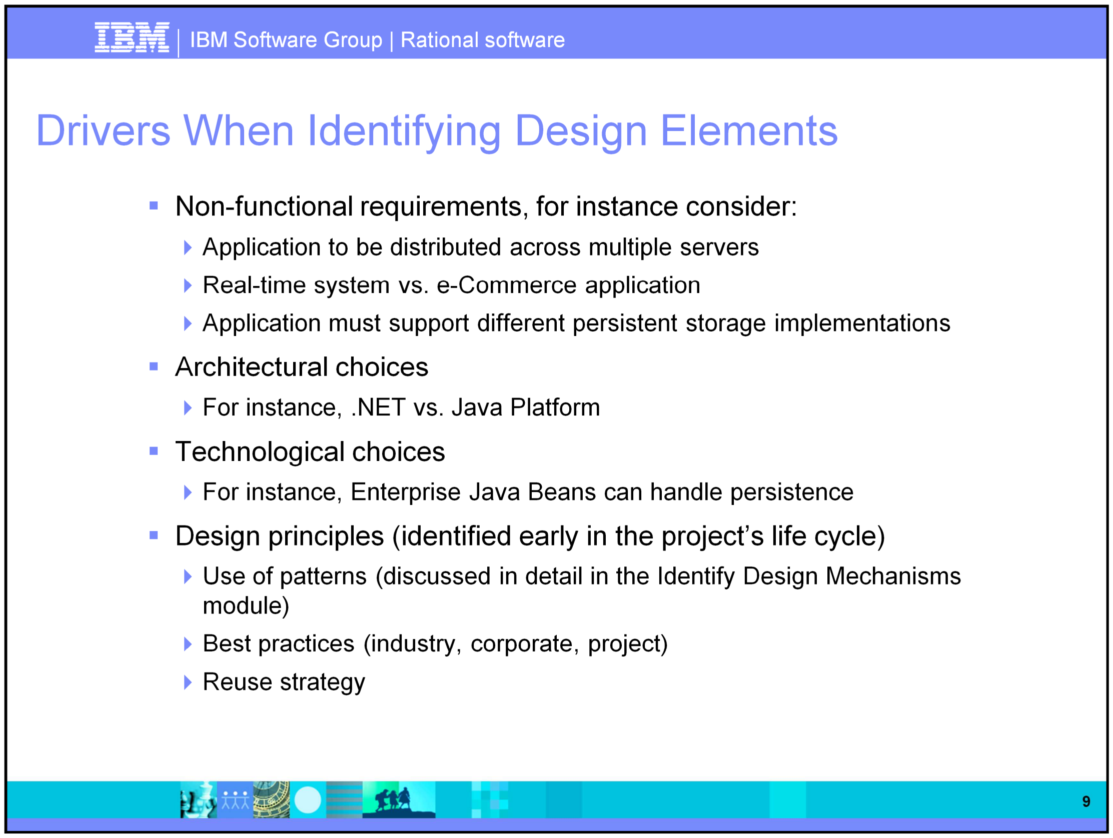
Analysis Classes vs. Design Elements

- 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).

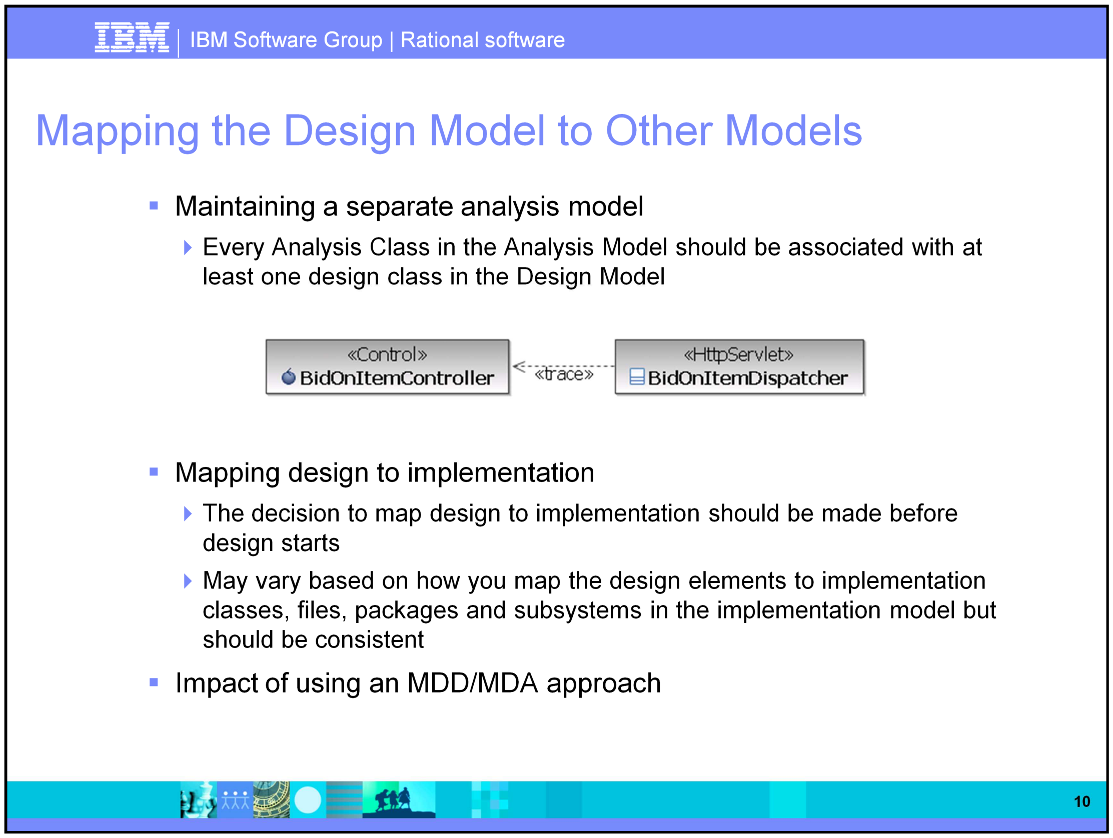


IBM Software Group | Rational software

Drivers When Identifying Design Elements

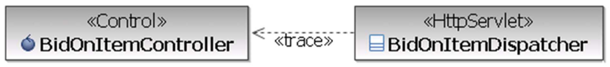
- Non-functional requirements, for instance consider:
  - Application to be distributed across multiple servers
  - Real-time system vs. e-Commerce application
  - Application must support different persistent storage implementations
- Architectural choices
  - For instance, .NET vs. Java Platform
- 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

9

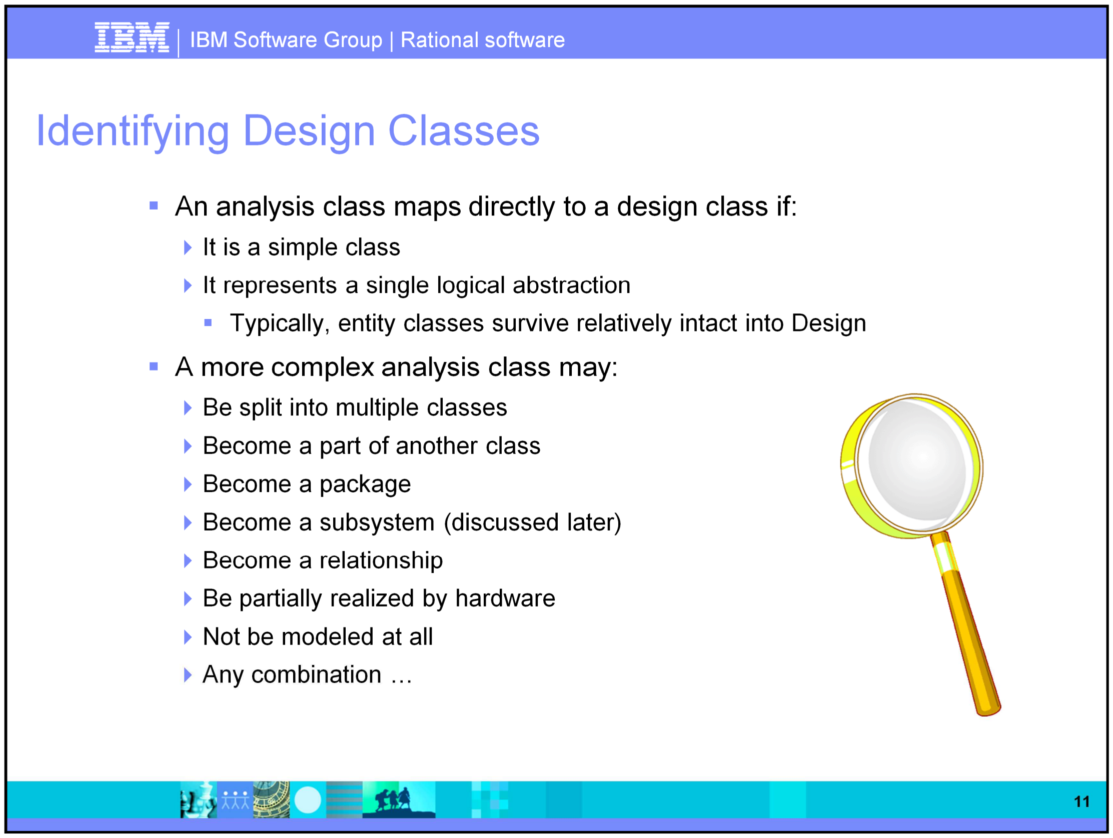


Mapping the Design Model to Other Models

- 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 | IBM Software Group | Rational software

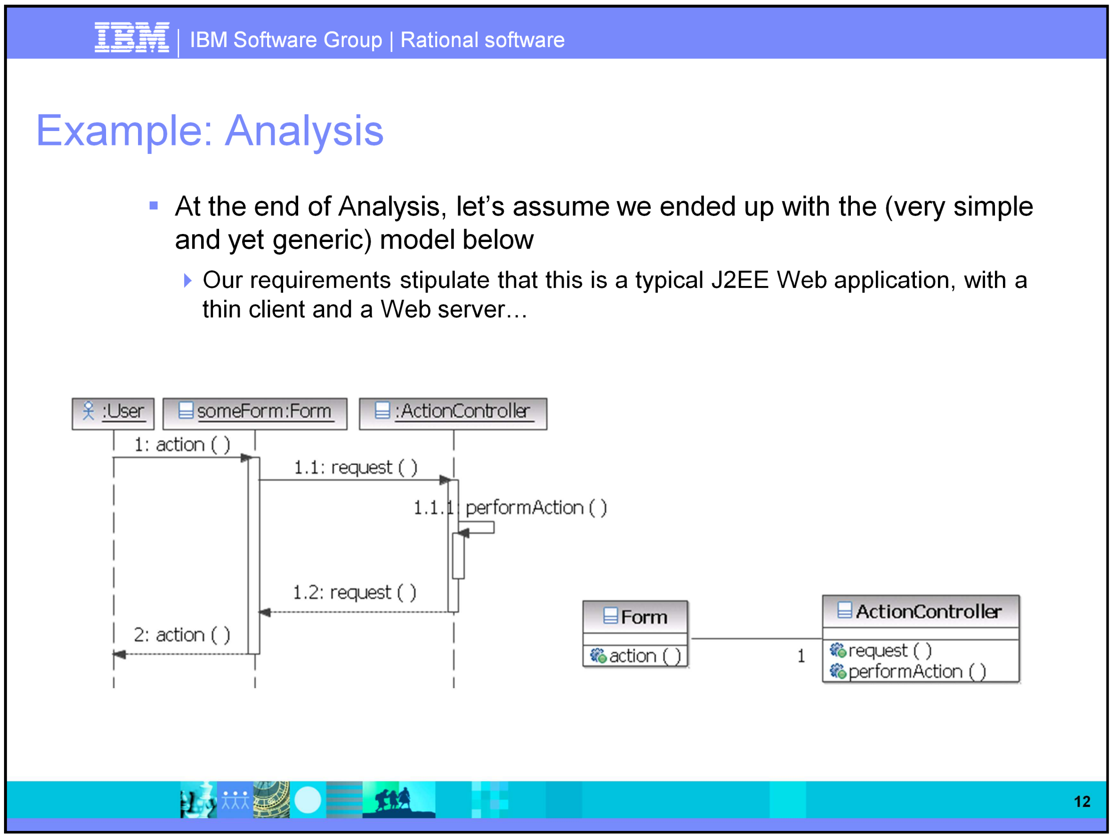
Identifying Design Classes

- 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

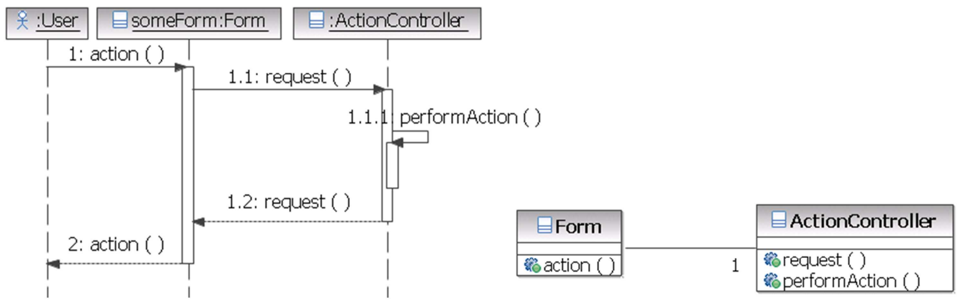
Some examples:

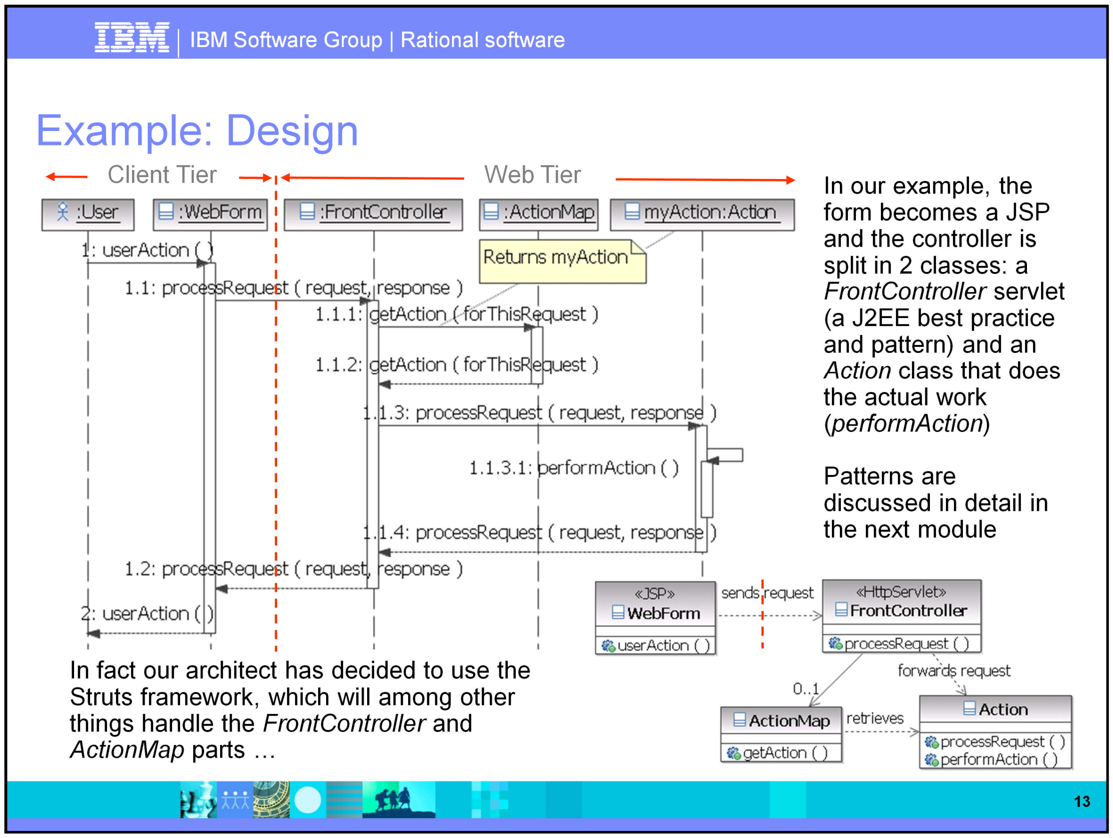
- A single boundary class representing a user interface may result in multiple classes, one per window.
- A control class may become a design class directly, or become a method within a design class.
- A single entity class may become multiple classes (for example, an aggregate with contained classes, 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
Client Tier
Web Tier
:User
:WebForm
:FrontController
:ActionMap
myAction:Action
Returns myAction
1: userAction ( )
1.1: processRequest (request, response)
1.1.1: getAction (forThisRequest)
1.1.2: getAction (forThisRequest)
1.1.3: processRequest (request, response)
1.1.3.1: performAction ( )
1.1.4: processRequest (request, response)
1.2: processRequest (request, response)
2: userAction ( )
«JSP» WebForm
userAction ( )
sends request
«HttpServlet» FrontController
processRequest ( )
forwards request
0..1
ActionMap
getAction ( )
retrieves
Action
processRequest ( )
performAction ( )
13

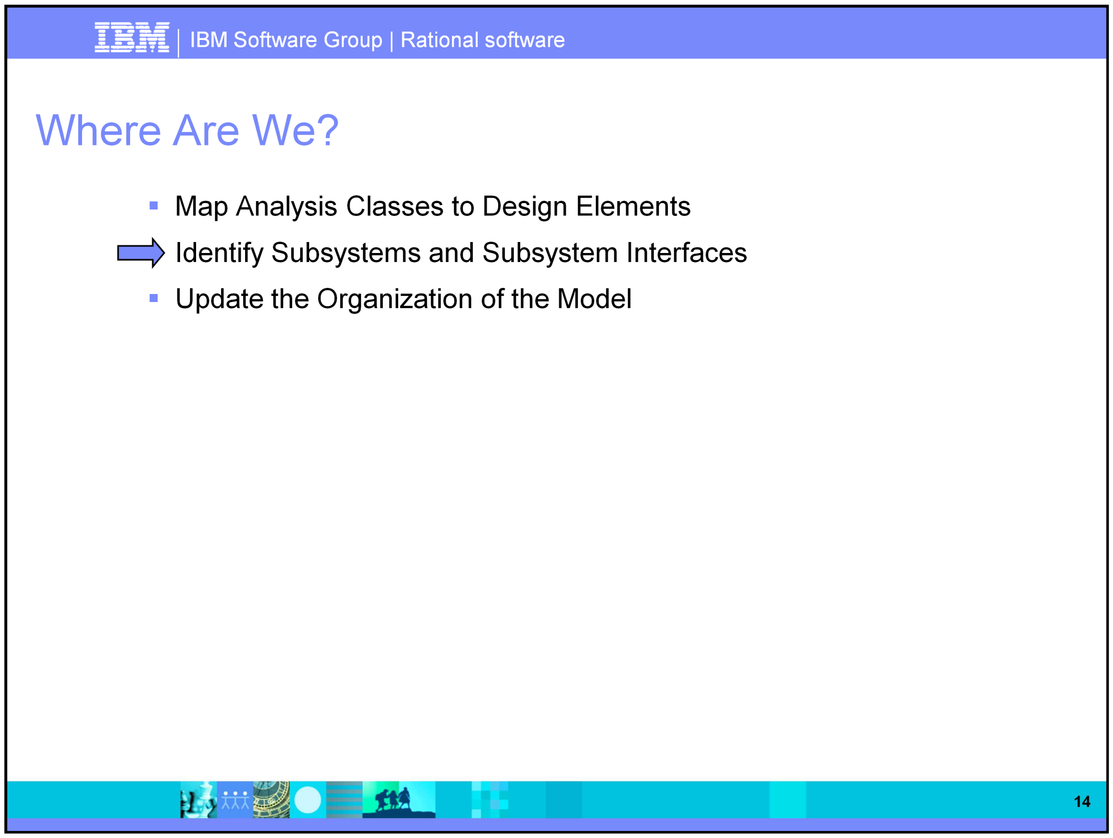
Example: Design

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 …

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.

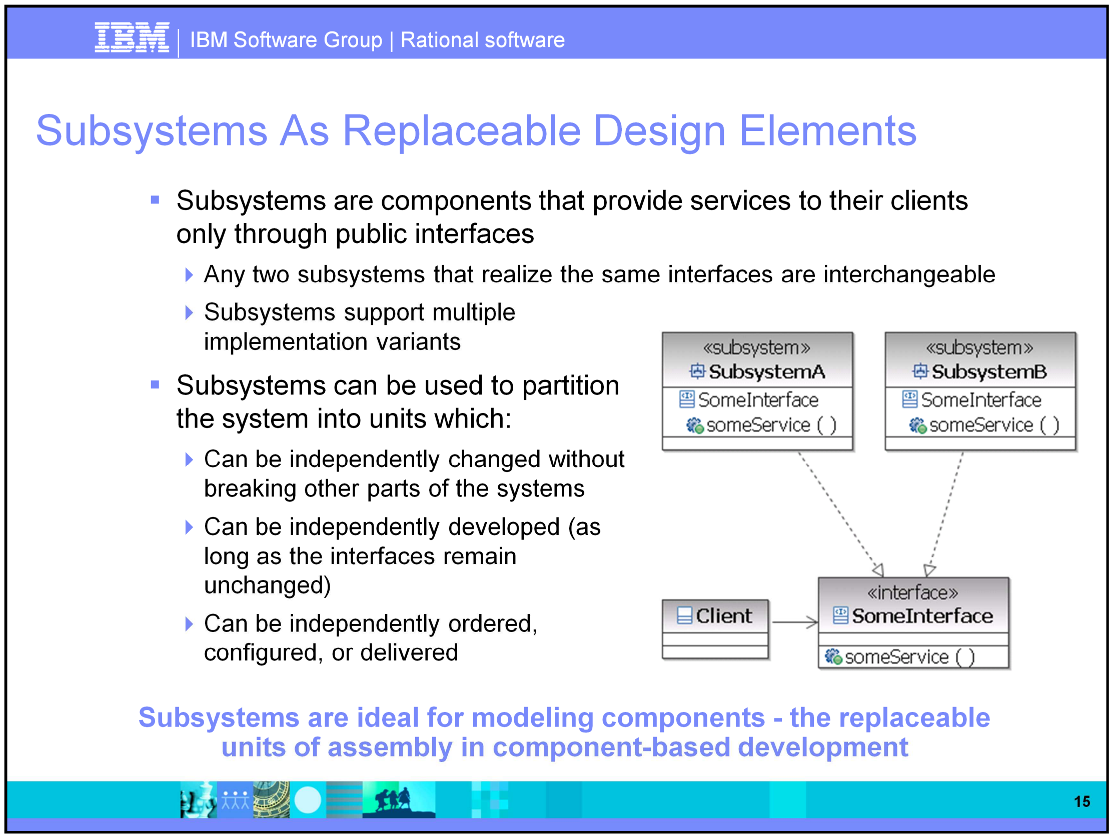


IBM Software Group | Rational software

Where Are We?

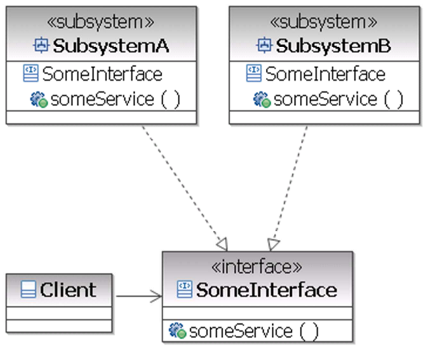
- Map Analysis Classes to Design Elements
- ⇨ Identify Subsystems and Subsystem Interfaces
- Update the Organization of the Model

14

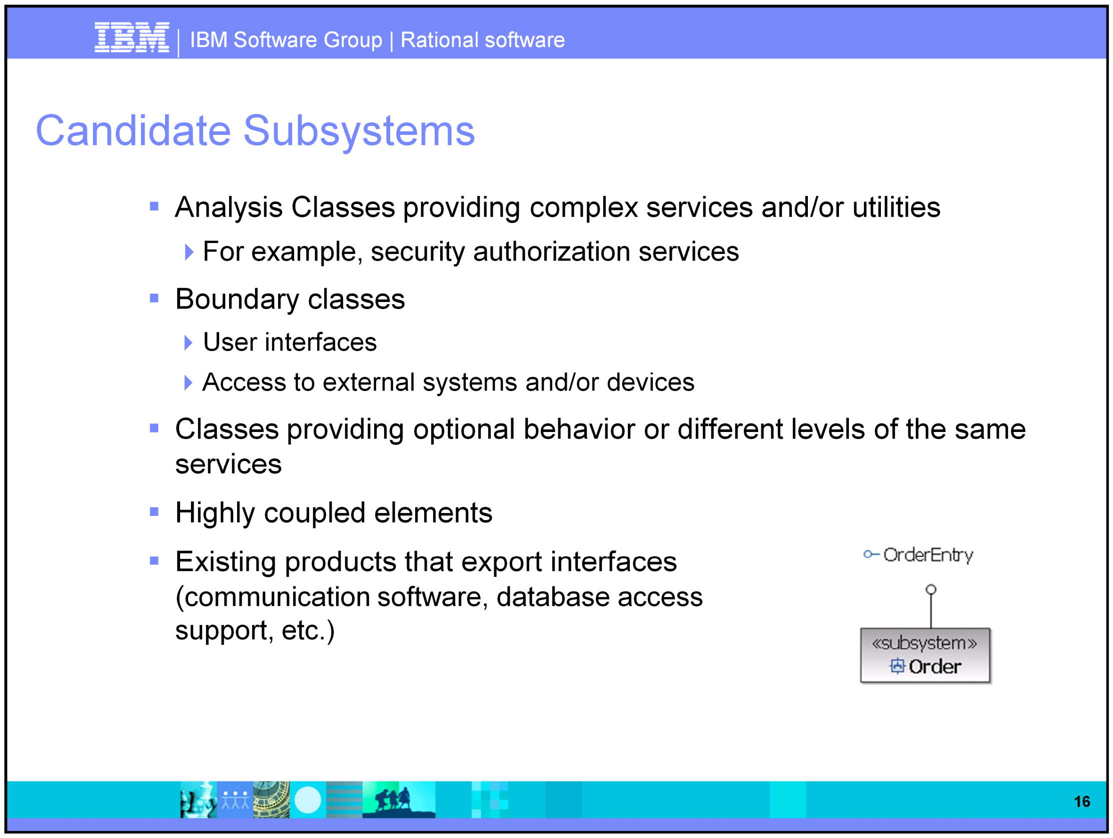


Subsystems As Replaceable Design Elements

- 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



IBM Software Group | Rational software

Candidate Subsystems

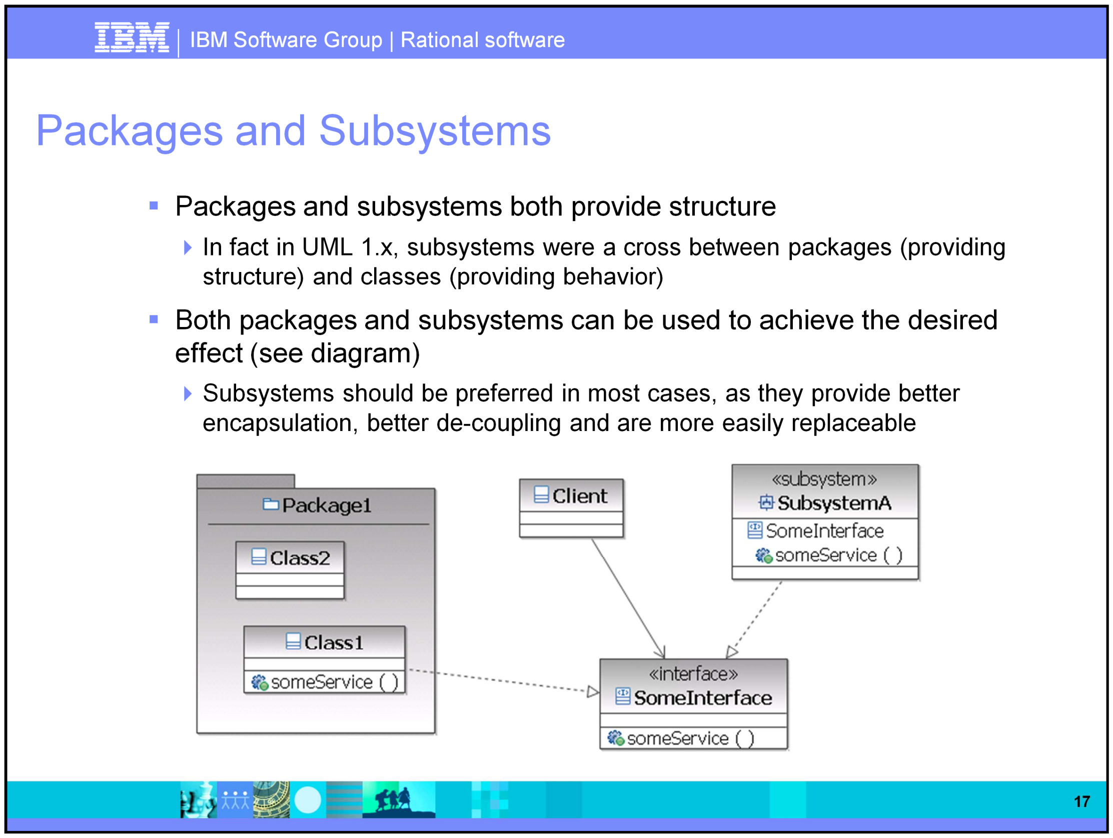
- Analysis Classes providing complex services and/or utilities
  - For example, security authorization services
- Boundary classes
  - User interfaces
  - Access to external systems and/or devices
- Classes providing optional behavior or different levels of the same services
- Highly coupled elements
- 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.



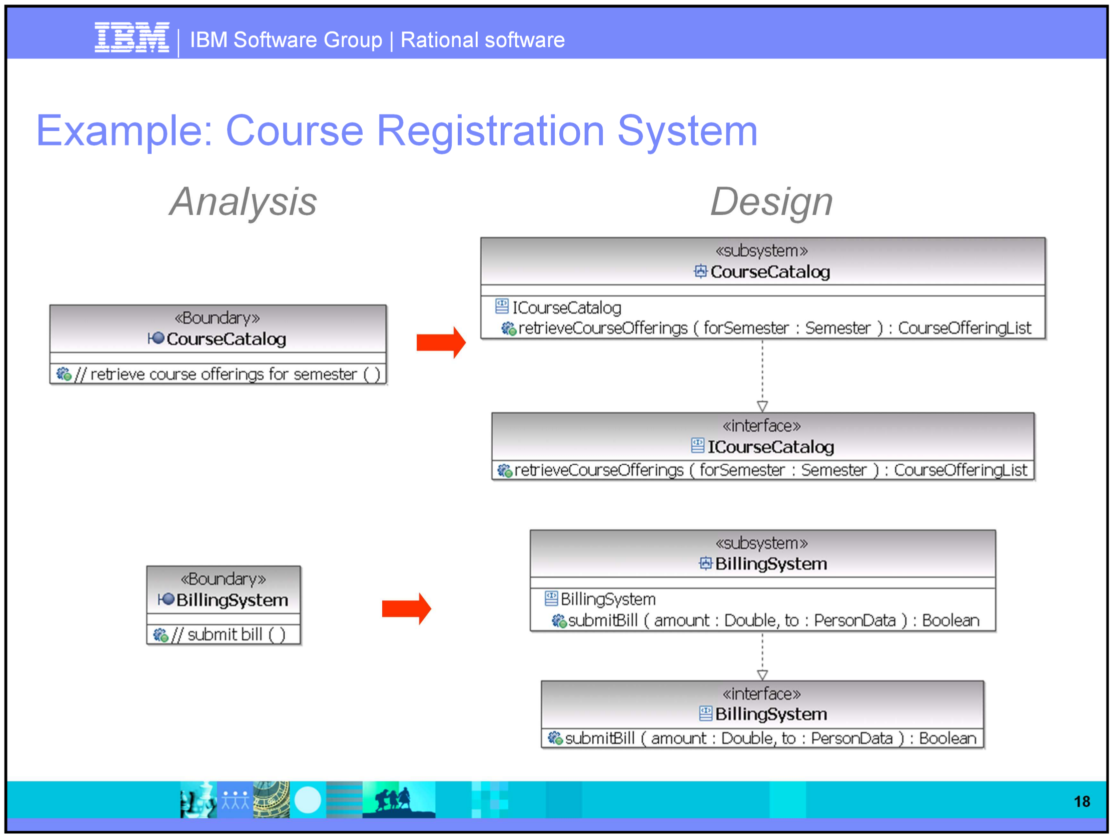
IBM | IBM Software Group | Rational software

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

17

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).


IBM Software Group | Rational software

Example: Course Registration System

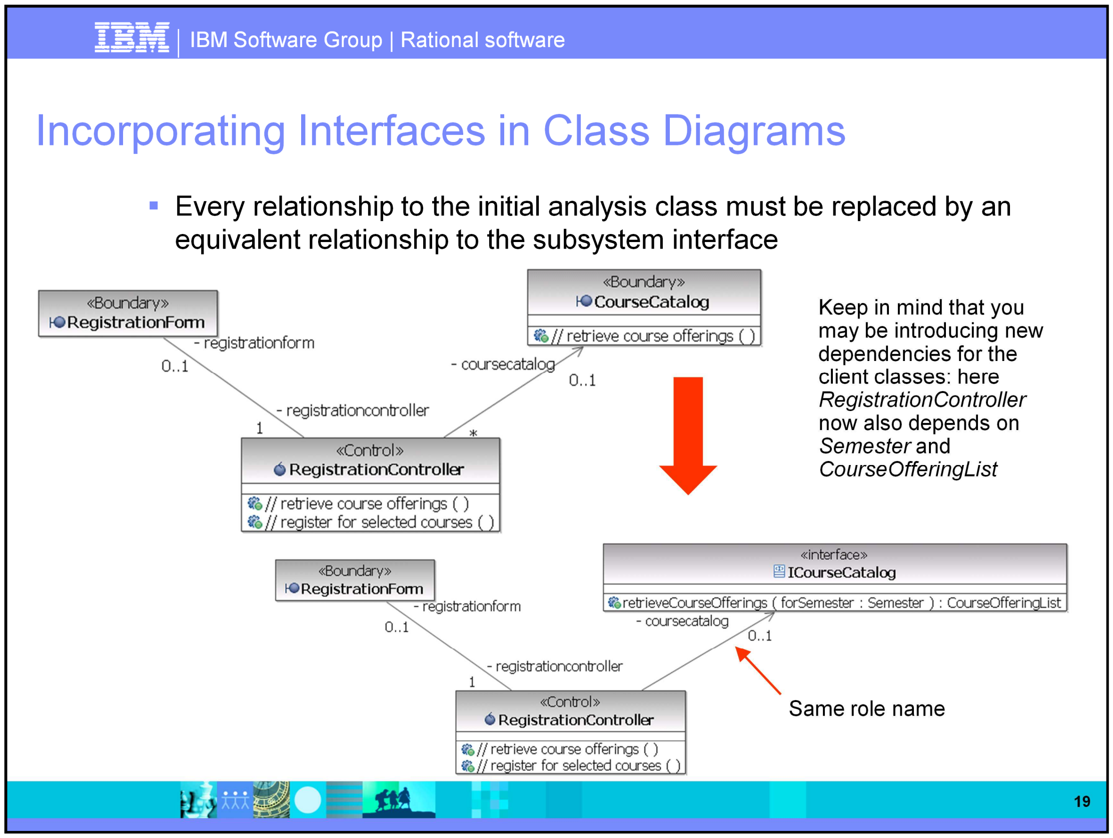
Analysis
Design
«Boundary»
CourseCatalog
// retrieve course offerings for semester ( )
«subsystem»
CourseCatalog
ICourseCatalog
retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
«interface»
ICourseCatalog
retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
«Boundary»
BillingSystem
// submit bill ( )
«subsystem»
BillingSystem
BillingSystem
submitBill (amount : Double, to : PersonData) : Boolean
«interface»
BillingSystem
submitBill (amount : Double, to : PersonData) : Boolean
18

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

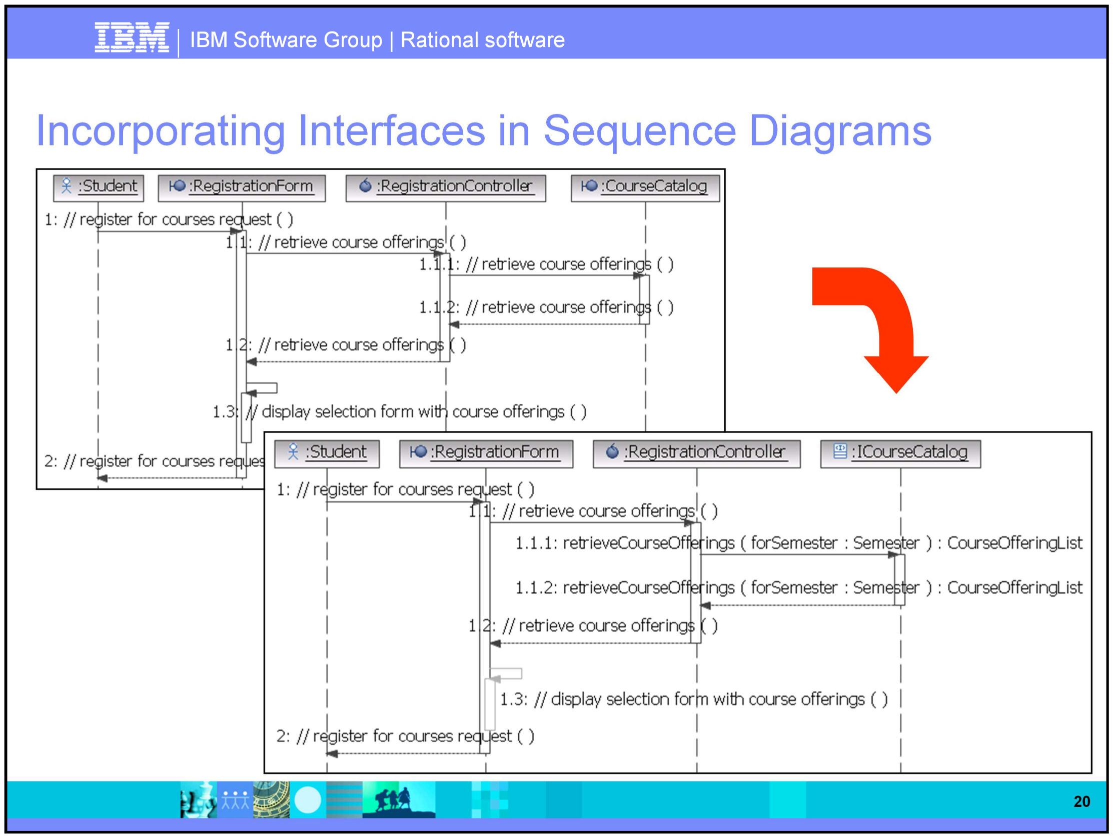
- Every relationship to the initial analysis class must be replaced by an equivalent relationship to the subsystem interface

Keep in mind that you may be introducing new dependencies for the client classes: here *RegistrationController* now also depends on *Semester* and *CourseOfferingList*

19

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

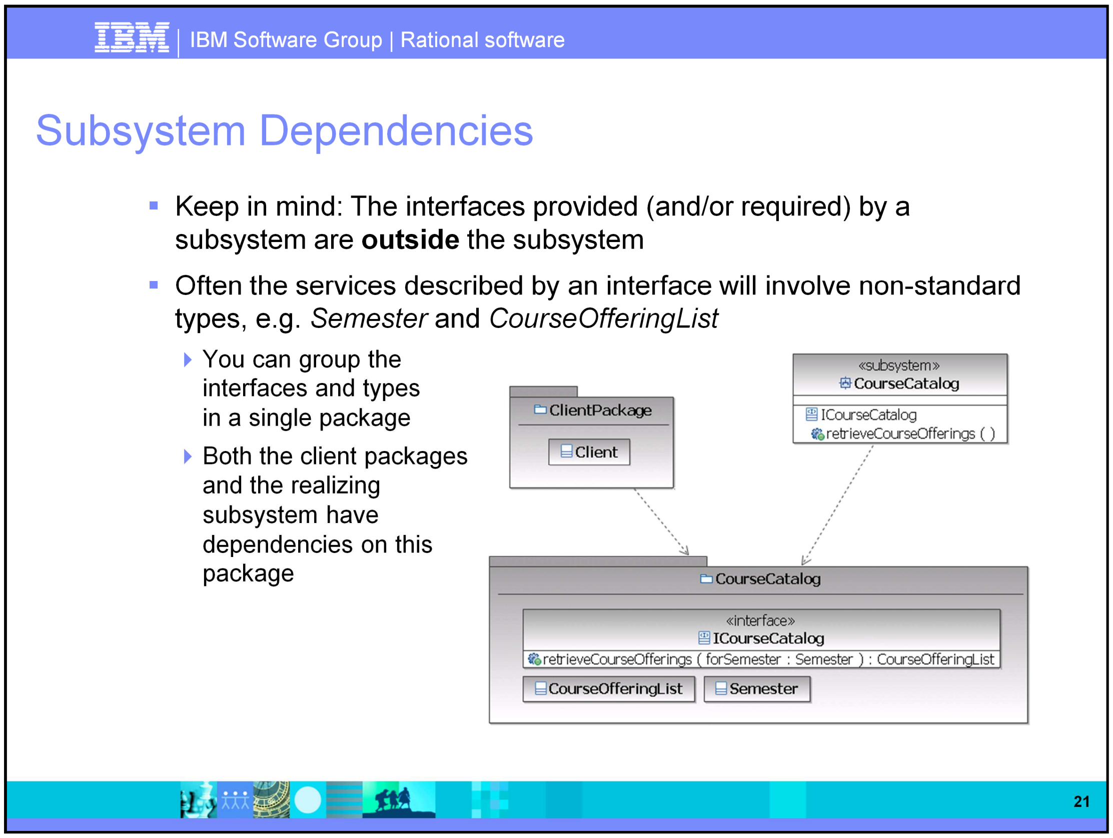
- Retrieve the interface to use and drag it to the diagram
- Select the relationship and move the target end from the analysis class to the interface
- Delete the analysis class from the diagram
- Delete the analysis class from the design model after all changes have been made


IBM Software Group | Rational software

Incorporating Interfaces in Sequence Diagrams

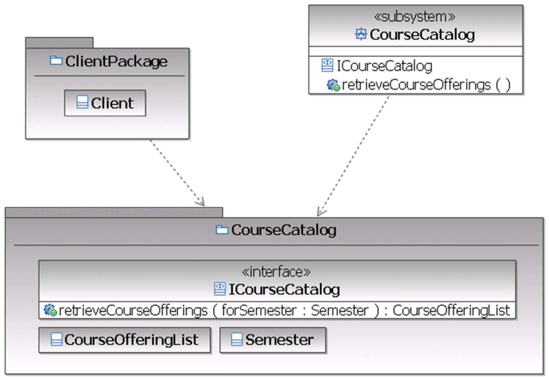
:Student
:RegistrationForm
:RegistrationController
:CourseCatalog
1: // register for courses request ( )
1.1: // retrieve course offerings ( )
1.1.1: // retrieve course offerings ( )
1.1.2: // retrieve course offerings ( )
1.2: // retrieve course offerings ( )
1.3: // display selection form with course offerings ( )
2: // register for courses request ( )
:Student
:RegistrationForm
:RegistrationController
:ICourseCatalog
1: // register for courses request ( )
1.1: // retrieve course offerings ( )
1.1.1: retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
1.1.2: retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
1.2: // retrieve course offerings ( )
1.3: // display selection form with course offerings ( )
2: // register for courses request ( )
20

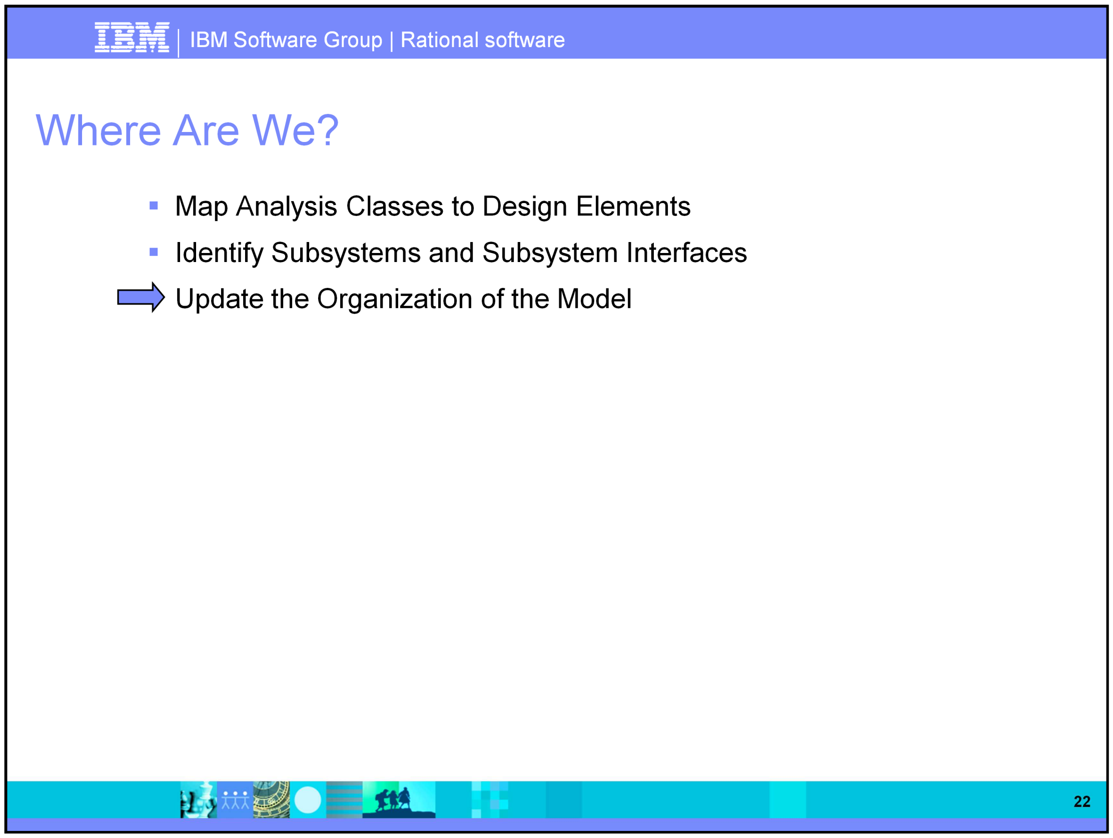
In RSA/RSM, simply drag the interface over the analysis object and update the message.



Subsystem Dependencies

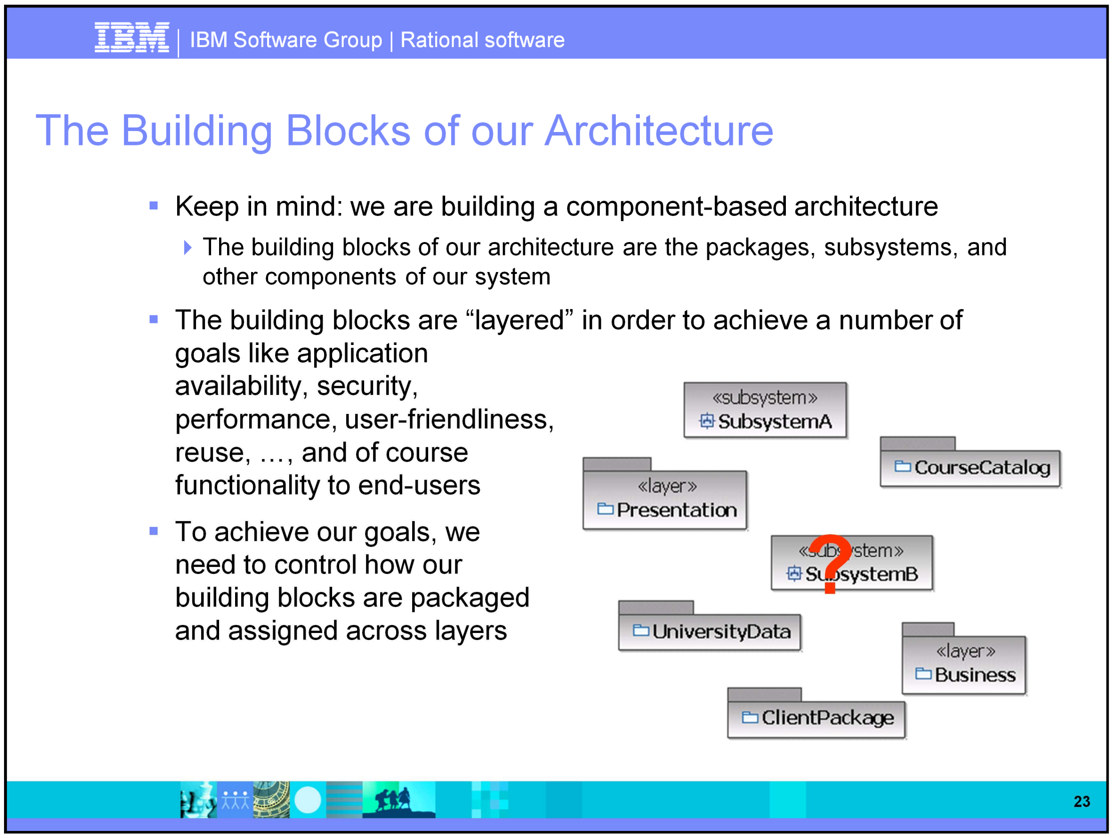
- Keep in mind: The interfaces provided (and/or required) by a subsystem are **outside** the subsystem
- Often the services described by an interface will involve non-standard types, e.g. *Semester* and *CourseOfferingList*
  - You can group the interfaces and types in a single package
  - Both the client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





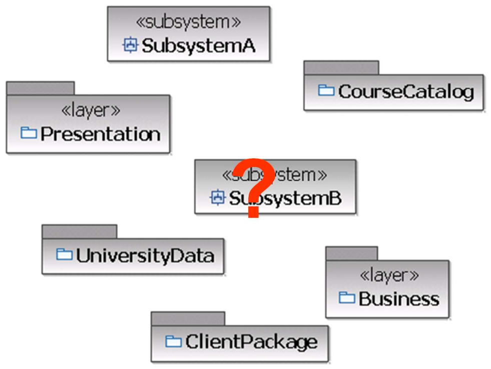
Where Are We?

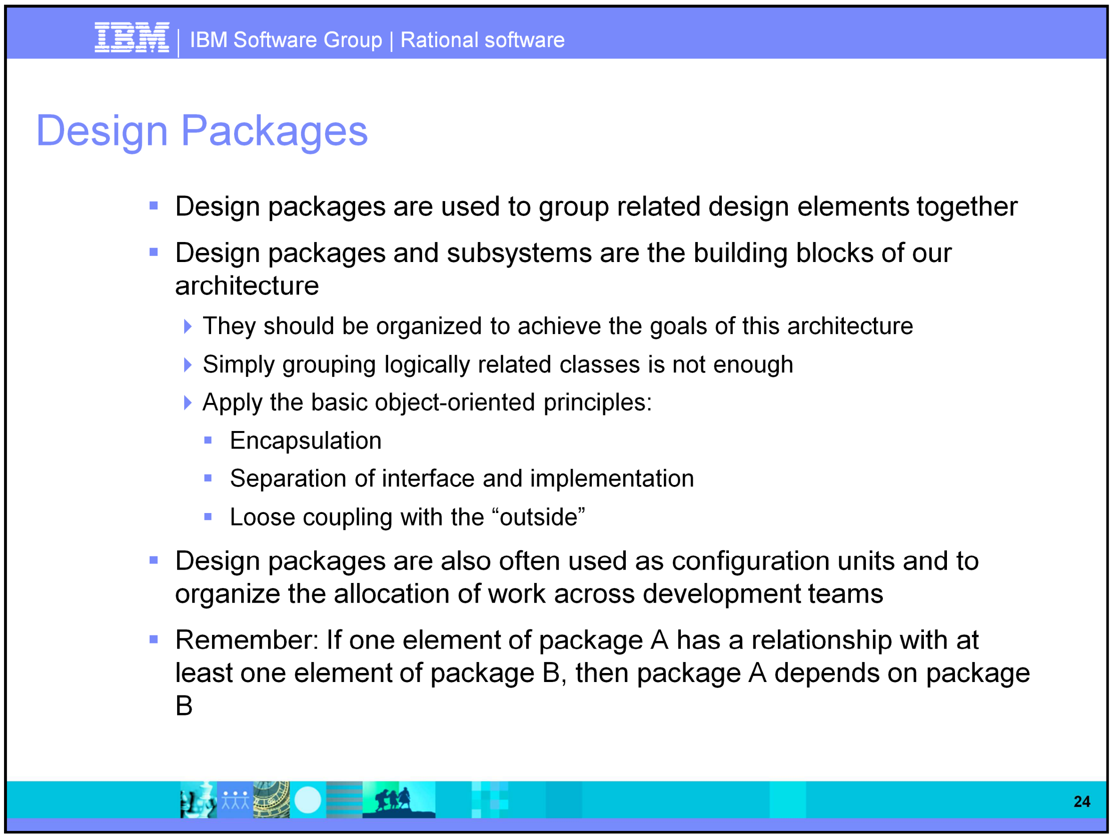
- Map Analysis Classes to Design Elements
- Identify Subsystems and Subsystem Interfaces
- ➡ Update the Organization of the Model



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



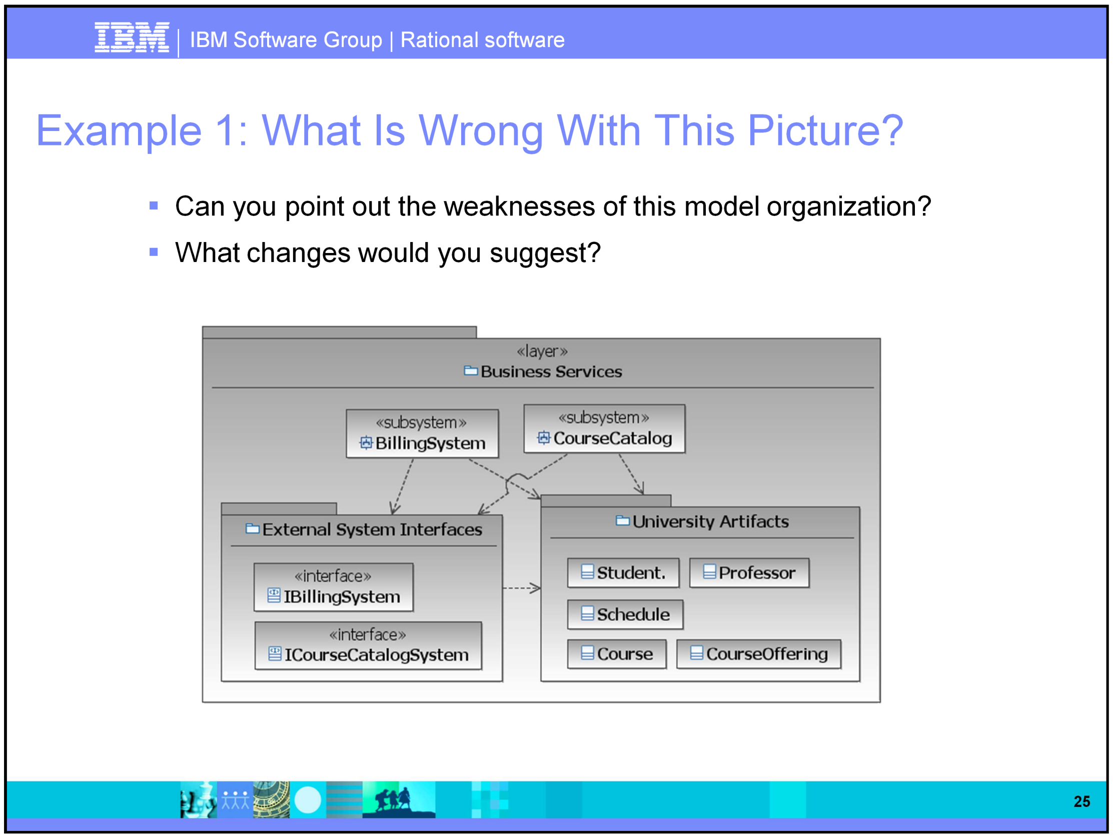


IBM | IBM Software Group | Rational software

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

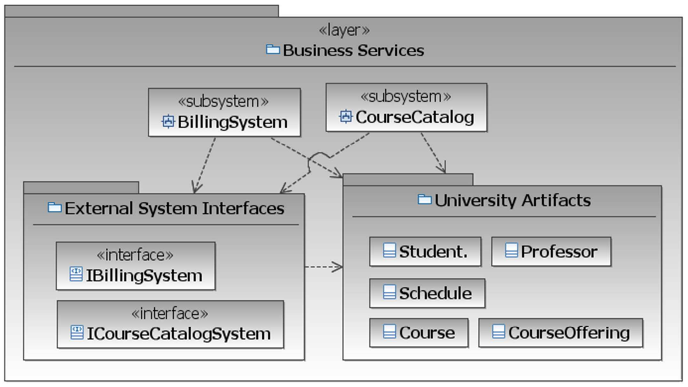
24



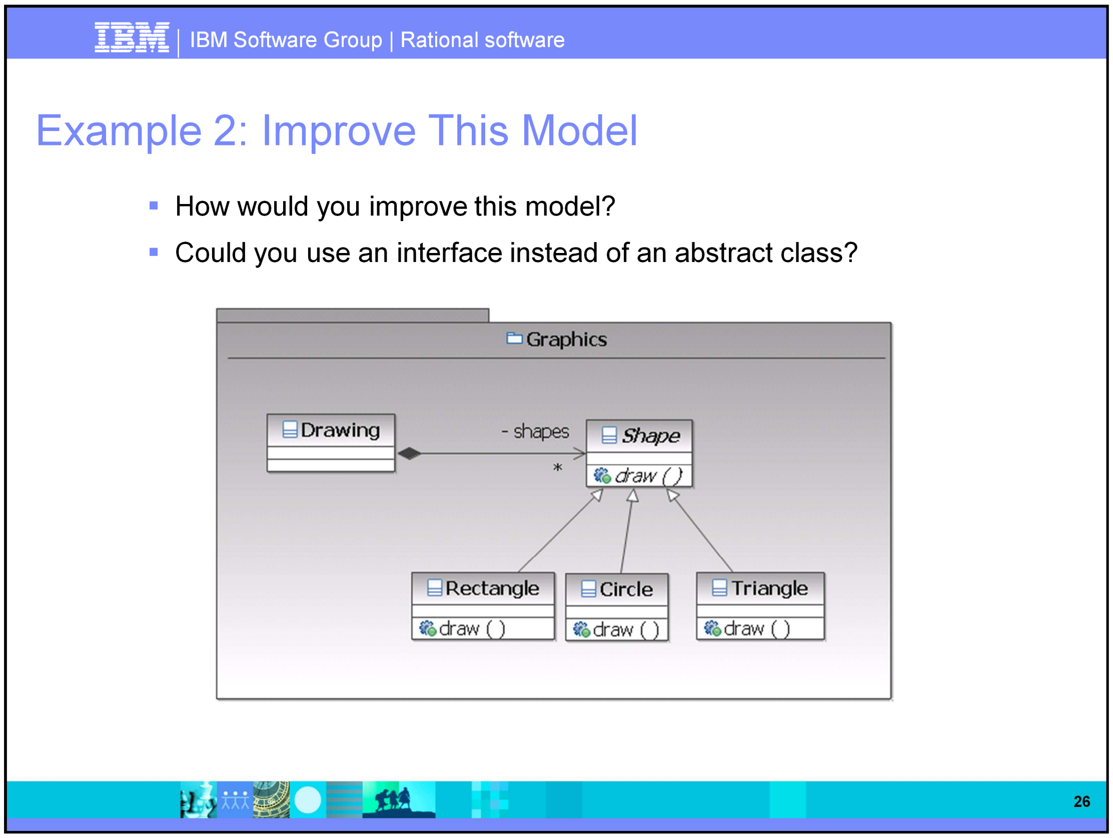
IBM Software Group | Rational software

Example 1: What Is Wrong With This Picture?

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



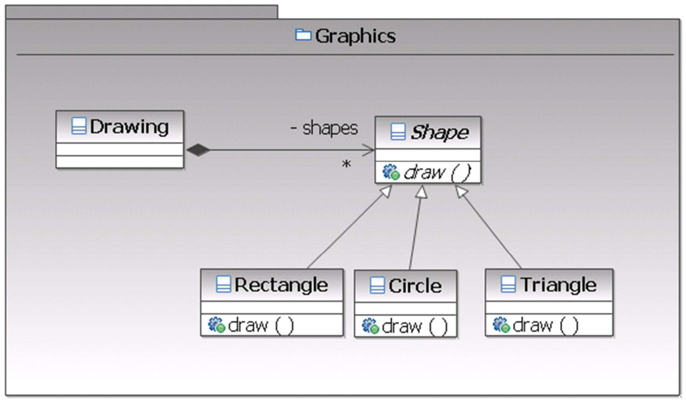
25



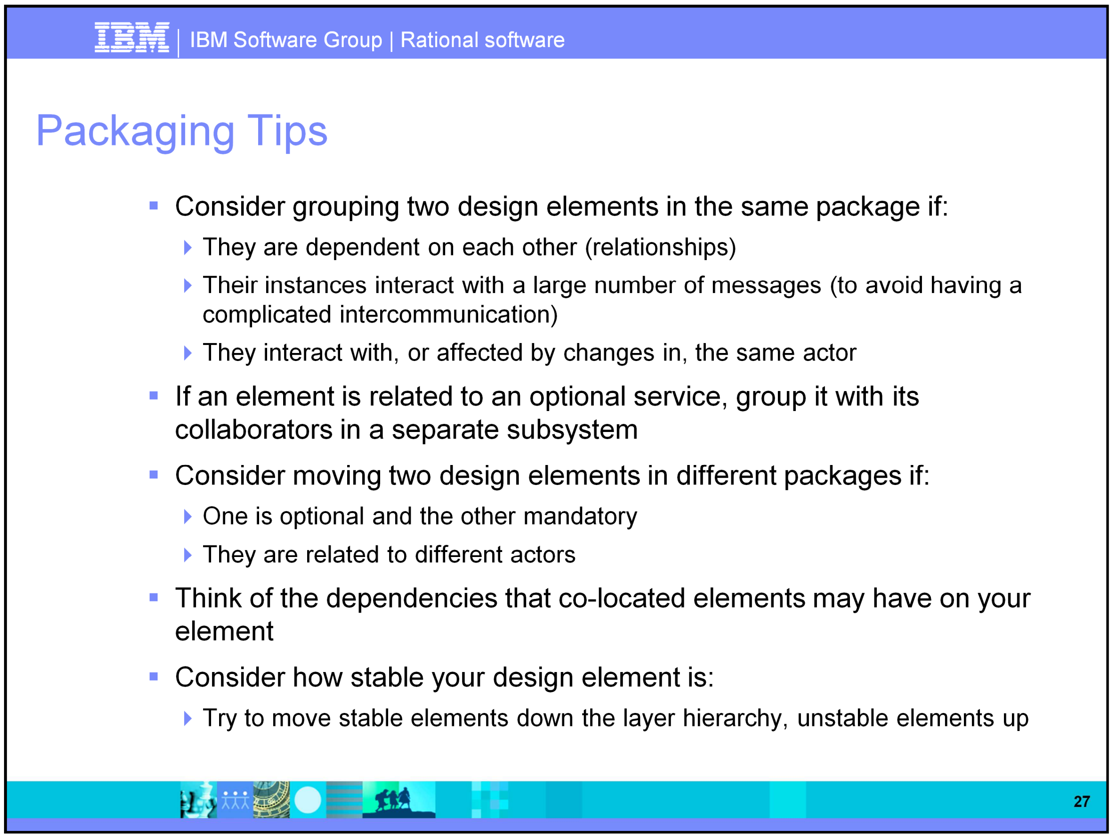
IBM | IBM Software Group | Rational software

Example 2: Improve This Model

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

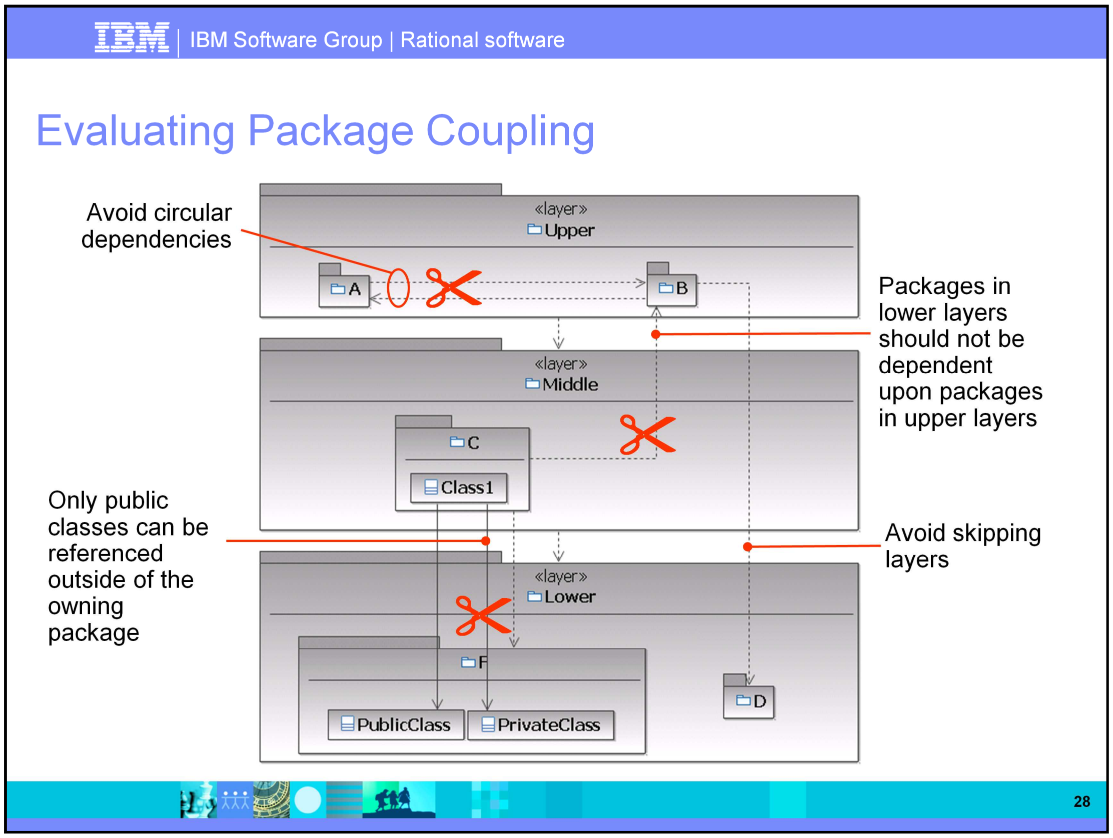


26



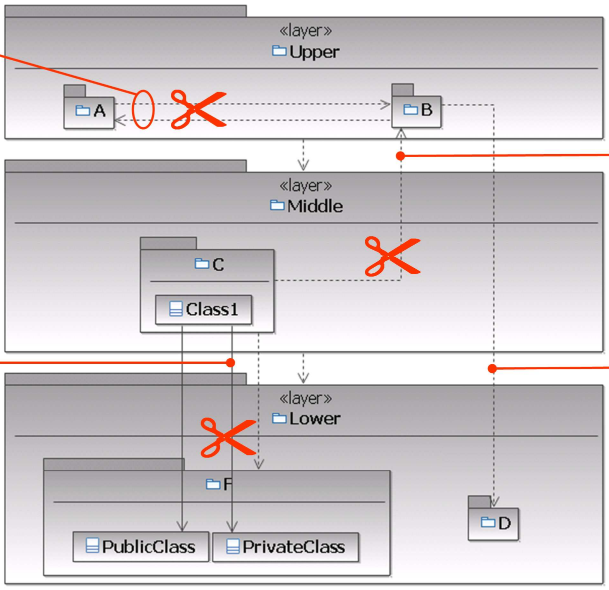
Packaging Tips

- 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



IBM | IBM Software Group | Rational software

Evaluating Package Coupling



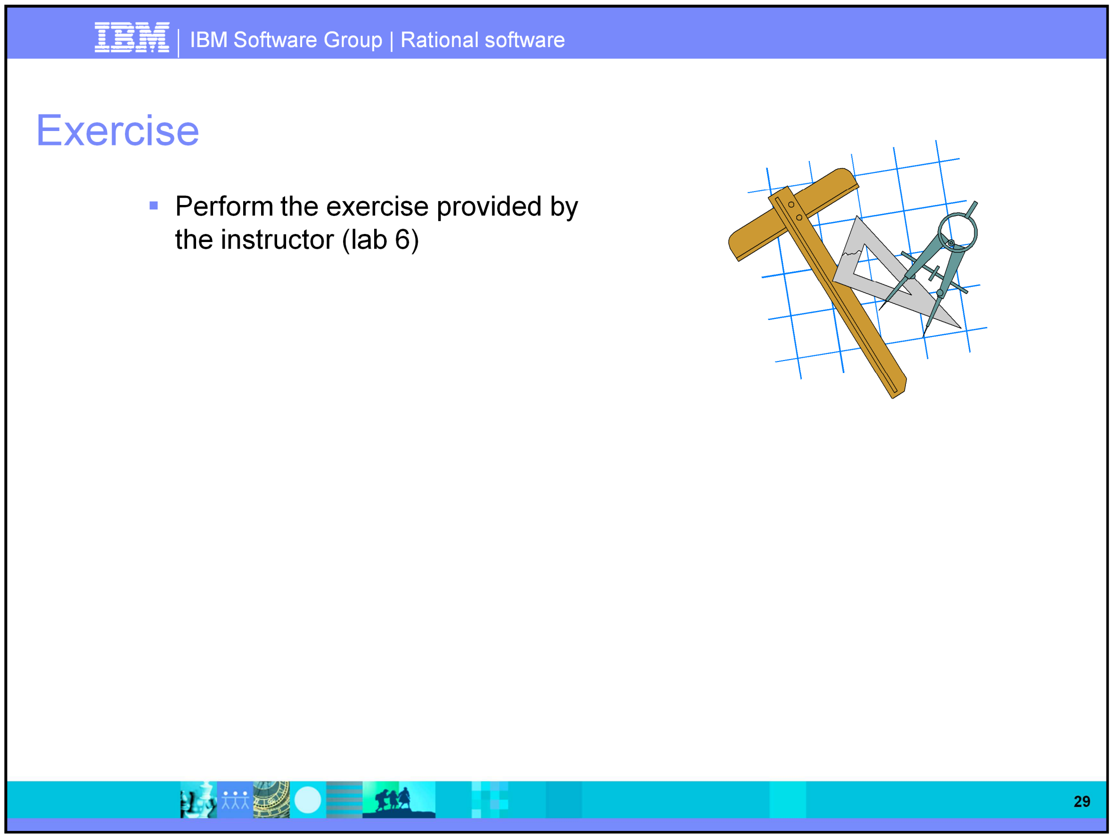
Avoid circular dependencies

Packages in lower layers should not be dependent upon packages in upper layers

Only public classes can be referenced outside of the owning package

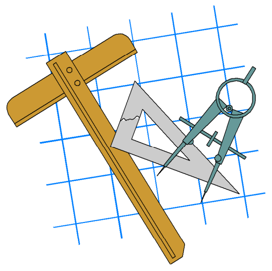
Avoid skipping layers

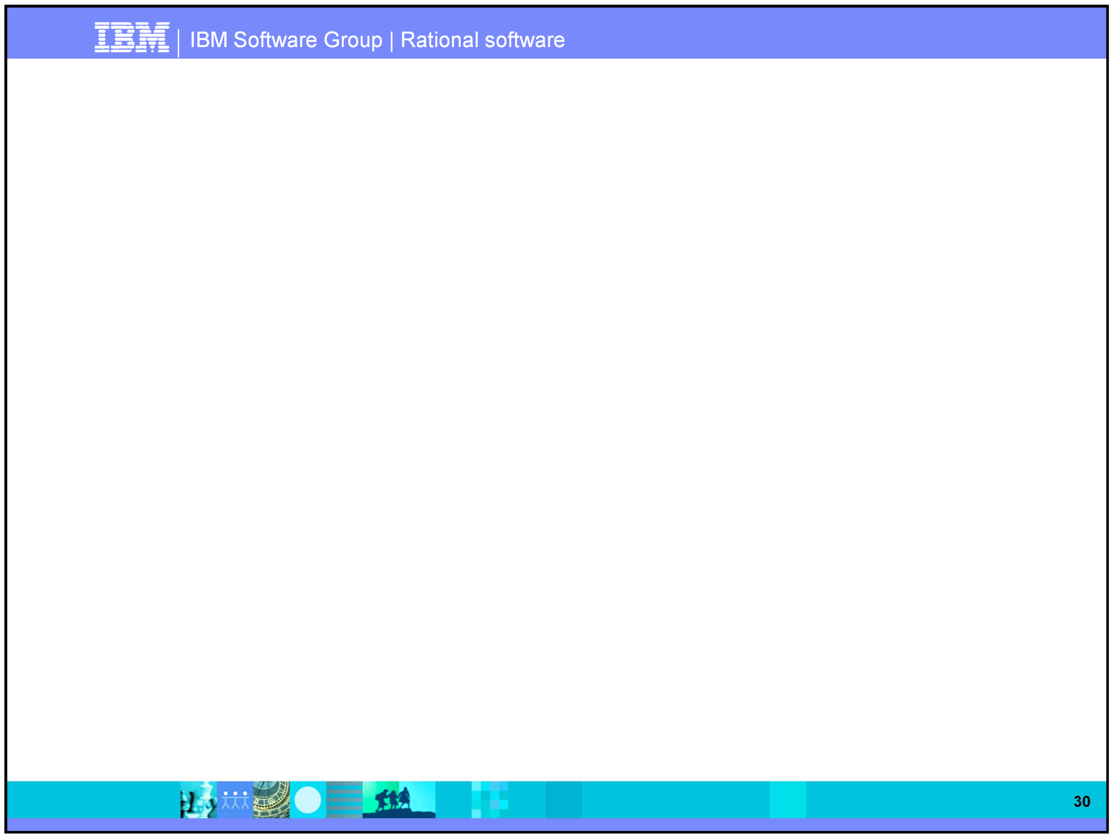
28



Exercise

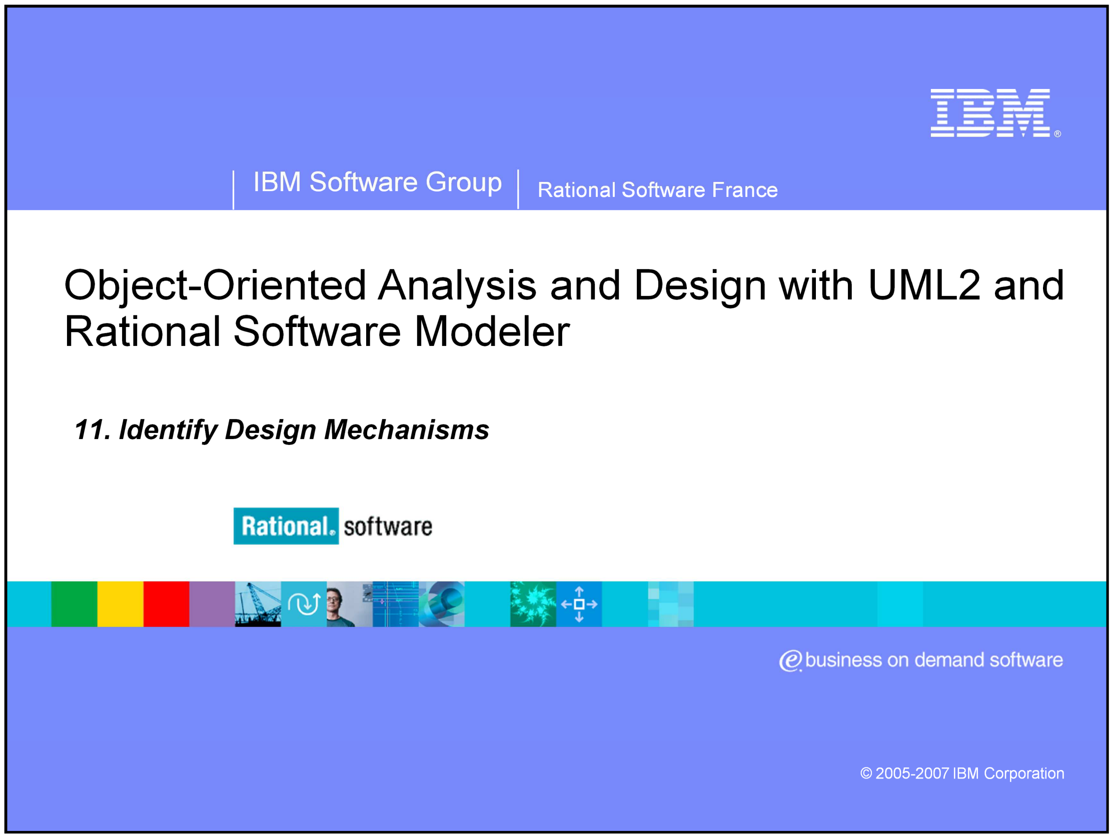
- Perform the exercise provided by the instructor (lab 6)





IBM Software Group | Rational software

30



IBM Software Group | Rational Software France

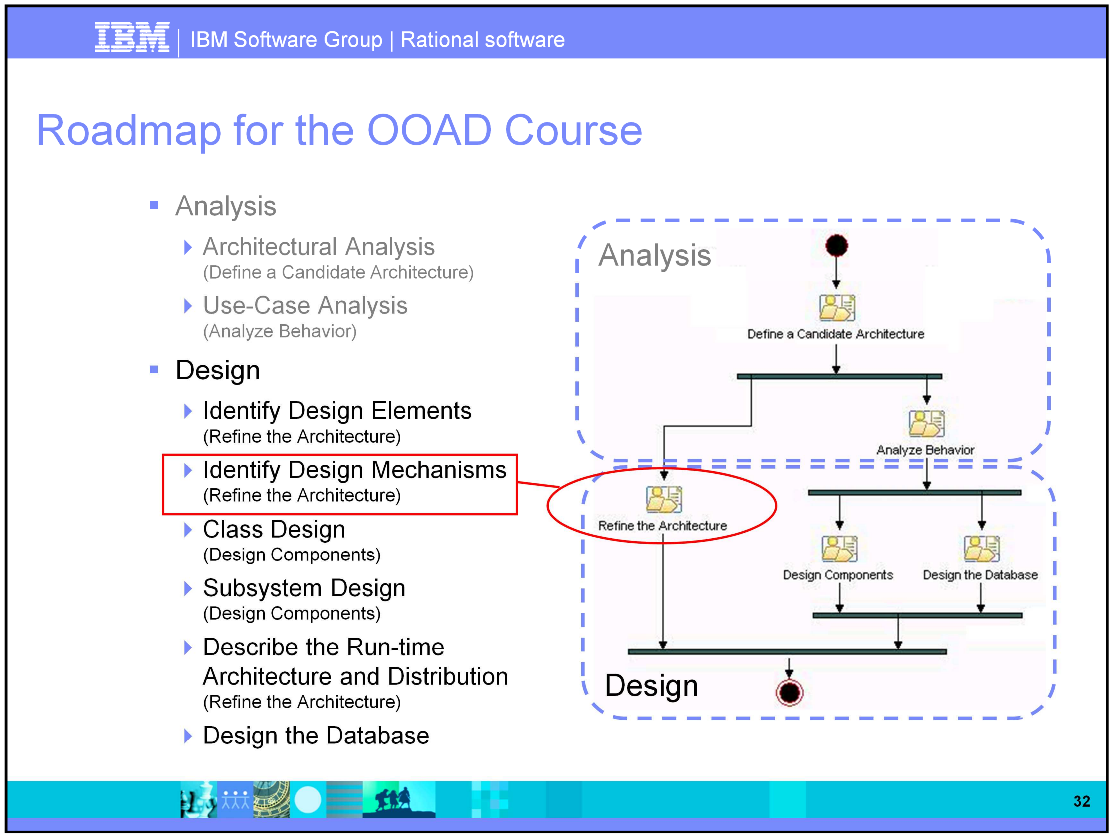
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

11. Identify Design Mechanisms

Rational software

e business on demand software

© 2005-2007 IBM Corporation

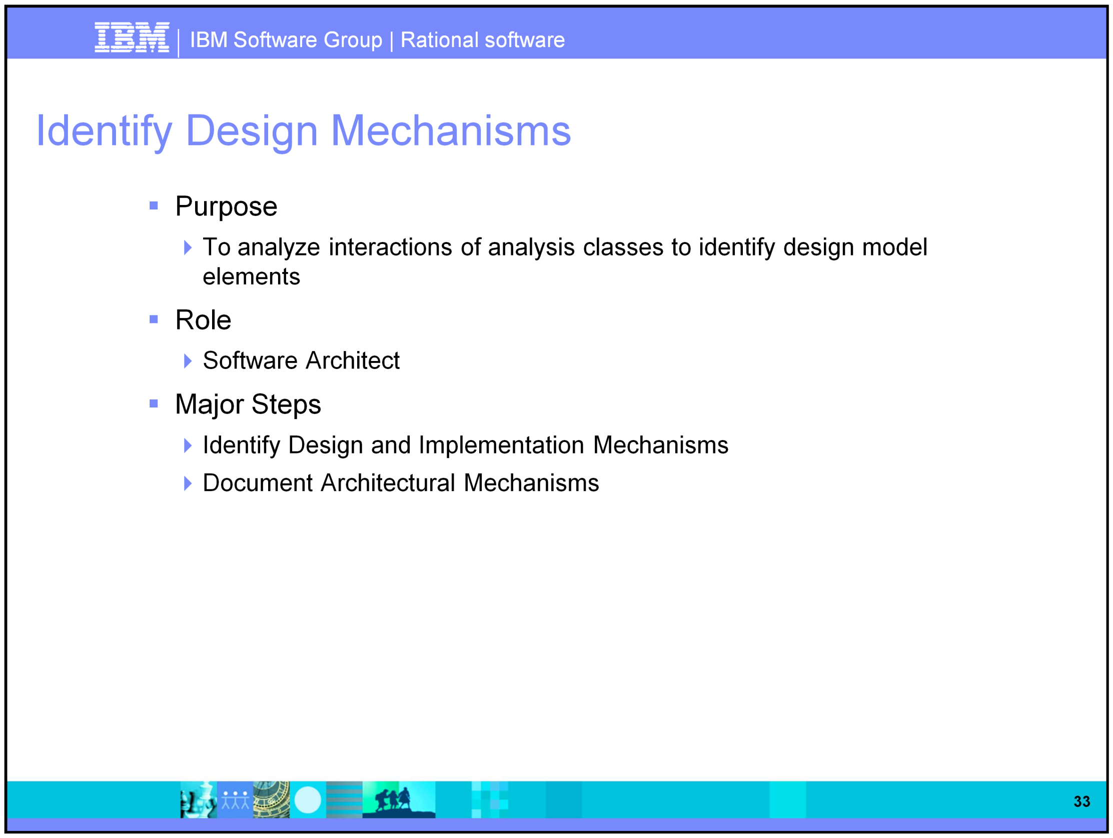


IBM | IBM Software Group | Rational software

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

32

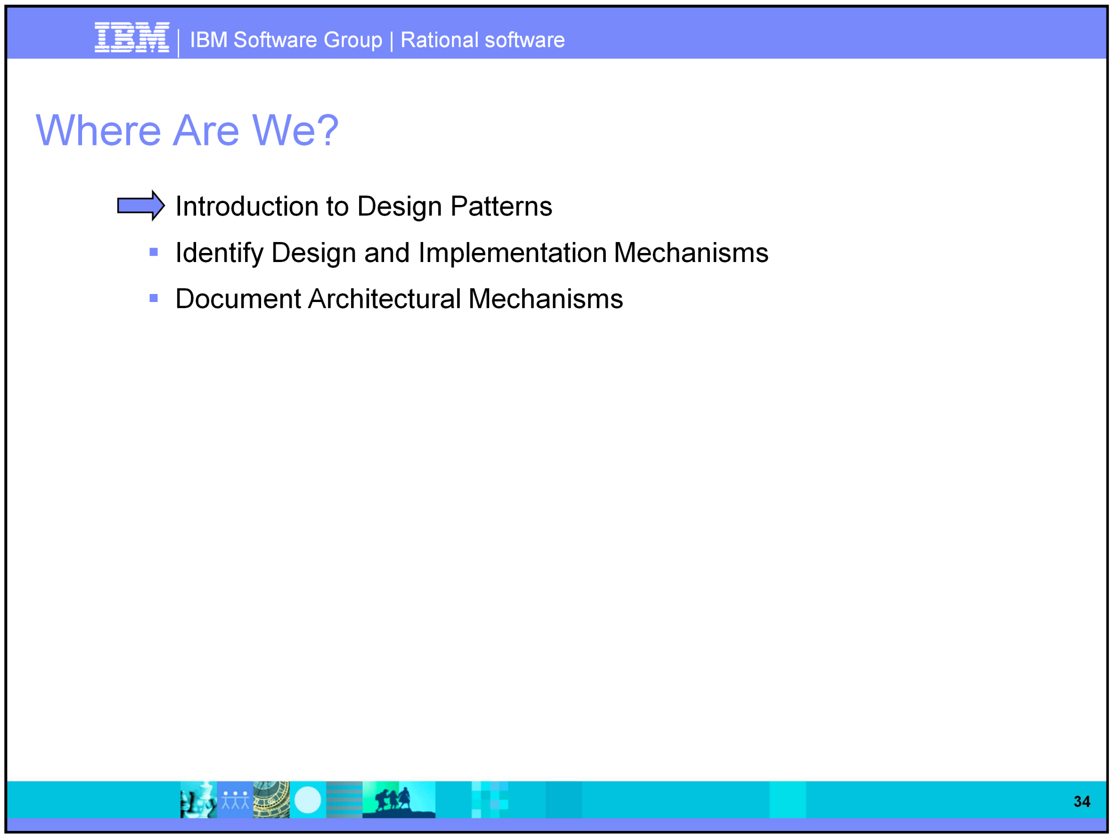


IBM Software Group | Rational software

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

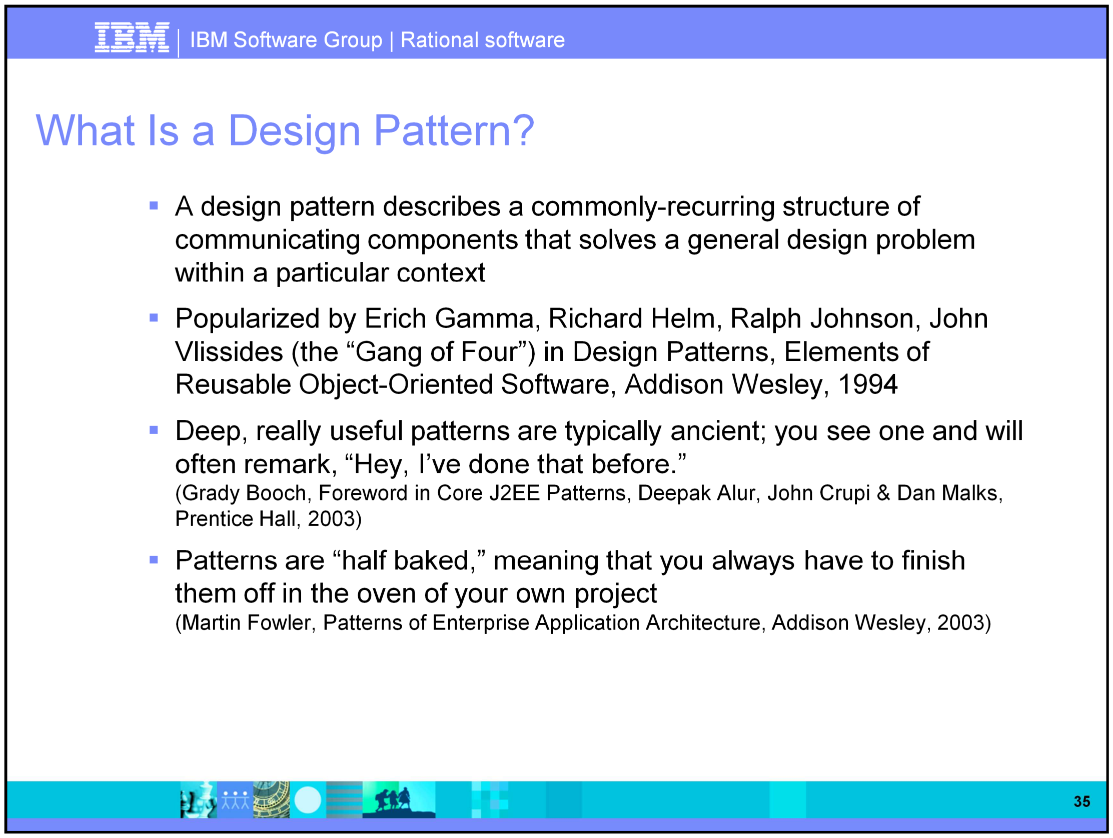


IBM | IBM Software Group | Rational software

Where Are We?

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

34



IBM Software Group | Rational software

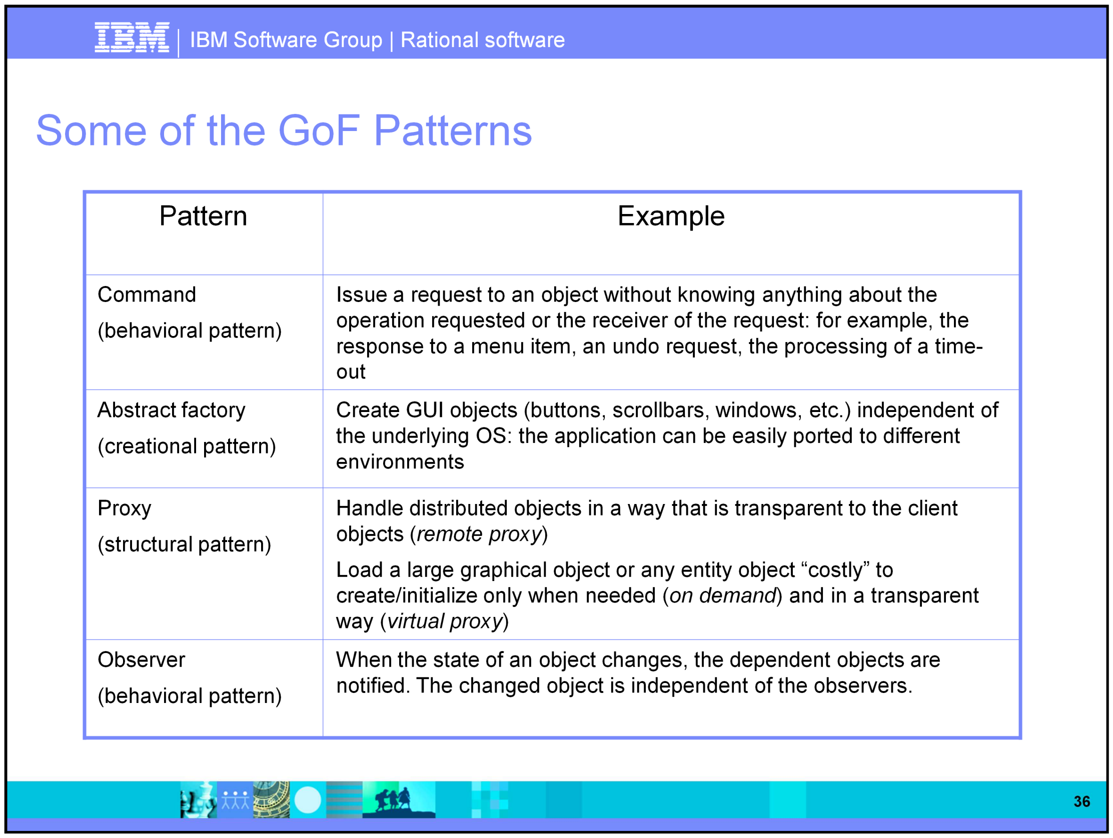
What Is a Design Pattern?

- 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

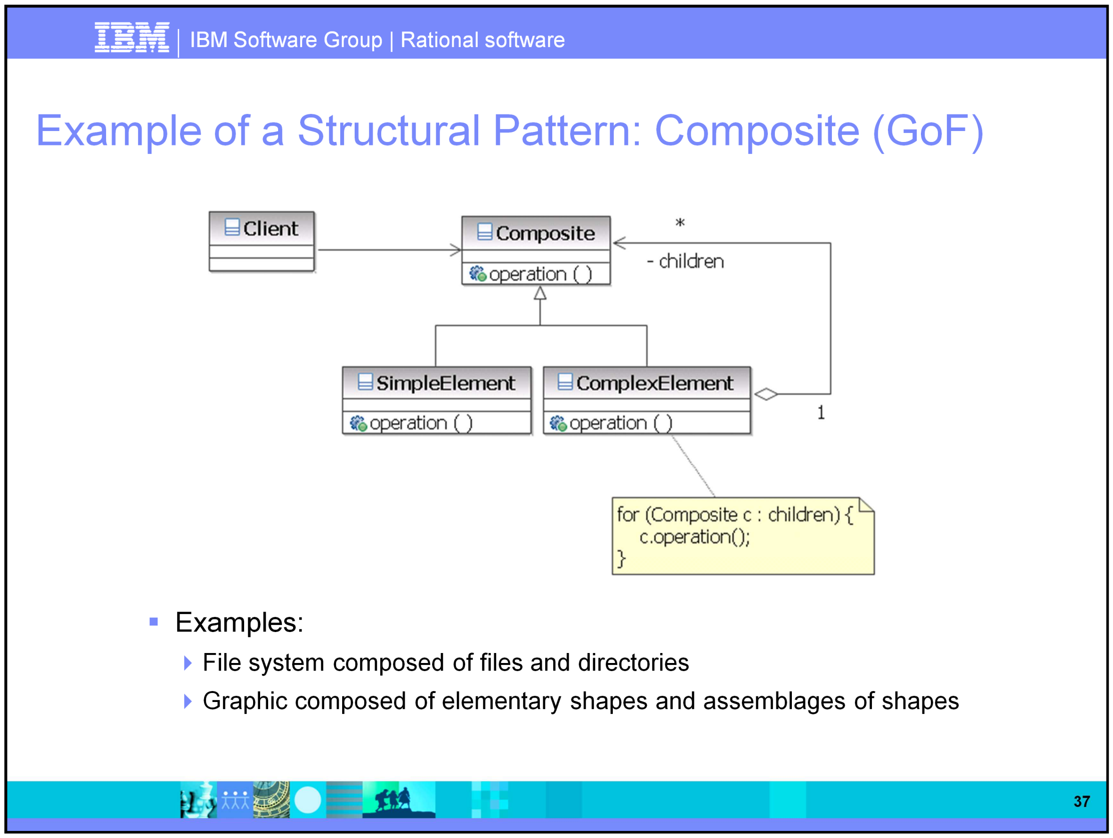
Design patterns are medium-to-small-scale patterns, smaller in scale than architectural patterns but typically independent of programming language. When a design pattern is bound, it forms a portion of a concrete design model (perhaps a portion of a design mechanism). Design patterns tend, because of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

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

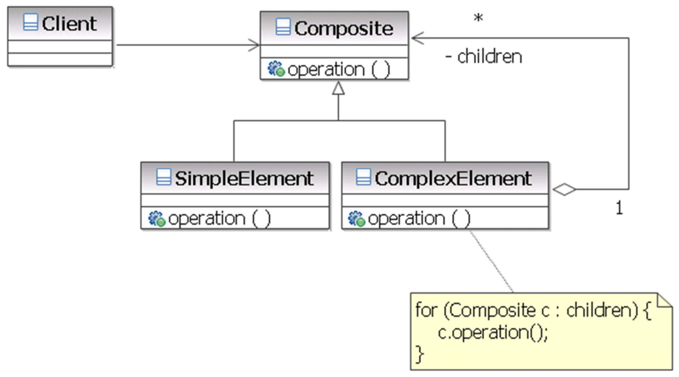


Some of the GoF Patterns

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.



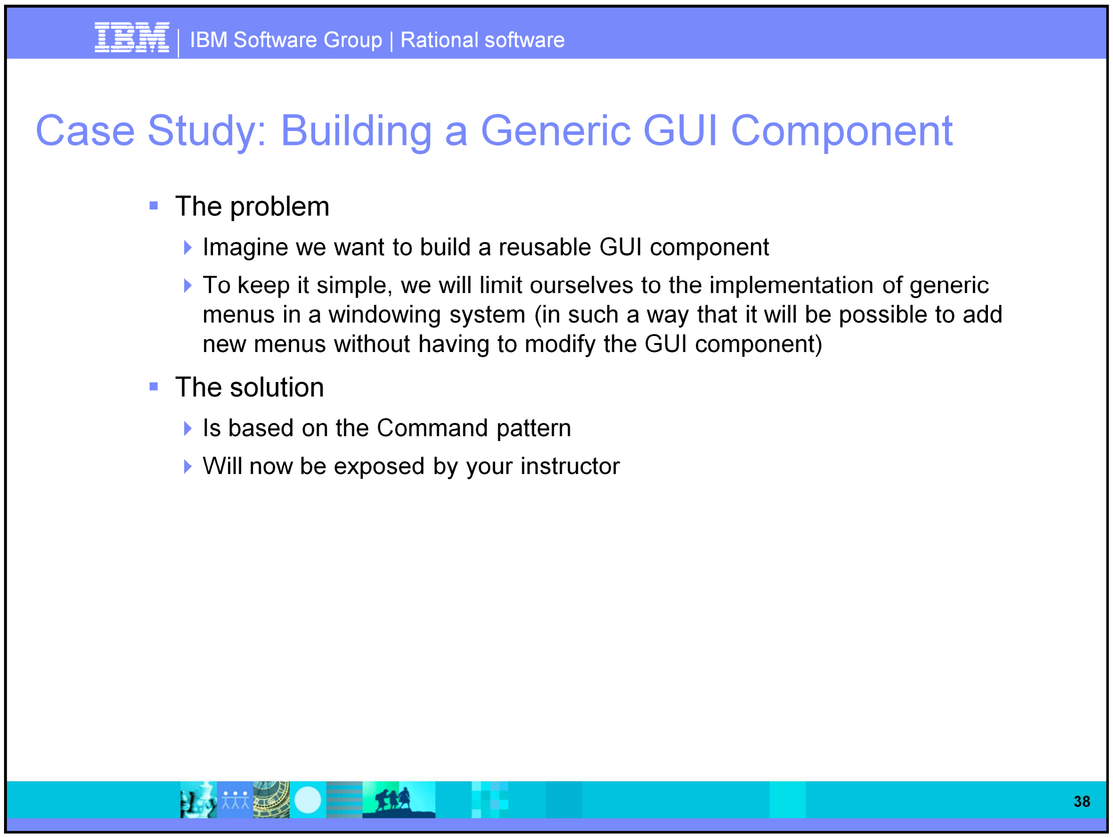
Example of a Structural Pattern: Composite (GoF)



```
for (Composite c : children) {
    c.operation();
}
```

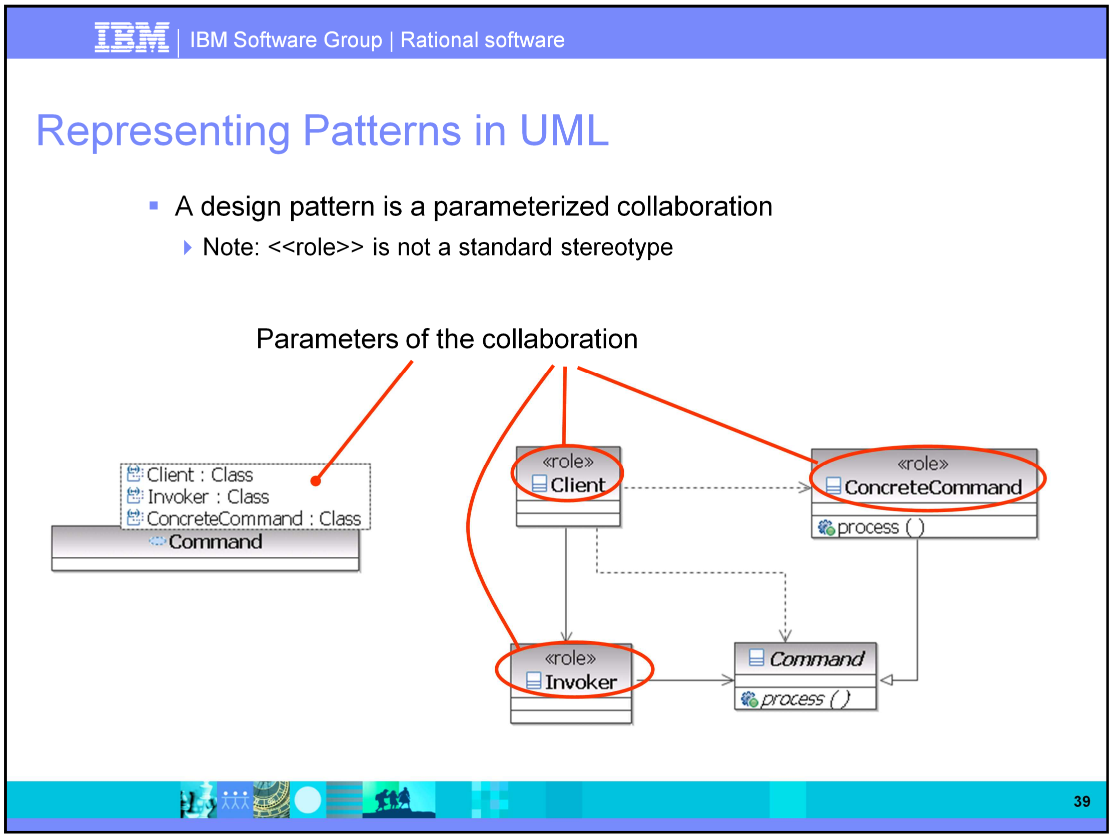
- Examples:
  - File system composed of files and directories
  - Graphic composed of elementary shapes and assemblages of shapes

37



Case Study: Building a Generic GUI Component

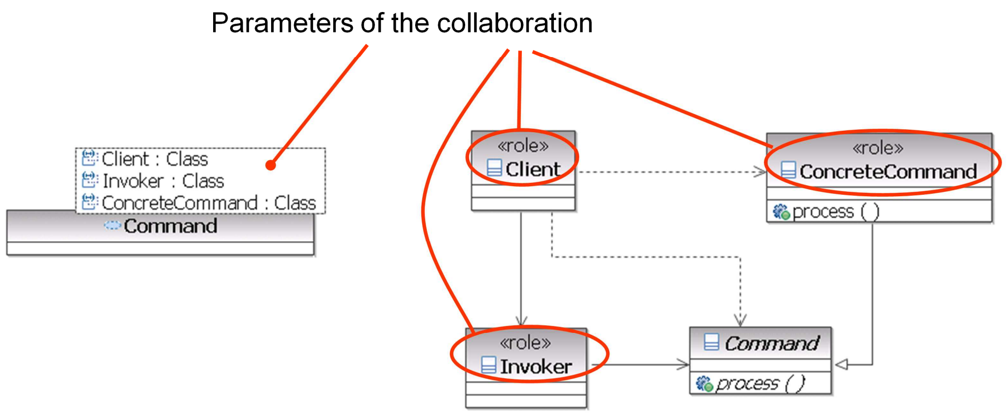
- The problem
  - Imagine we want to build a reusable GUI component
  - To keep it simple, we will limit ourselves to the implementation of generic menus in a windowing system (in such a way that it will be possible to add new menus without having to modify the GUI component)
- The solution
  - Is based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



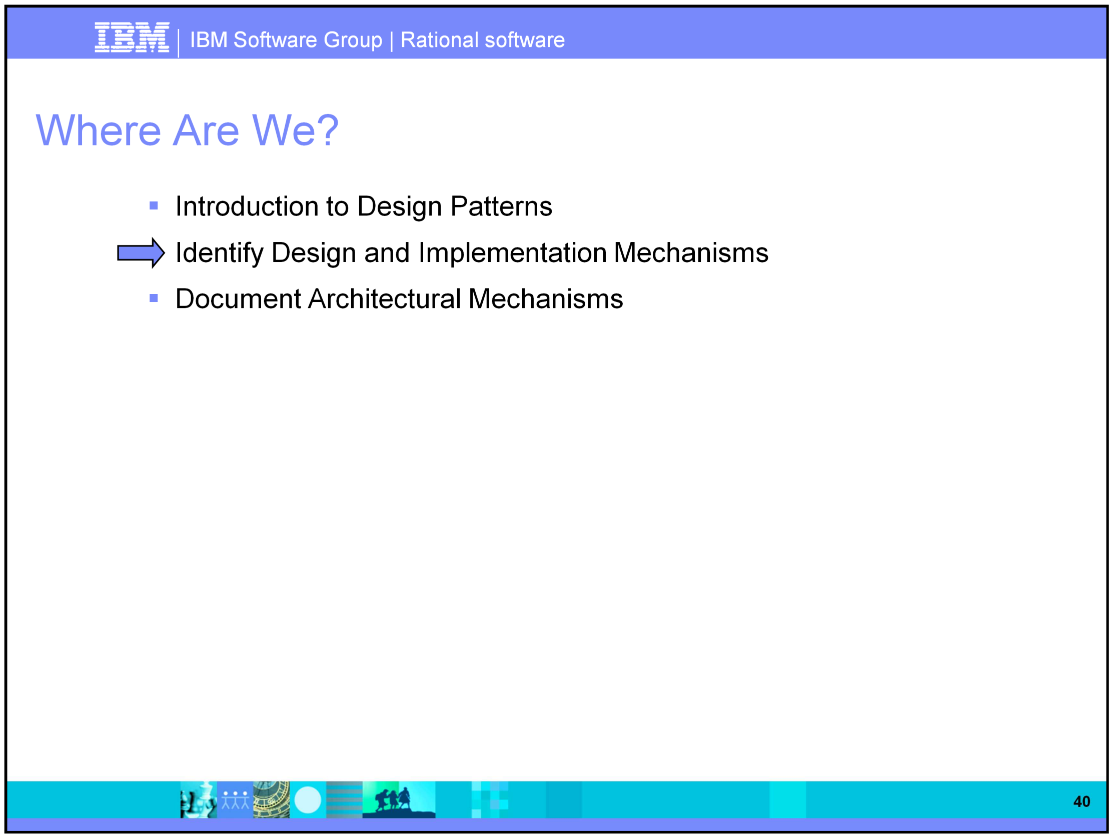
IBM Software Group | Rational software

Representing Patterns in UML

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

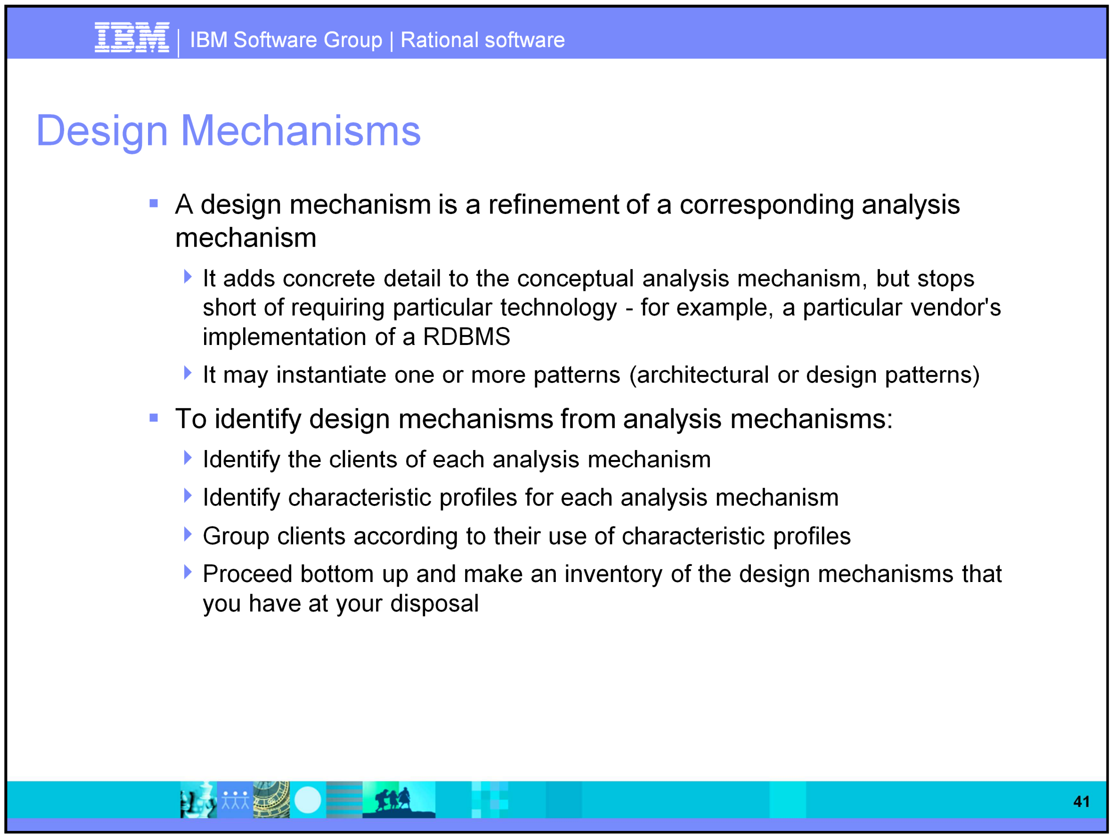


39



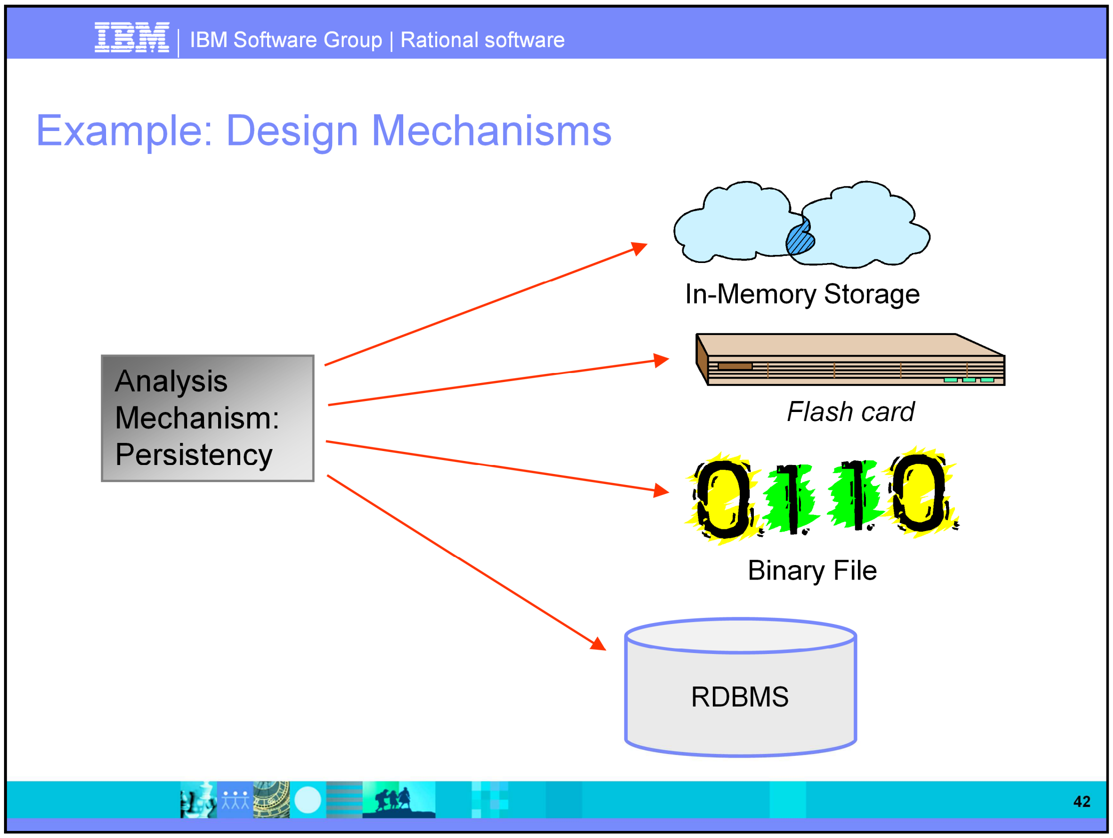
Where Are We?

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



Design Mechanisms

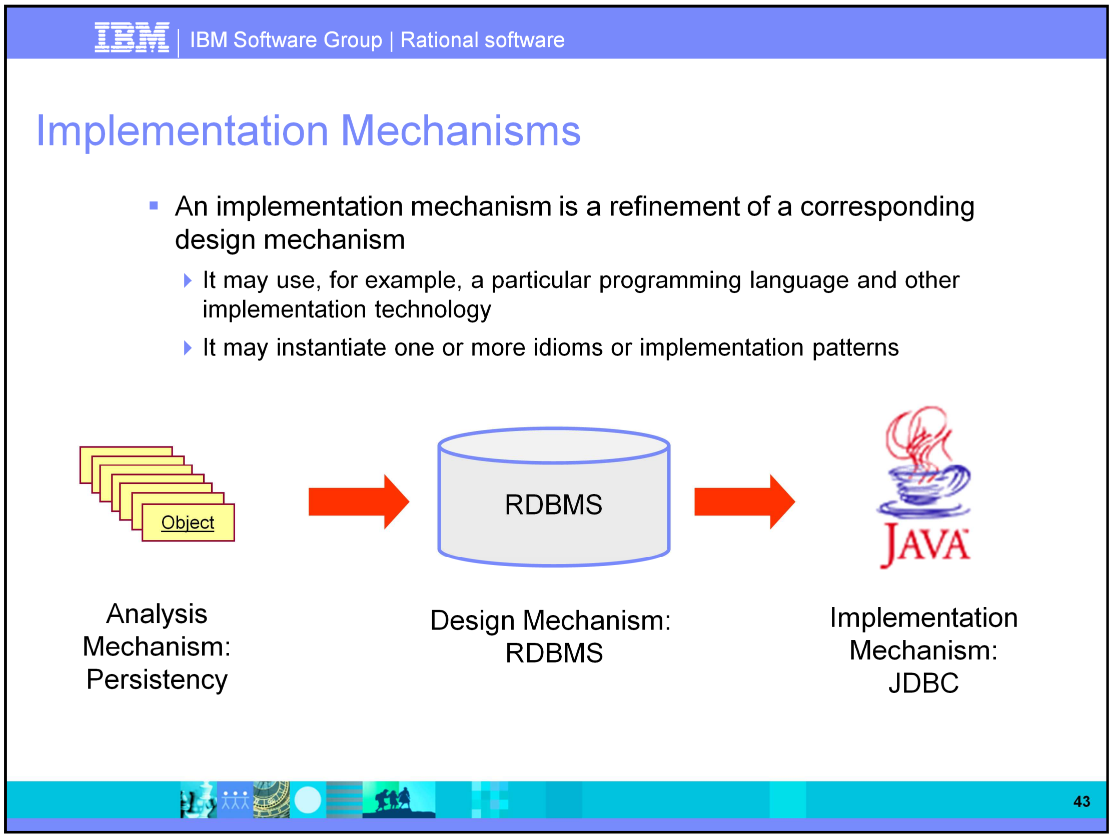
- A design mechanism is a refinement of a corresponding analysis mechanism
  - It adds concrete detail to the conceptual analysis mechanism, but stops short of requiring particular technology - for example, a particular vendor's implementation of a RDBMS
  - It may instantiate one or more patterns (architectural 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 | IBM Software Group | Rational software

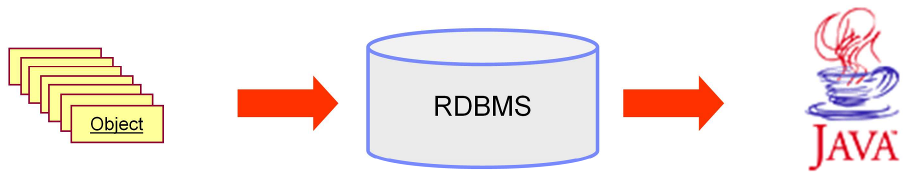
Example: Design Mechanisms

42



Implementation Mechanisms

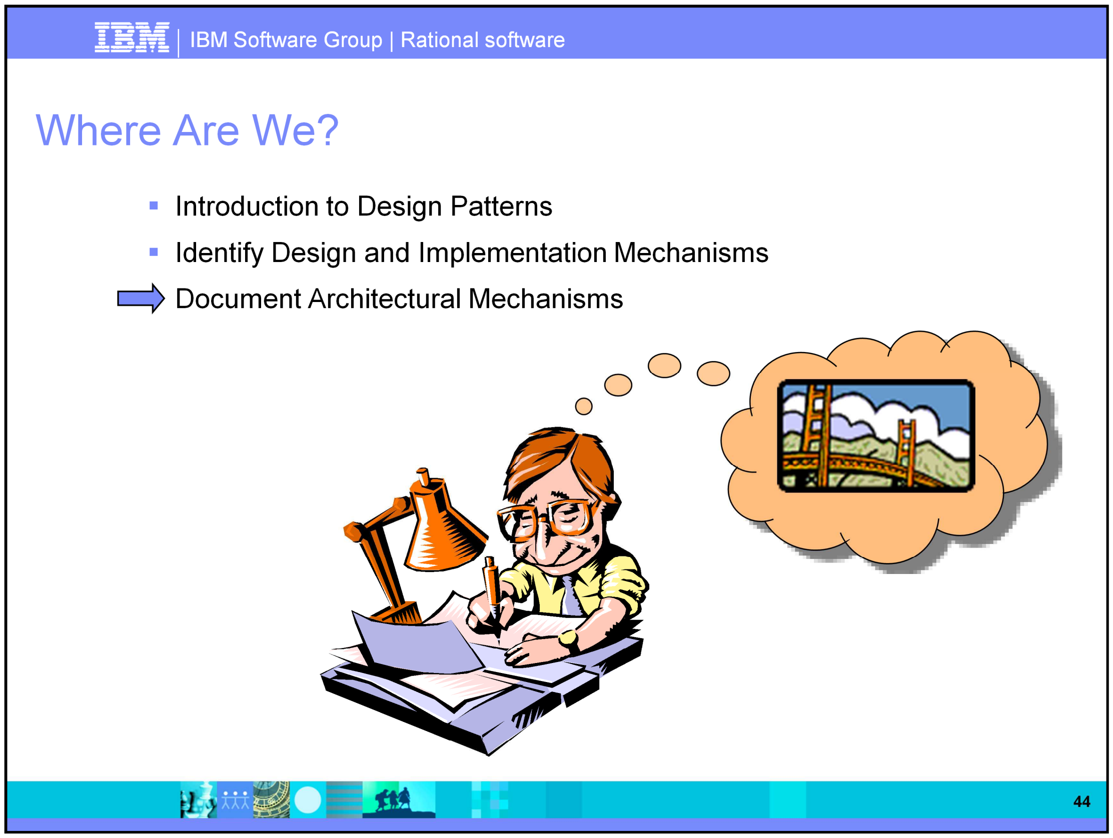
- An implementation mechanism is a refinement of a corresponding design mechanism
  - It may use, for example, a particular programming language and other implementation technology
  - It may instantiate one or more idioms or implementation patterns



Analysis Mechanism: Persistency

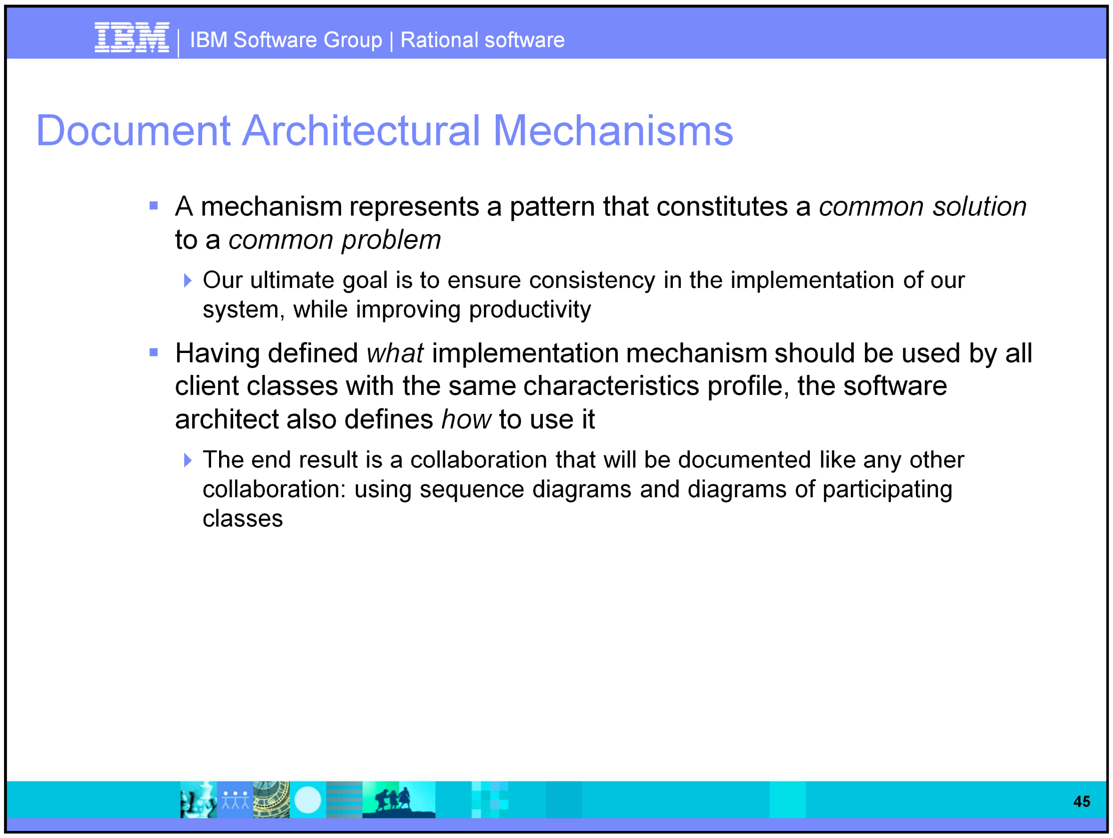
Design Mechanism: RDBMS

Implementation Mechanism: JDBC



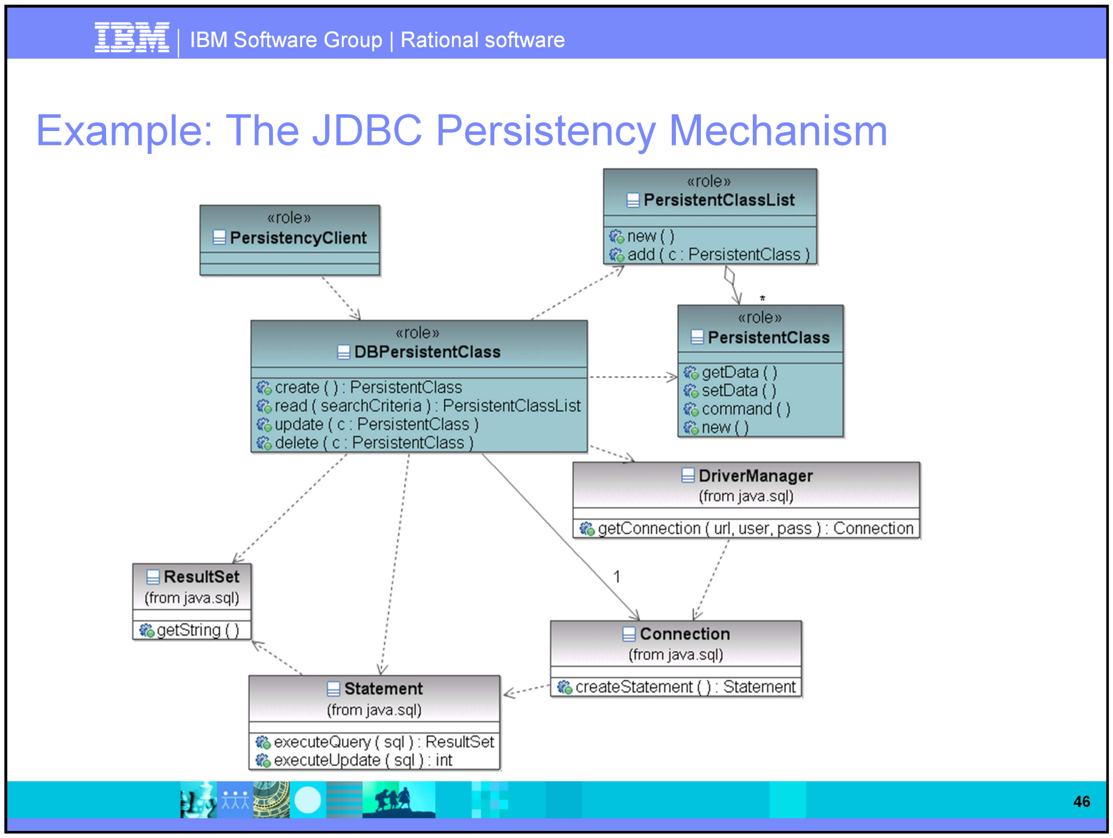
Where Are We?

- 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


IBM Software Group | Rational software

Example: The JDBC Persistency Mechanism

«role»
PersistencyClient
«role»
PersistentClassList
new ( )
add (c : PersistentClass)
*
«role»
PersistentClass
getData ( )
setData ( )
command ( )
new ( )
«role»
DBPersistentClass
create ( ) : PersistentClass
read (searchCriteria) : PersistentClassList
update (c : PersistentClass)
delete (c : PersistentClass)
DriverManager
(from java.sql)
getConnection (url, user, pass) : Connection
1
ResultSet
(from java.sql)
getString ( )
Connection
(from java.sql)
createStatement ( ) : Statement
Statement
(from java.sql)
executeQuery (sql) : ResultSet
executeUpdate (sql) : int

46

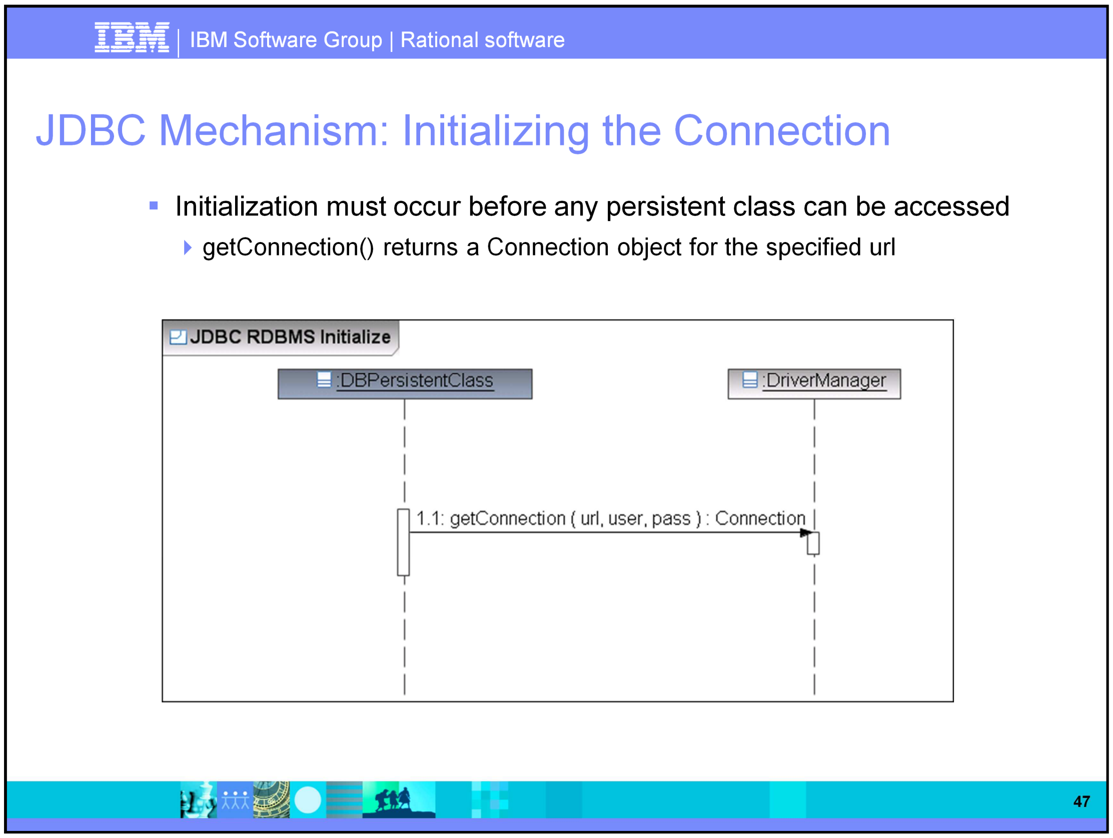
The next few slides demonstrate the JDBC mechanism chosen for our persistent classes in our example.

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.

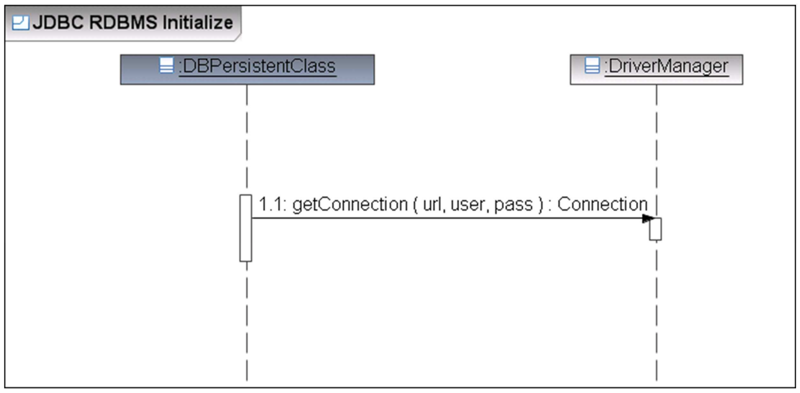
The PersistentClassList is used to return a set of persistent objects as a result of a database query (for example, DBClass.read()).

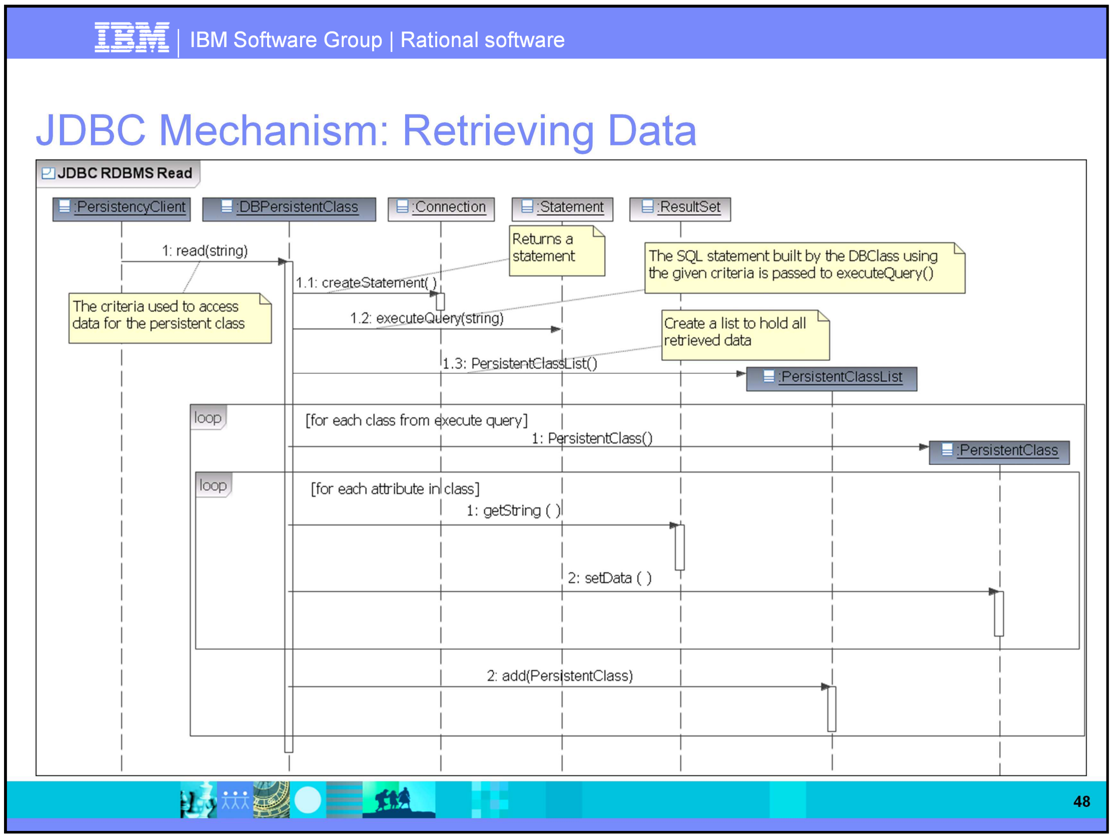
The <<role>> stereotype was used for anything that should be regarded as a placeholder for the actual design element to be supplied by the developer. This convention makes it easier to apply the mechanism, because it is easier to recognize what the designer must supply.



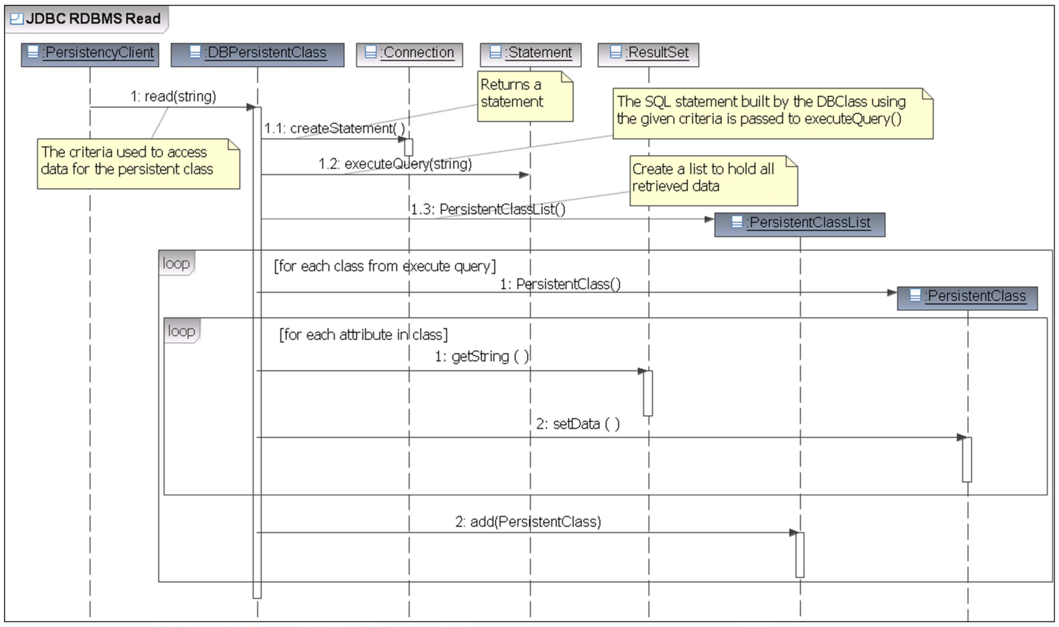
JDBC Mechanism: Initializing the Connection

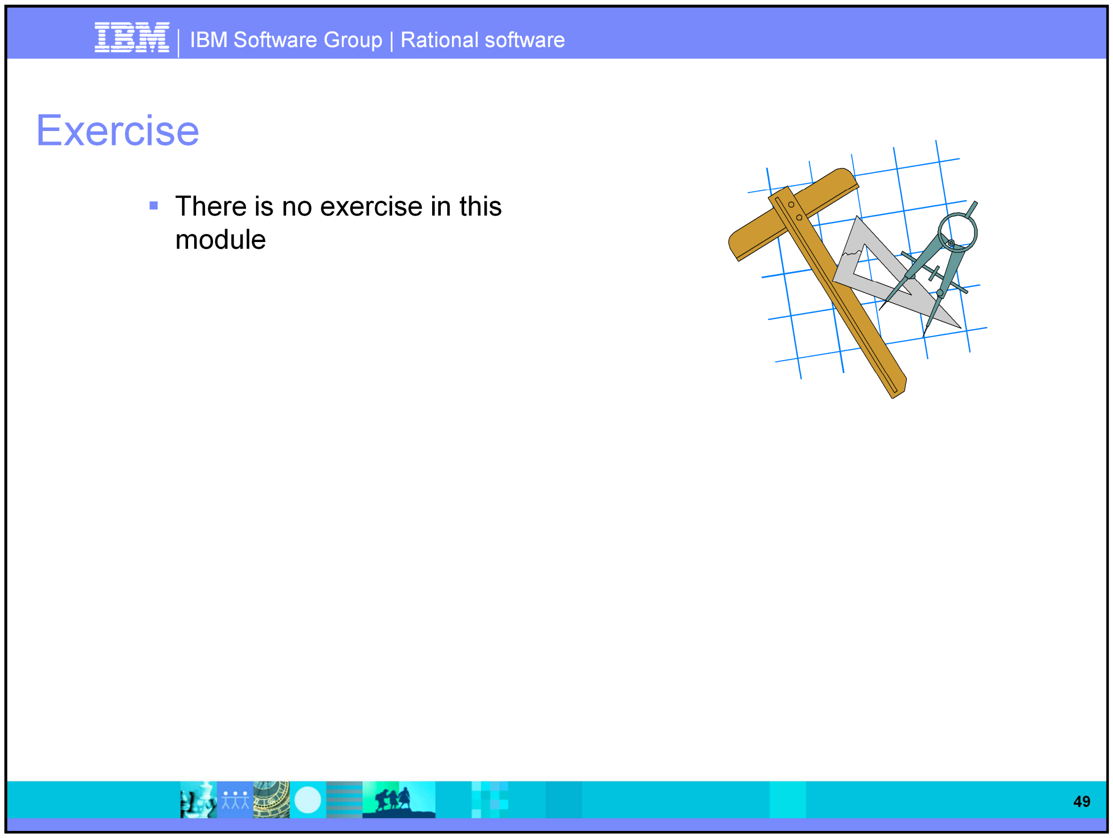
- Initialization must occur before any persistent class can be accessed
  - getConnection() returns a Connection object for the specified url





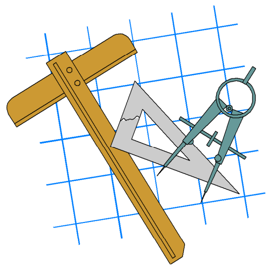
JDBC Mechanism: Retrieving Data

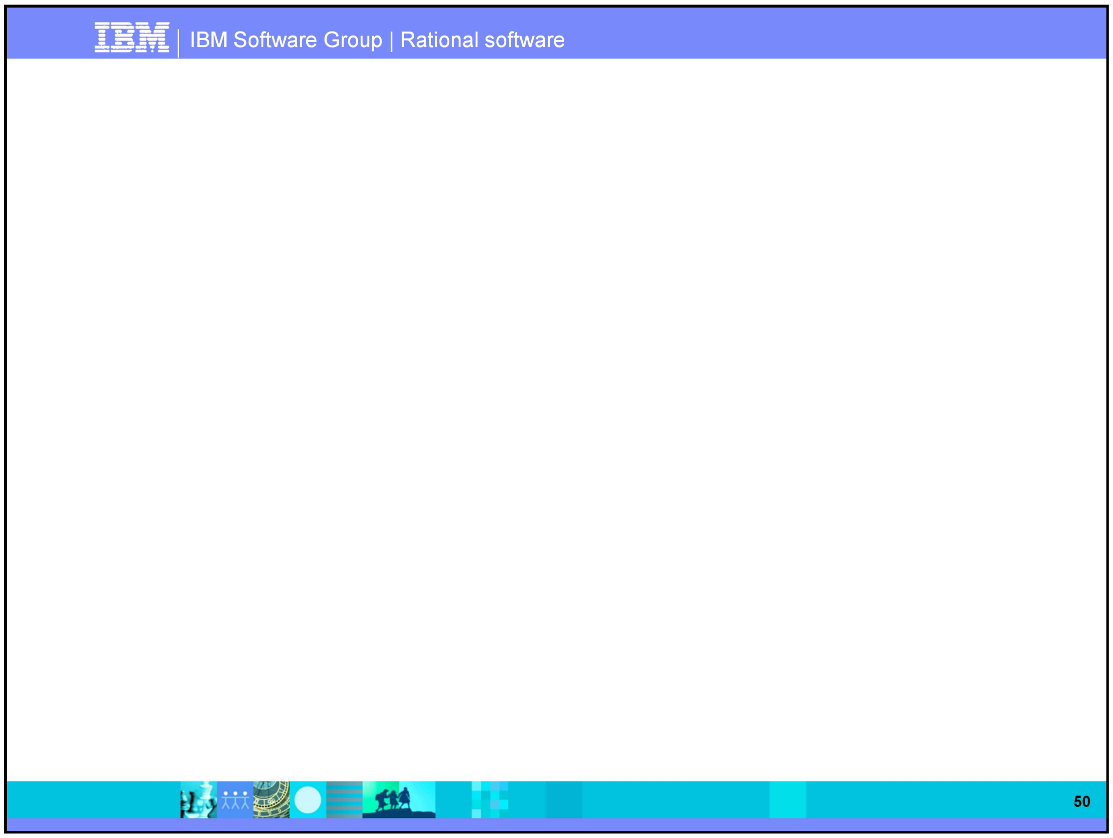

JDBC RDBMS Read
:PersistencyClient
:DBPersistentClass
:Connection
:Statement
:ResultSet
Returns a statement
The SQL statement built by the DBClass using the given criteria is passed to executeQuery()
1: read(string)
1.1: createStatement( )
The criteria used to access data for the persistent class
1.2: executeQuery(string)
Create a list to hold all retrieved data
1.3: PersistentClassList()
:PersistentClassList
loop
[for each class from execute query]
1: PersistentClass()
:PersistentClass
loop
[for each attribute in class]
1: getString ( )
2: setData ( )
2: add(PersistentClass)
48

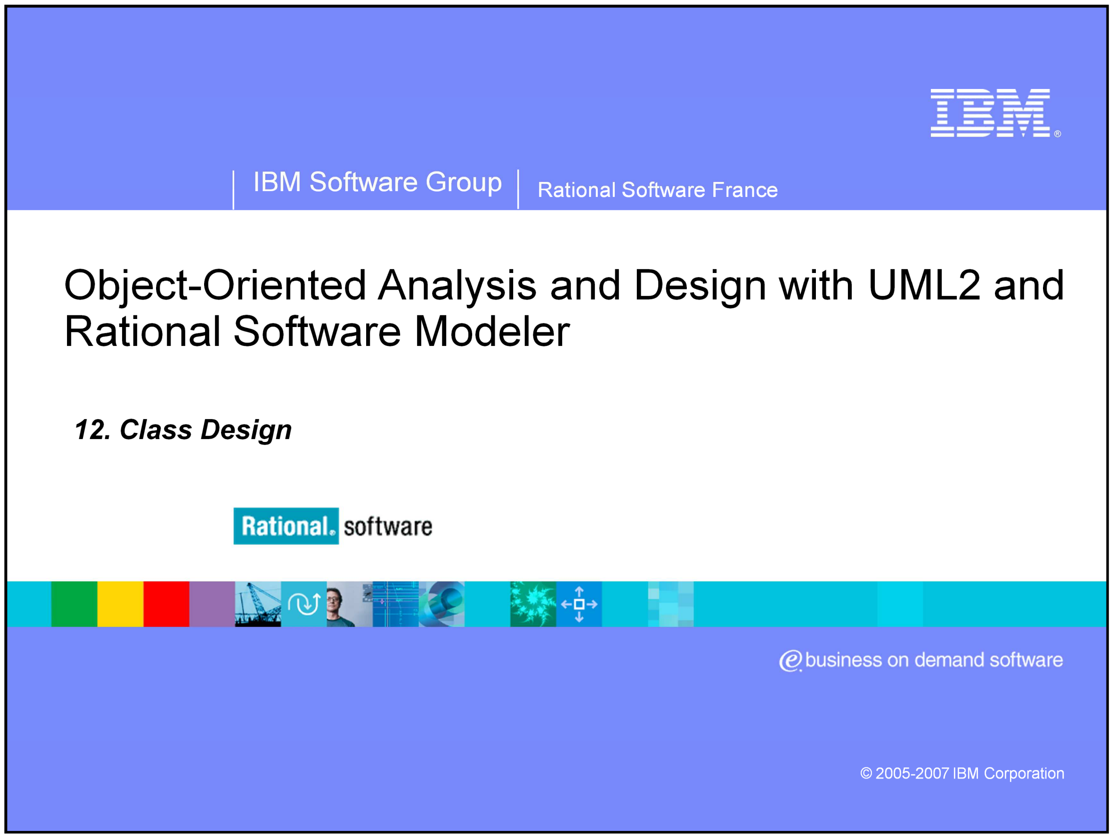


Exercise

- There is no exercise in this module




IBM Software Group | Rational software
50



IBM Software Group | Rational Software France

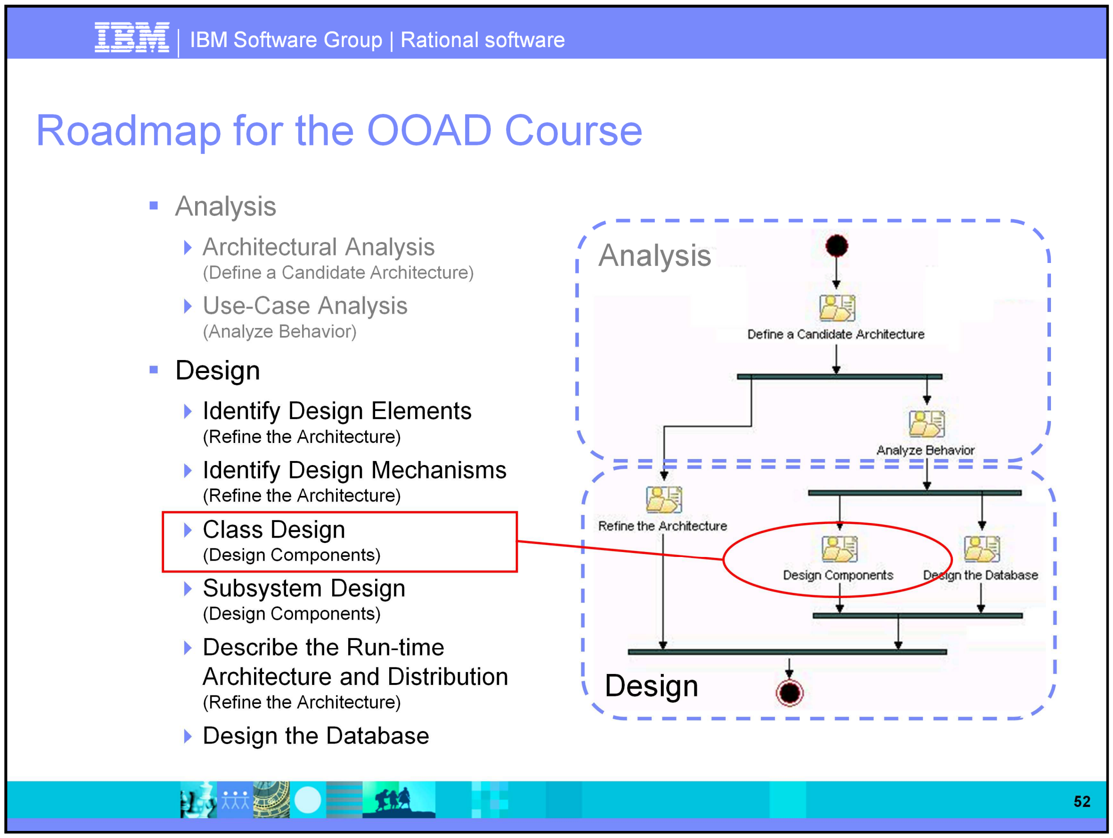
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

12. Class Design

Rational. software

e business on demand software

© 2005-2007 IBM Corporation

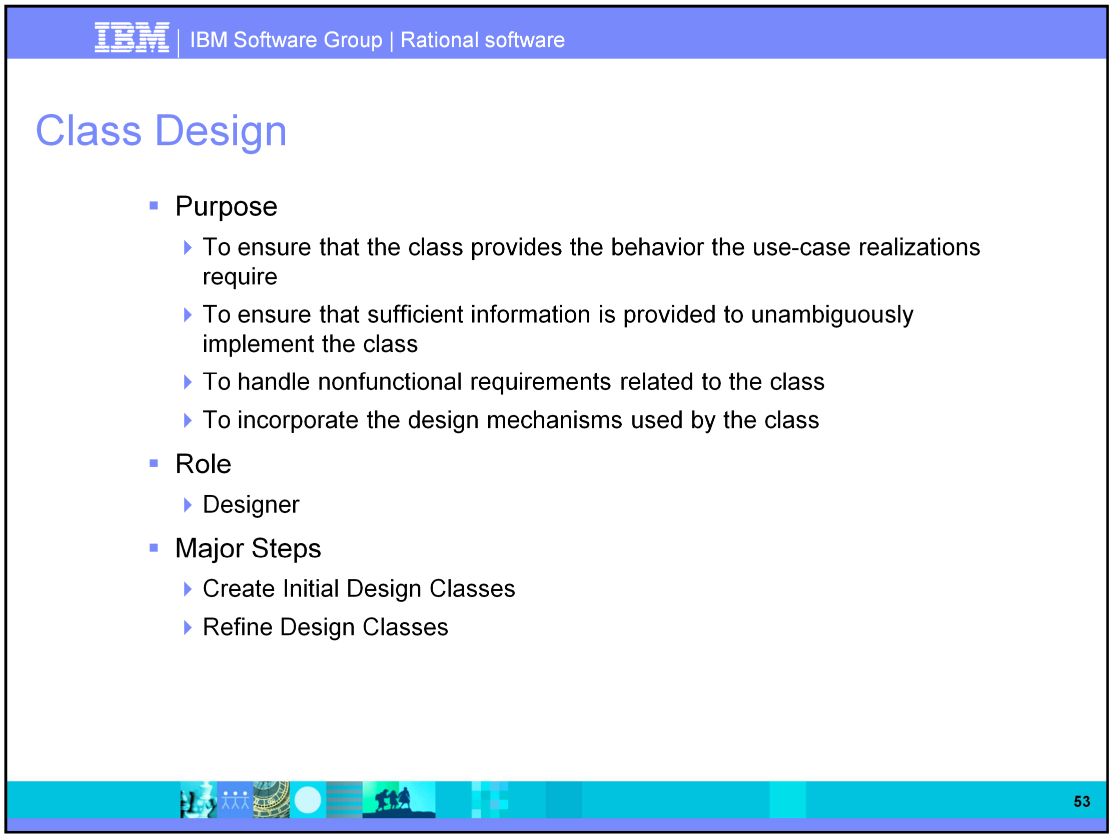


IBM Software Group | Rational software

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

52

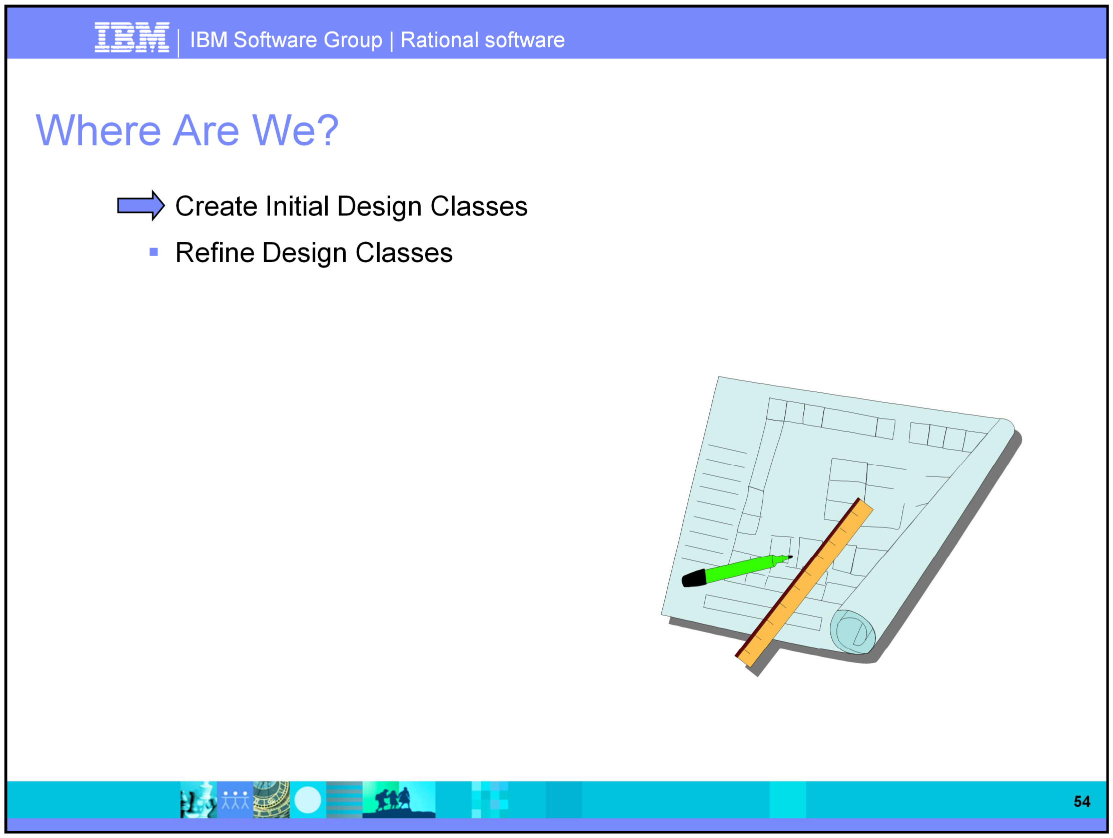


IBM Software Group | Rational software

Class Design

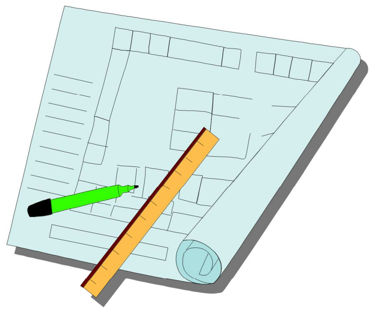
- Purpose
  - To ensure that the class provides the behavior the use-case realizations require
  - To ensure that sufficient information is provided to unambiguously implement the class
  - To handle nonfunctional requirements related to the class
  - To incorporate the design mechanisms used by the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

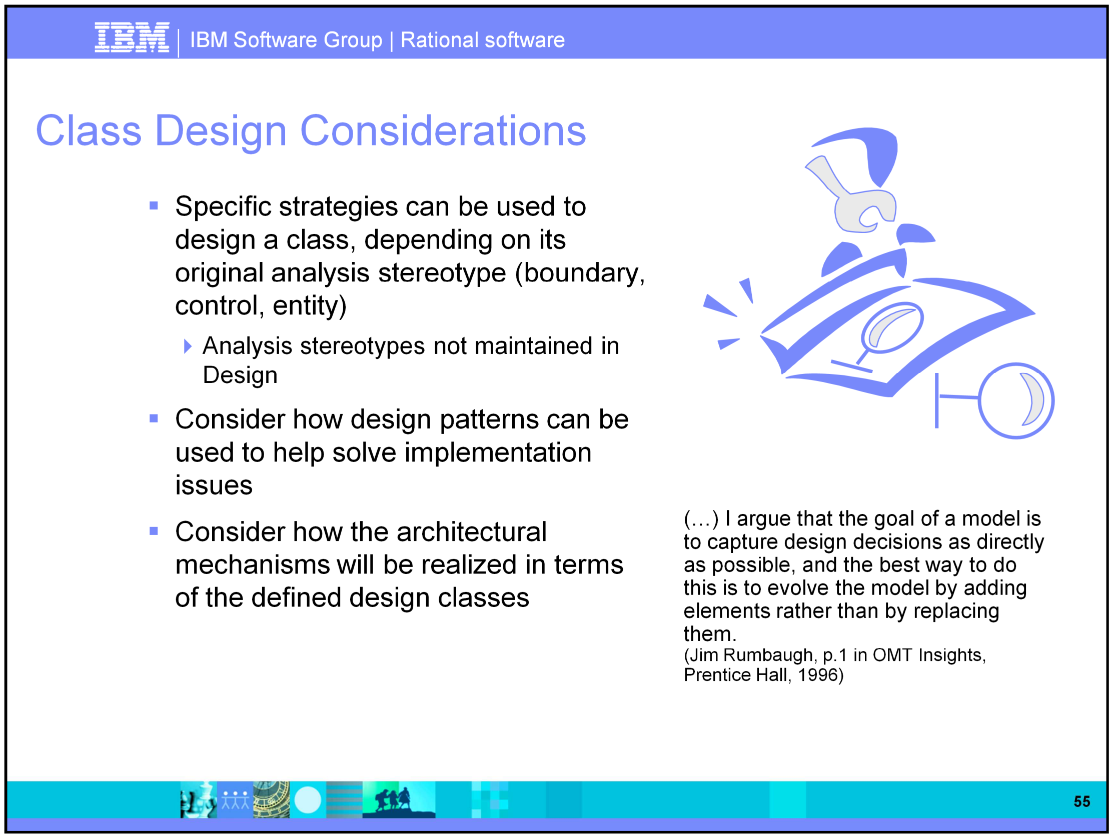
53



Where Are We?

- ➡ Create Initial Design Classes
- Refine Design Classes




IBM | IBM Software Group | Rational software

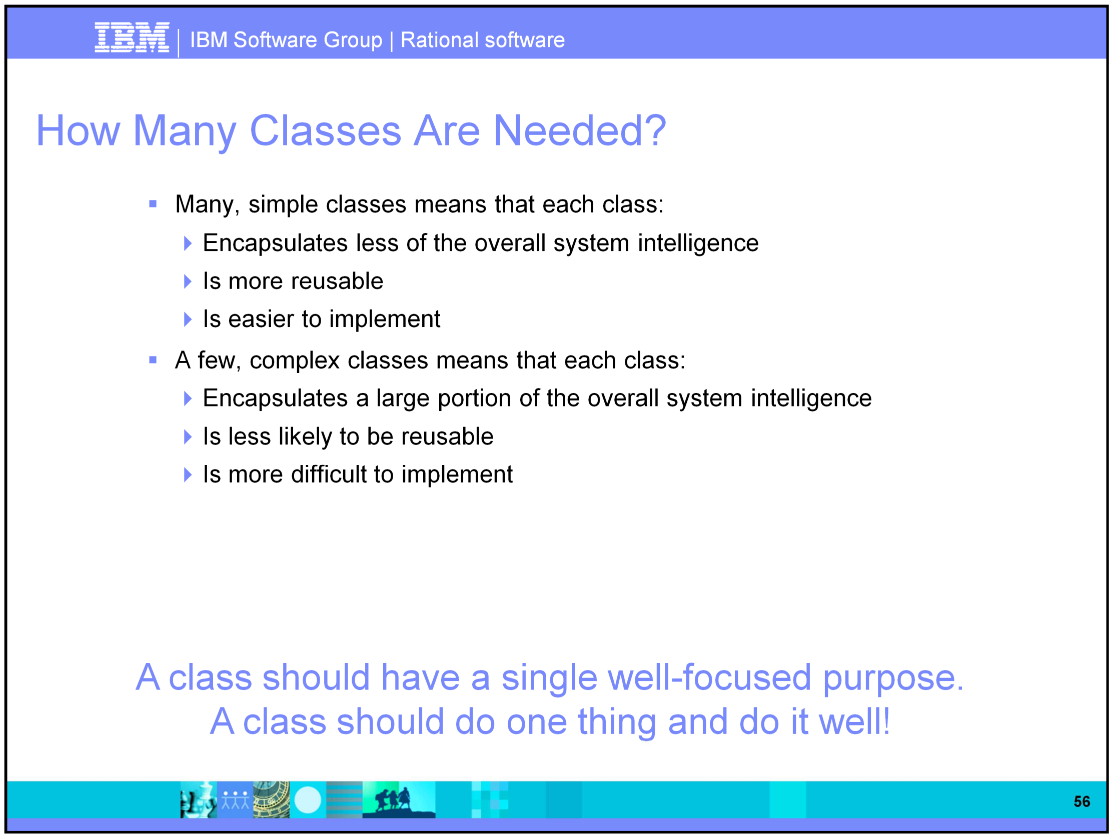
Class Design Considerations

- 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)

55

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.



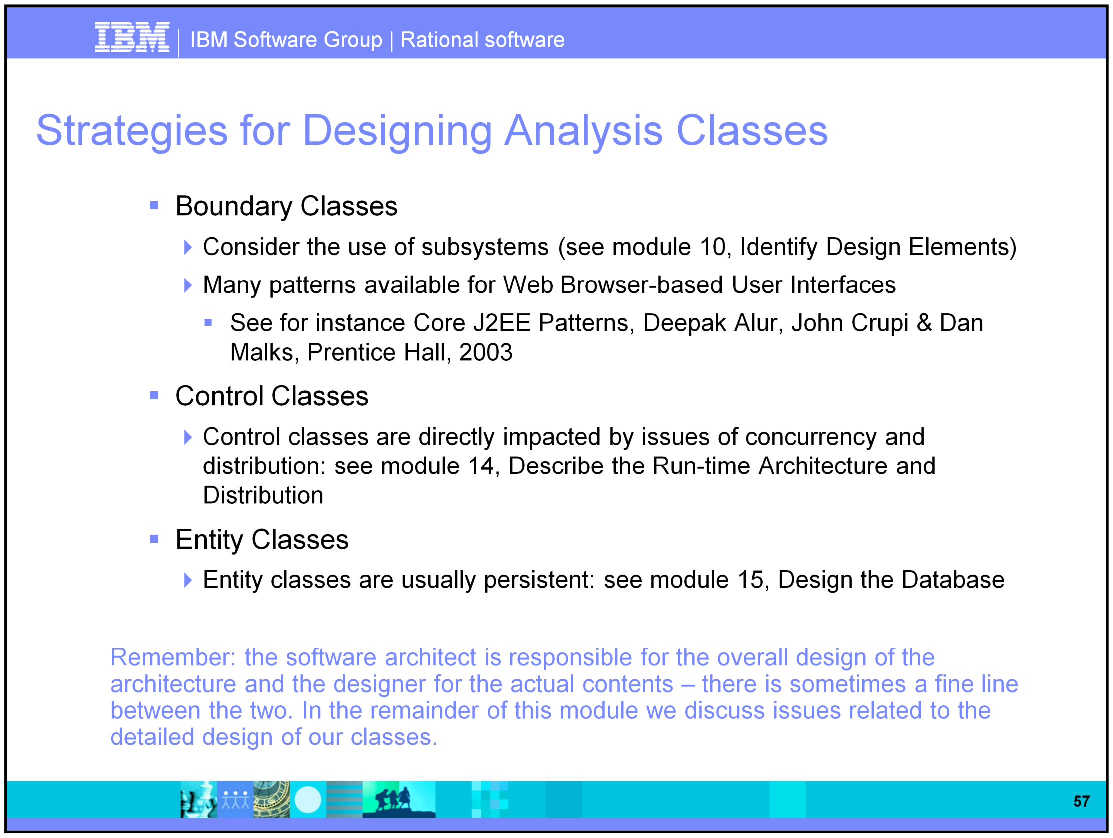
IBM Software Group | Rational software

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!

56



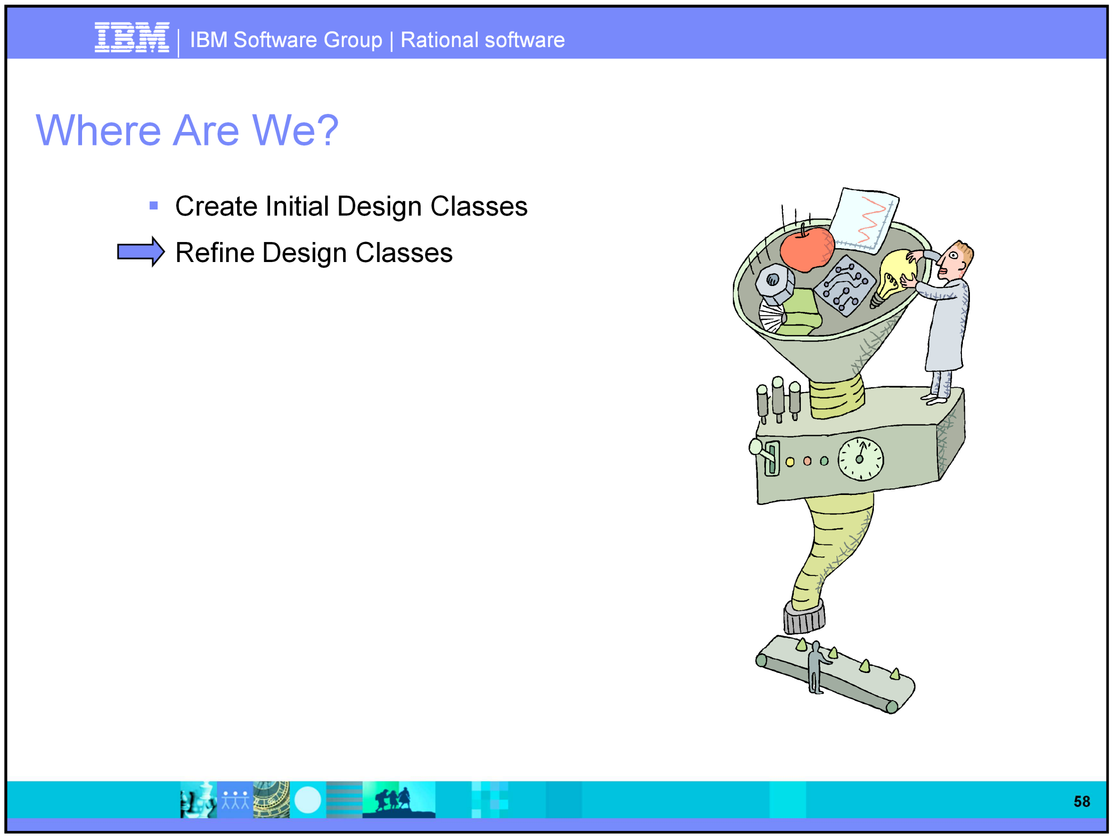
IBM | IBM Software Group | Rational software

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

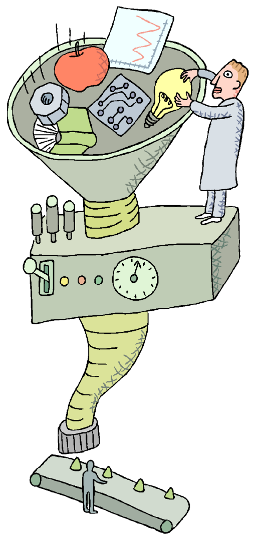
Remember: the software architect is responsible for the overall design of the architecture and the designer for the actual contents – there is sometimes a fine line between the two. In the remainder of this module we discuss issues related to the detailed design of our classes.

57



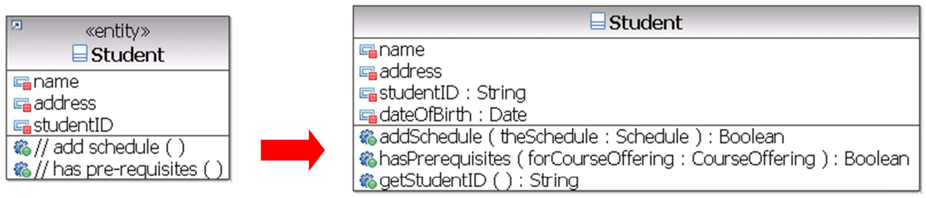
Where Are We?

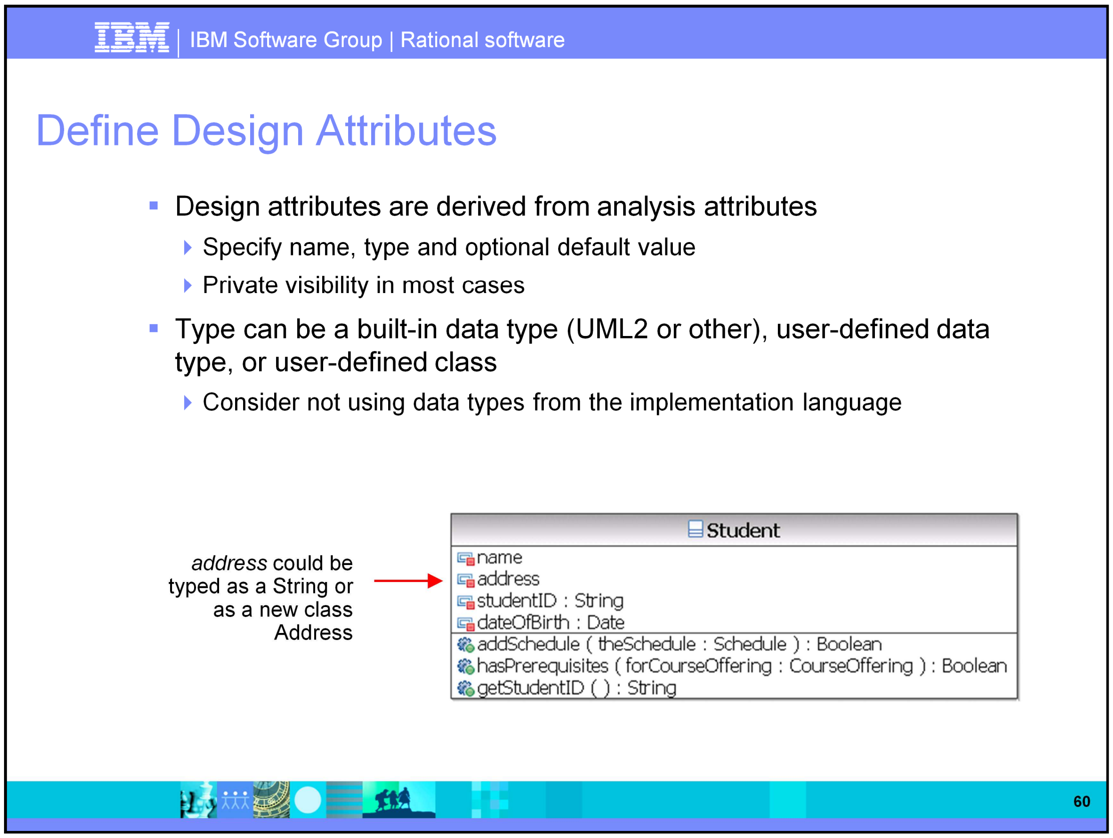
- Create Initial Design Classes
- ➡ Refine Design Classes



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)


«entity»
Student
name
address
studentID
// add schedule ( )
// has pre-requisites ( )
Student
name
address
studentID : String
dateOfBirth : Date
addSchedule (theSchedule : Schedule) : Boolean
hasPrerequisites (forCourseOffering : CourseOffering) : Boolean
getStudentID ( ) : String

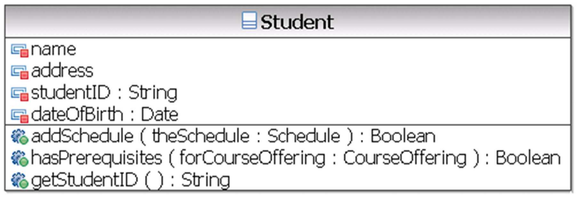


IBM Software Group | Rational software

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

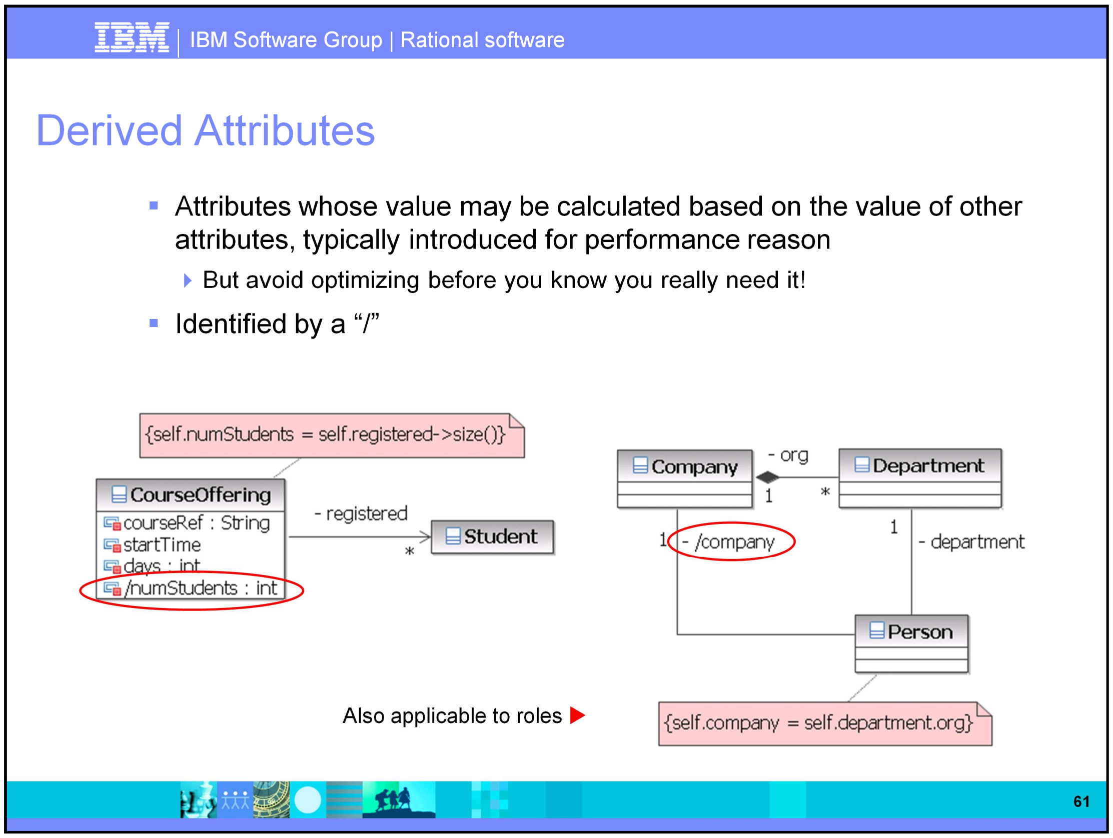
address could be typed as a String or as a new class Address



Student

name
address
studentID : String
dateOfBirth : Date
addSchedule (theSchedule : Schedule) : Boolean
hasPrerequisites (forCourseOffering : CourseOffering) : Boolean
getStudentID ( ) : String

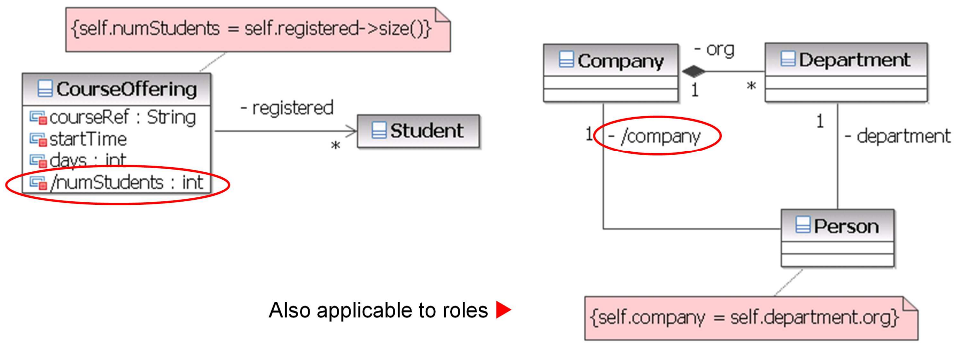
60



IBM | IBM Software Group | Rational software

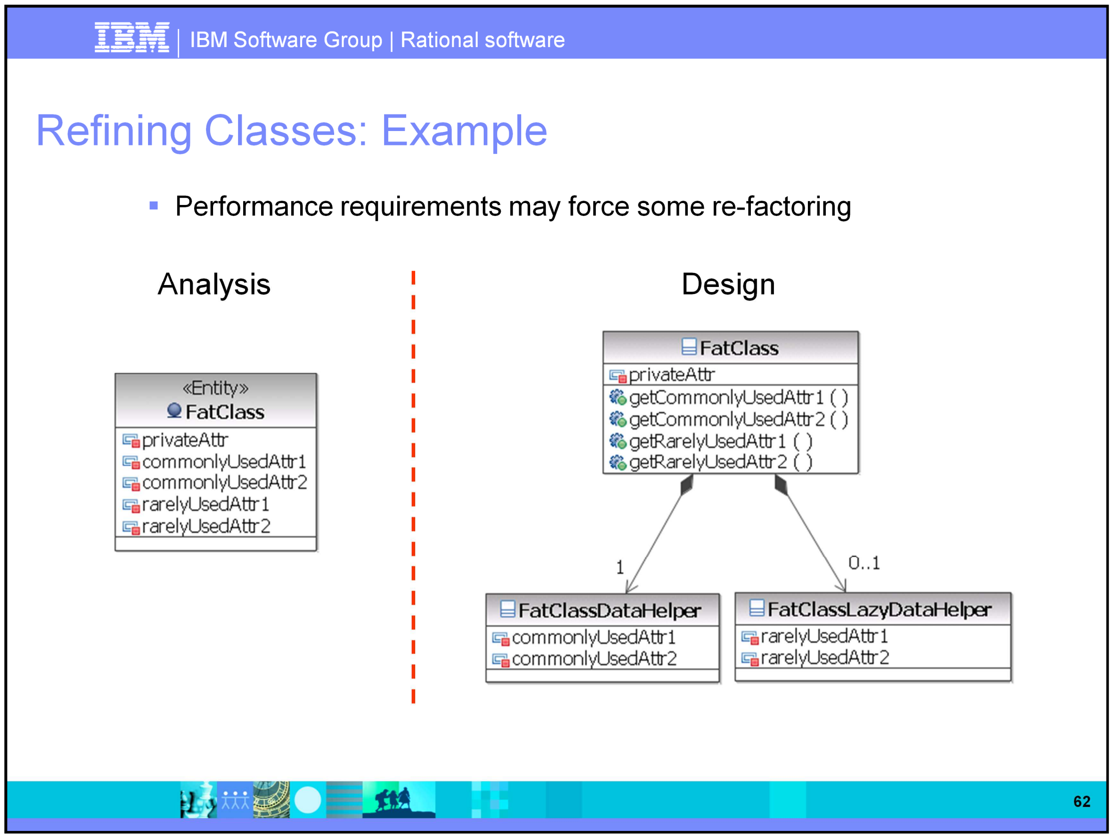
Derived Attributes

- 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 ▶

61



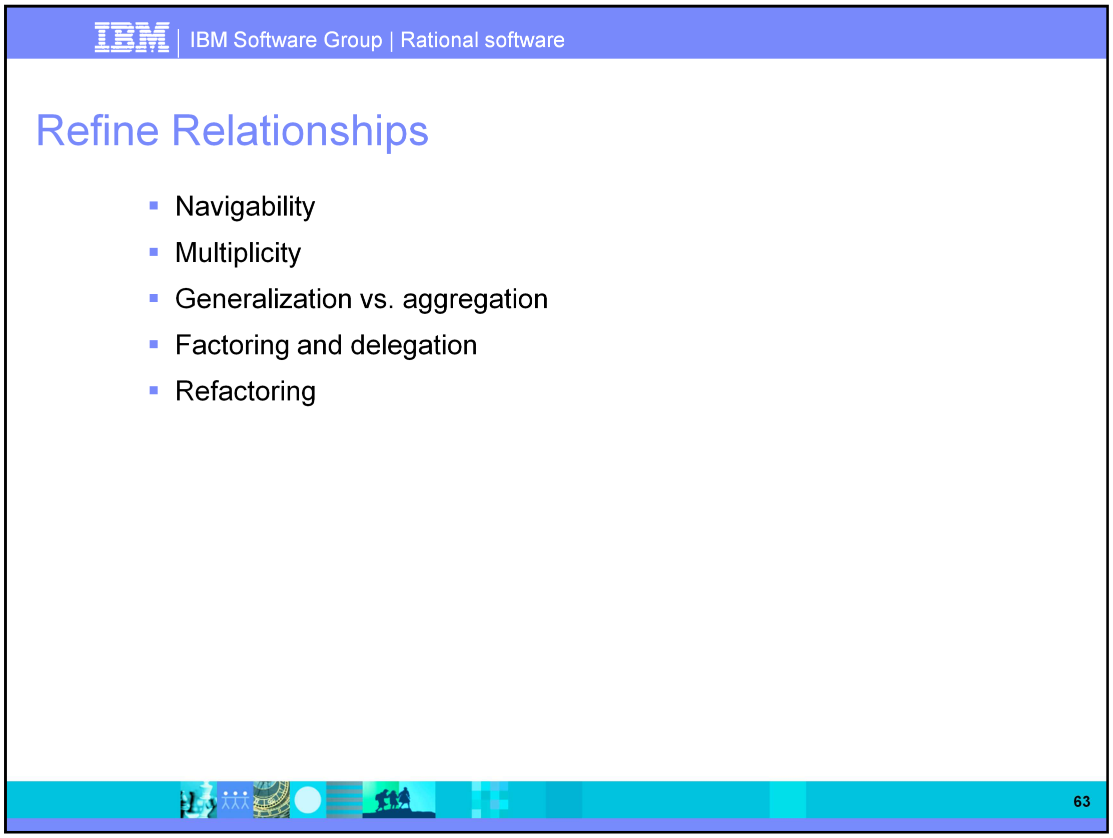
Refining Classes: Example

- Performance requirements may force some re-factoring

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.

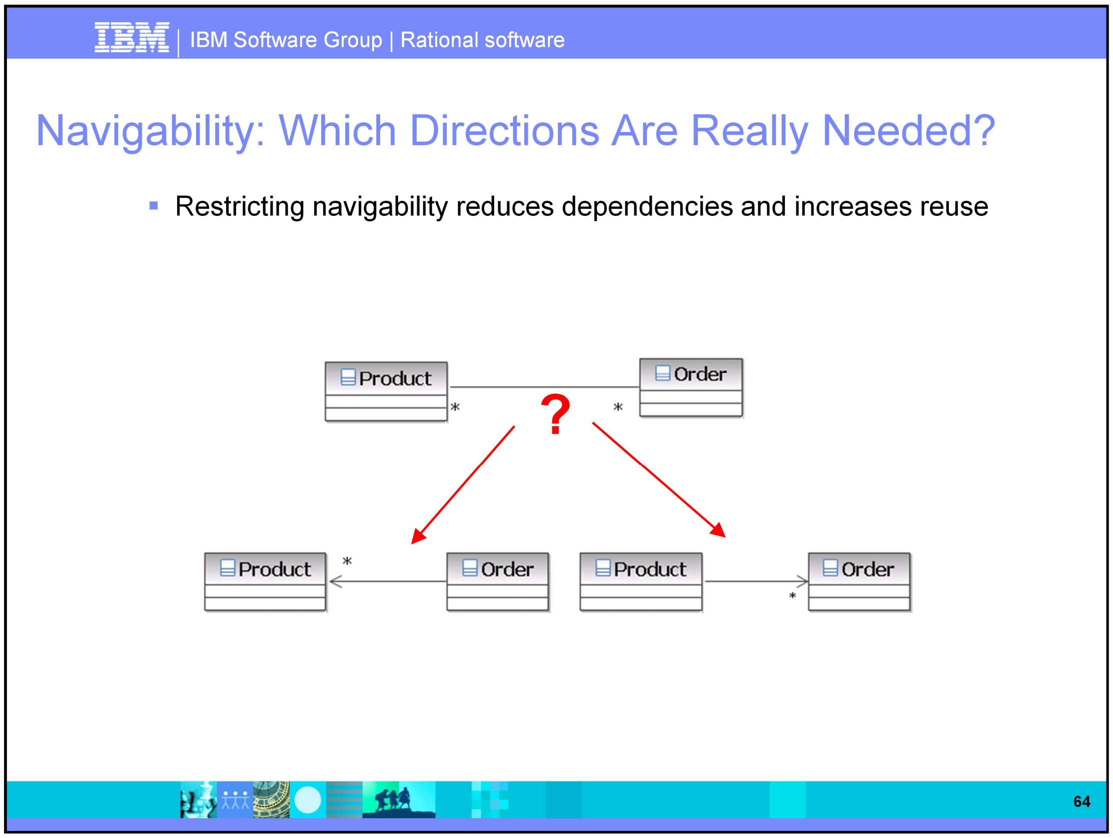


IBM Software Group | Rational software

Refine Relationships

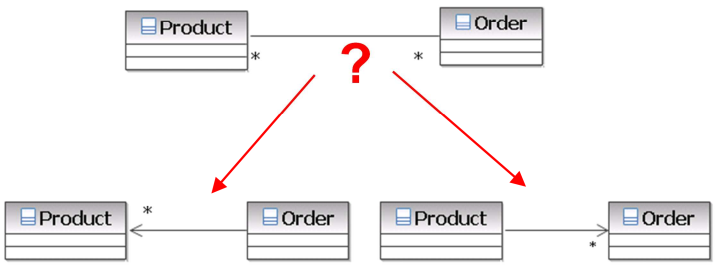
- Navigability
- Multiplicity
- Generalization vs. aggregation
- Factoring and delegation
- Refactoring

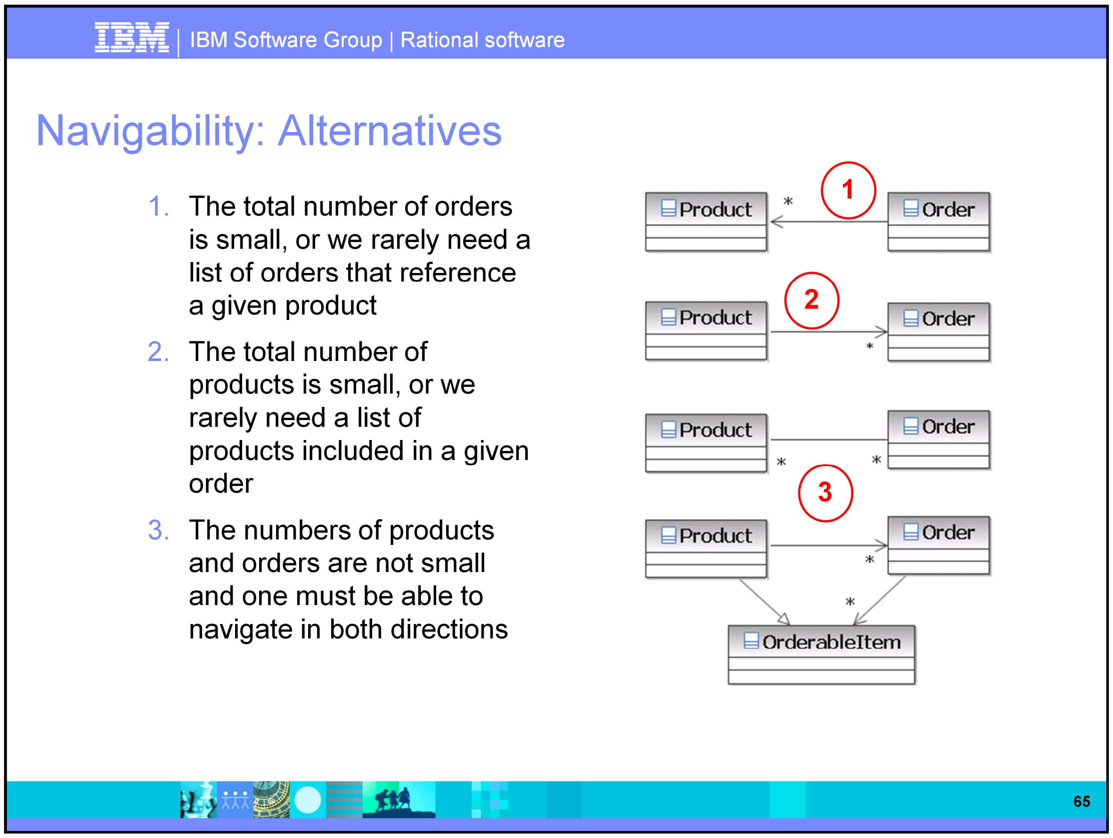
63



Navigability: Which Directions Are Really Needed?

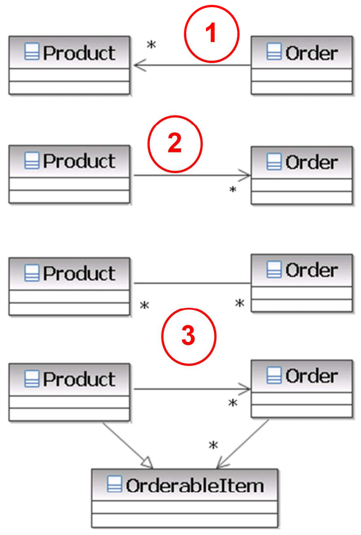
- Restricting navigability reduces dependencies and increases reuse

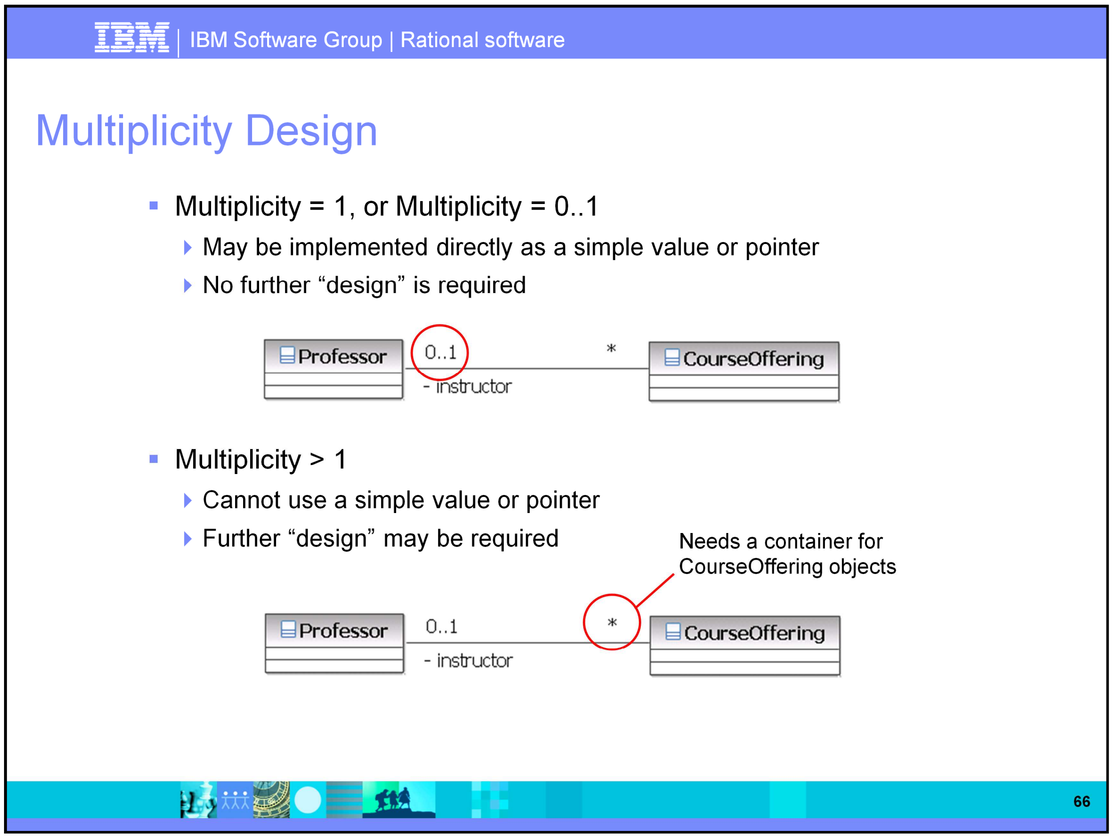




Navigability: Alternatives

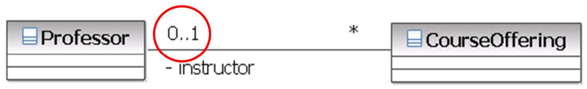
1. The total number of orders is small, or we rarely need a list of orders that reference a given product
2. The total number of products is small, or we rarely need a list of products included in a given order
3. The numbers of products and orders are not small and one must be able to navigate in both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



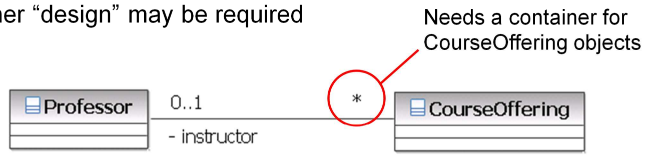


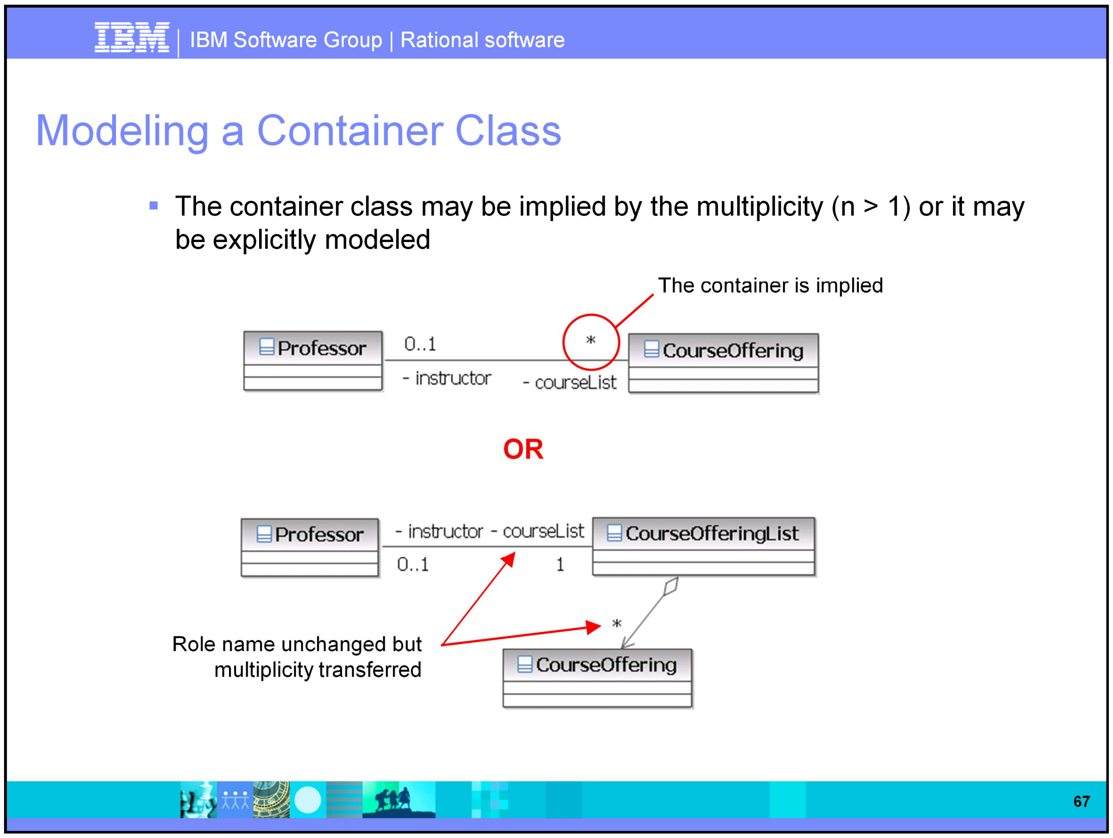
Multiplicity Design

- 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

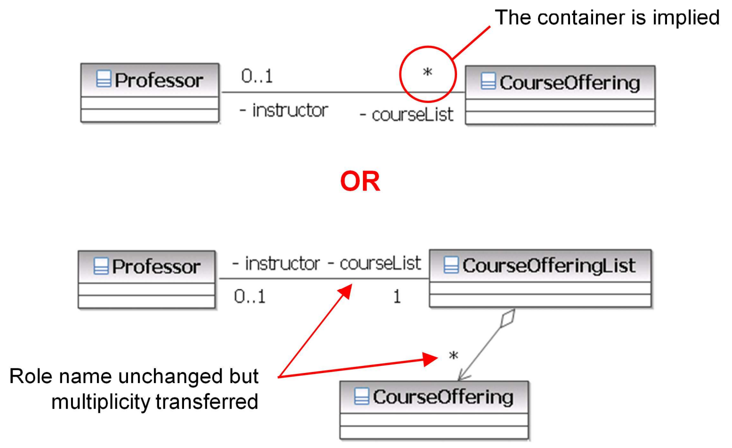




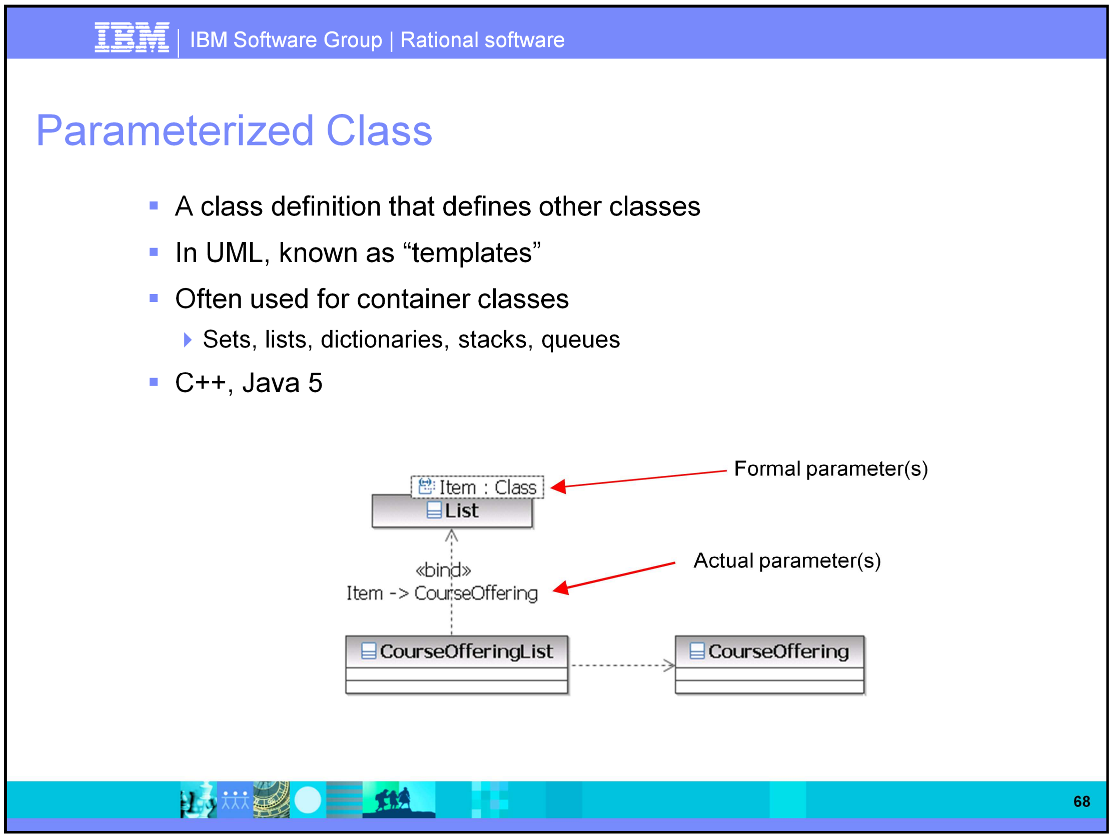
IBM | IBM Software Group | Rational software

Modeling a Container Class

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

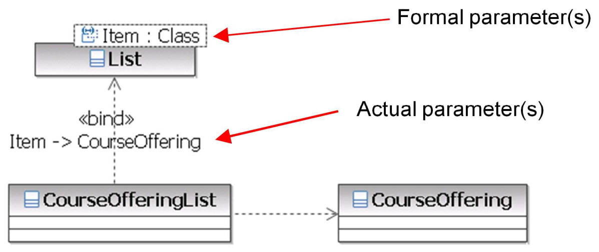


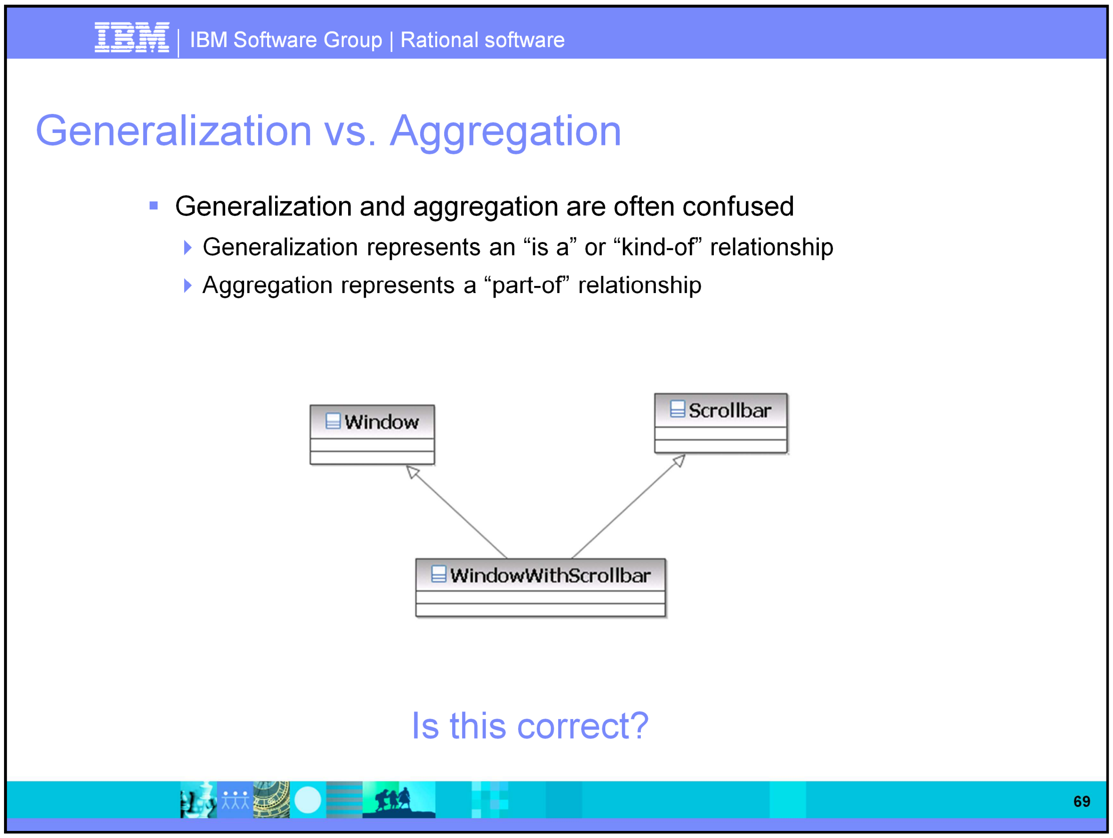
67



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

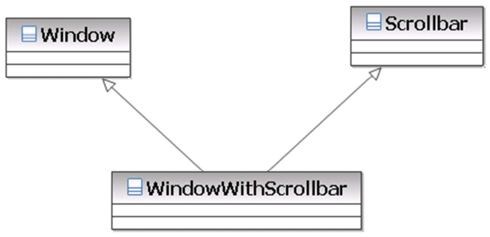




IBM | IBM Software Group | Rational software

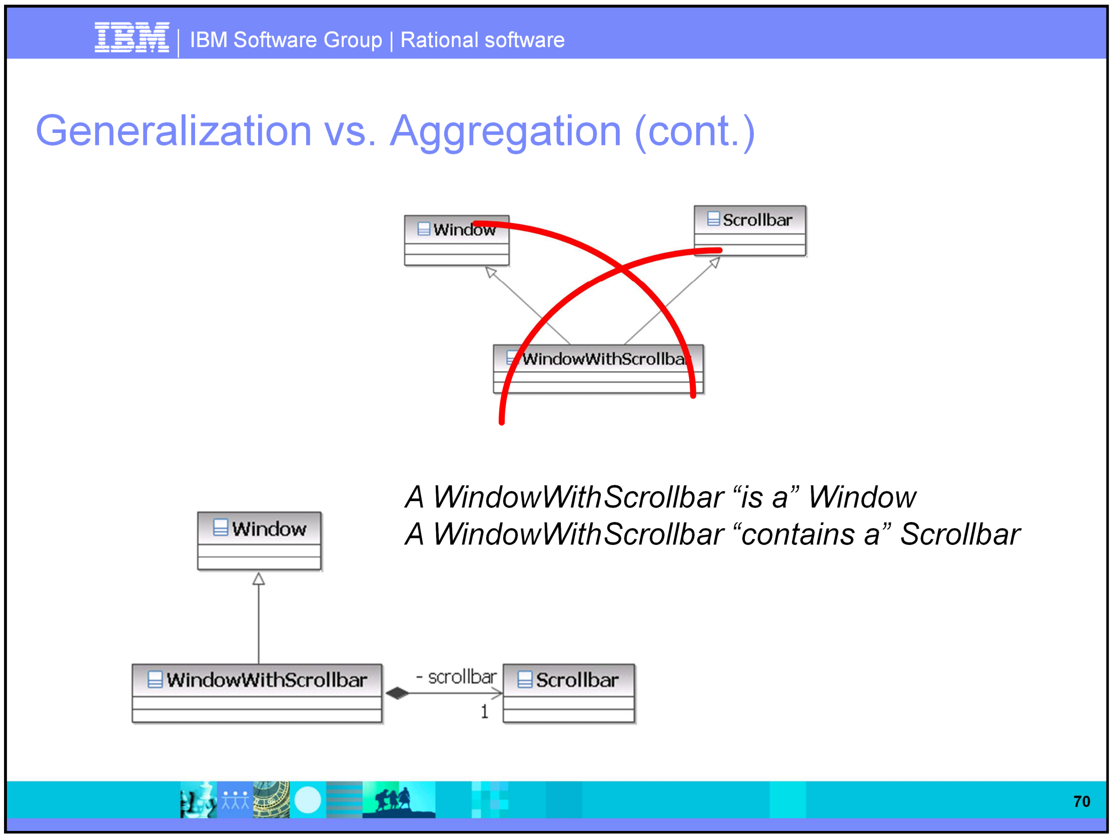
Generalization vs. Aggregation

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

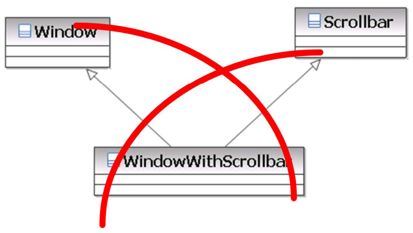


Is this correct?

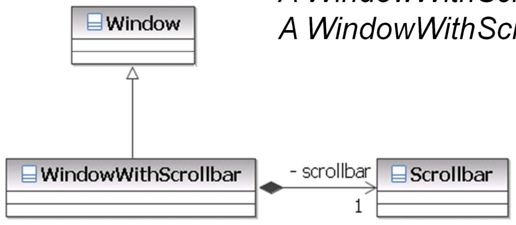
69

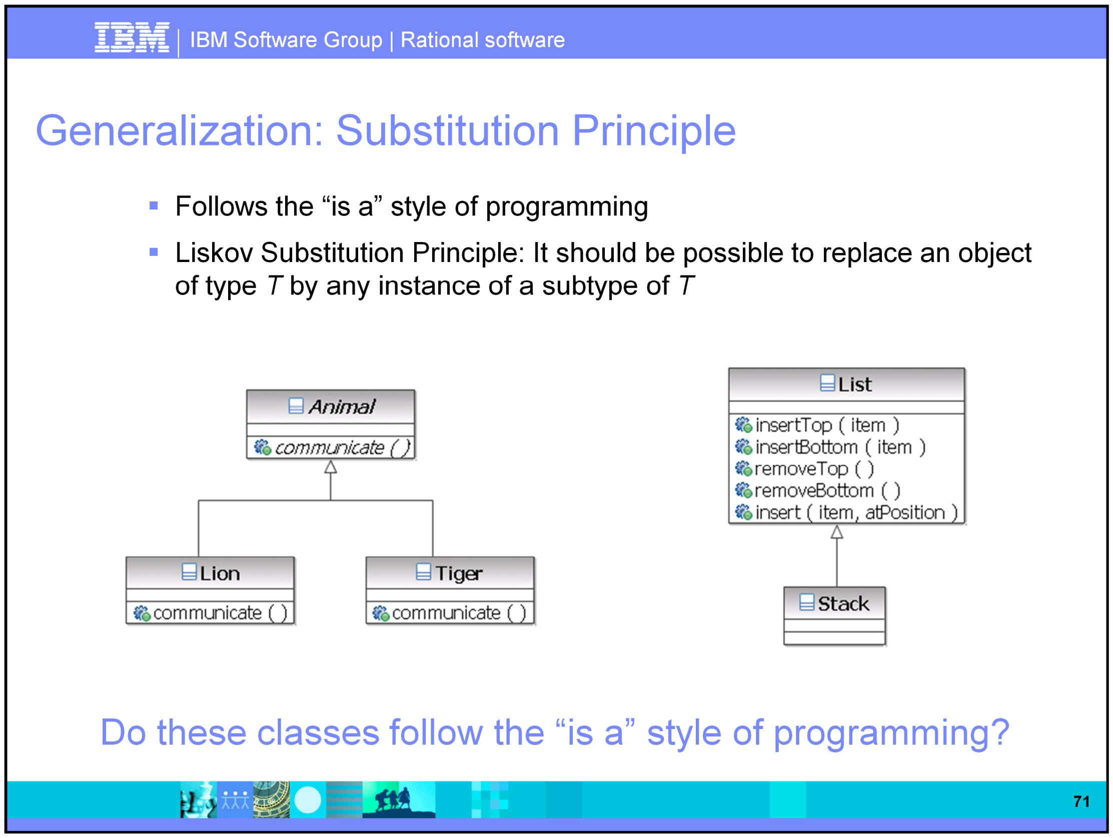


Generalization vs. Aggregation (cont.)



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

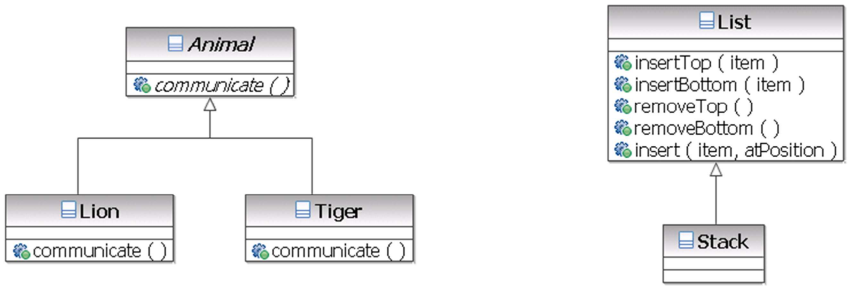




IBM | IBM Software Group | Rational software

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*



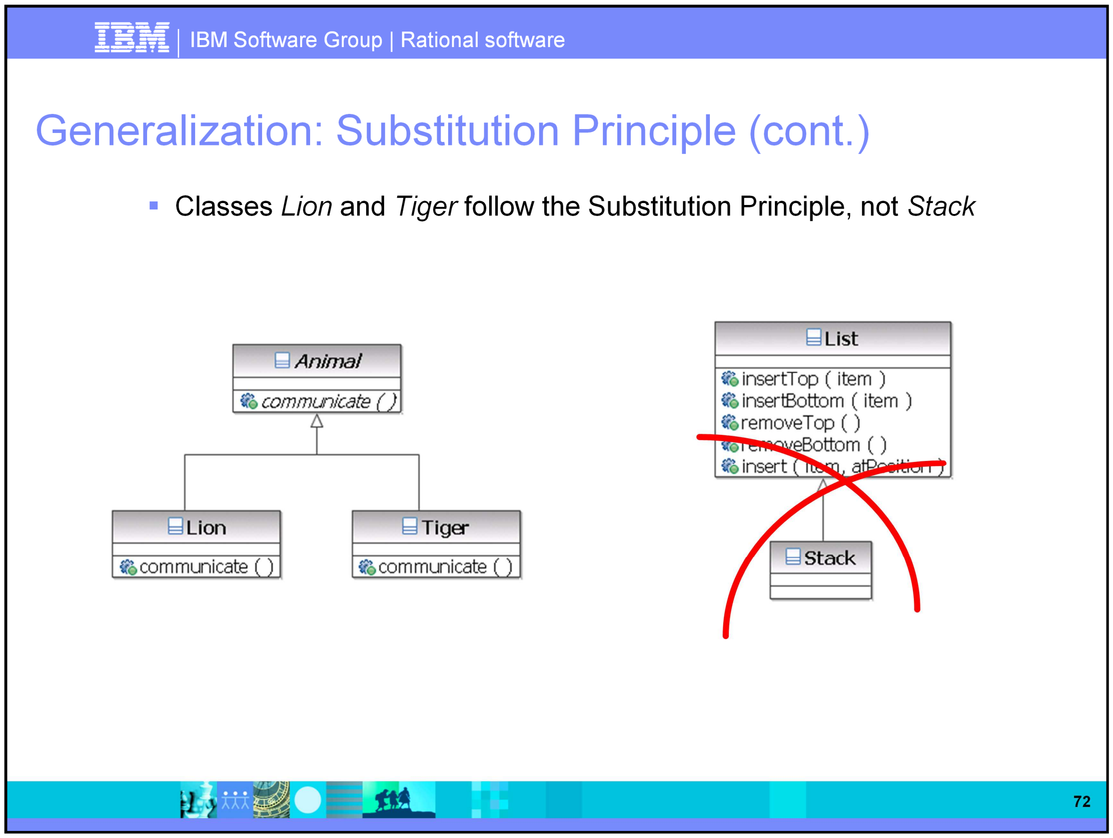
Do these classes follow the “is a” style of programming?

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.”

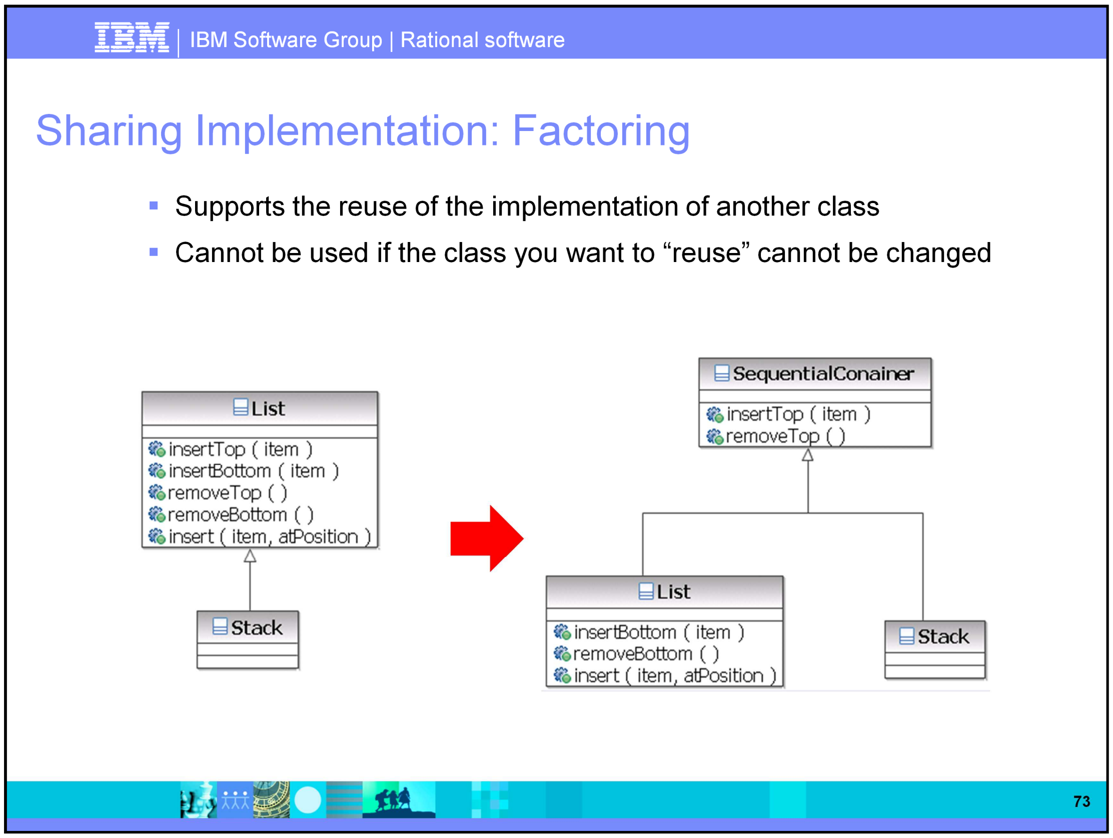


Generalization: Substitution Principle (cont.)

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

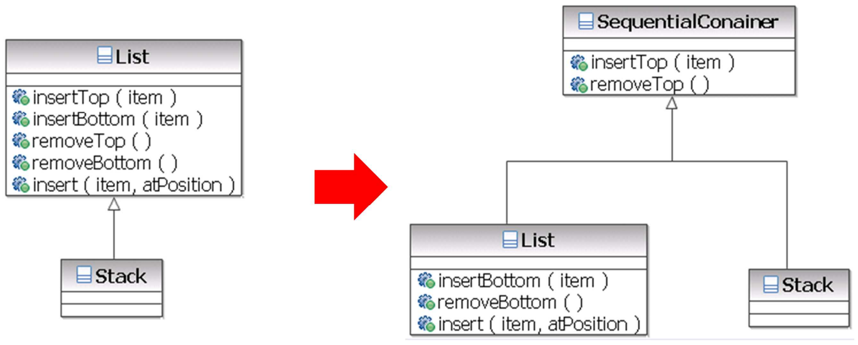
The classes on the left-hand side of the diagram do follow the "is a" style of programming: a Lion is an Animal and a Tiger is an animal.

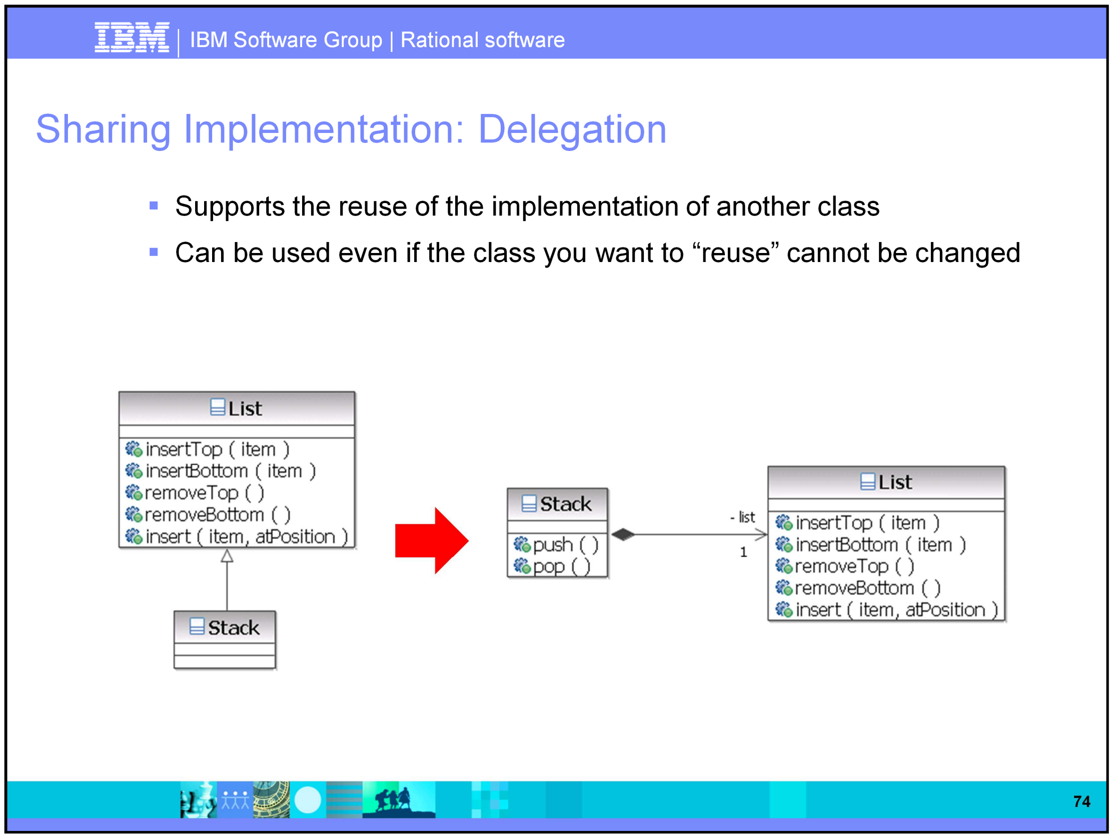
The classes on the right side of the diagram do *not* follow the “is a” style of programming: a Stack is not a List. Stack needs some of the behavior of a List but not all of the behavior. If a method expects a List, then the operation insert(position) should be successful. If the method is passed a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



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

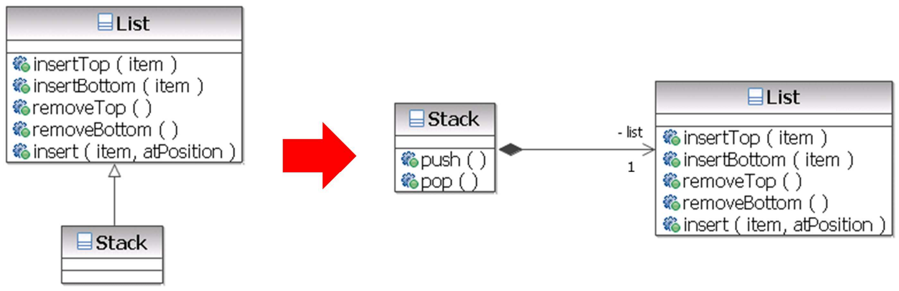




IBM Software Group | Rational software

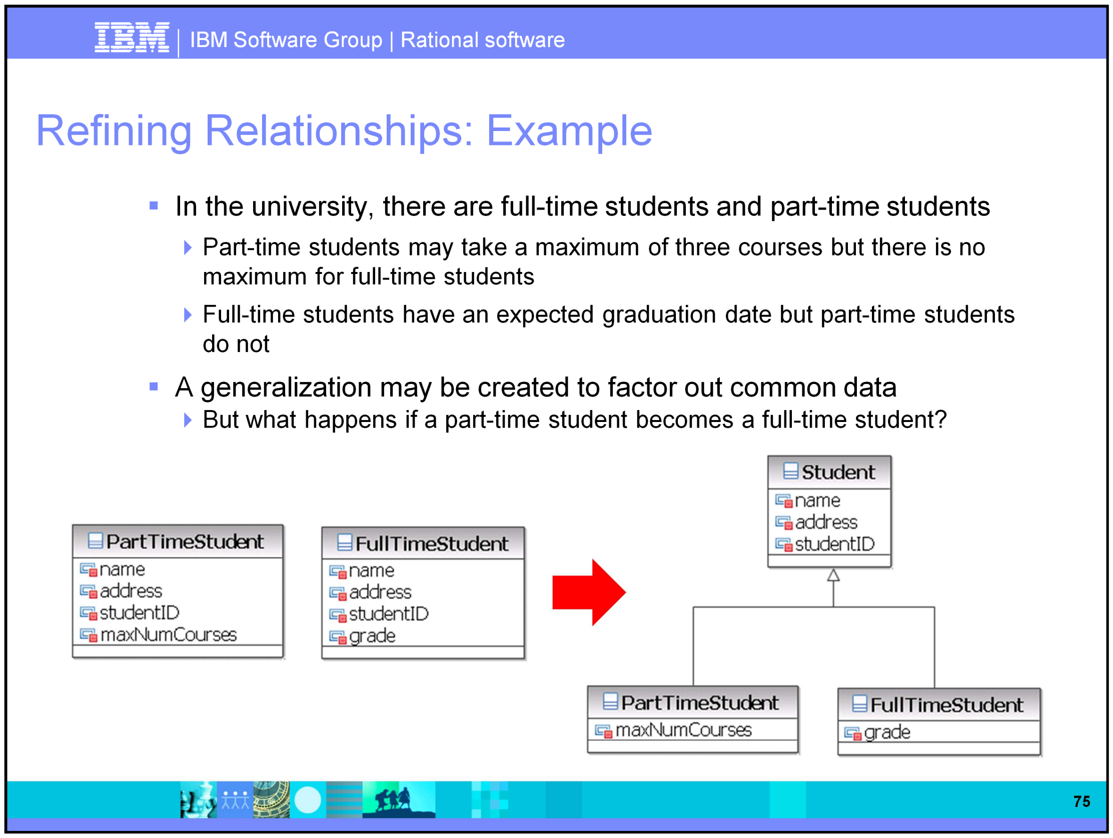
Sharing Implementation: Delegation

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



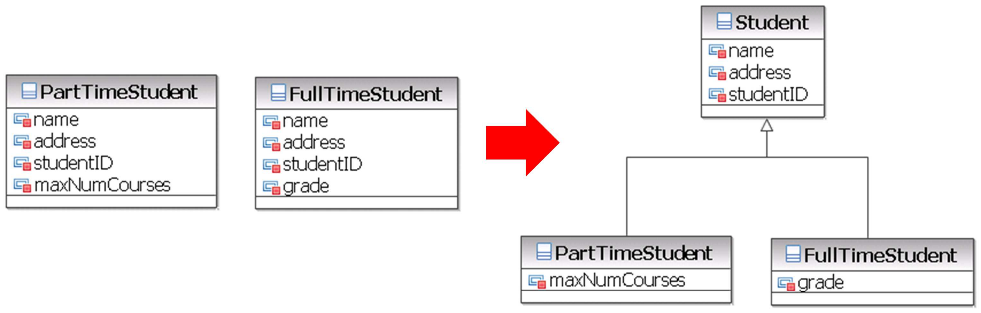
74

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.



Refining Relationships: Example

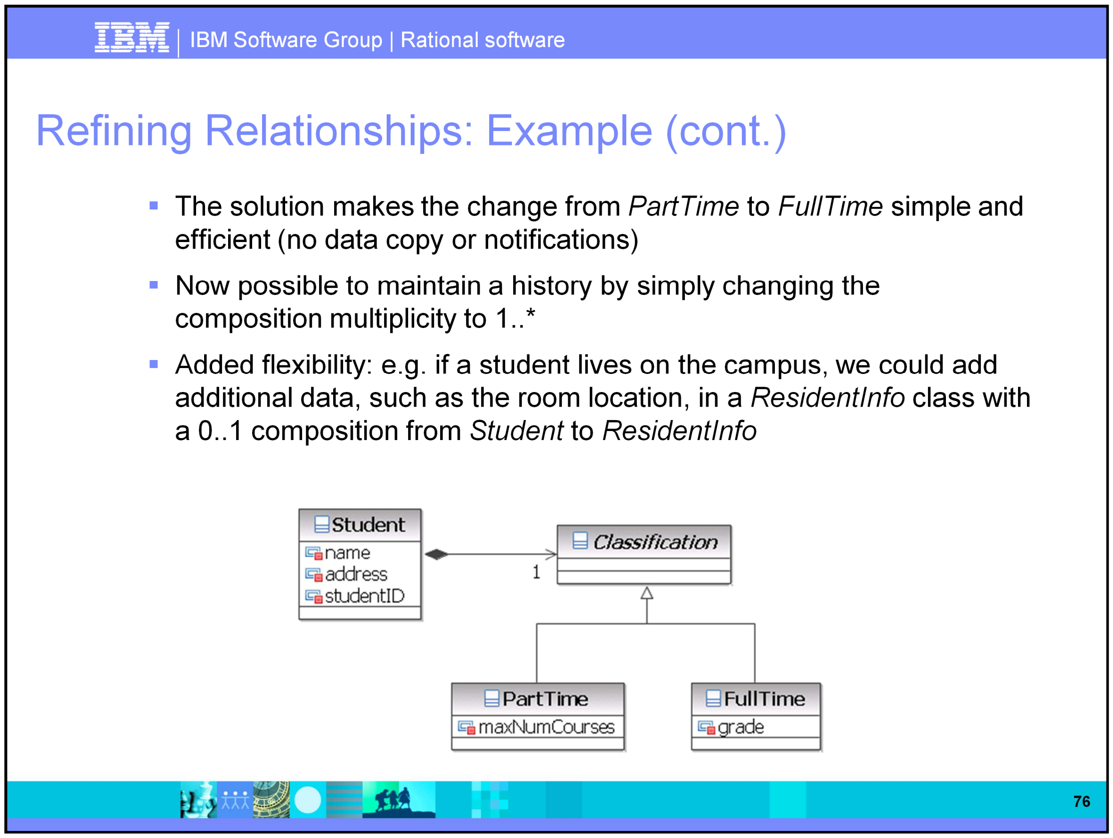
- In the university, there are full-time students and part-time students
  - Part-time students may take a maximum of three courses but there is no maximum for full-time students
  - Full-time students have an expected graduation date but part-time students do not
- A generalization may be created to factor out common data
  - But what happens if a part-time student becomes a full-time student?



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.

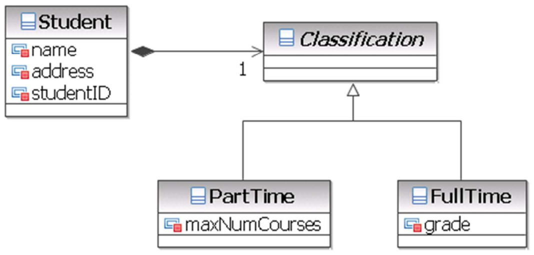
And what happens if in addition there is a requirement to maintain a history of the student.



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*

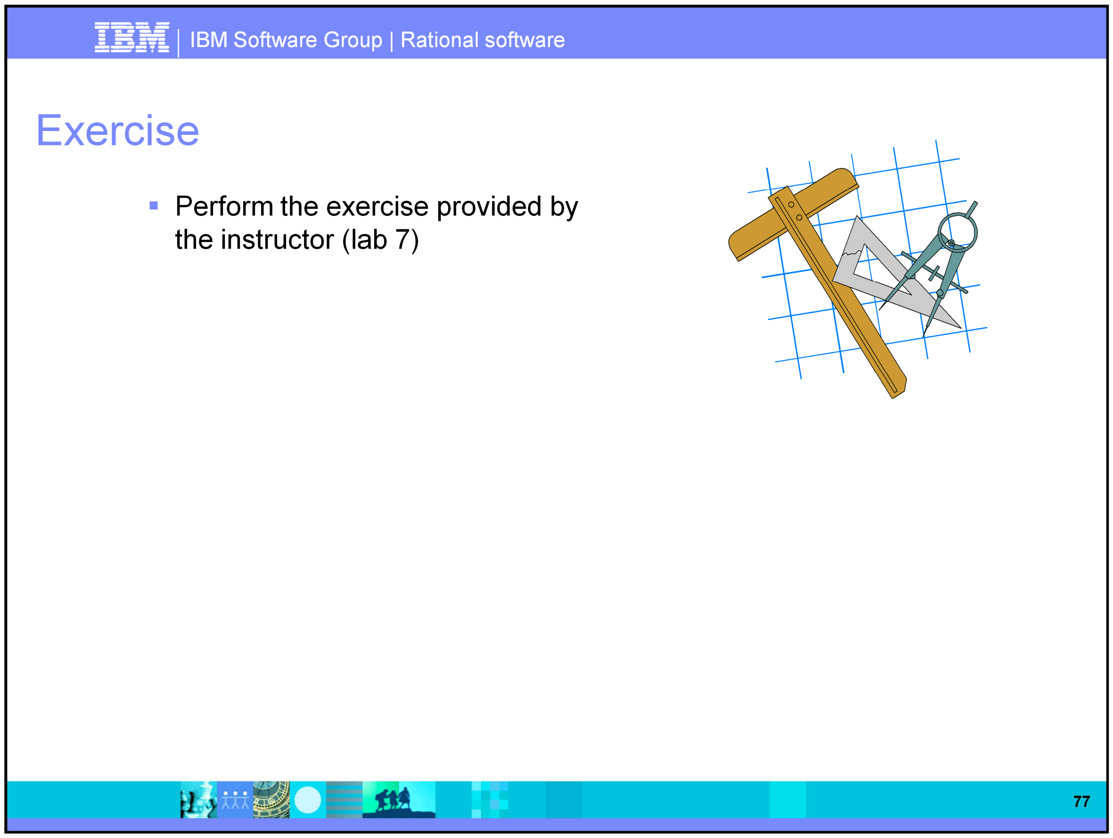


76

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.

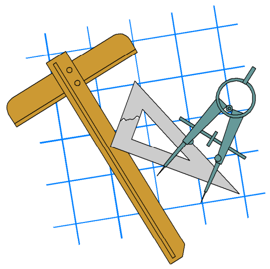
What's more, this structure adds to the flexibility of the model: imagine for instance that the student lives on the campus. In this case, we could add additional data, such as the room location, in a *ResidentInfo* class with a 0..1 composition from *Student* to *ResidentInfo.*

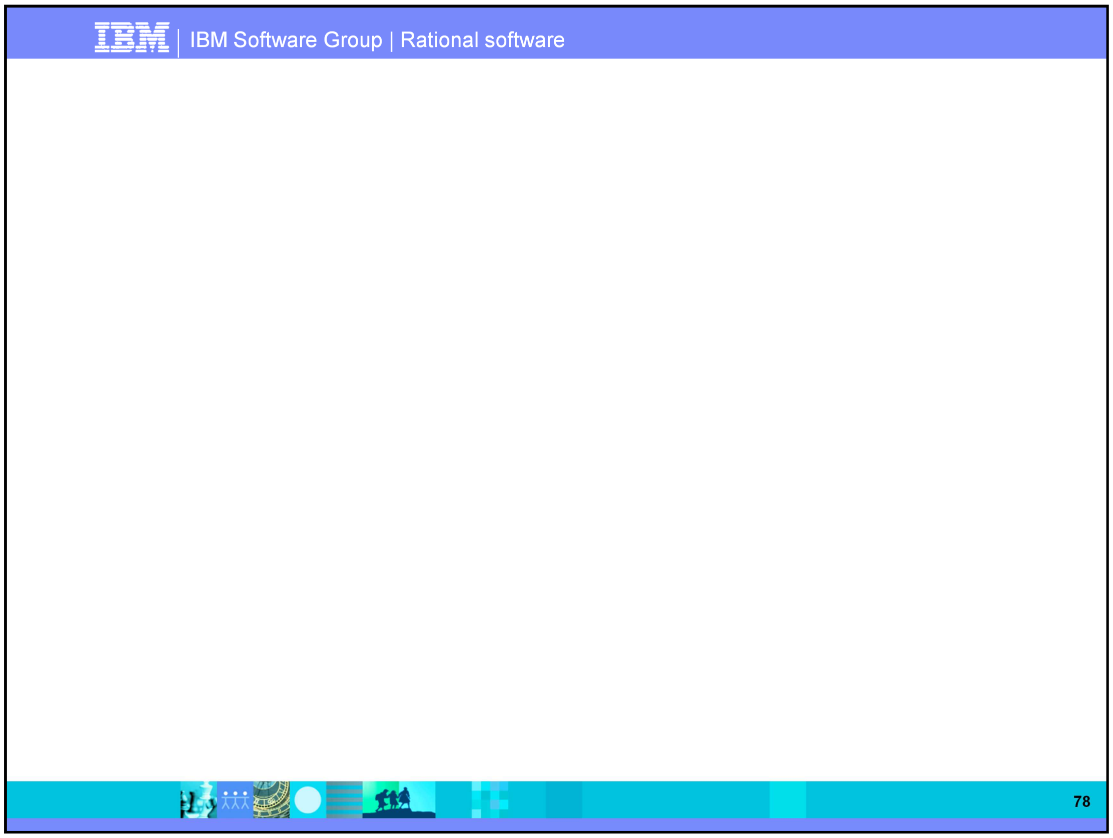
Note: The *State* pattern uses this structure in which a class *State* is introduced instead of *Classification*. The aggregate (the equivalent of *Student* in our diagram) can then invoke operations without having to know the current state. When there is a change of state, the aggregate receives a new *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

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





IBM Software Group | Rational software

78