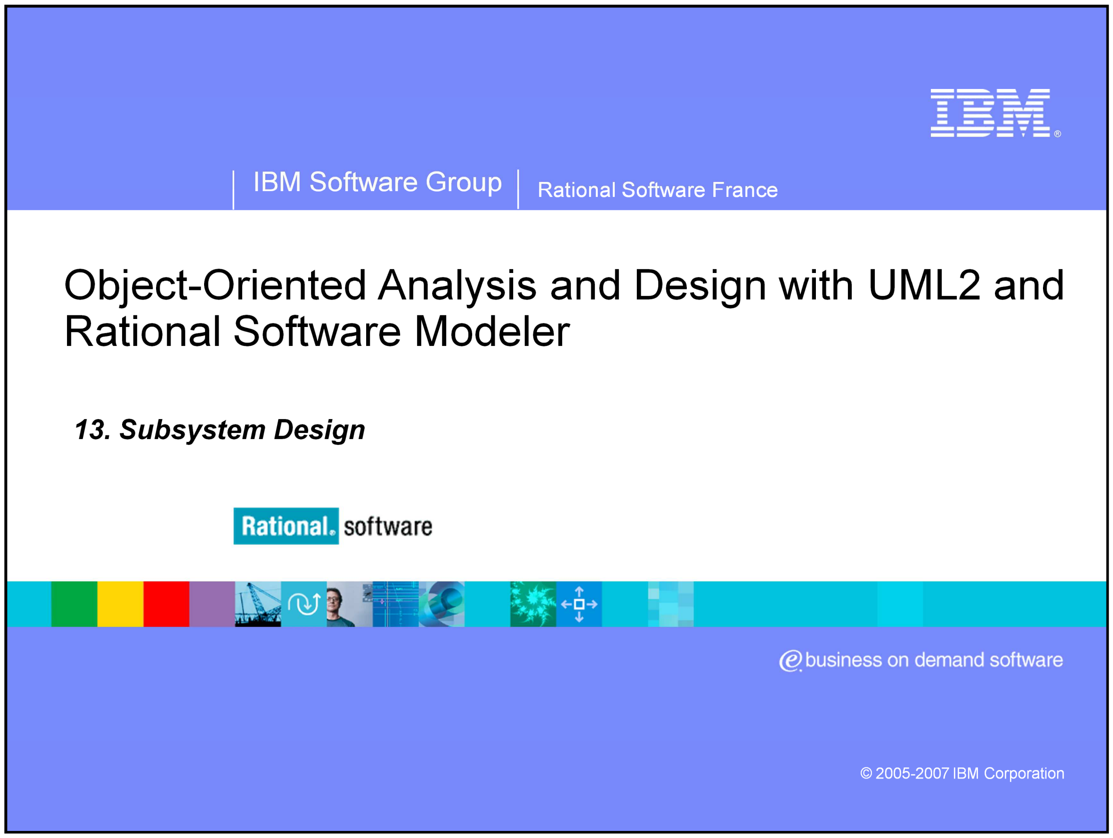


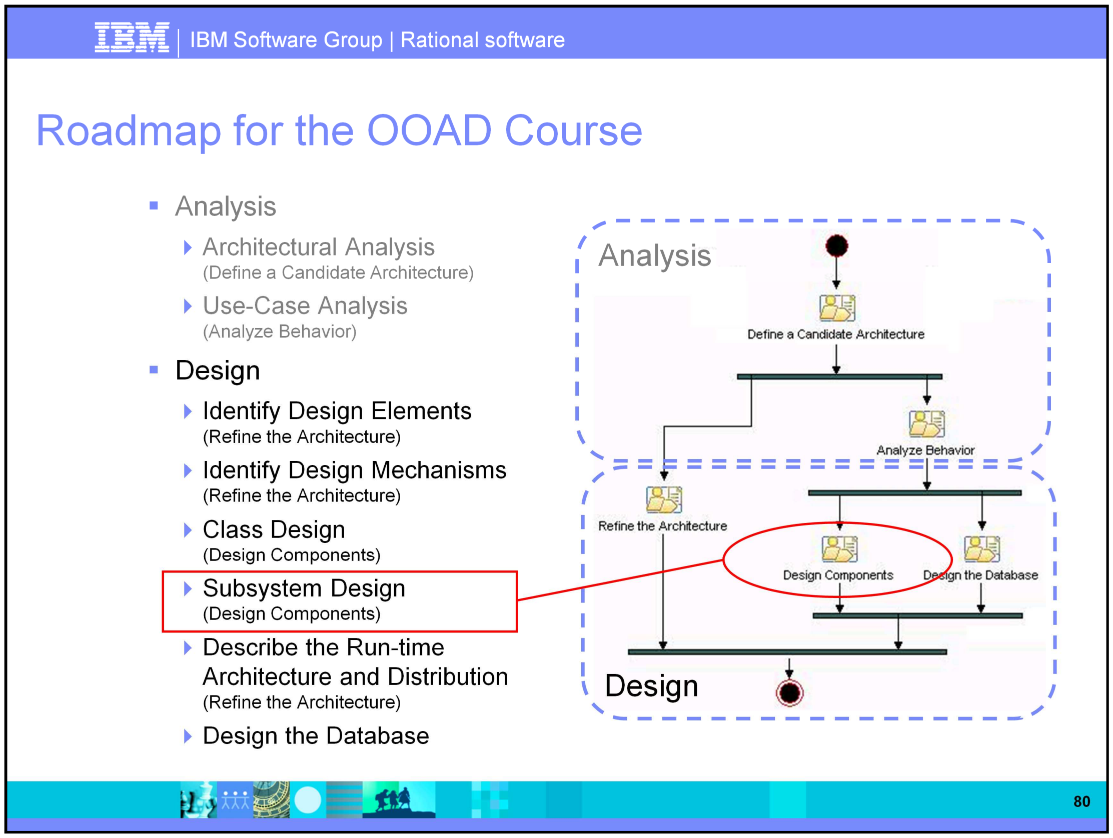
IBM Software Group | Rational Software France

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

13. Subsystem Design

Rational. software

e business on demand software

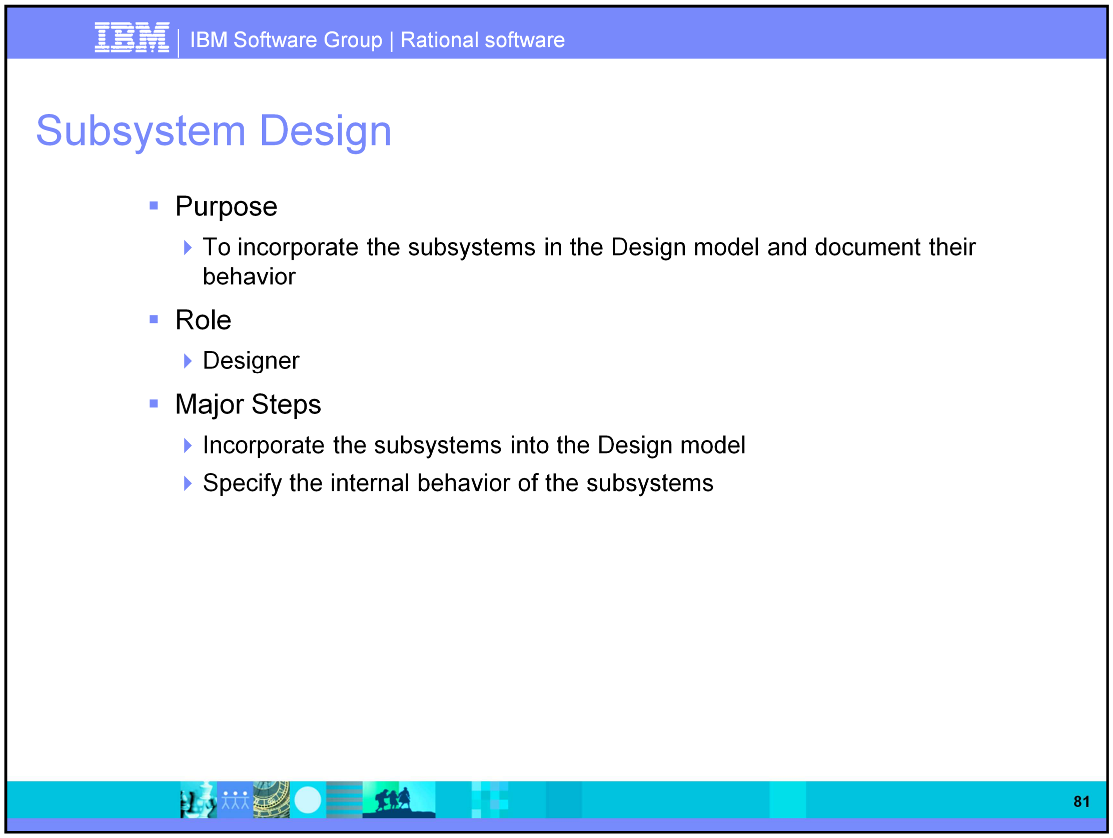


IBM | IBM Software Group | Rational software

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

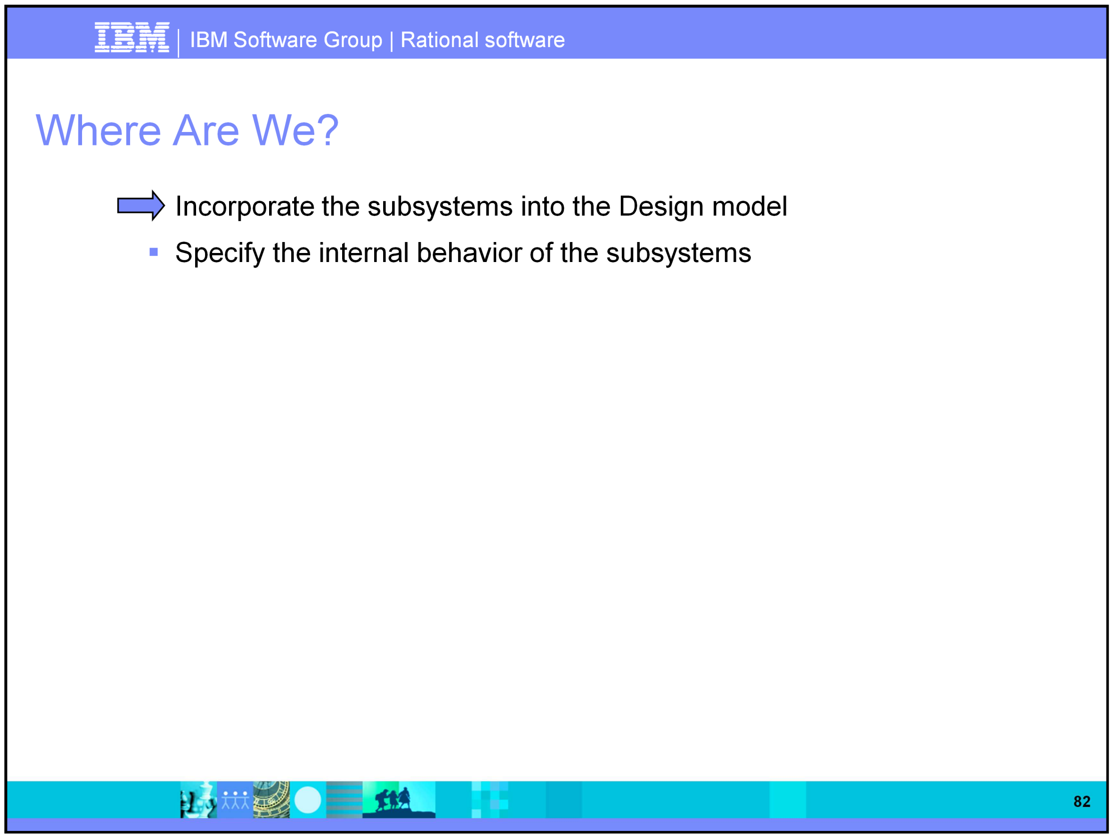


IBM Software Group | Rational software

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

81

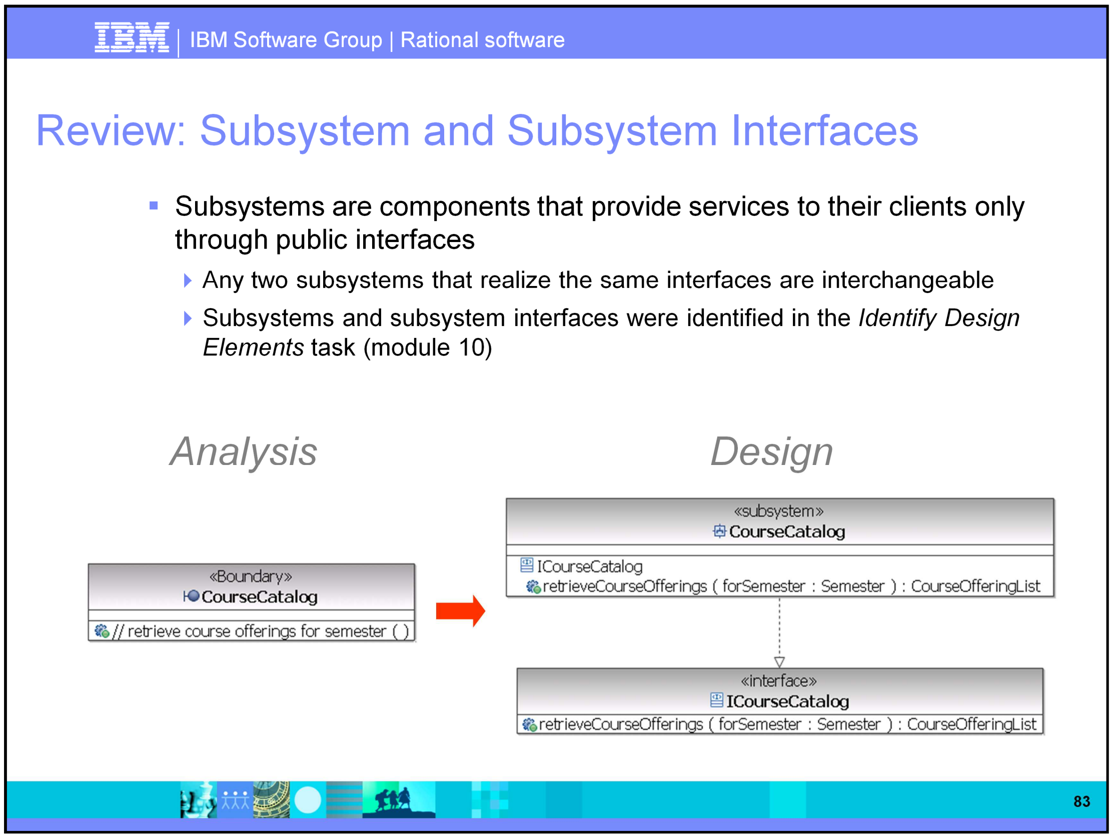


IBM Software Group | Rational software

Where Are We?

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

82

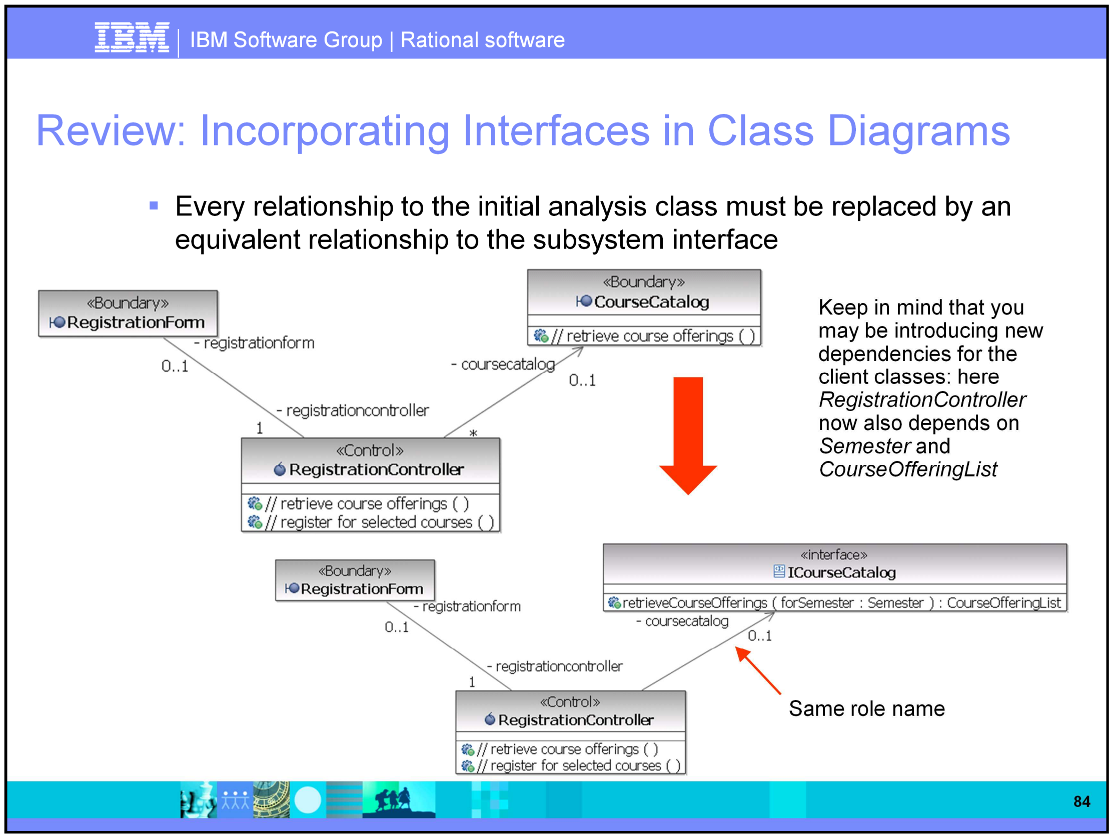


IBM | IBM Software Group | Rational software

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
«Boundary»
CourseCatalog
// retrieve course offerings for semester ( )
«subsystem»
CourseCatalog
ICourseCatalog
retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
«interface»
ICourseCatalog
retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
83



IBM Software Group | Rational software

Review: Incorporating Interfaces in Class Diagrams

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

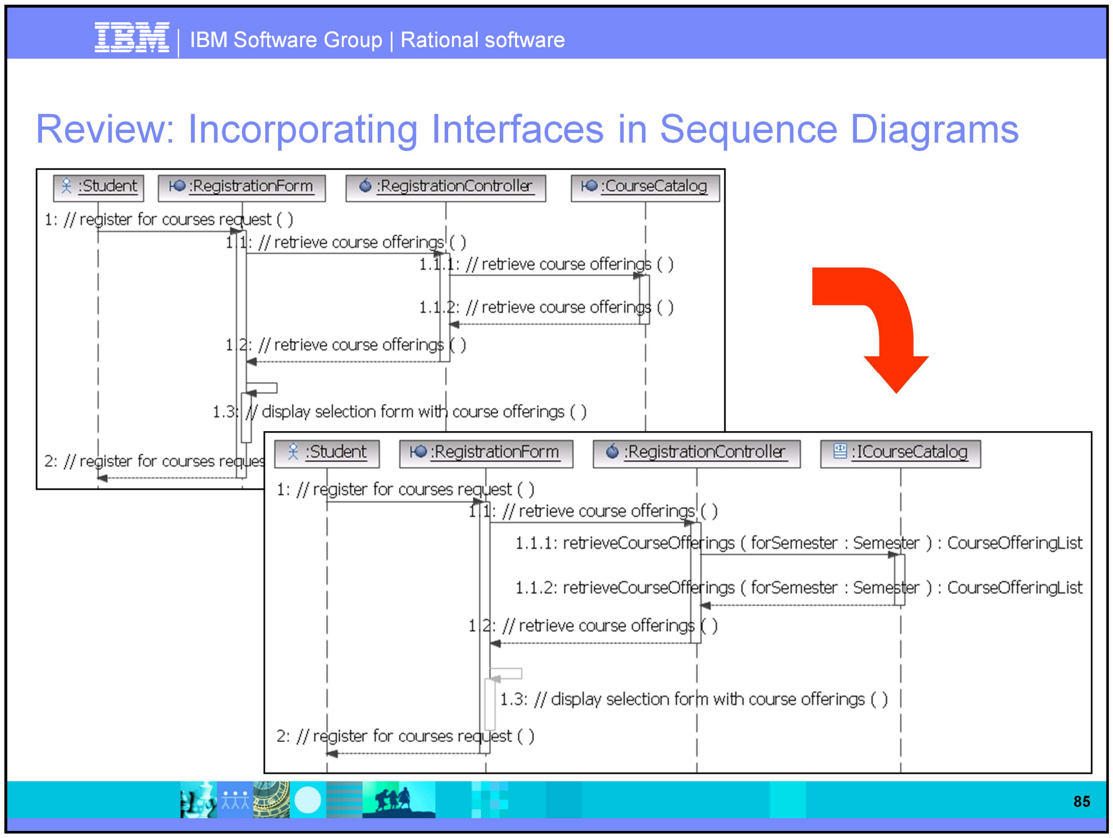
Keep in mind that you may be introducing new dependencies for the client classes: here *RegistrationController* now also depends on *Semester* and *CourseOfferingList*

Same role name

84

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

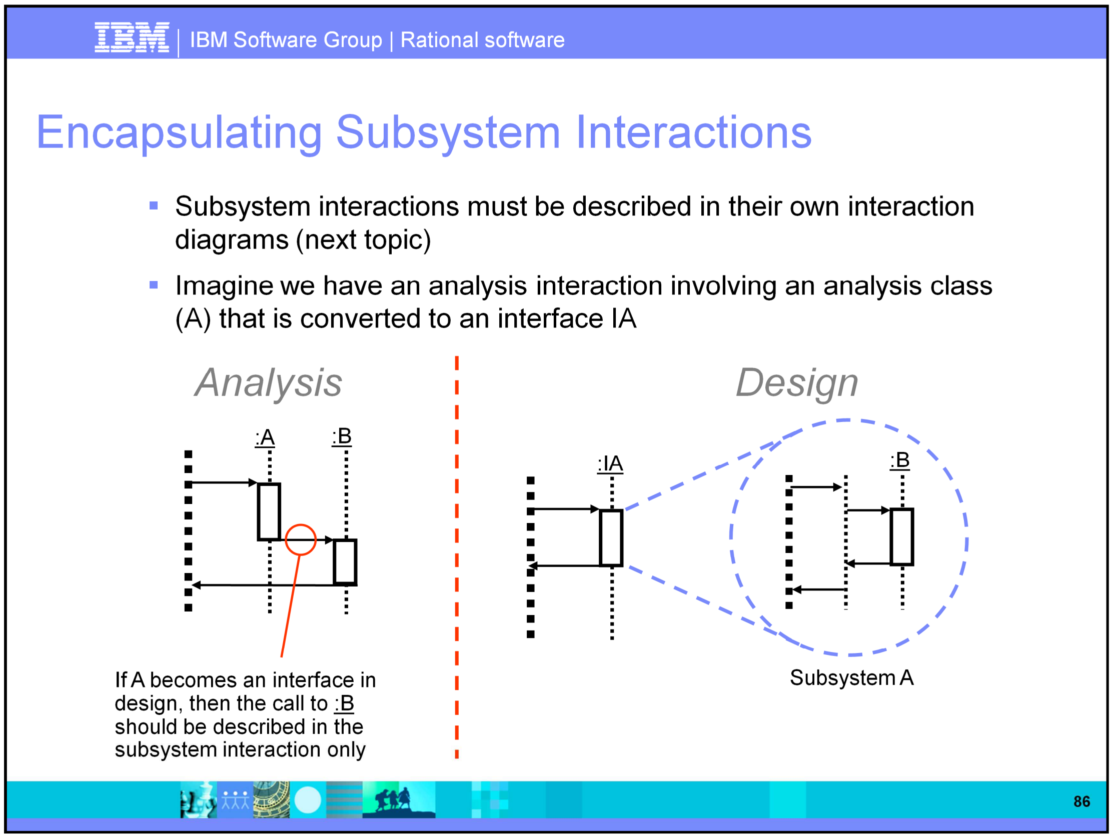
- Retrieve the interface to use and drag it to the diagram
- Select the relationship and move the target end from the analysis class to the interface
- Delete the analysis class from the diagram
- Delete the analysis class from the design model after all changes have been made


IBM Software Group | Rational software

Review: 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 reques
: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 ( )
85

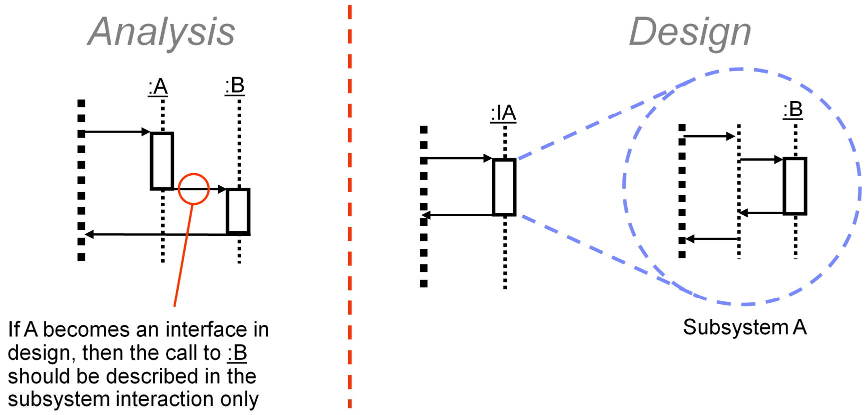
In RSA/RSM, simply drag the interface over the analysis object and update the message.



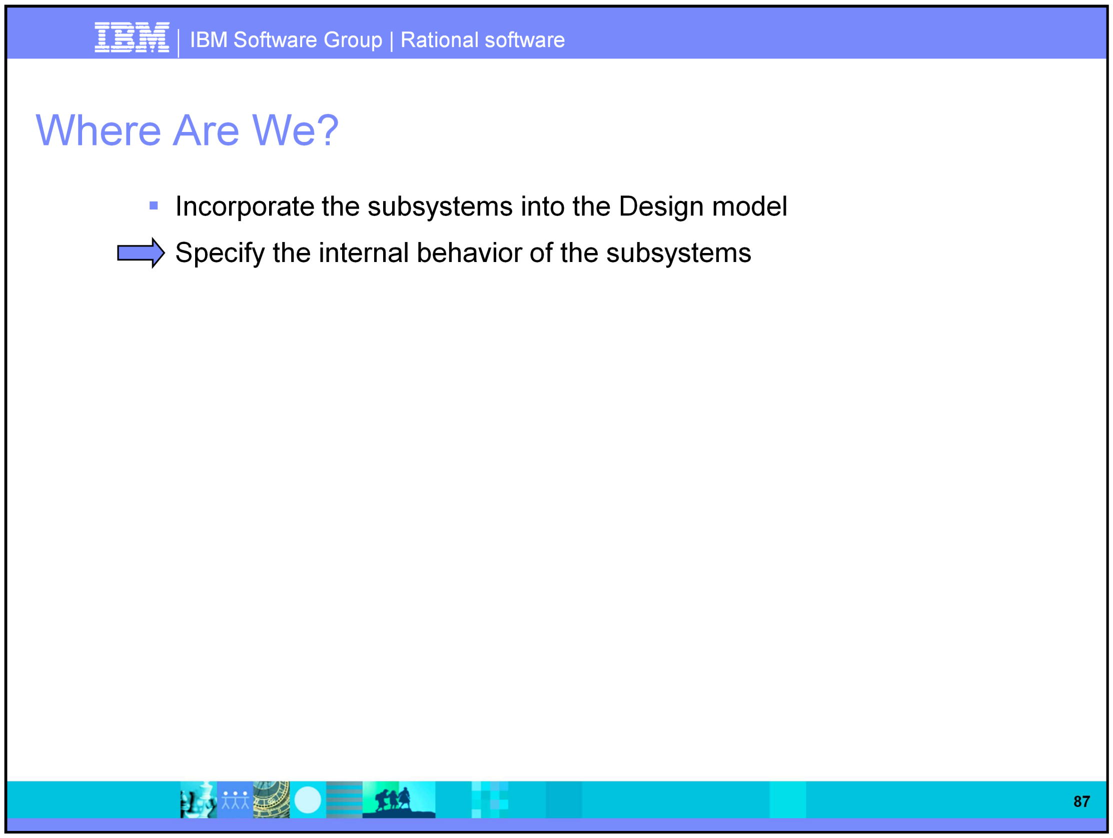
IBM | IBM Software Group | Rational software

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

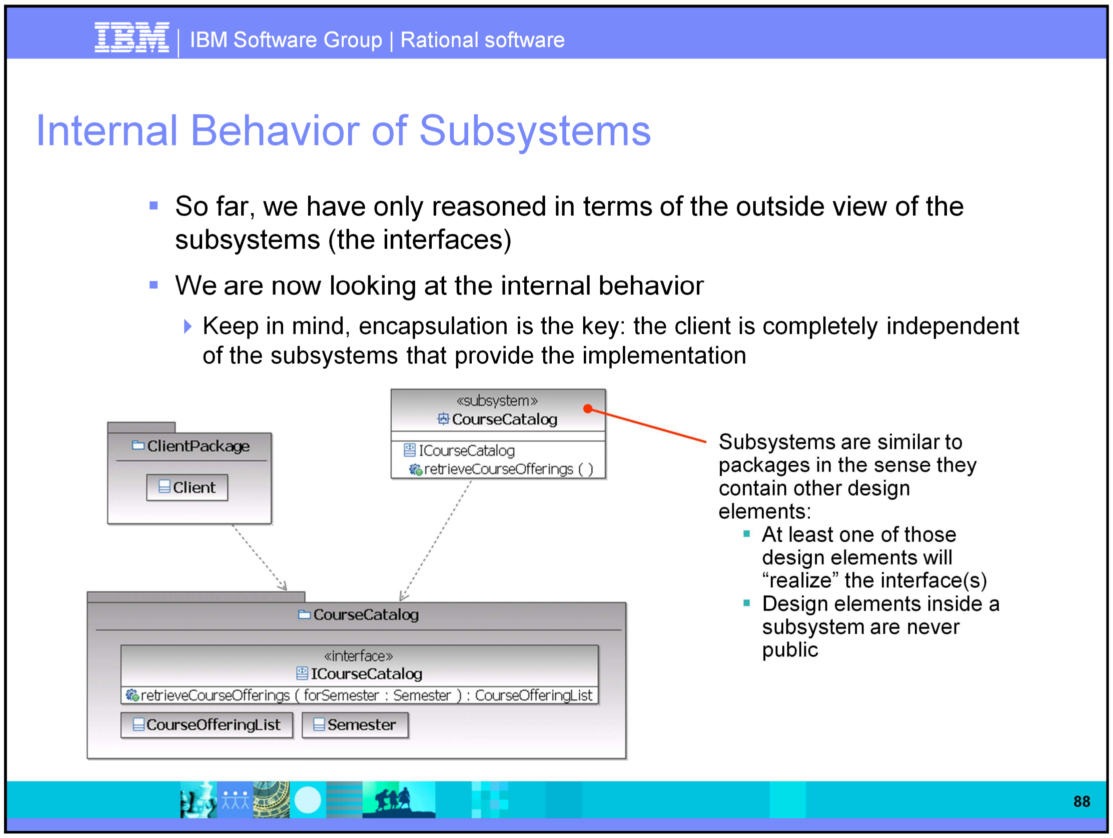

Analysis
:A
:B
If A becomes an interface in design, then the call to :B should be described in the subsystem interaction only
Design
:IA
:B
Subsystem A

86



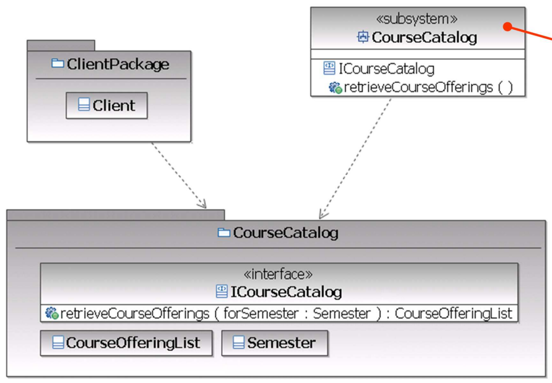
Where Are We?

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

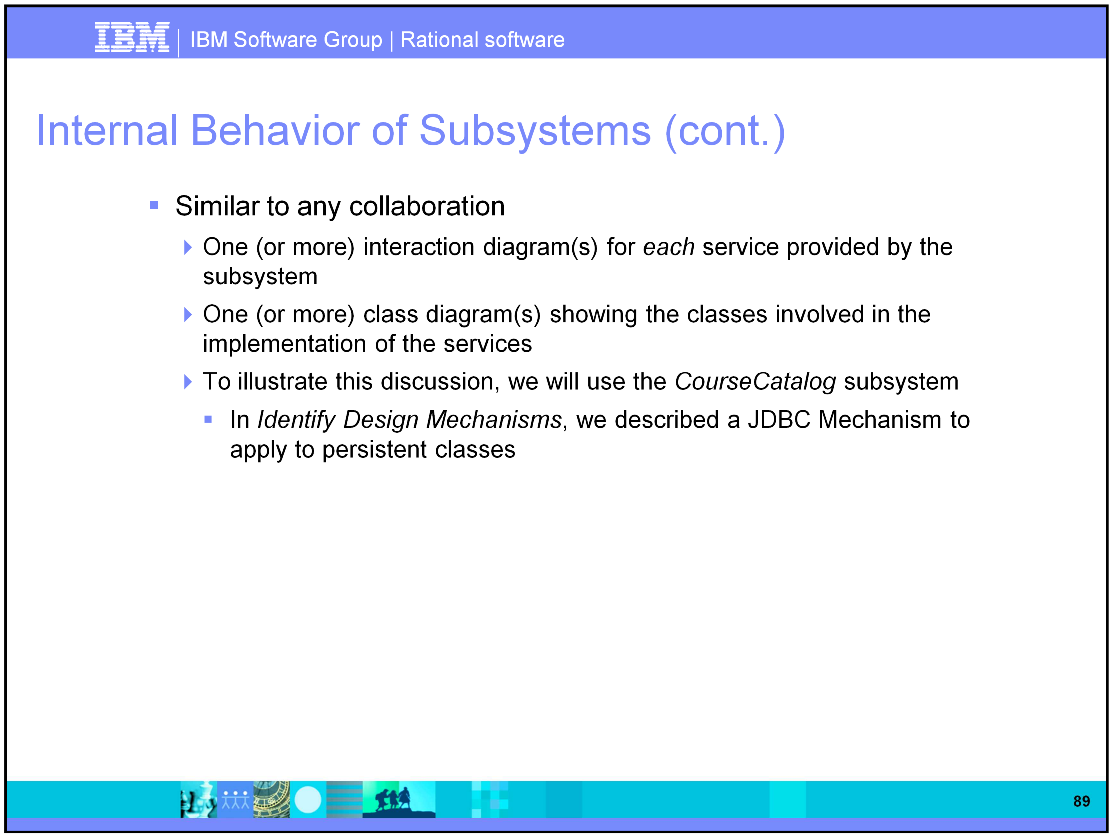


Internal Behavior of Subsystems

- 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

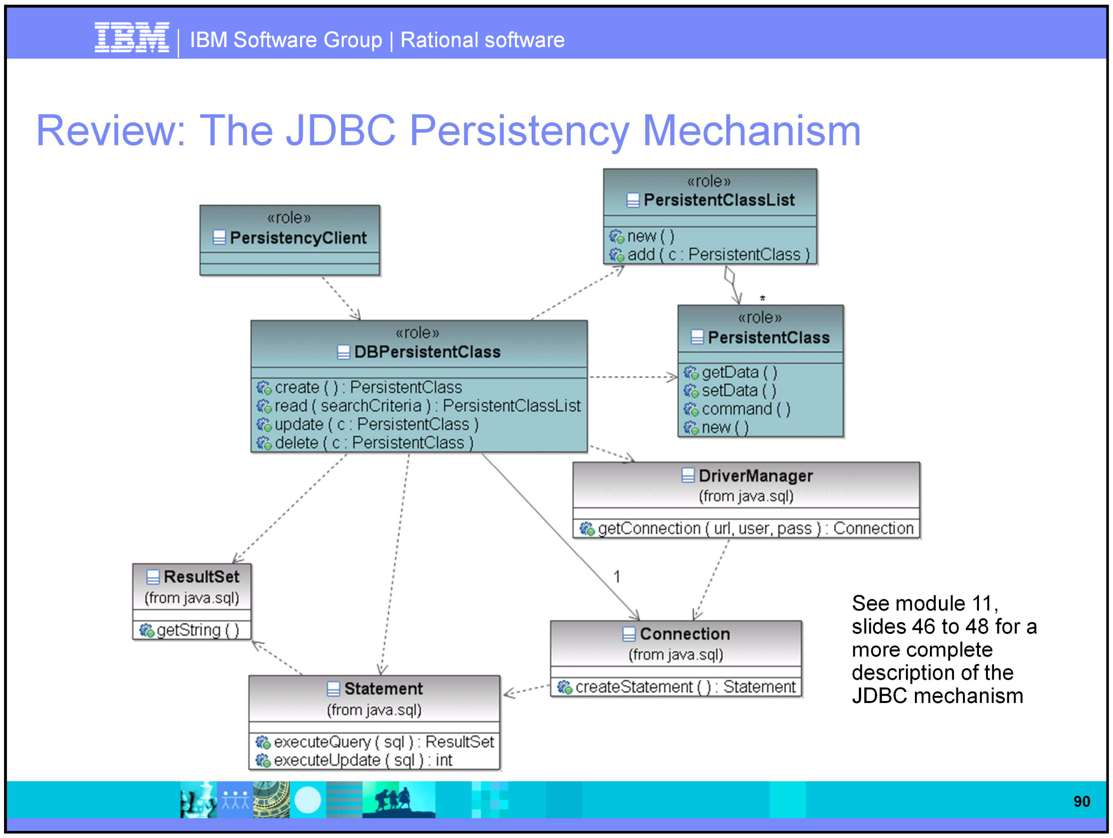


IBM Software Group | Rational software

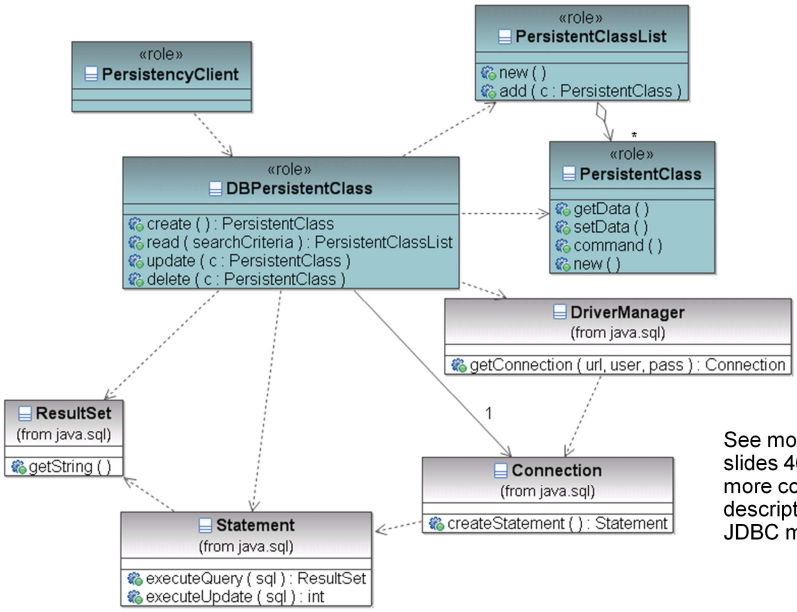
Internal Behavior of Subsystems (cont.)

- 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

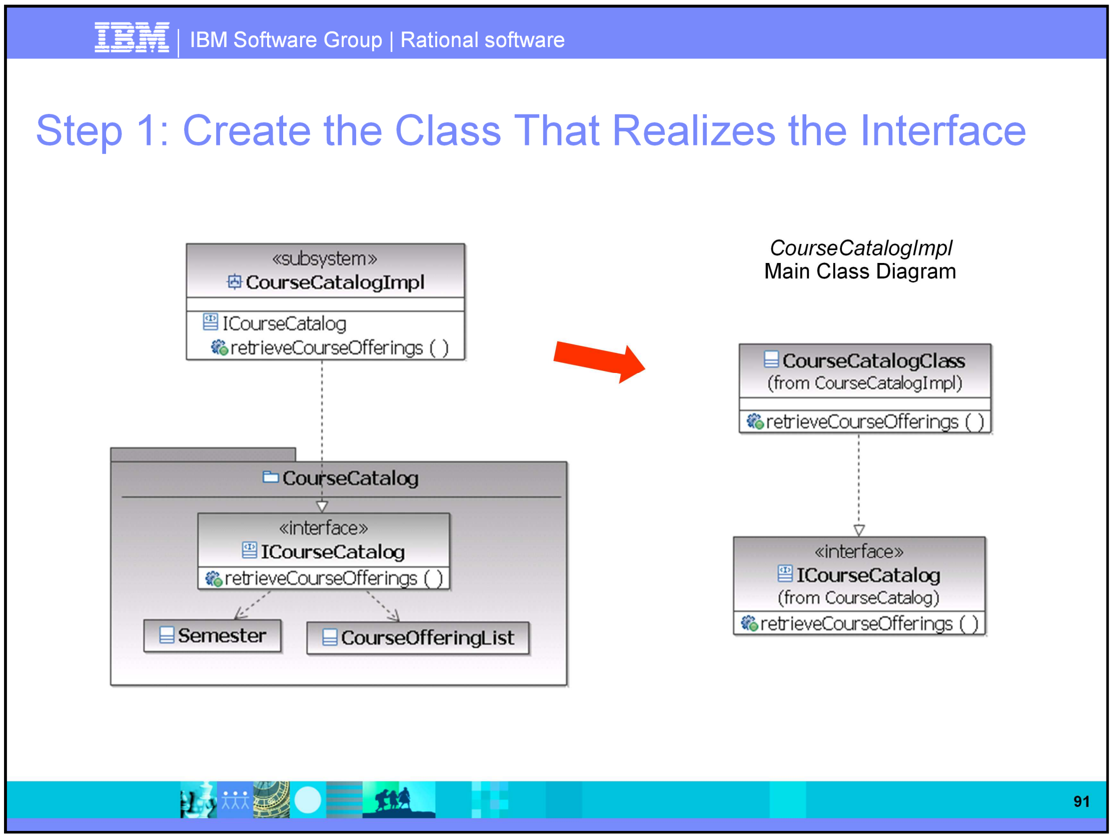


Review: The JDBC Persistency Mechanism

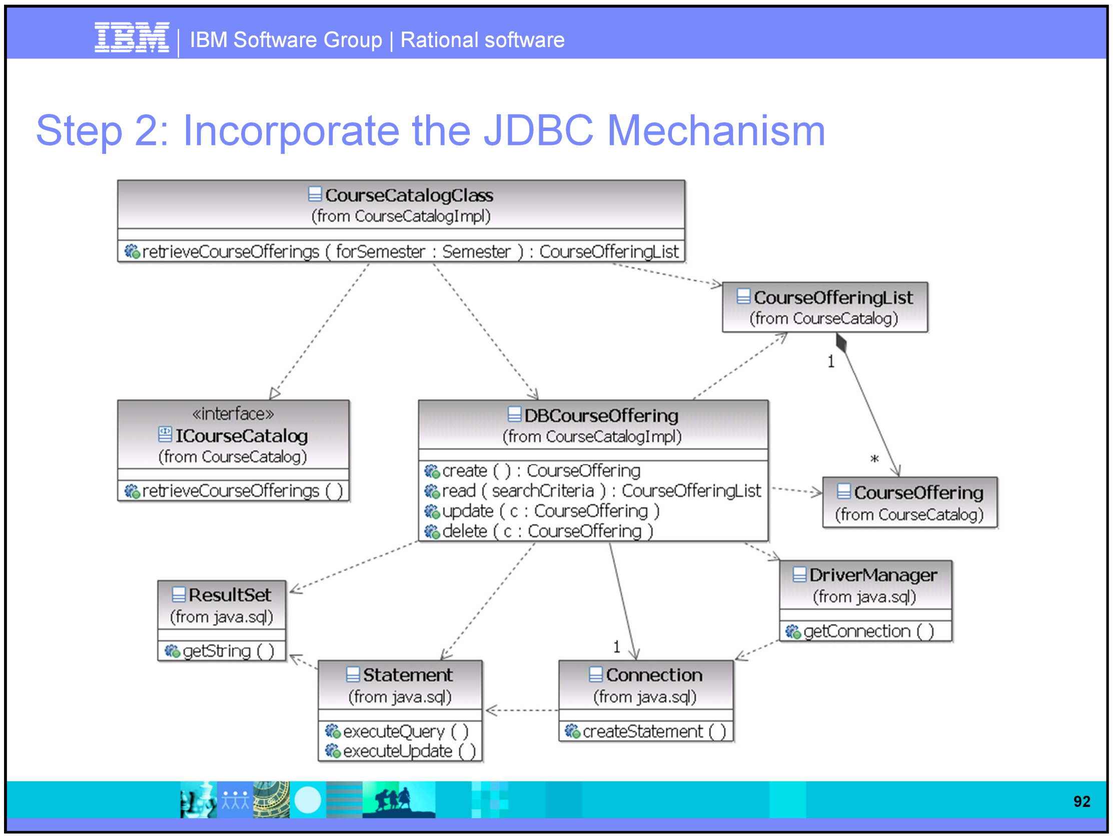

«role» PersistencyClient
«role» PersistentClassList
new ( )
add (c : PersistentClass)
«role» DBPersistentClass
create ( ) : PersistentClass
read (searchCriteria) : PersistentClassList
update (c : PersistentClass)
delete (c : PersistentClass)
«role» PersistentClass
getData ( )
setData ( )
command ( )
new ( )
*
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

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

90

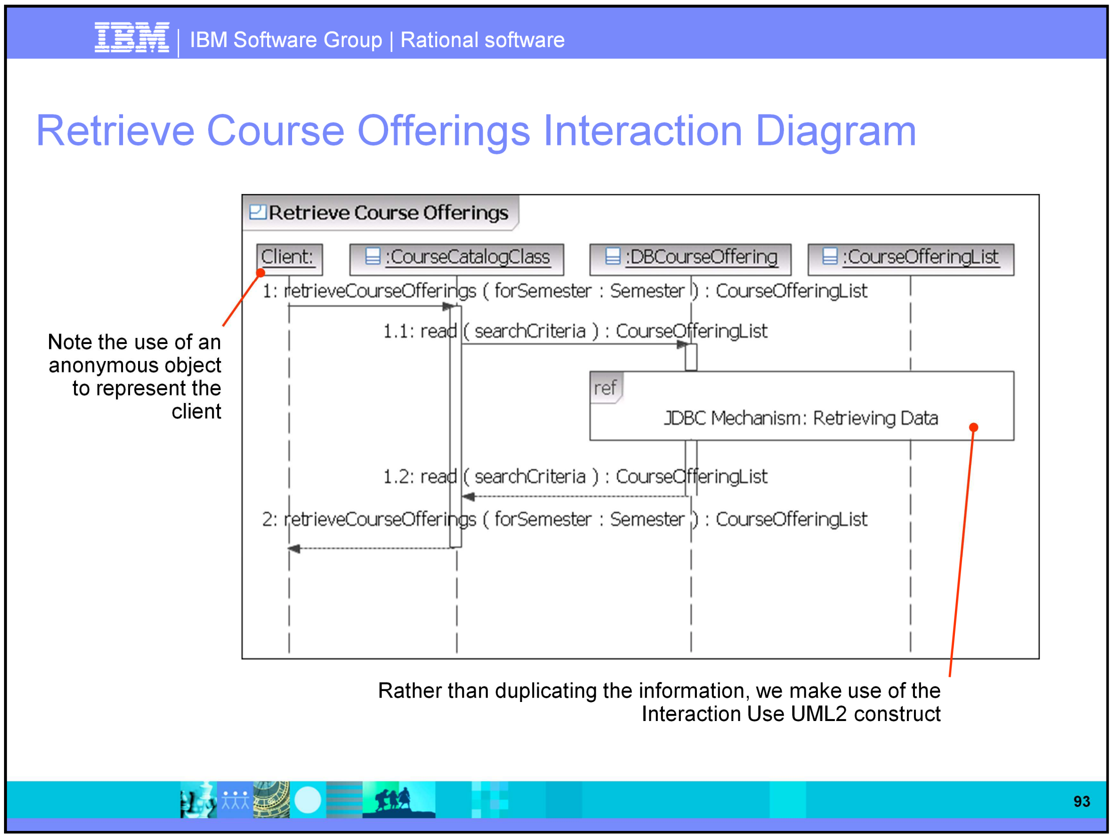


Step 1: Create the Class That Realizes the Interface



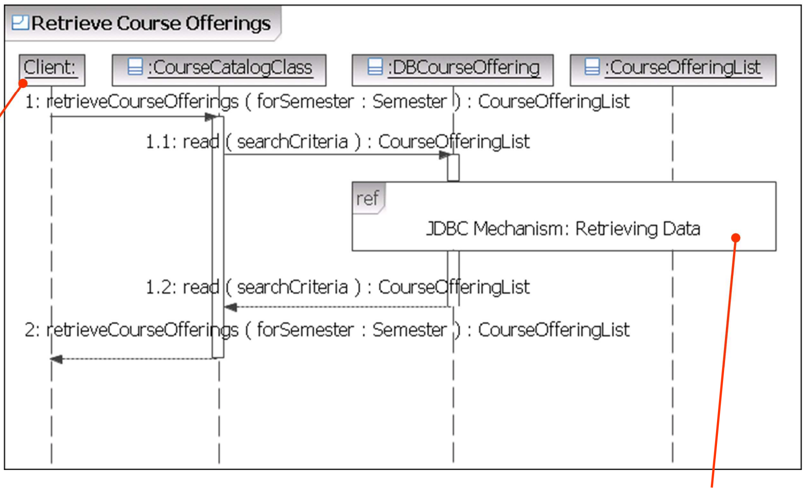
IBM Software Group | Rational software

Step 2: Incorporate the JDBC Mechanism



IBM Software Group | Rational software

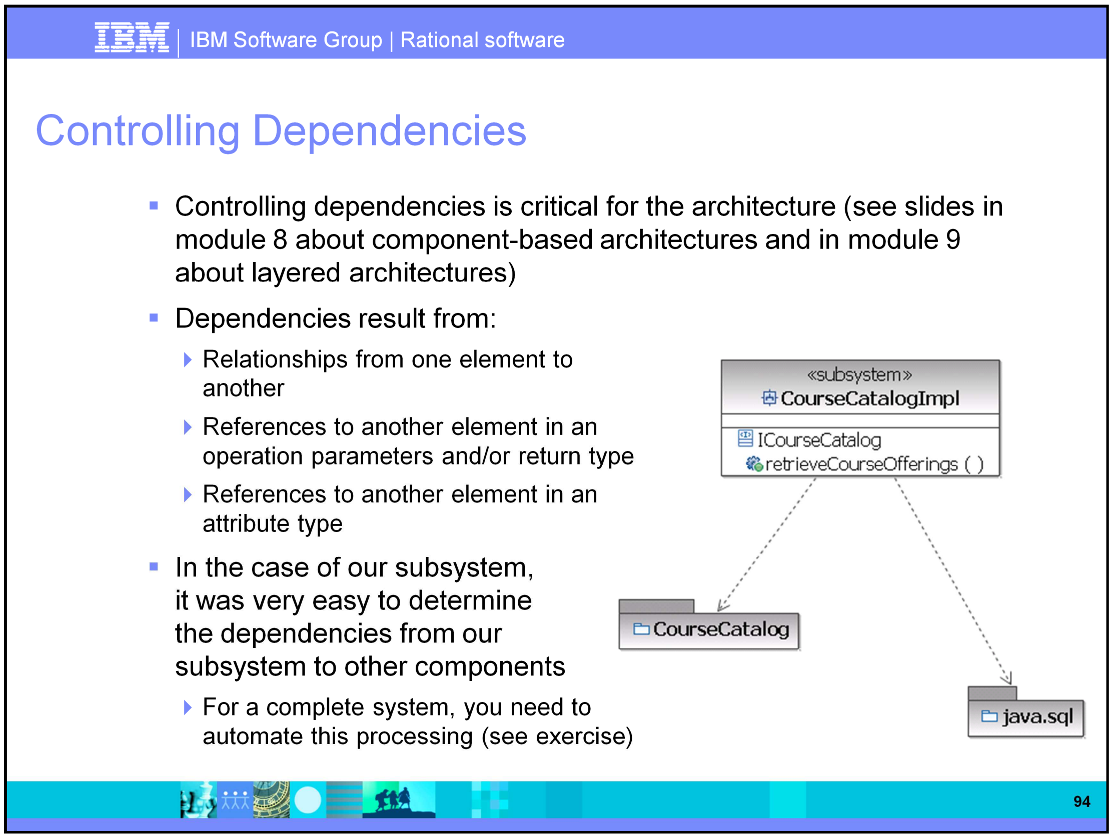
Retrieve Course Offerings Interaction Diagram


Retrieve Course Offerings
Client:
:CourseCatalogClass
:DBCourseOffering
:CourseOfferingList
1: retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList
1.1: read (searchCriteria) : CourseOfferingList
ref
JDBC Mechanism: Retrieving Data
1.2: read (searchCriteria) : CourseOfferingList
2: retrieveCourseOfferings (forSemester : Semester) : CourseOfferingList

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

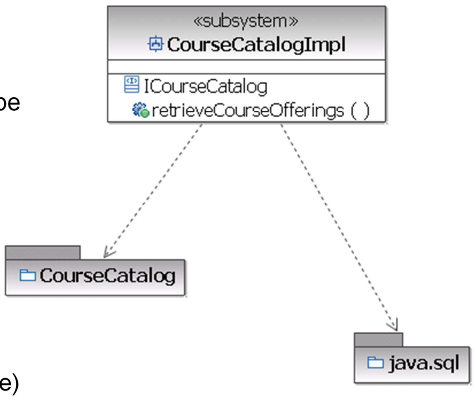
93



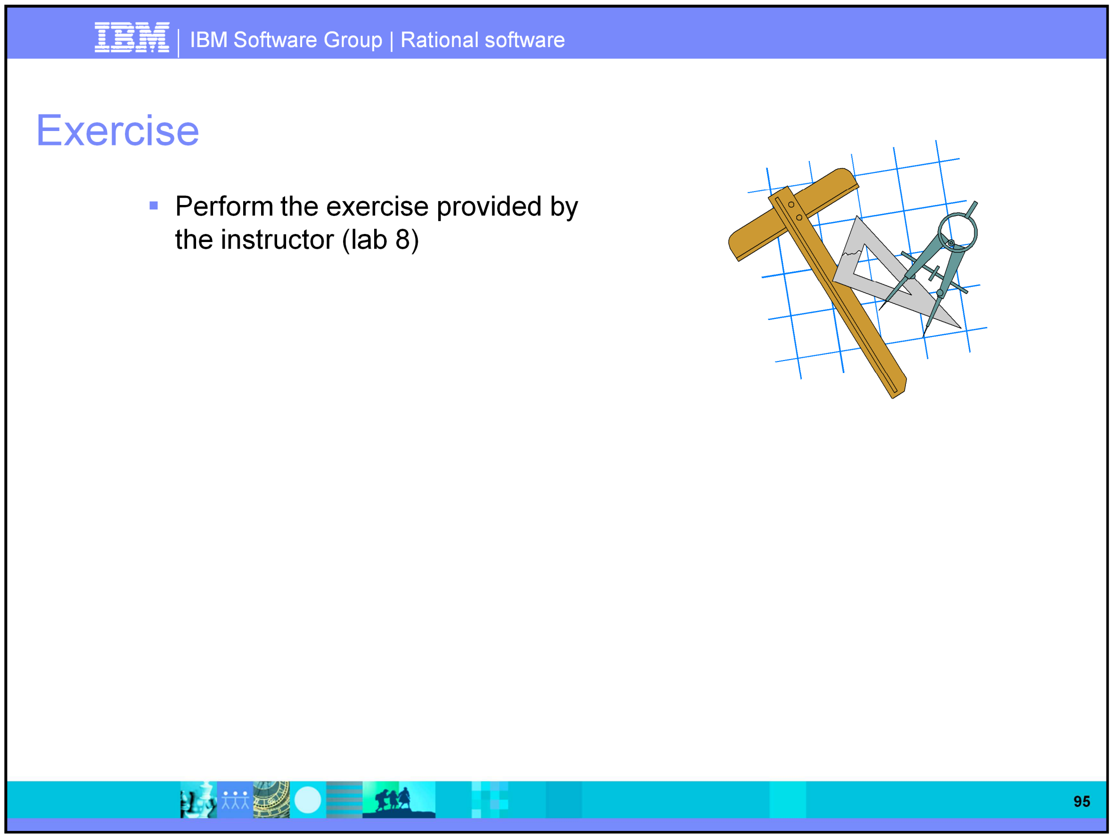
IBM | IBM Software Group | Rational software

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

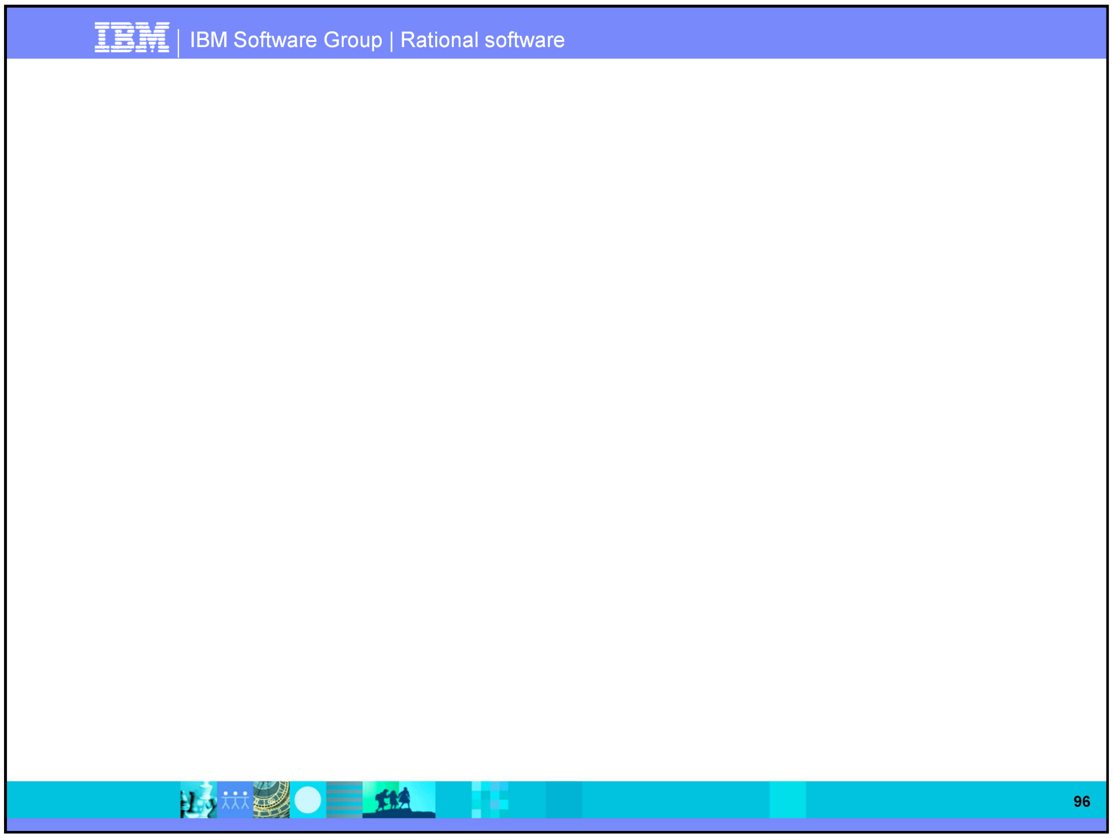


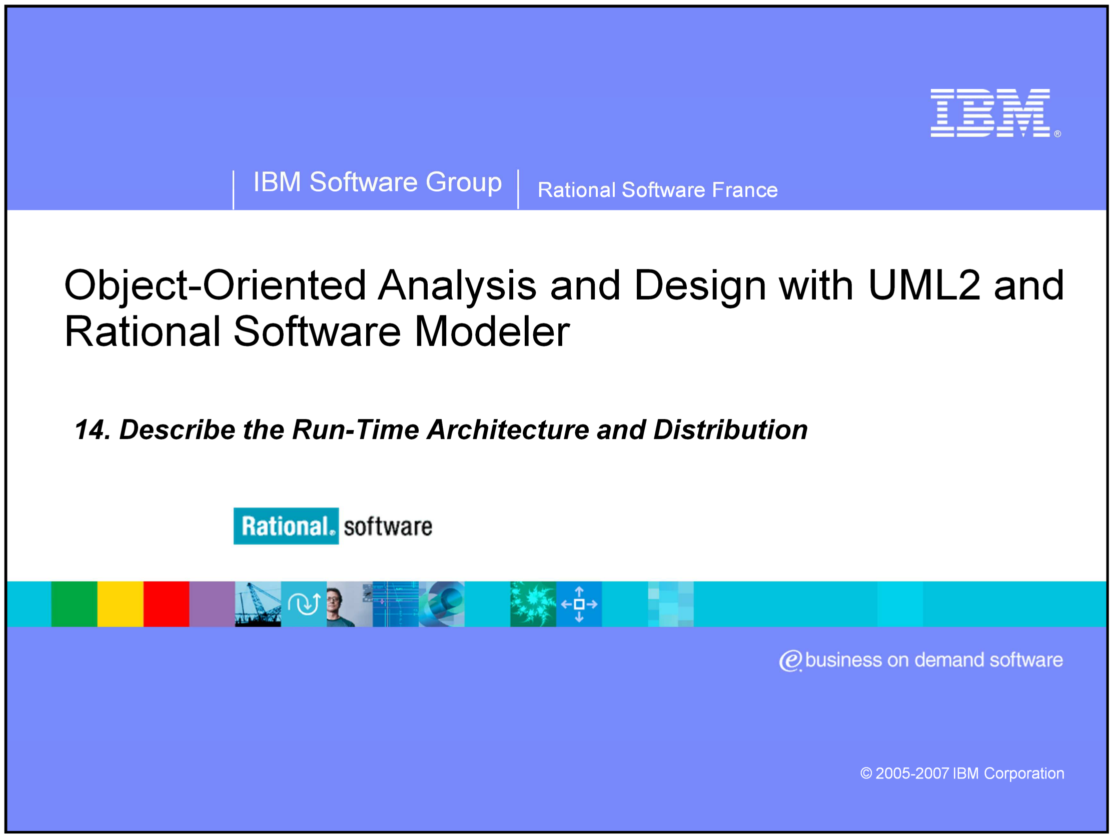
IBM Software Group | Rational software

Exercise

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

95





IBM Software Group | Rational Software France

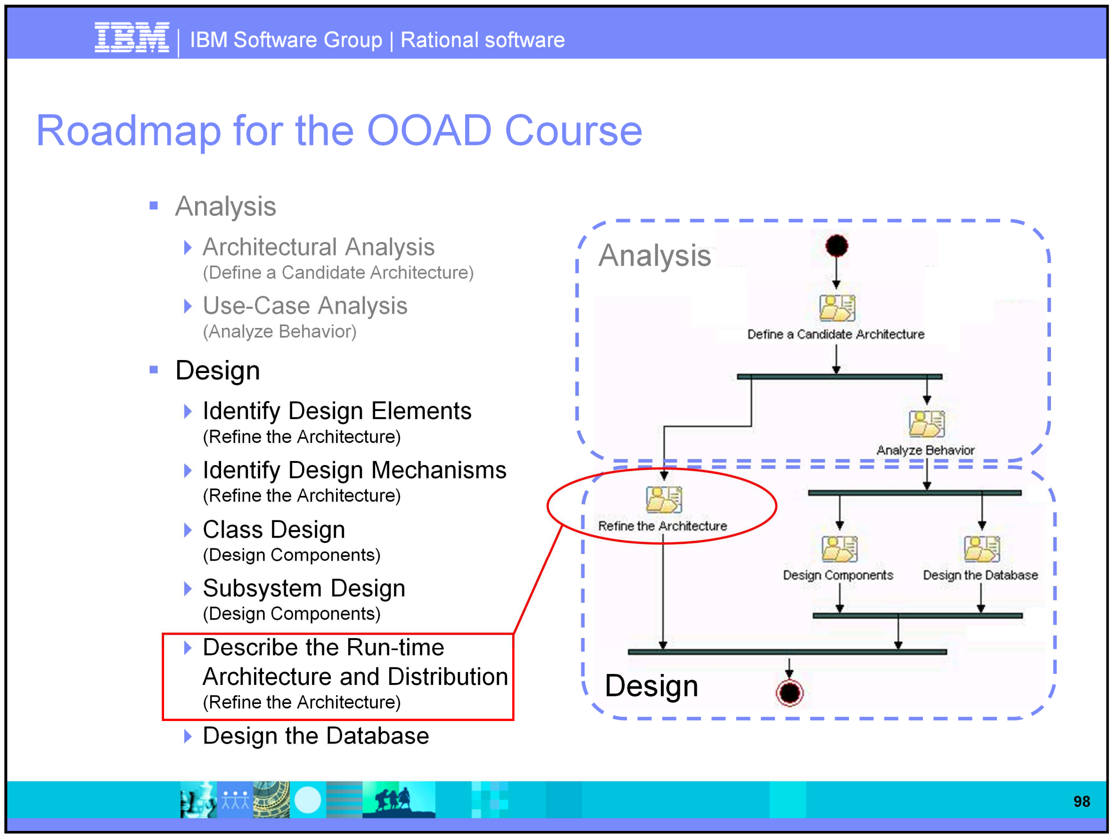
Object-Oriented Analysis and Design with UML2 and Rational Software Modeler

14. Describe the Run-Time Architecture and Distribution

Rational. software

e business on demand software

© 2005-2007 IBM Corporation

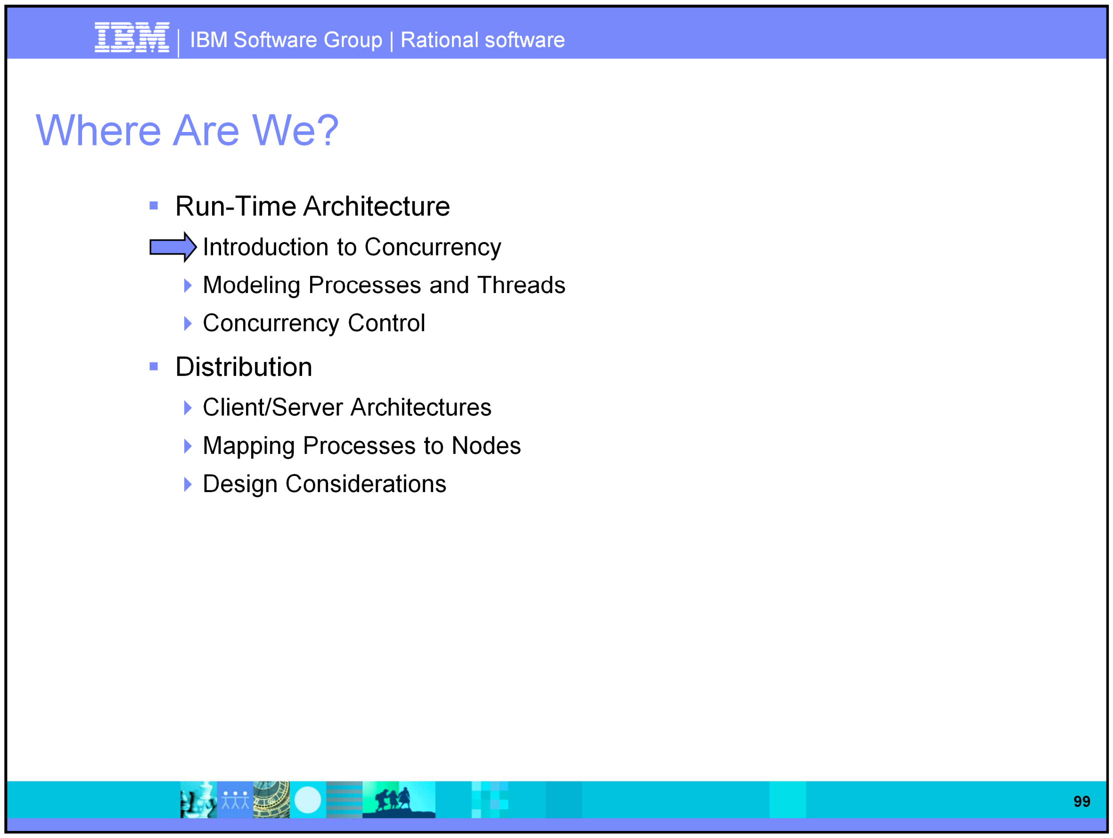


IBM | IBM Software Group | Rational software

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

98

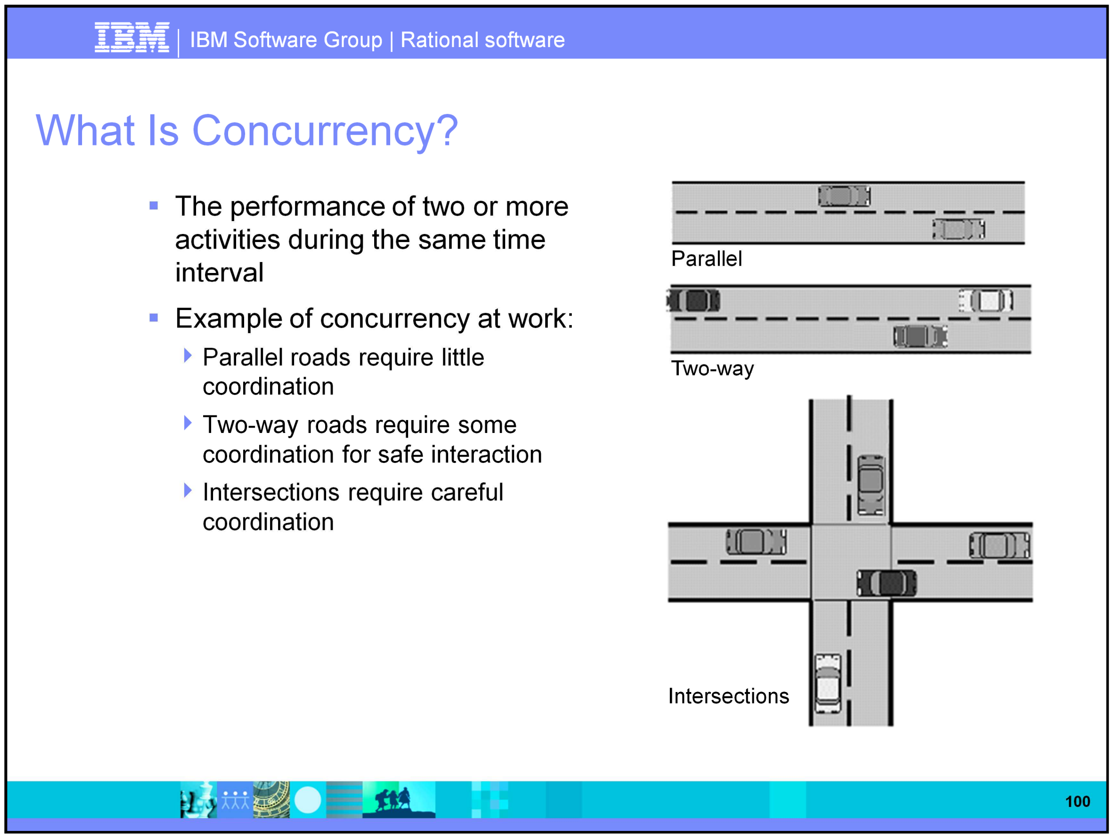


IBM | IBM Software Group | Rational software

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

99



What Is Concurrency?

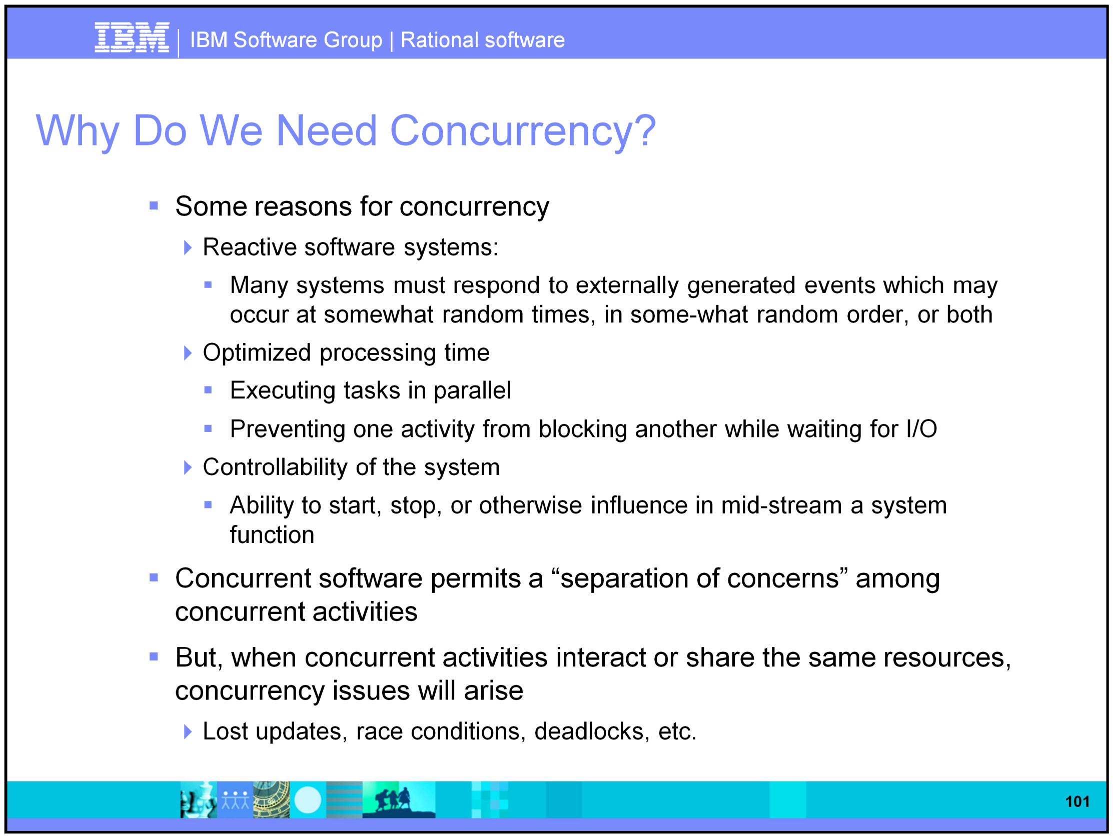
- The performance of two or more activities during the same time interval
- Example of concurrency at work:
  - Parallel roads require little coordination
  - Two-way roads require some coordination for safe interaction
  - Intersections require careful coordination

Parallel
Two-way
Intersections

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.

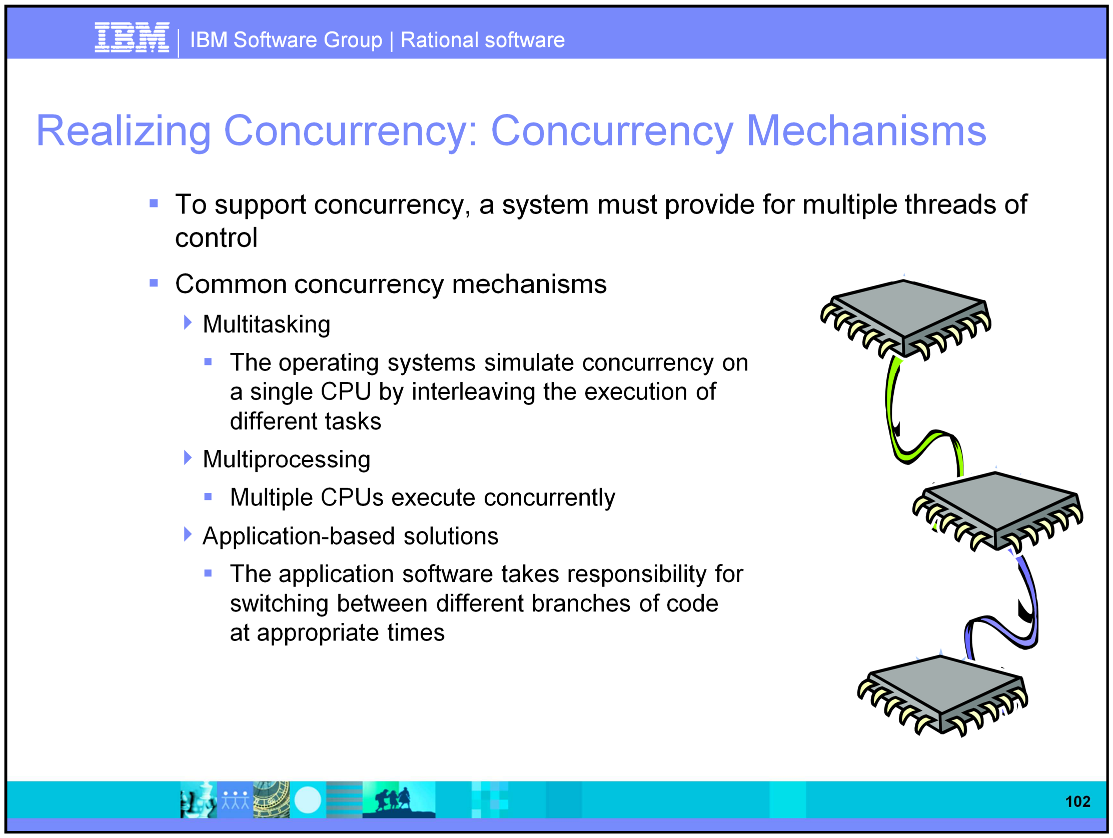
101

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 | IBM Software Group | Rational software

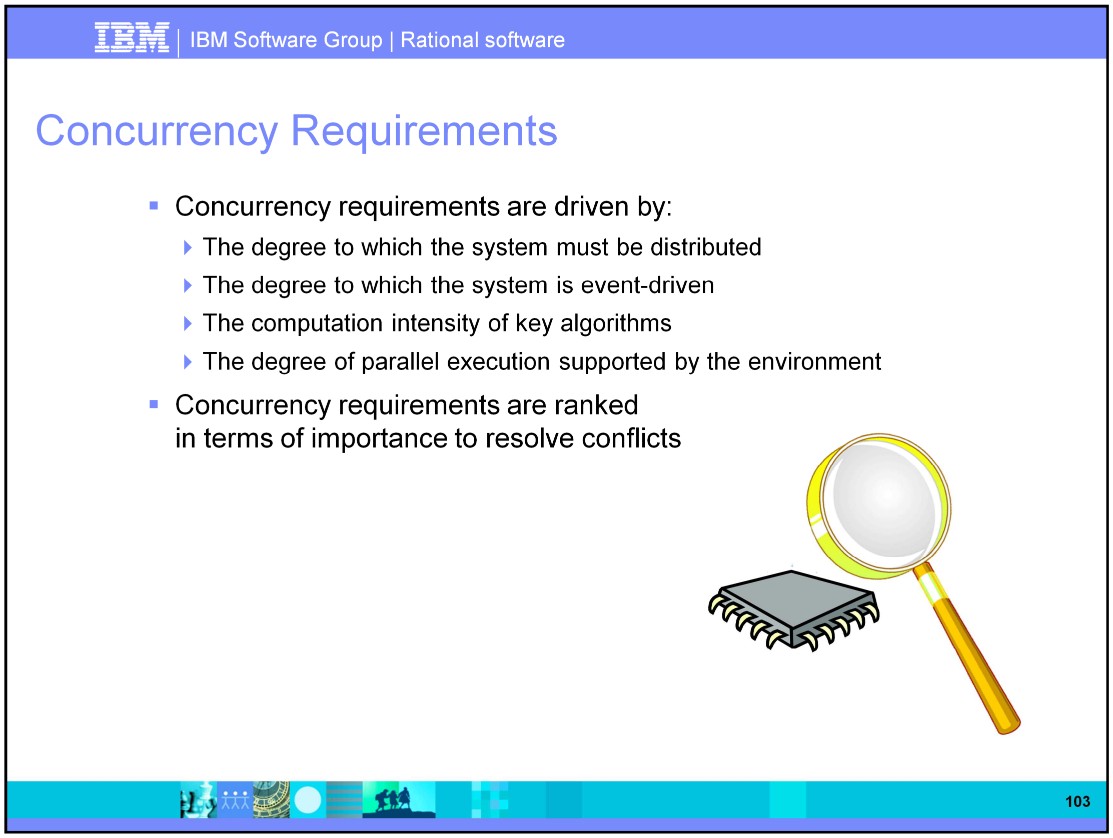
Realizing Concurrency: Concurrency Mechanisms

- 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.



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

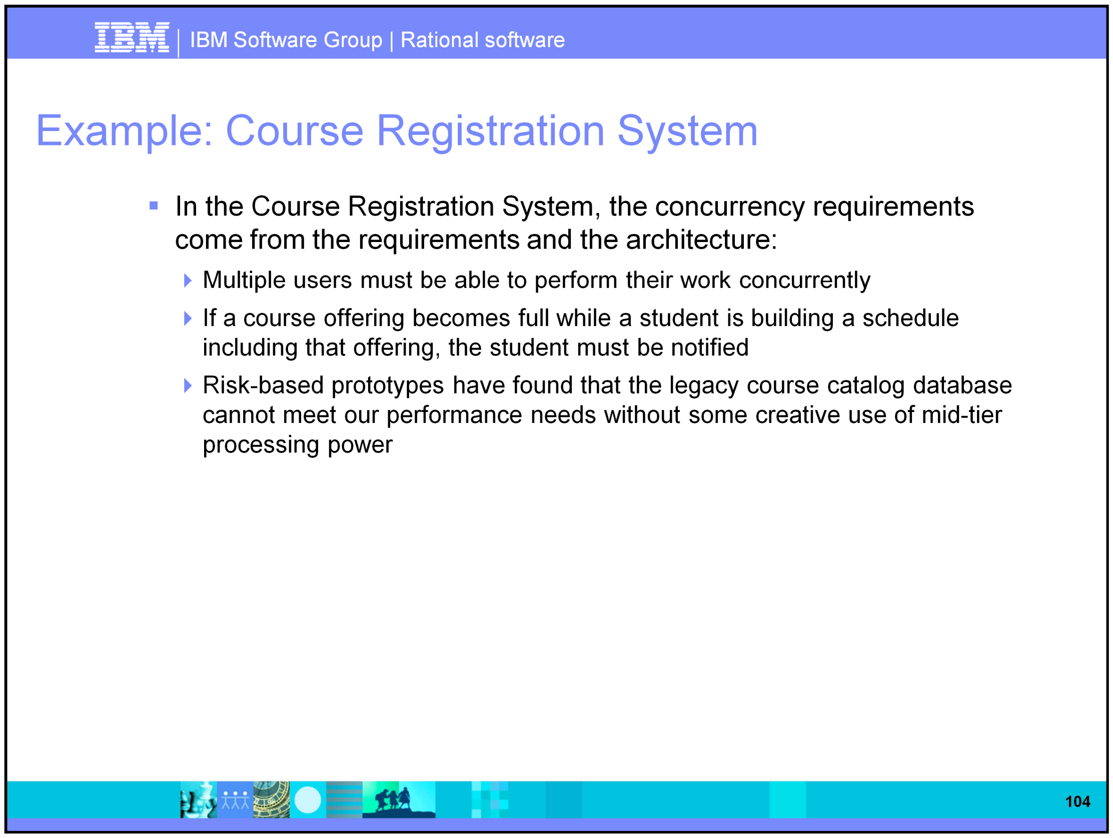
Concurrency requirements define the extent to which parallel execution of tasks is required for the system. These requirements help shape the architecture.

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.


IBM Software Group | Rational software

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

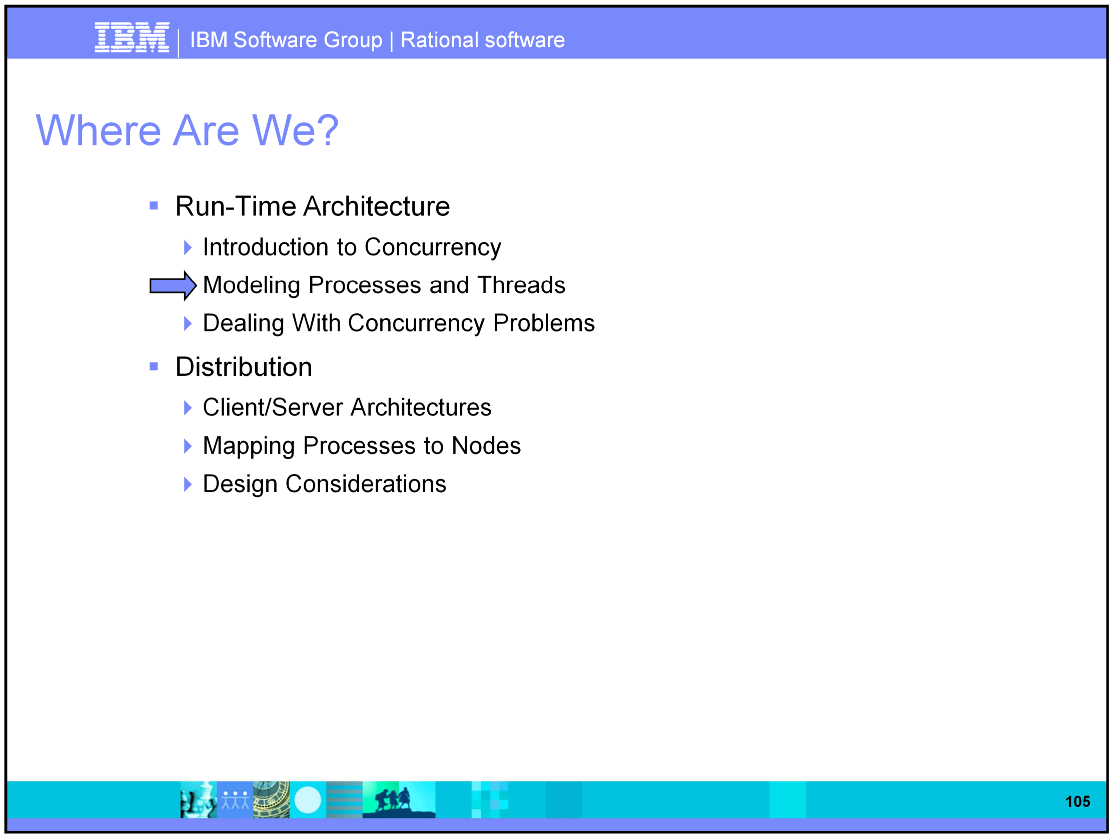
104

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.

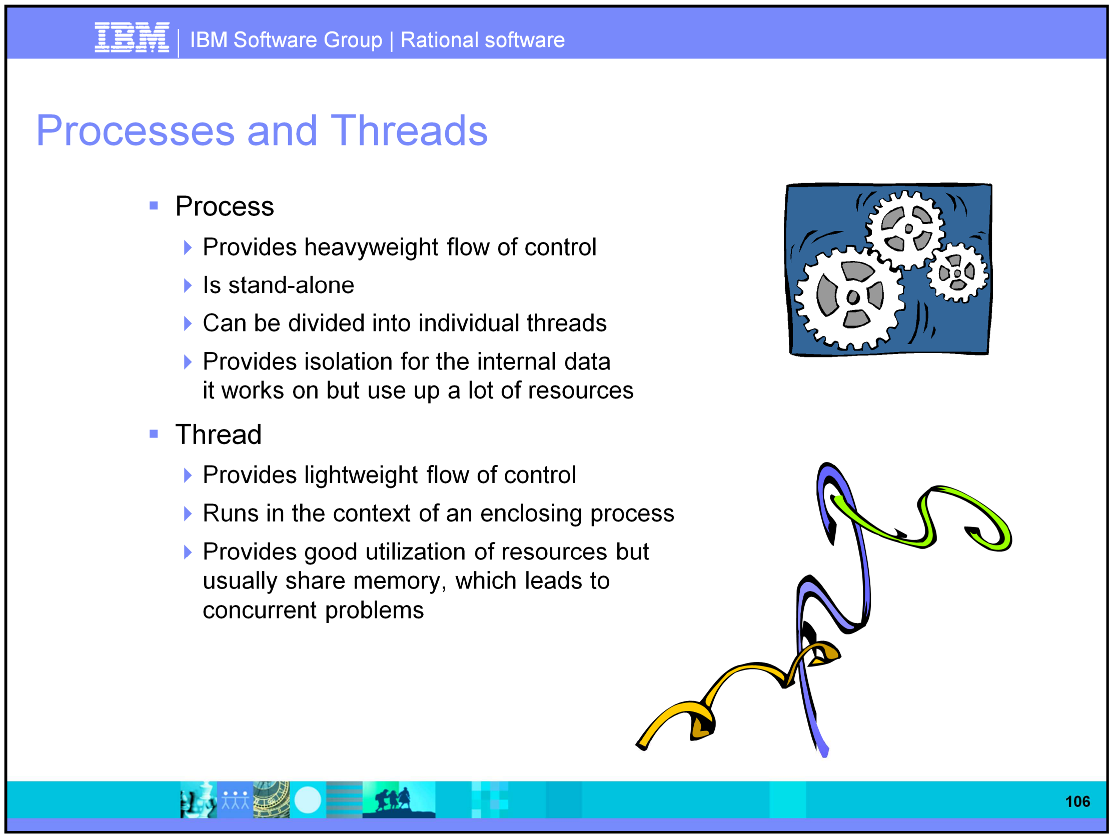


IBM Software Group | Rational software

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

105



IBM Software Group | Rational software

Processes and Threads

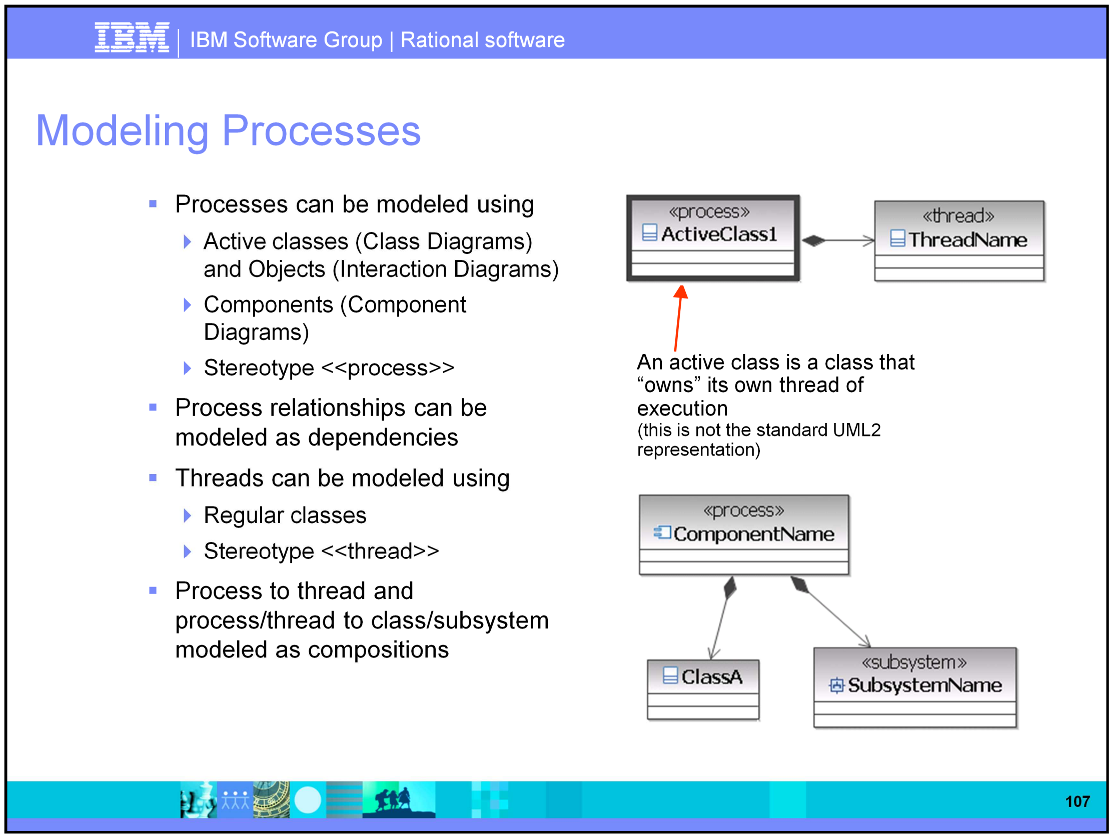
- Process
  - Provides heavyweight flow of control
  - Is stand-alone
  - Can be divided into individual threads
  - Provides isolation for the internal data it works on but use up a lot of resources
- Thread
  - Provides lightweight flow of control
  - Runs in the context of an enclosing process
  - Provides good utilization of resources but usually share memory, which leads to concurrent problems

106

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.



IBM | IBM Software Group | Rational software

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)

107

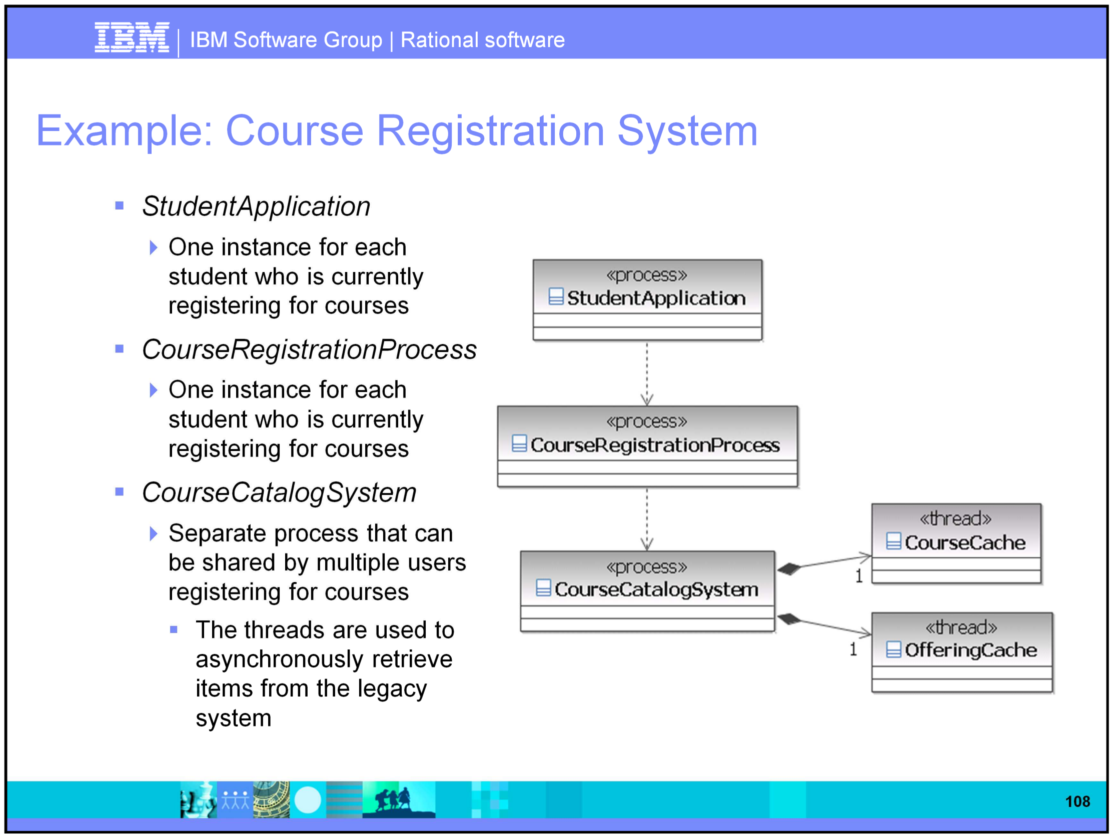
You can use "active" classes to model processes and threads. An active class is a class that "owns" its own thread of execution and can initiate control activity, contrasted with passive classes that can only be acted upon. Active classes can execute in parallel (that is, concurrently) with other active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


IBM Software Group | Rational software

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

«process» StudentApplication
«process» CourseRegistrationProcess
«process» CourseCatalogSystem
«thread» CourseCache
«thread» OfferingCache
1
1

108

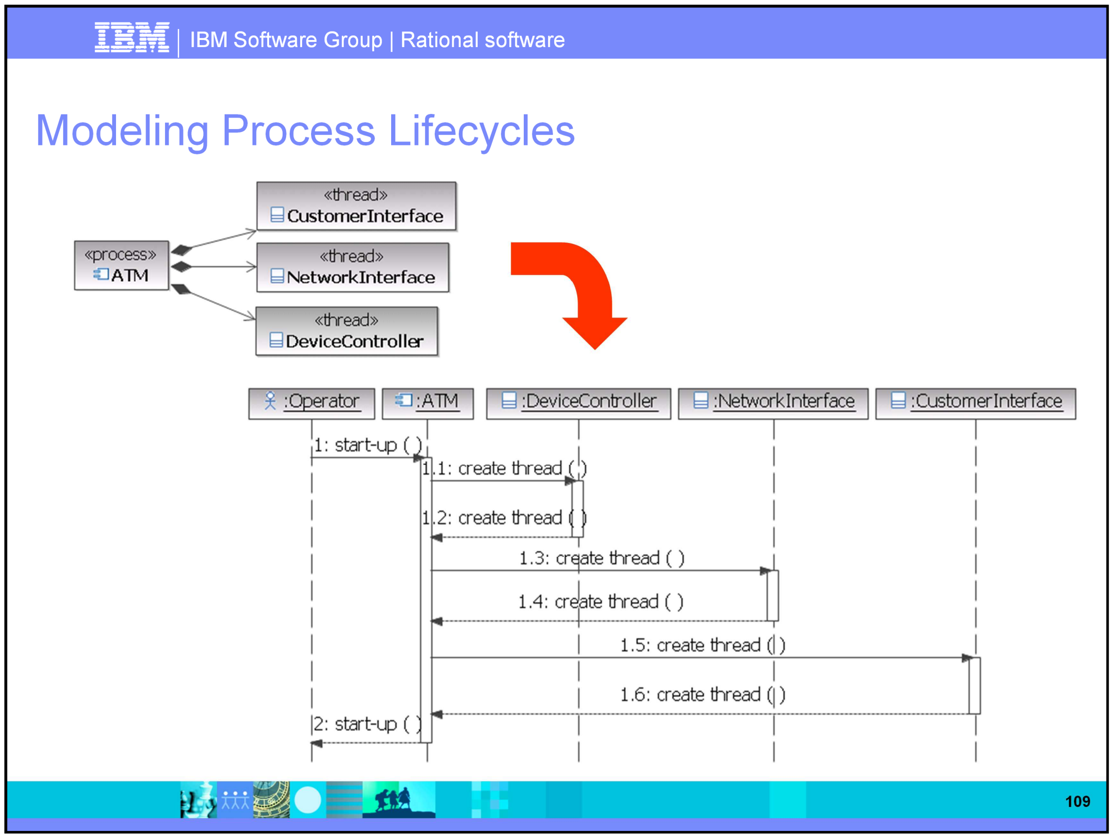
The above example demonstrates how processes and threads are modeled. Processes and threads are represented as stereotyped classes. Separate processes have dependencies among them. When there are threads within a process composition is used. The composition relationship indicates that the threads are 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.

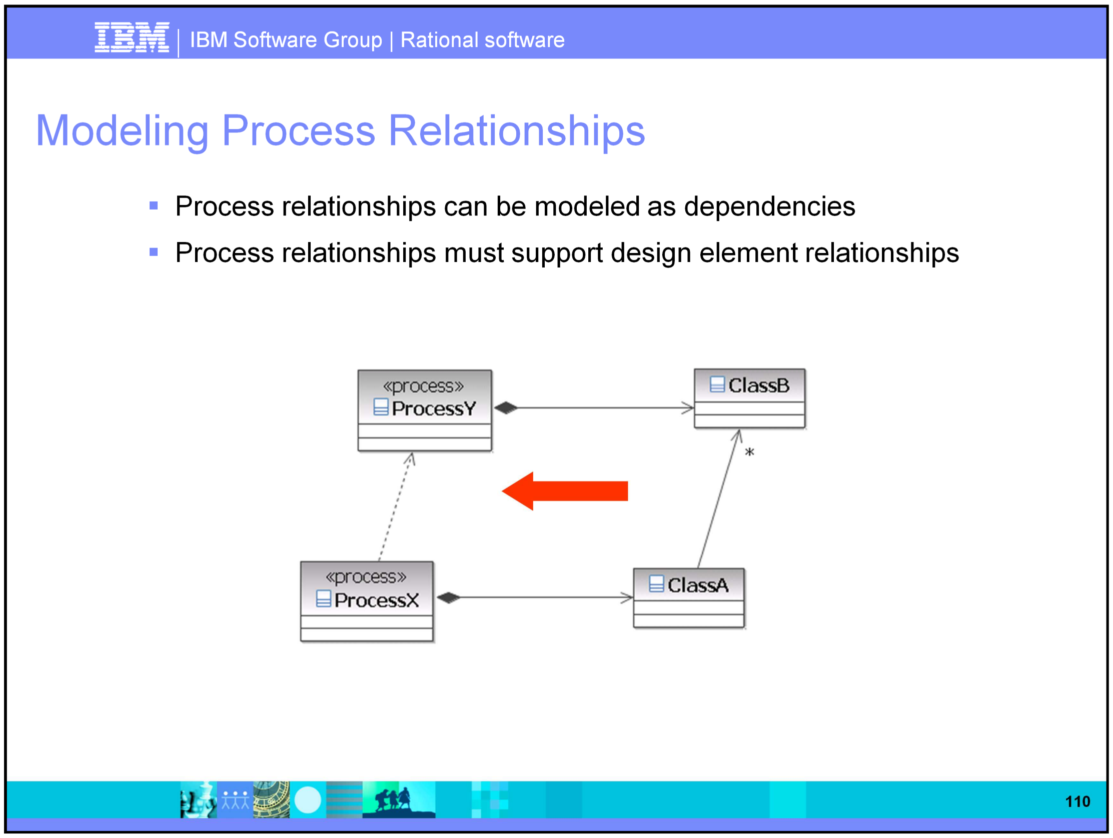
The CourseRegistrationProcess talks to the separate CourseCatalogSystemAccess process, which manages access to the legacy system. CourseCatalogSystemAccess is a separate process that can be shared by multiple users registering for courses. This allows for a cache of recently retrieved courses and offerings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



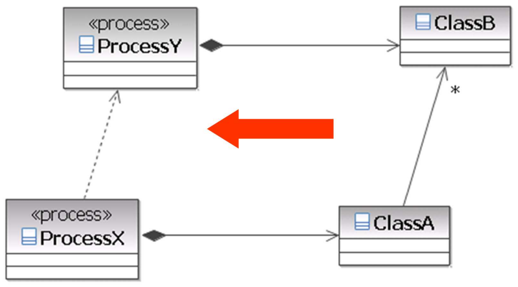
Modeling Process Lifecycles

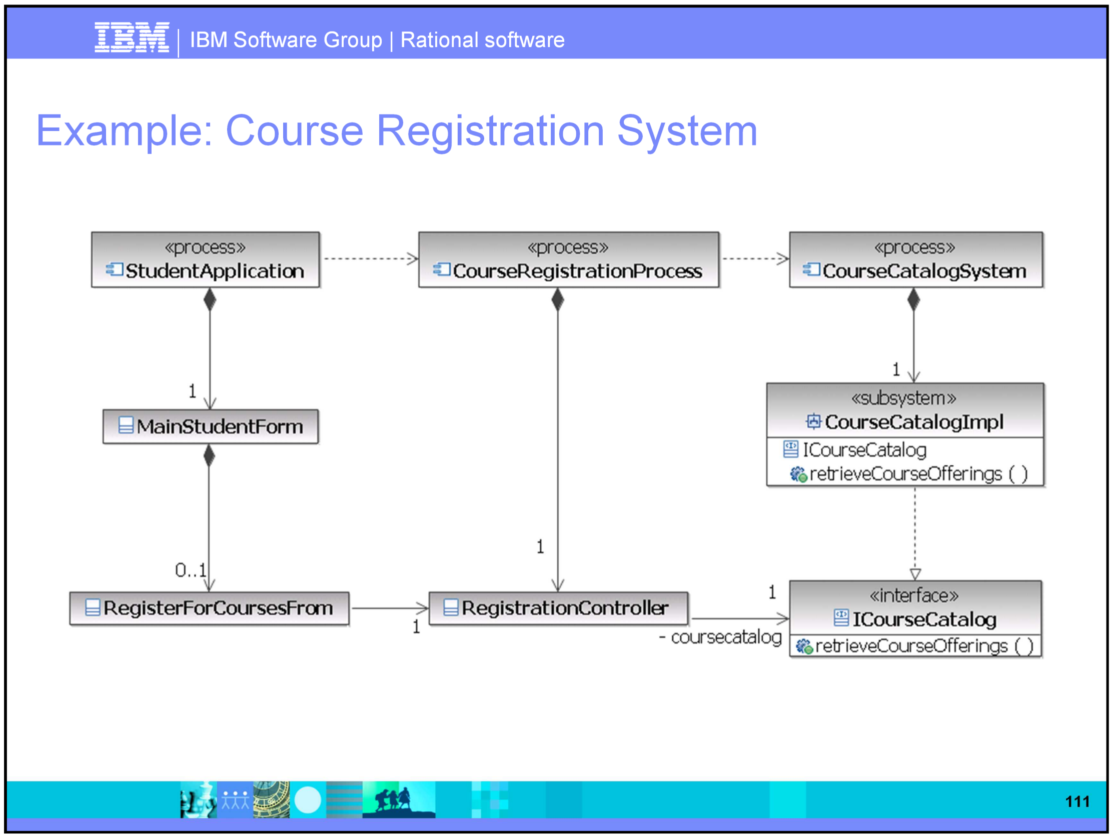
In the Automated Teller Machine, asynchronous events must be handled coming from three different sources: the user of the system, the ATM devices (in the case of a jam in the cash dispenser, for example), or the ATM Network (in the case of a shutdown directive from the network). To handle these asynchronous 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

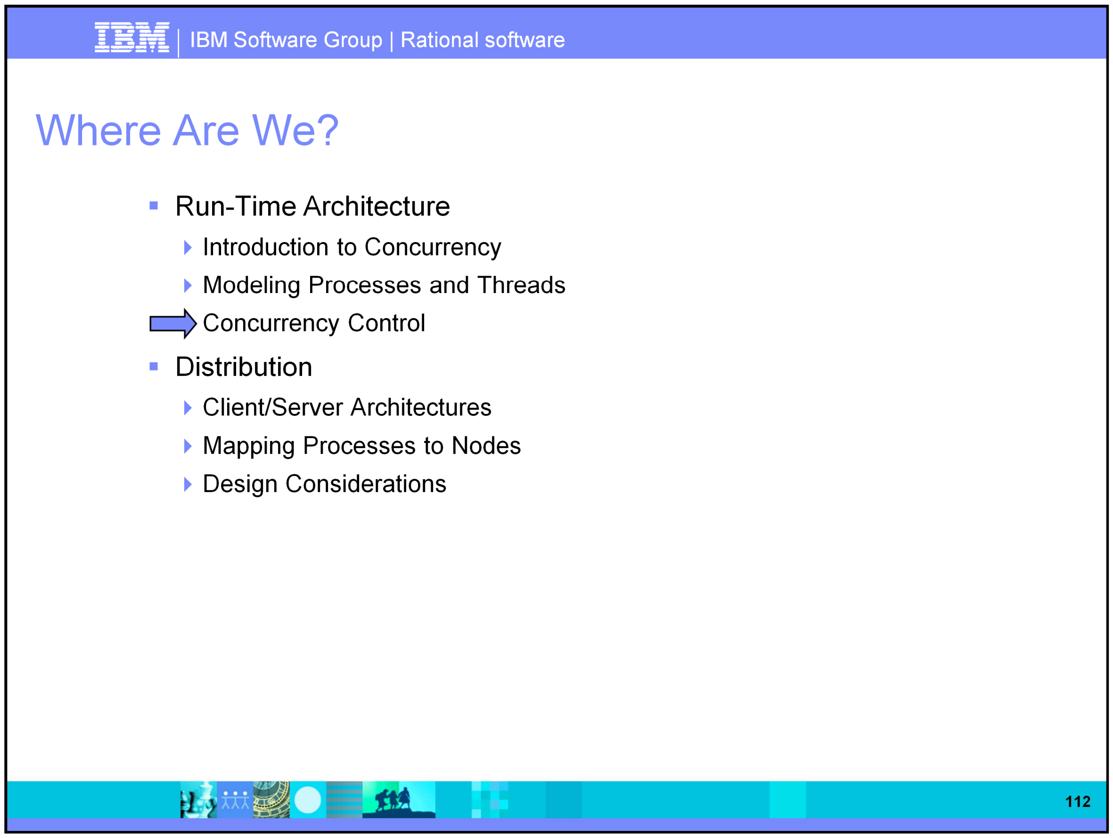




IBM | IBM Software Group | Rational software

Example: Course Registration System

111

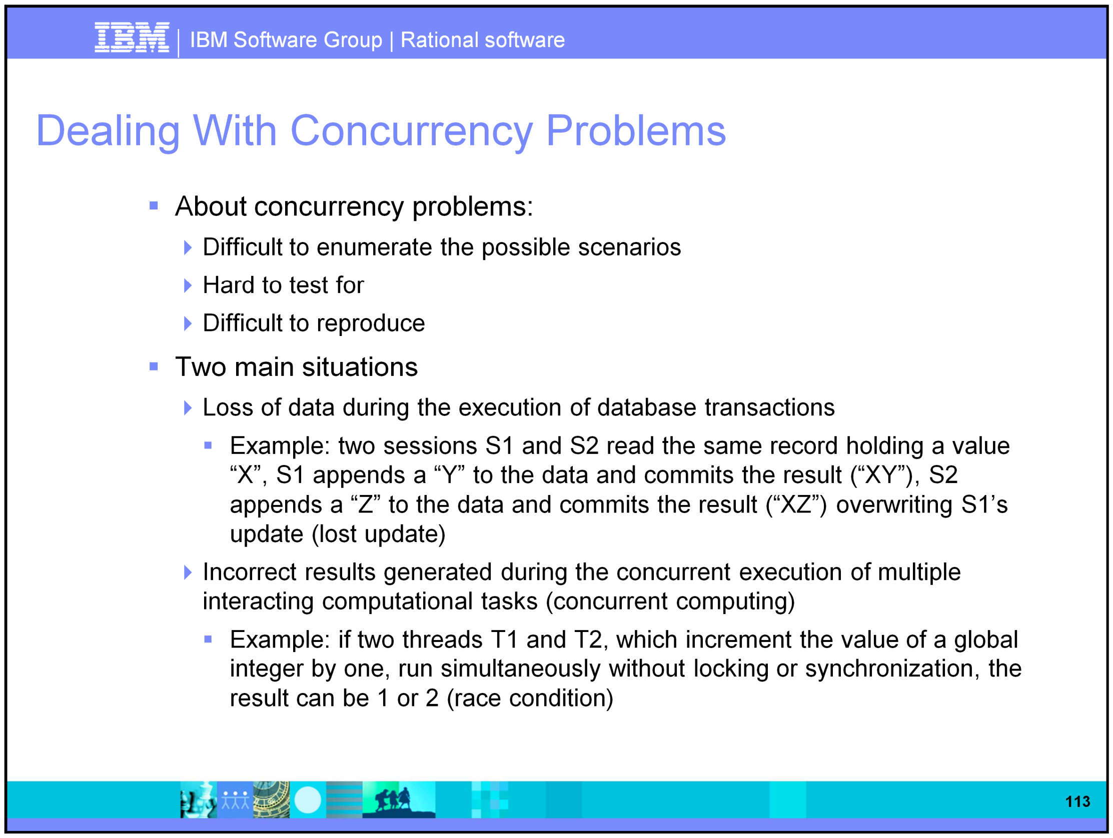


IBM | IBM Software Group | Rational software

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

112


IBM Software Group | Rational software

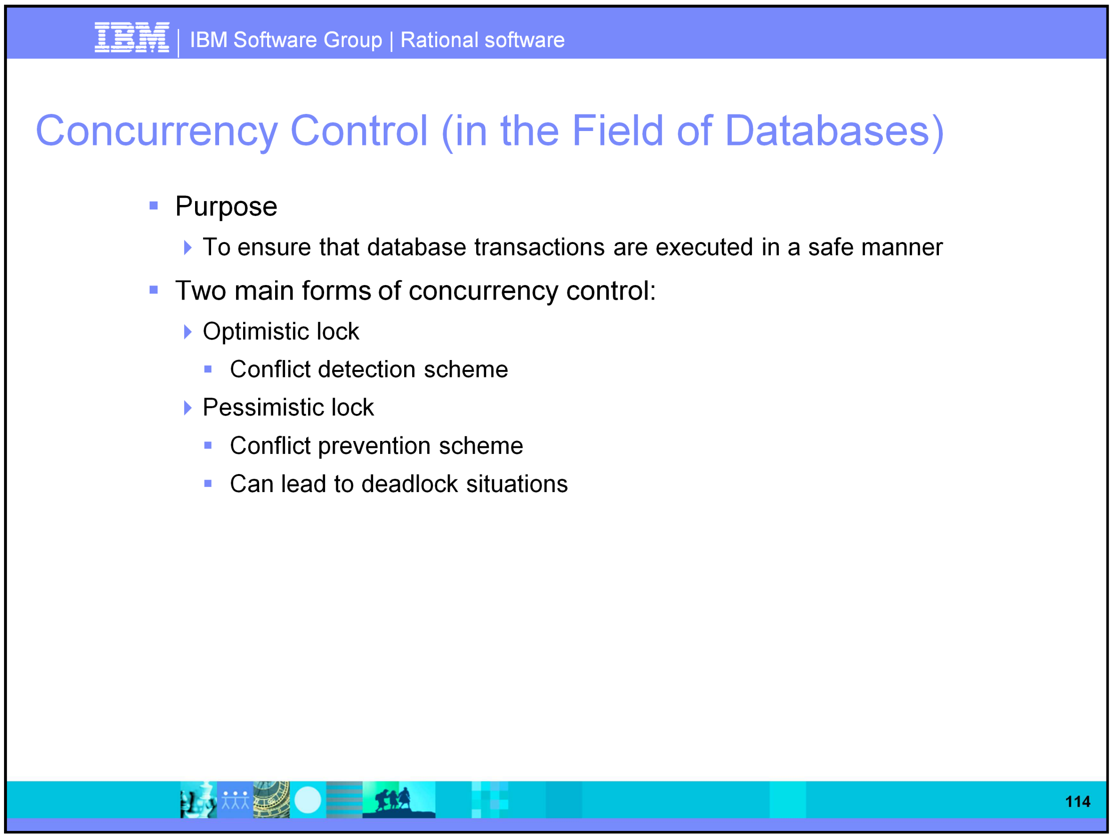
Dealing With Concurrency Problems

- 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)

113

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

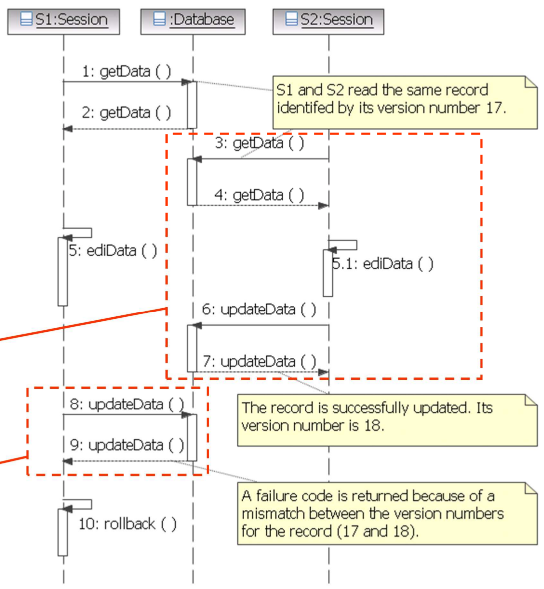
IBM | IBM Software Group | Rational software

Optimistic Lock Pattern

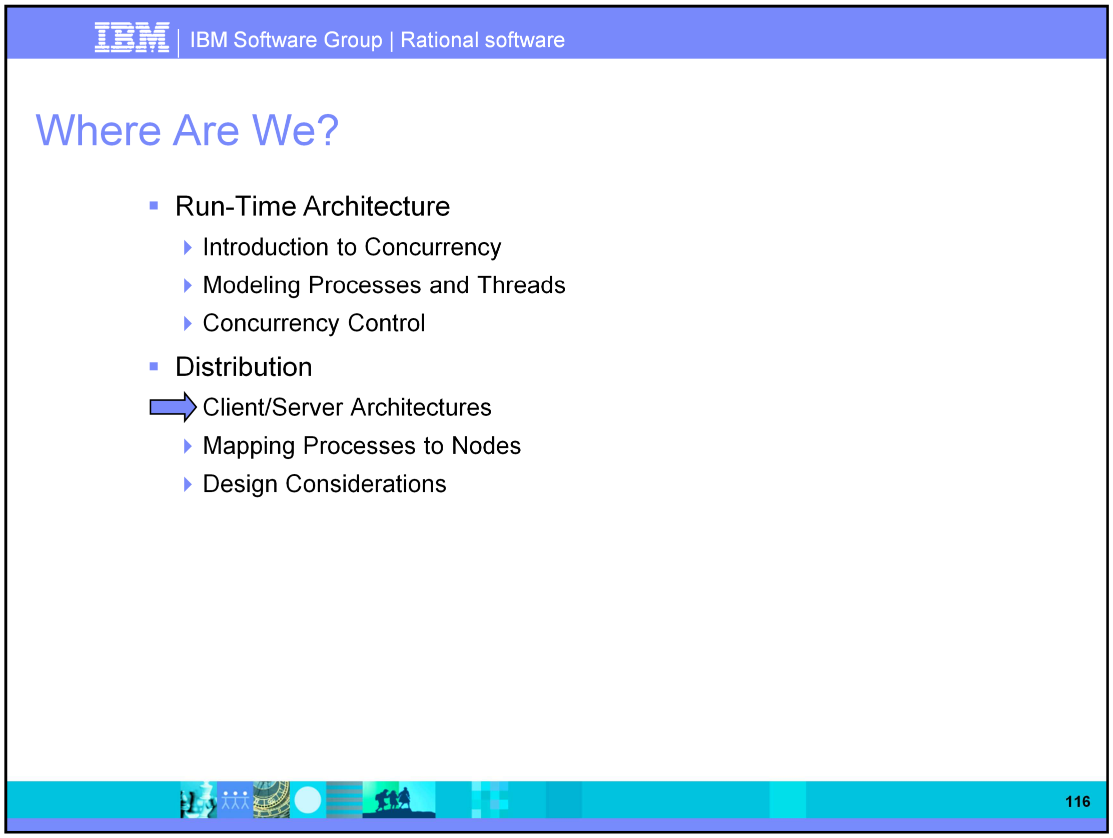
- Source: Martin Fowler, Optimistic Offline Lock in Patterns of Enterprise Application Architecture, Addison Wesley, 2003

Business Transaction Boundary

System Transaction Boundary


S1:Session
:Database
S2:Session
1: getData ( )
2: getData ( )
S1 and S2 read the same record identifed by its version number 17.
3: getData ( )
4: getData ( )
5: ediData ( )
5.1: ediData ( )
6: updateData ( )
7: updateData ( )
8: updateData ( )
9: updateData ( )
The record is successfully updated. Its version number is 18.
A failure code is returned because of a mismatch between the version numbers for the record (17 and 18).
10: rollback ( )

115



IBM Software Group | Rational software

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

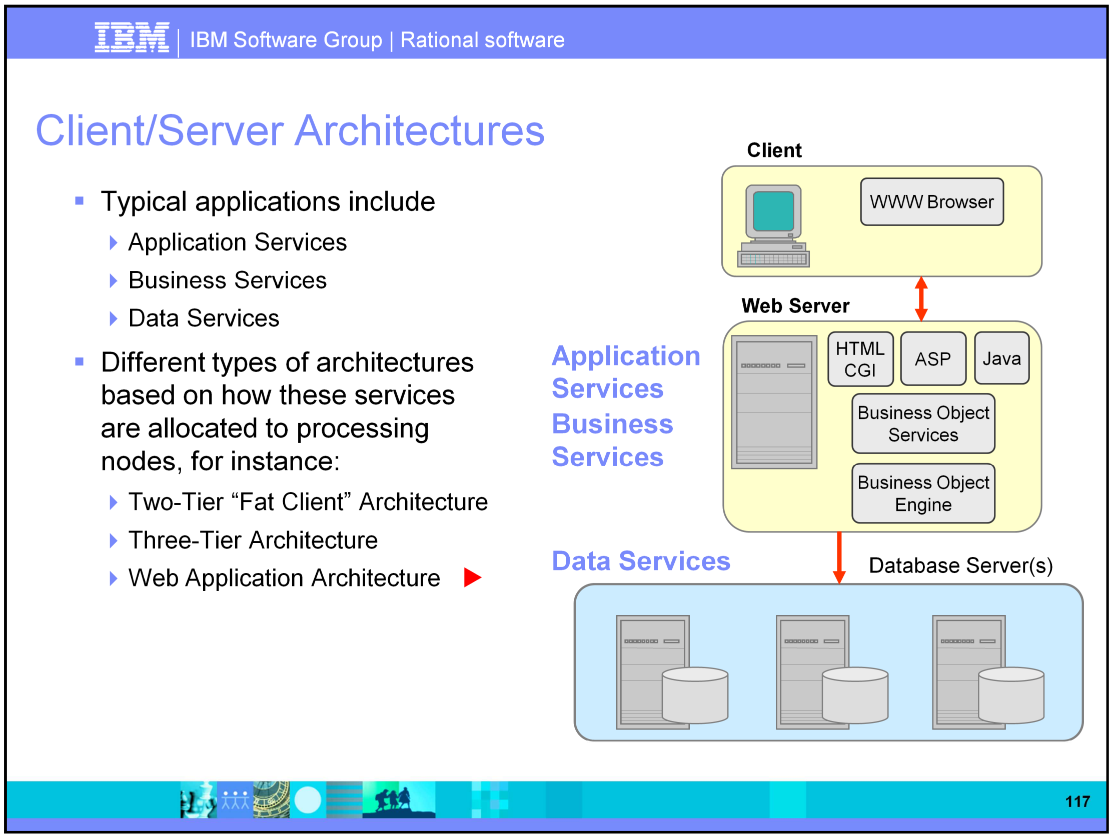
116

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.

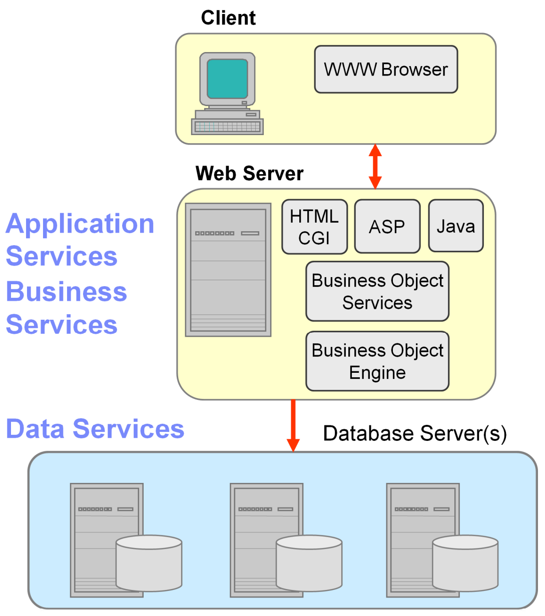
The server usually provides services to several clients simultaneously. These services are typically database, security, or print services. A system can consist of several different types of servers. For example: *database servers*, handling database machines such as Oracle, DB2; *print servers*, handling 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).



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



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.

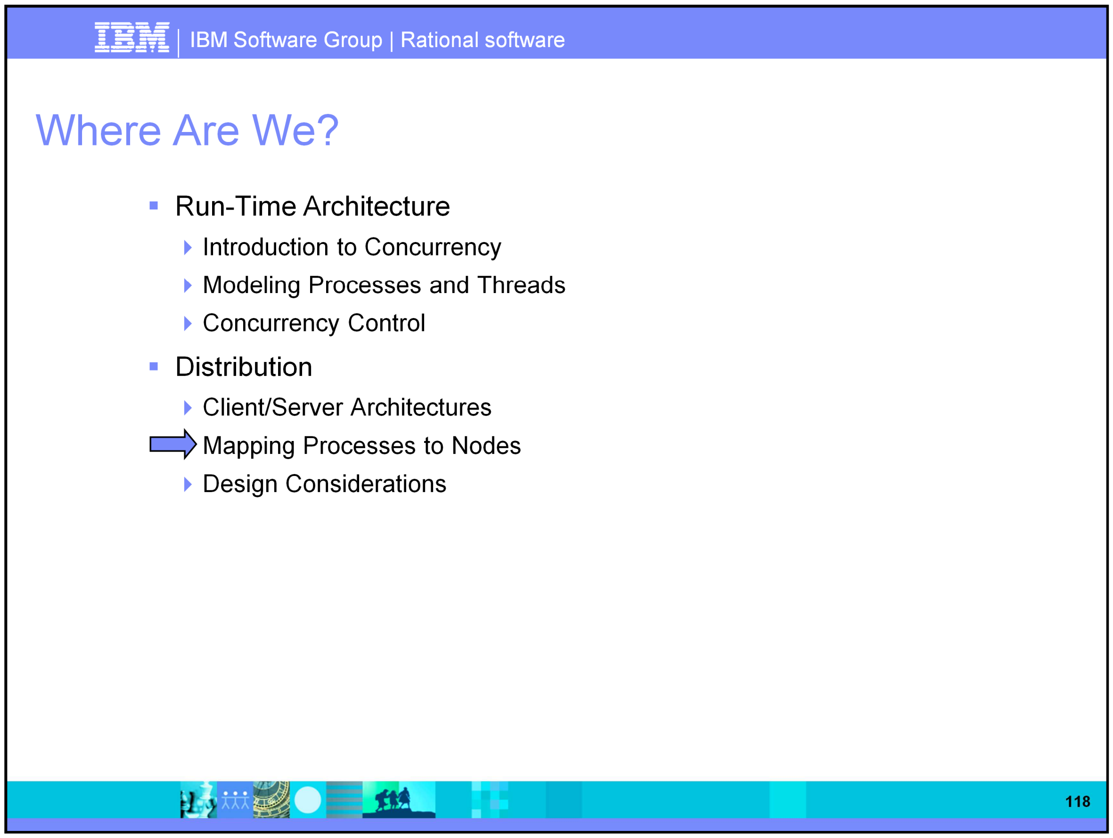
Data services tend to be implemented using database server technology, which normally executes on one or more high-performance, high-bandwidth nodes that serve hundreds or thousands of users, connected over a network.

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.

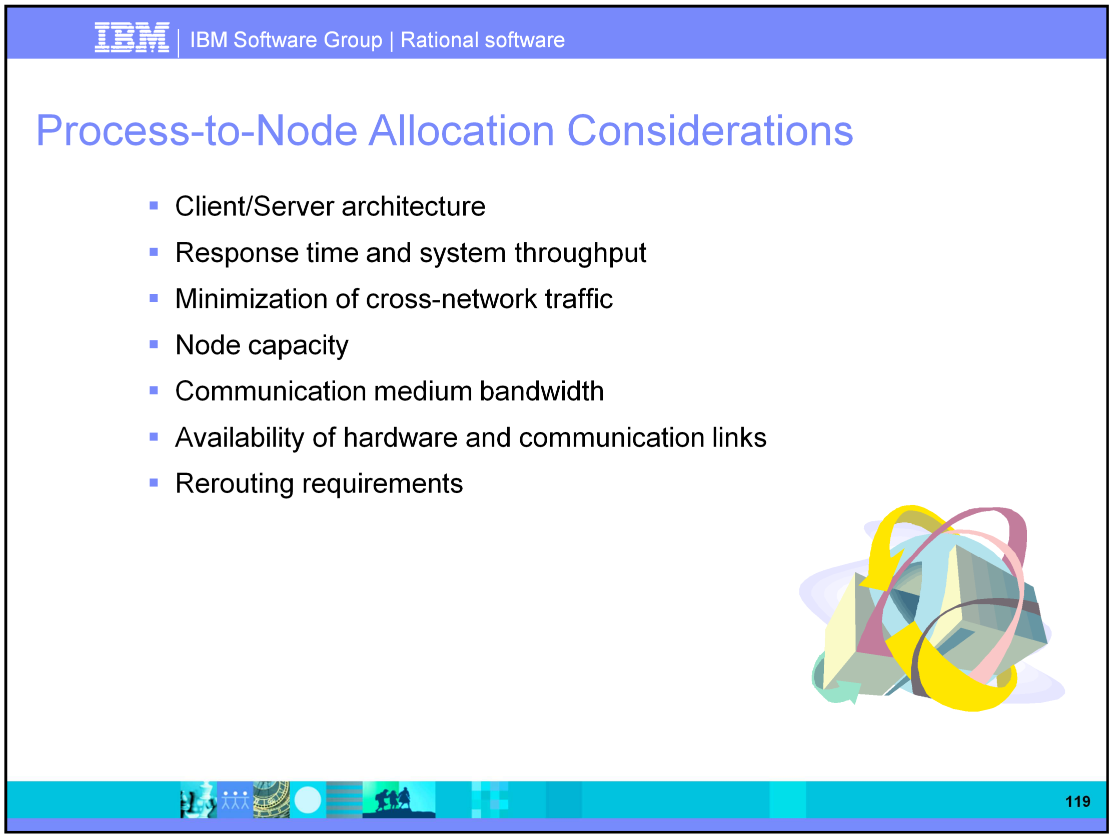
At the other end of the spectrum from the fat client is the typical **Web Application** (which might be characterized as fat server or “anorexic client”). Since the client is simply a Web browser running a set of HTML pages and Java applets, Java Beans, or ActiveX components, there is very little application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

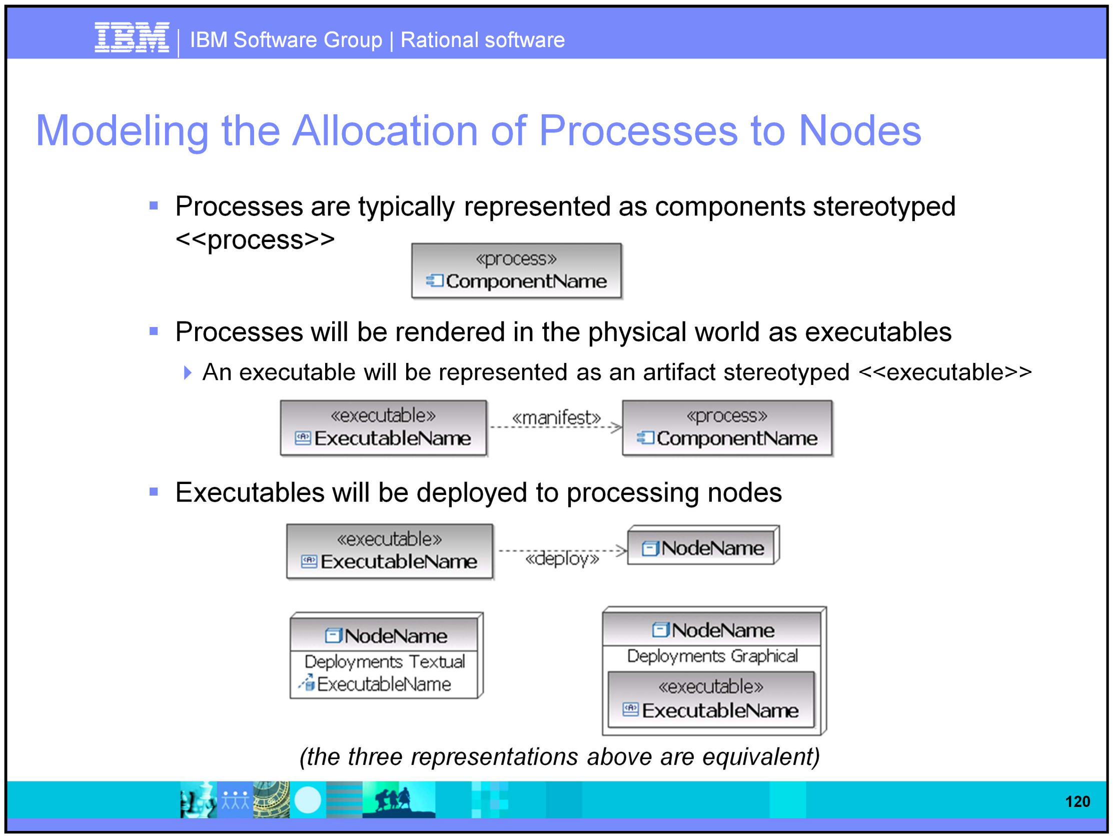
Processes must be assigned to a hardware device for execution in order to distribute the workload of the system.

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

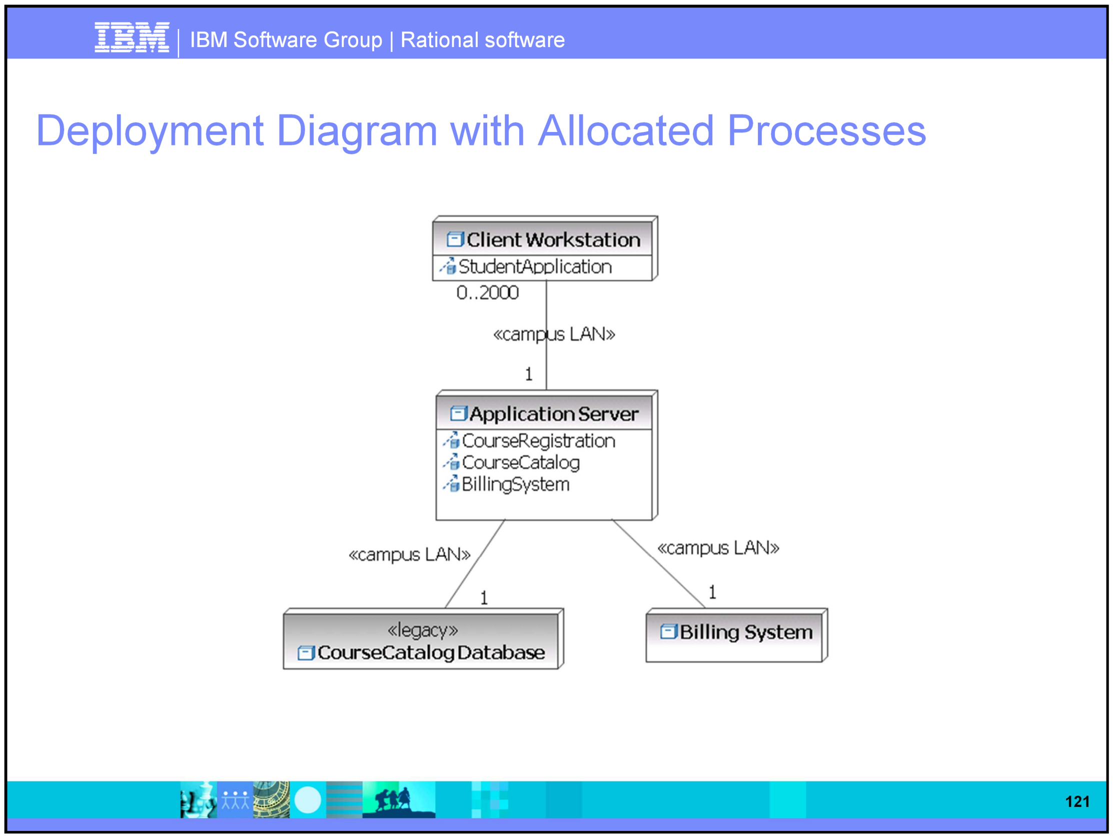
(the three representations above are equivalent)

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.

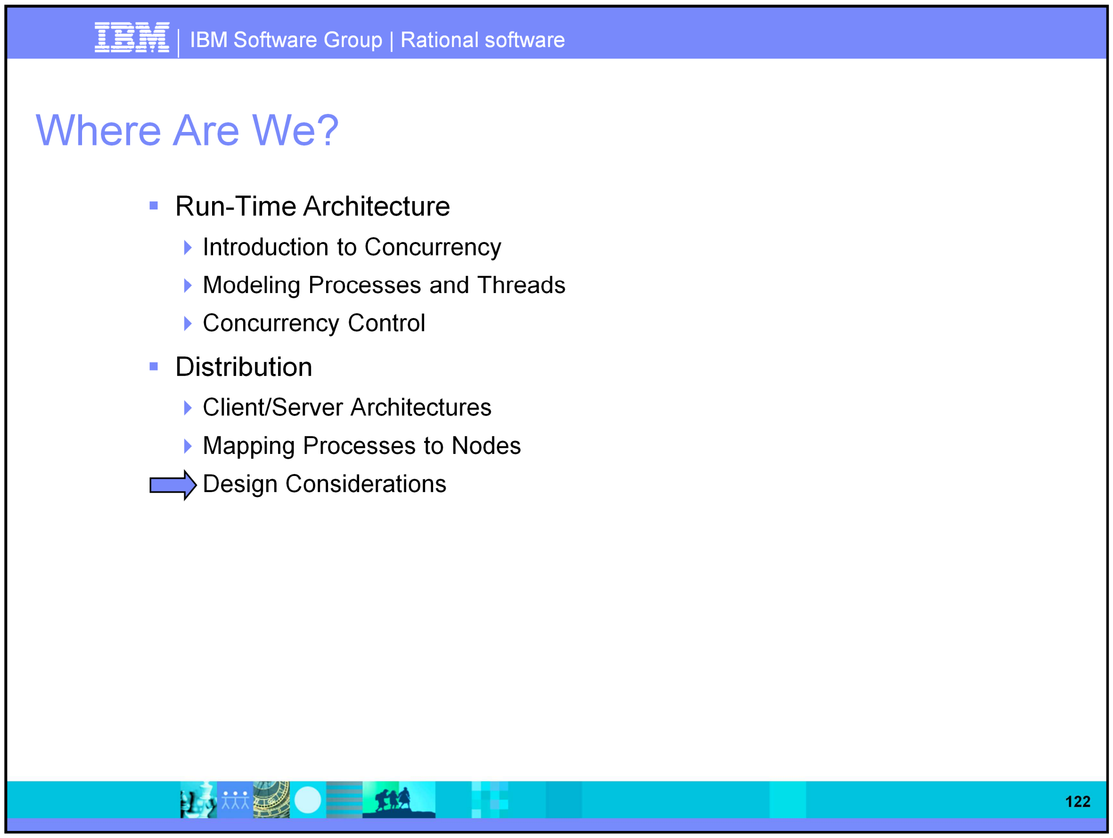


IBM Software Group | Rational software

Deployment Diagram with Allocated Processes

121

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.

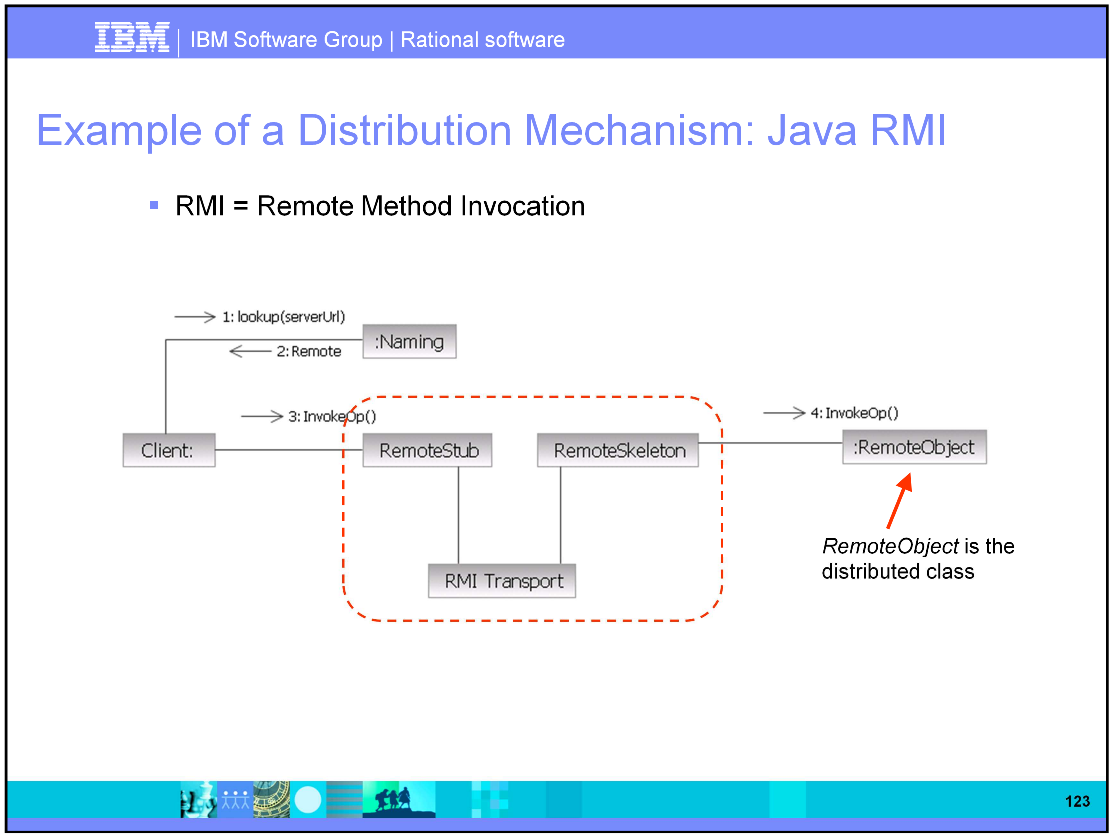


IBM Software Group | Rational software

Where Are We?

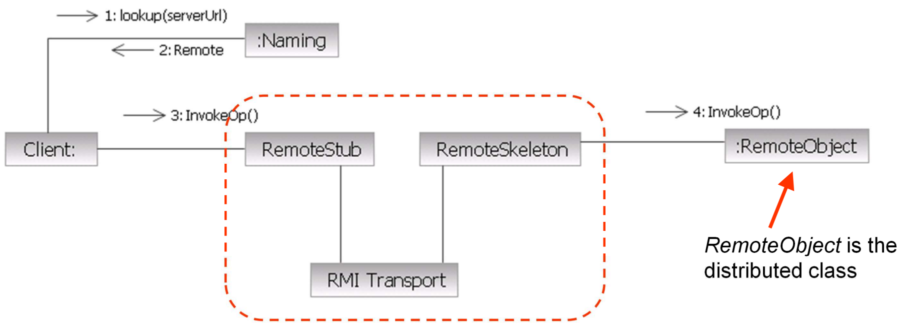
- Run-Time Architecture
  - Introduction to Concurrency
  - Modeling Processes and Threads
  - Concurrency Control
- Distribution
  - Client/Server Architectures
  - Mapping Processes to Nodes
  - Design Considerations

122



Example of a Distribution Mechanism: Java RMI

- RMI = Remote Method Invocation



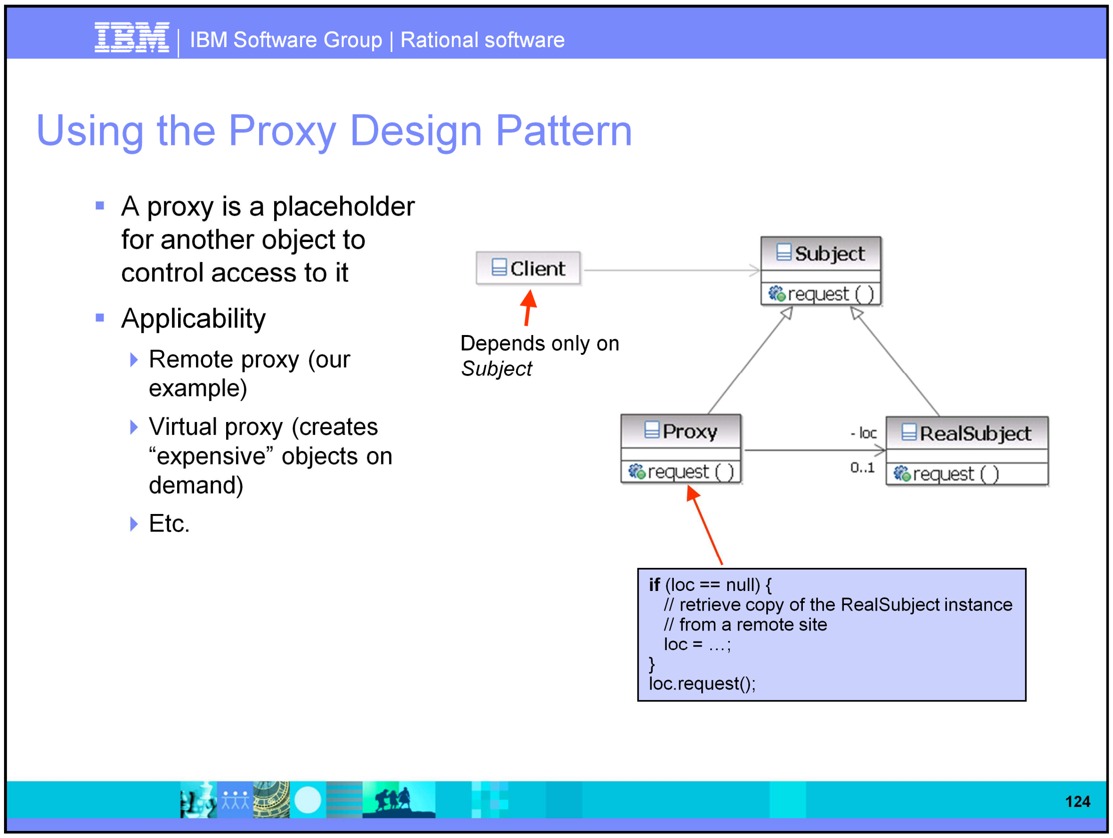
Remote Method Invocation (RMI) is a Java-specific mechanism that allows client objects to invoke operations on server objects as if they were local. The only catch is that, with basic RMI, you must know where the server object resides.

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.

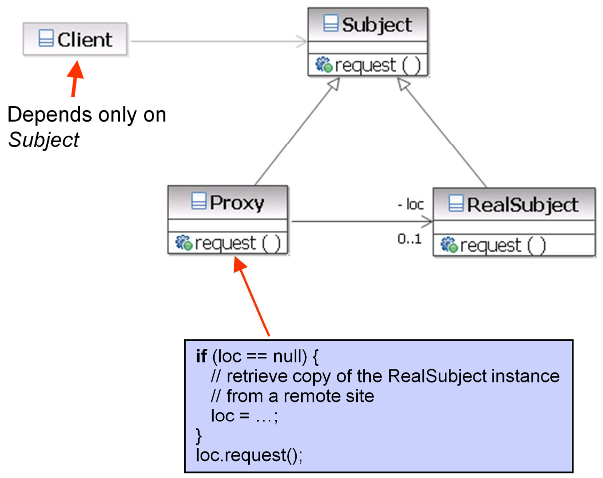
RemoteStub and *RemoteSkeleton* are automatically generated. To get them, you run the compiled distributed class through the *rmic* compiler to generate the stubs and skeletons. You then must add the code to look up the object on the server. The lookup returns a reference to the auto-generated *RemoteStub*.

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.

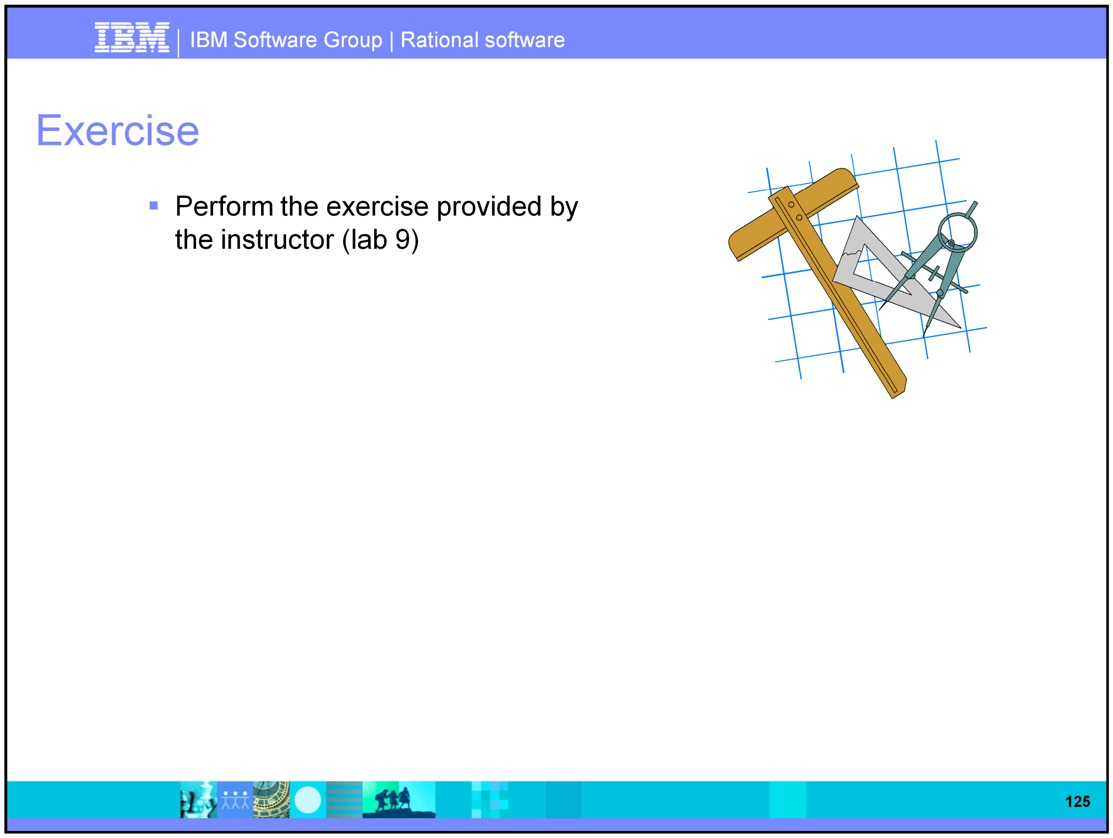


Using the Proxy Design Pattern

- 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.



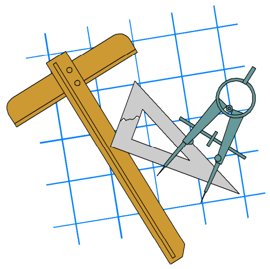
```
if (loc == null) {
   // retrieve copy of the RealSubject instance
   // from a remote site
   loc = …;
}
loc.request();
```

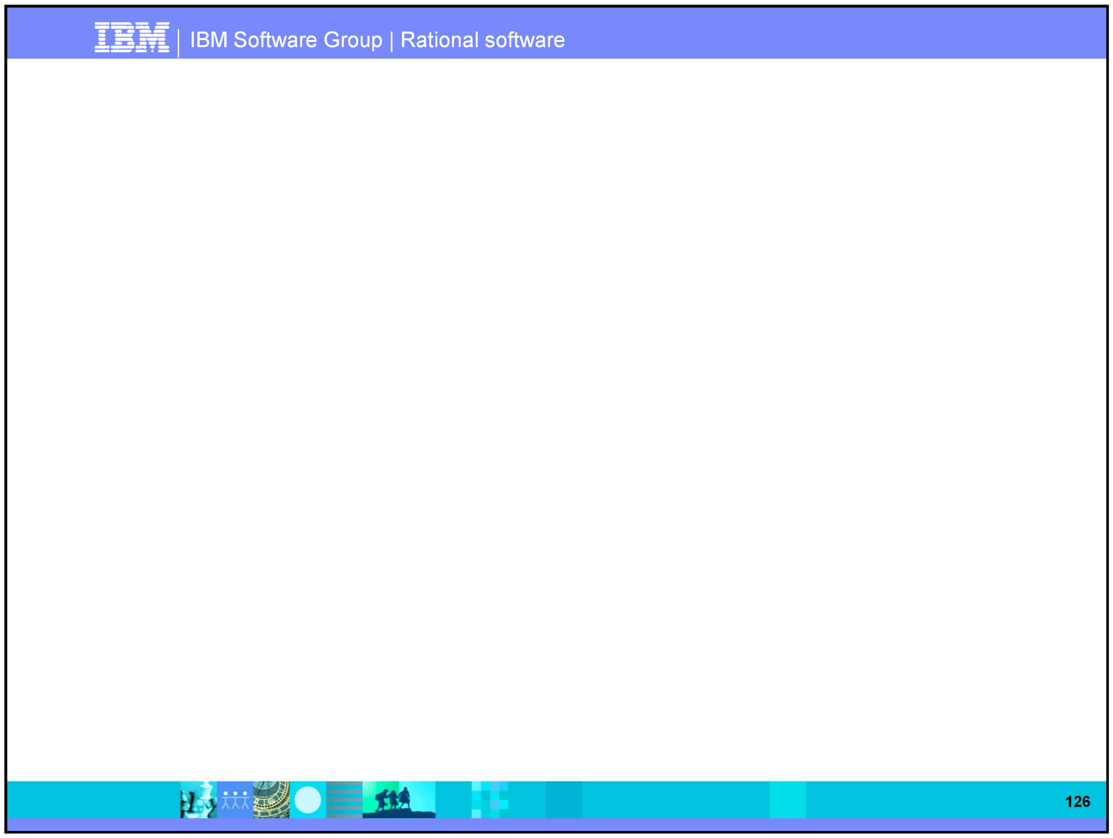


IBM Software Group | Rational software

Exercise

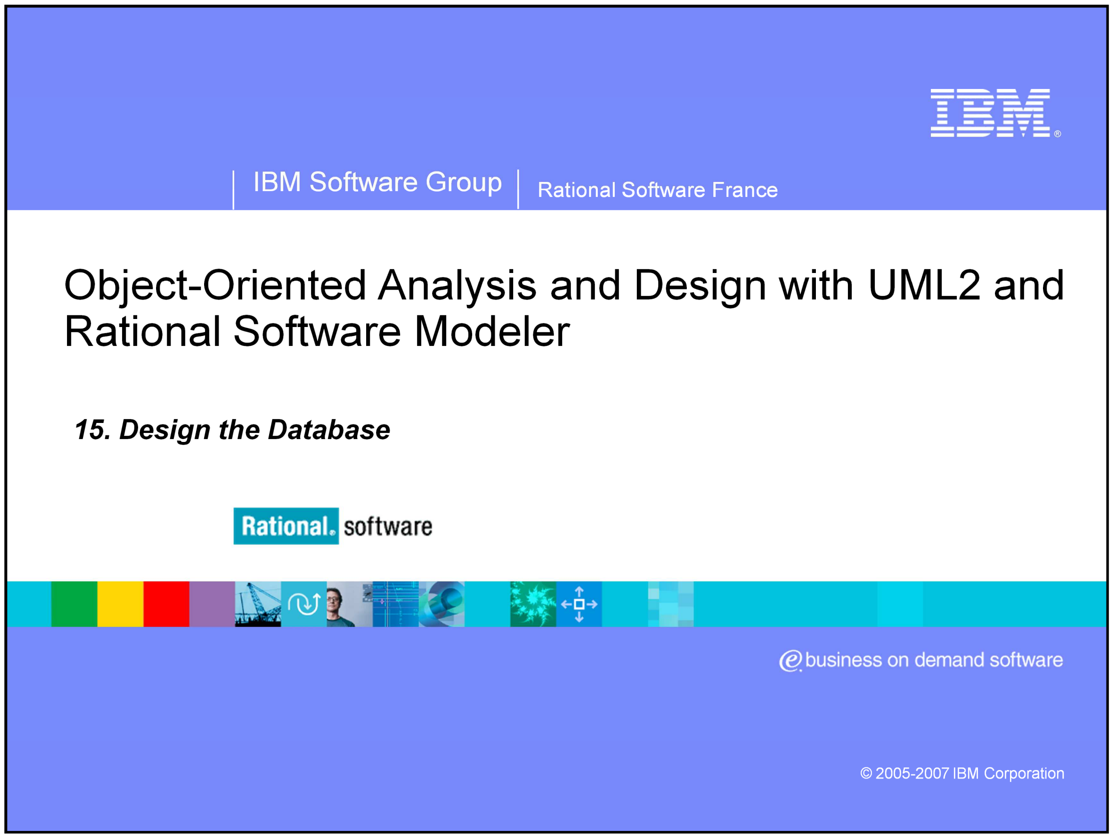
- Perform the exercise provided by the instructor (lab 9)





IBM Software Group | Rational software

126



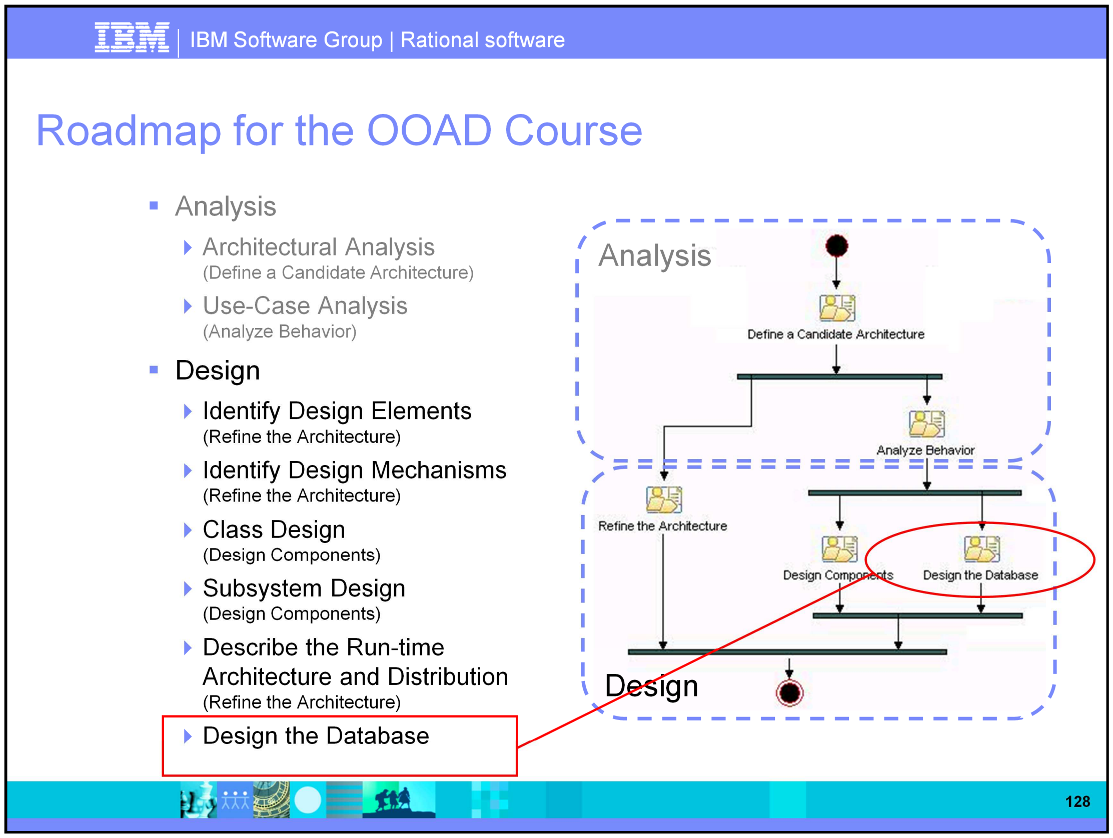
IBM Software Group | Rational Software France

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

15. Design the Database

Rational. software

business on demand software

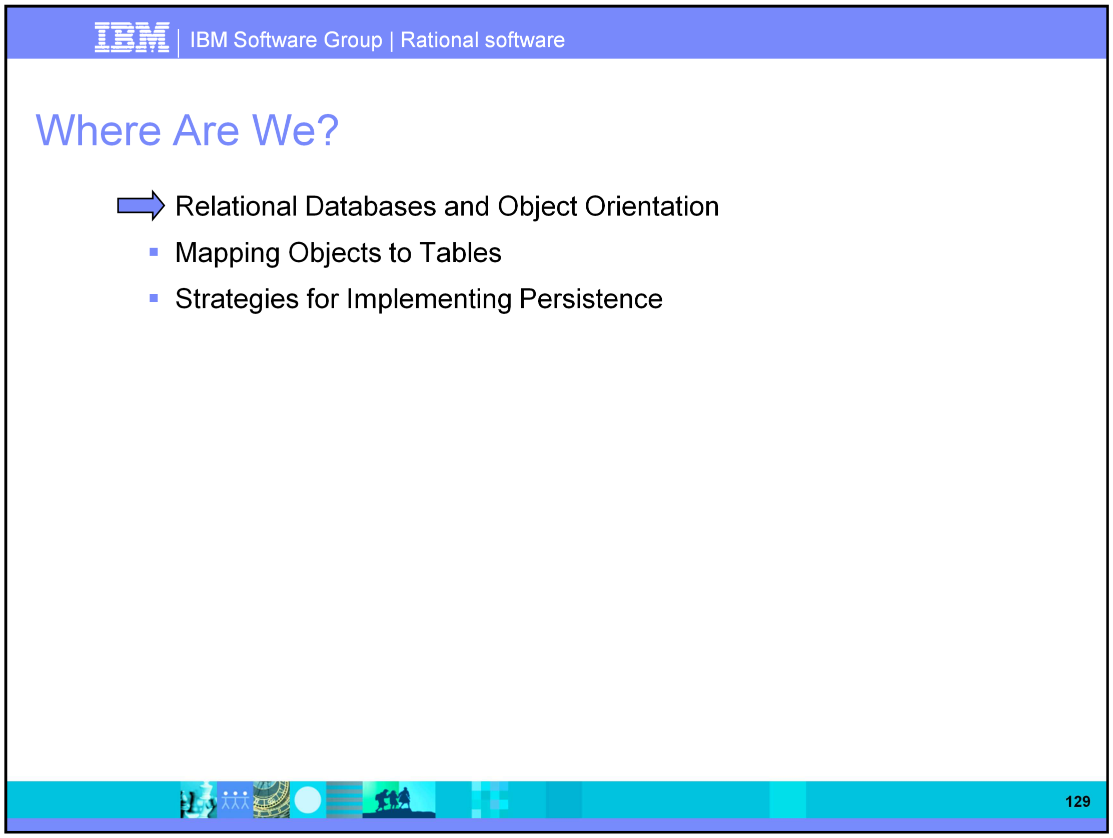


IBM | IBM Software Group | Rational software

Roadmap for the OOAD Course

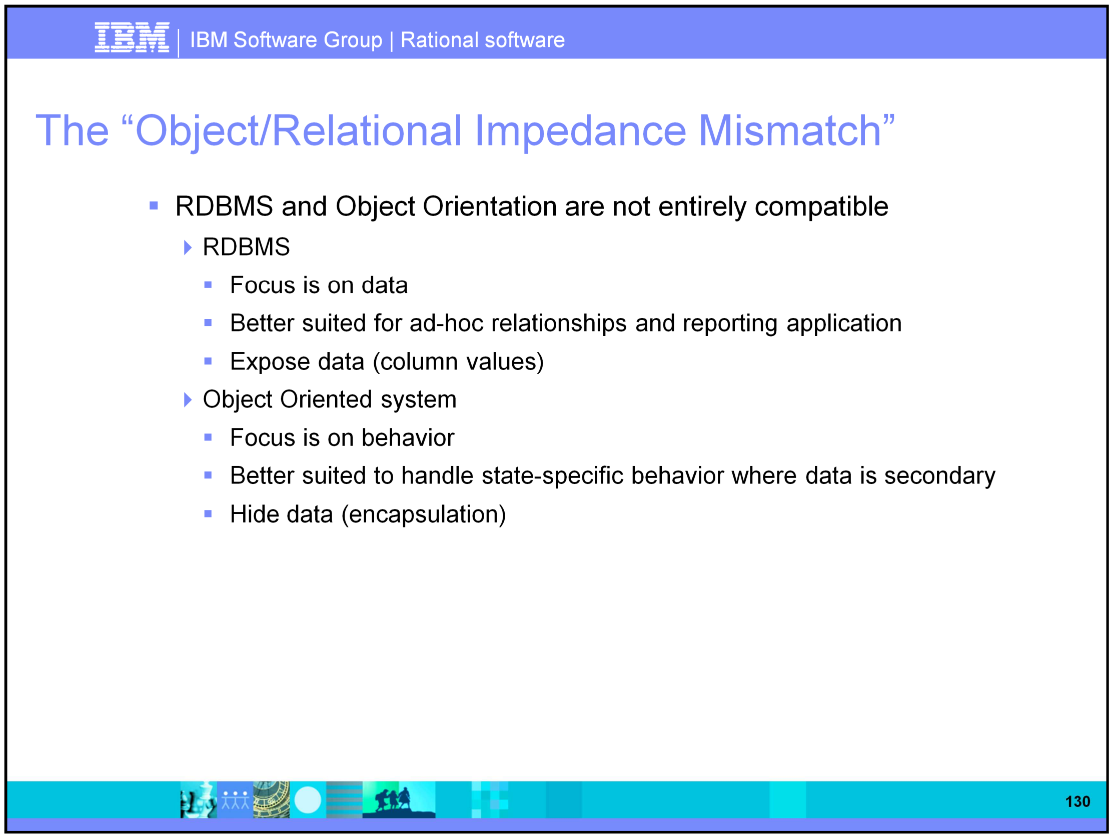
- Analysis
  - Architectural Analysis
    (Define a Candidate Architecture)
  - Use-Case Analysis
    (Analyze Behavior)
- Design
  - Identify Design Elements
    (Refine the Architecture)
  - Identify Design Mechanisms
    (Refine the Architecture)
  - Class Design
    (Design Components)
  - Subsystem 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 Software Group | Rational software

The "Object/Relational Impedance Mismatch"

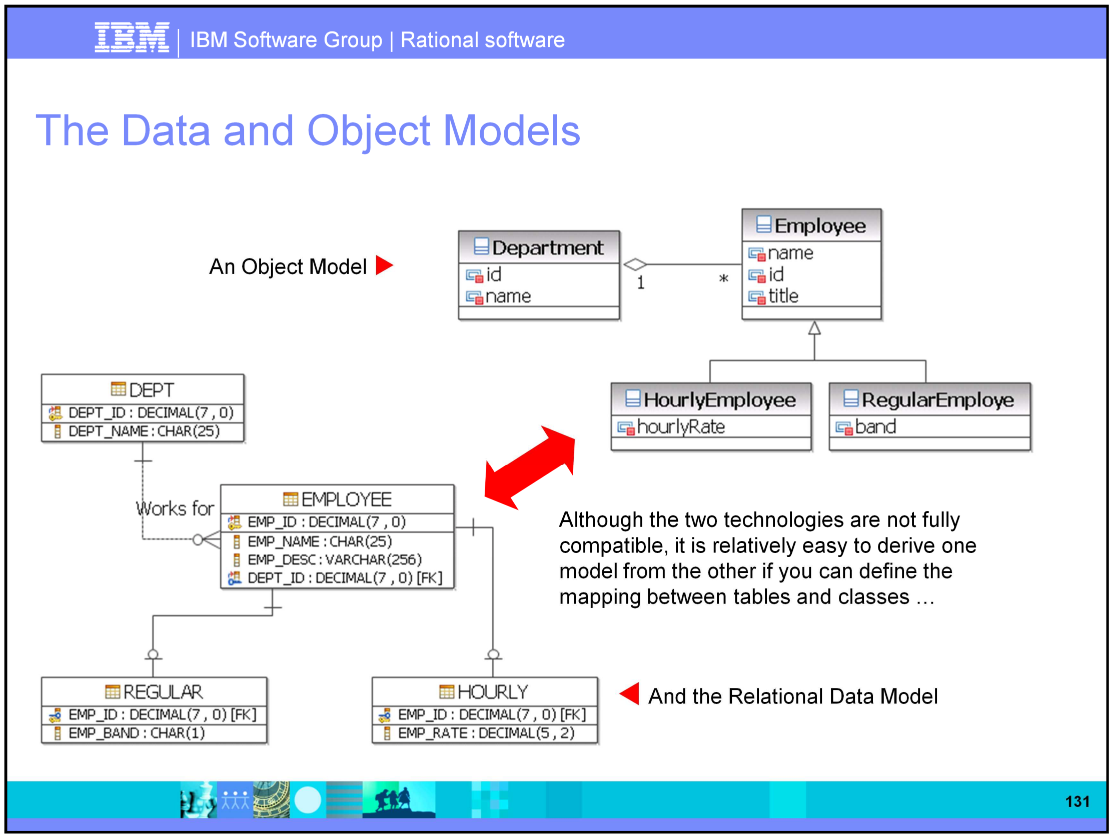
- RDBMS and Object Orientation are not entirely compatible
  - RDBMS
    - Focus is on data
    - Better suited for ad-hoc relationships and reporting application
    - Expose data (column values)
  - Object Oriented system
    - Focus is on behavior
    - Better suited to handle state-specific behavior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

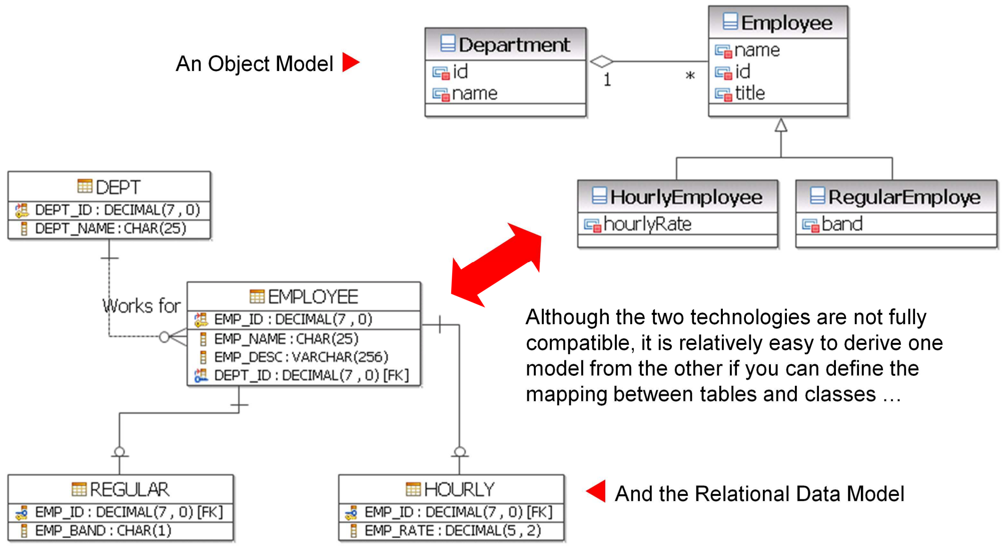
The real fact of the matter is that a lot of information is stored in relational databases, and if object-oriented applications want access to that data, they need to be able to read and write to an RDBMS. In addition, object-oriented systems often need to share data with non-object-oriented systems. 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 Software Group | Rational software

The Data and Object Models

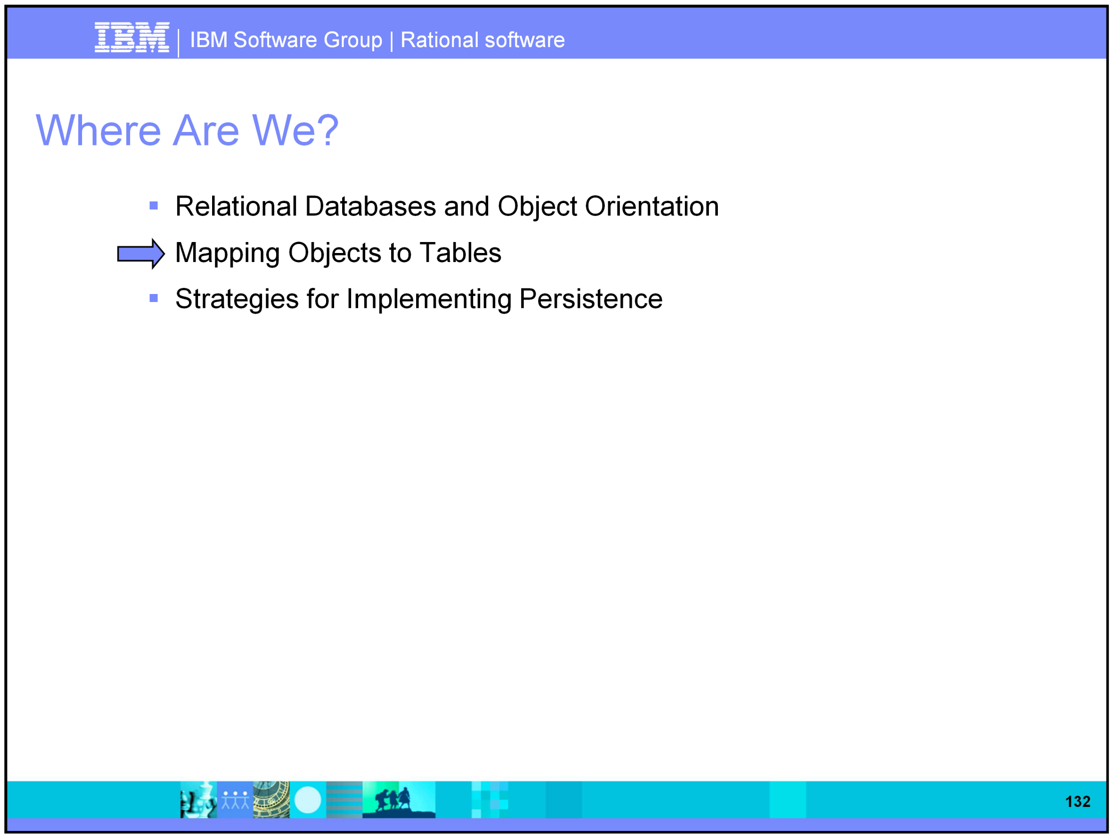


An Object Model ▶

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

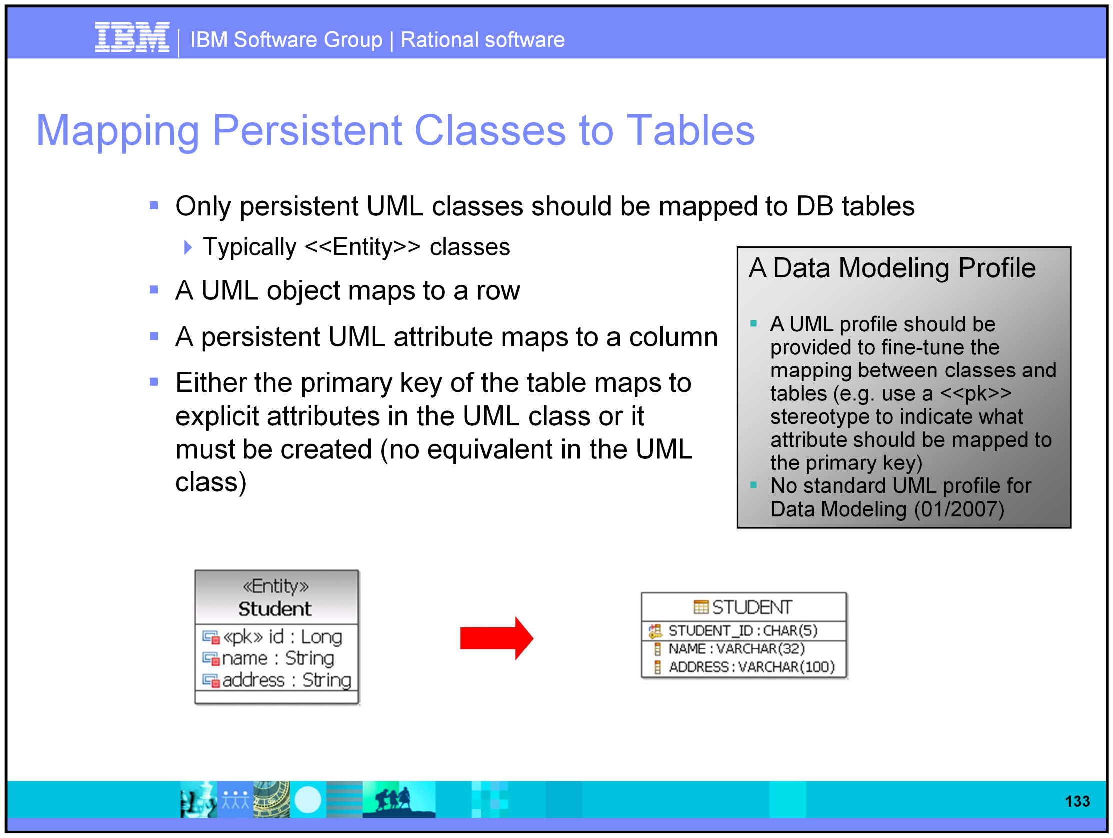


IBM | IBM Software Group | Rational software

Where Are We?

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

132

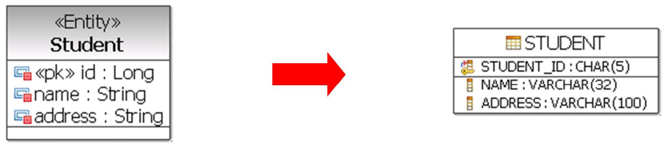

IBM | IBM Software Group | Rational software
133

Mapping Persistent Classes to Tables

- Only persistent UML classes should be mapped to DB tables
  - Typically <<Entity>> classes
- A UML object maps to a row
- A persistent UML attribute maps to a column
- Either the primary key of the table maps to explicit attributes in the UML class or it must be created (no equivalent in the UML class)

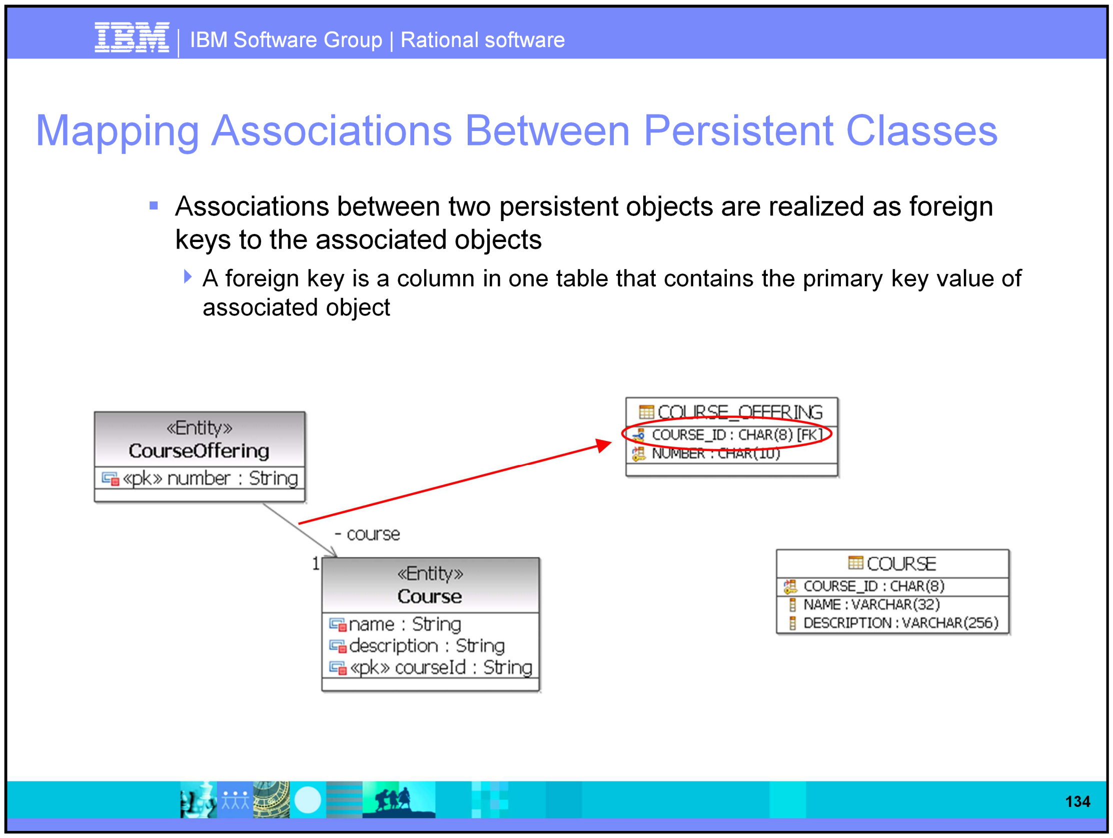
A Data Modeling Profile

- A UML profile should be provided to fine-tune the mapping between classes and tables (e.g. use a <<pk>> stereotype to indicate what attribute should be mapped to the primary key)
- No standard UML profile for Data Modeling (01/2007)



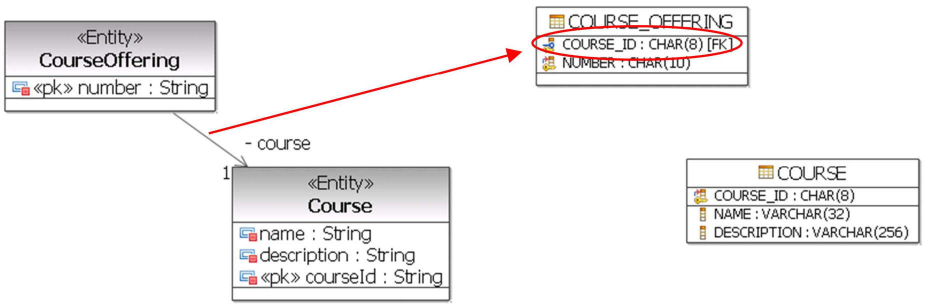
The persistent classes in the Design Model represent the information the system must store. Conceptually, these classes might resemble a relational design (for example, the classes in the Design Model might be reflected in some fashion as entities in the relational schema). As we move from elaboration 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.



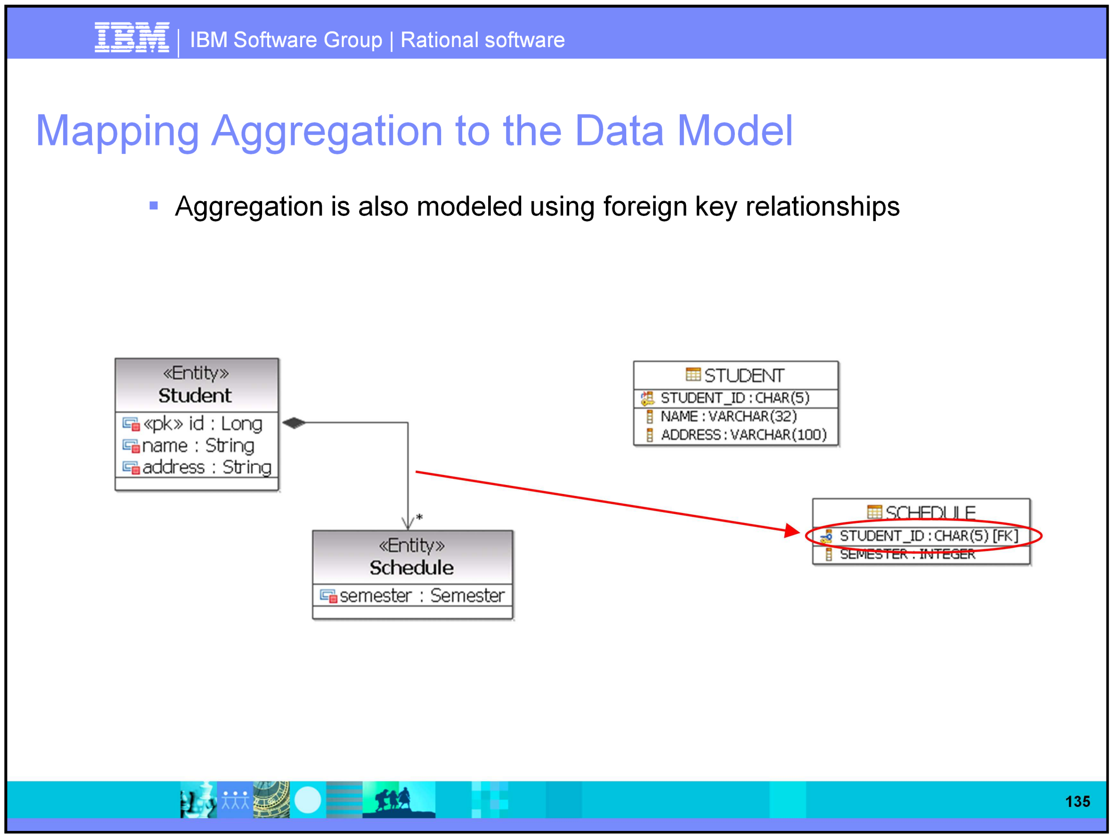
Mapping Associations Between Persistent Classes

- 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



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

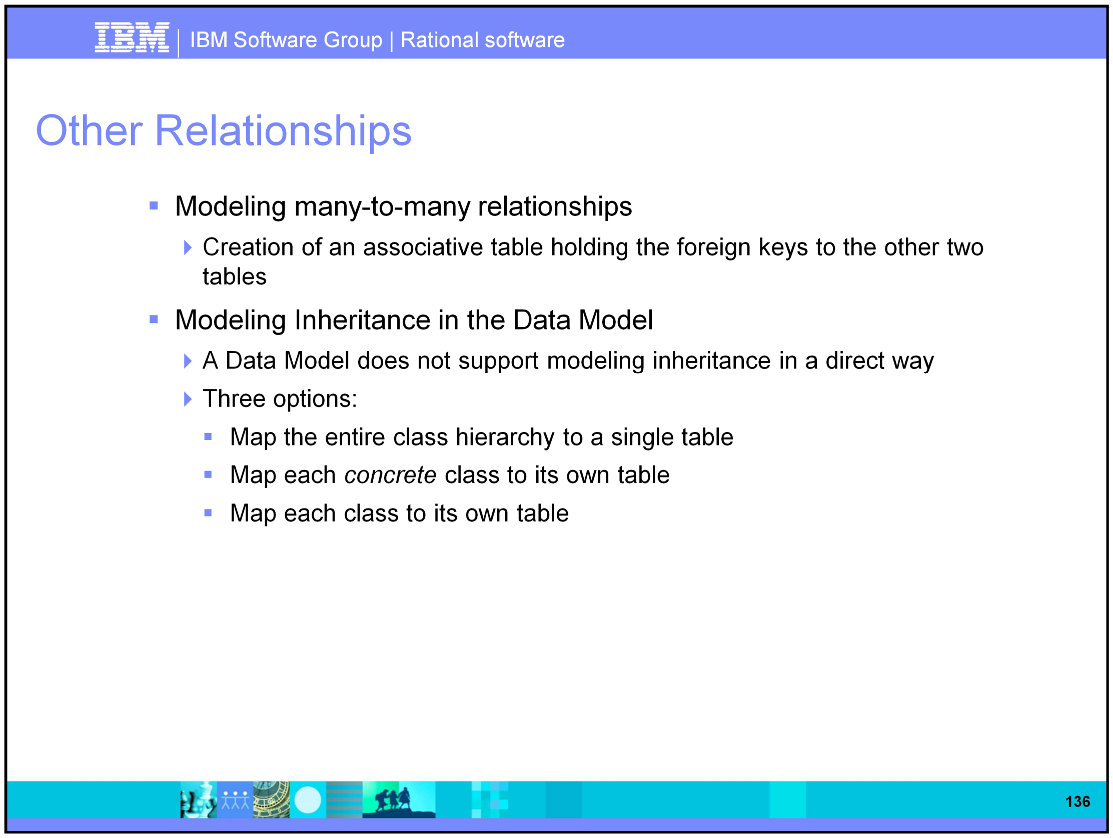
- Aggregation is also modeled using foreign key relationships

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

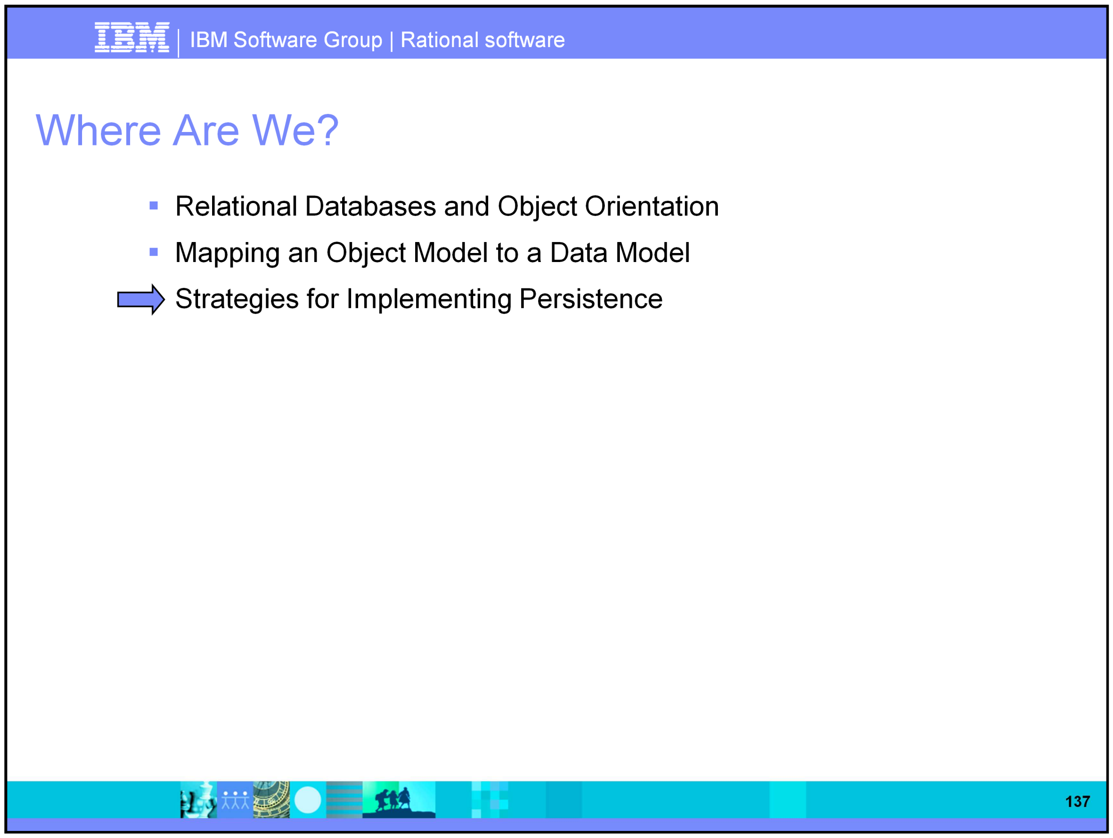
Other Relationships

- 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.

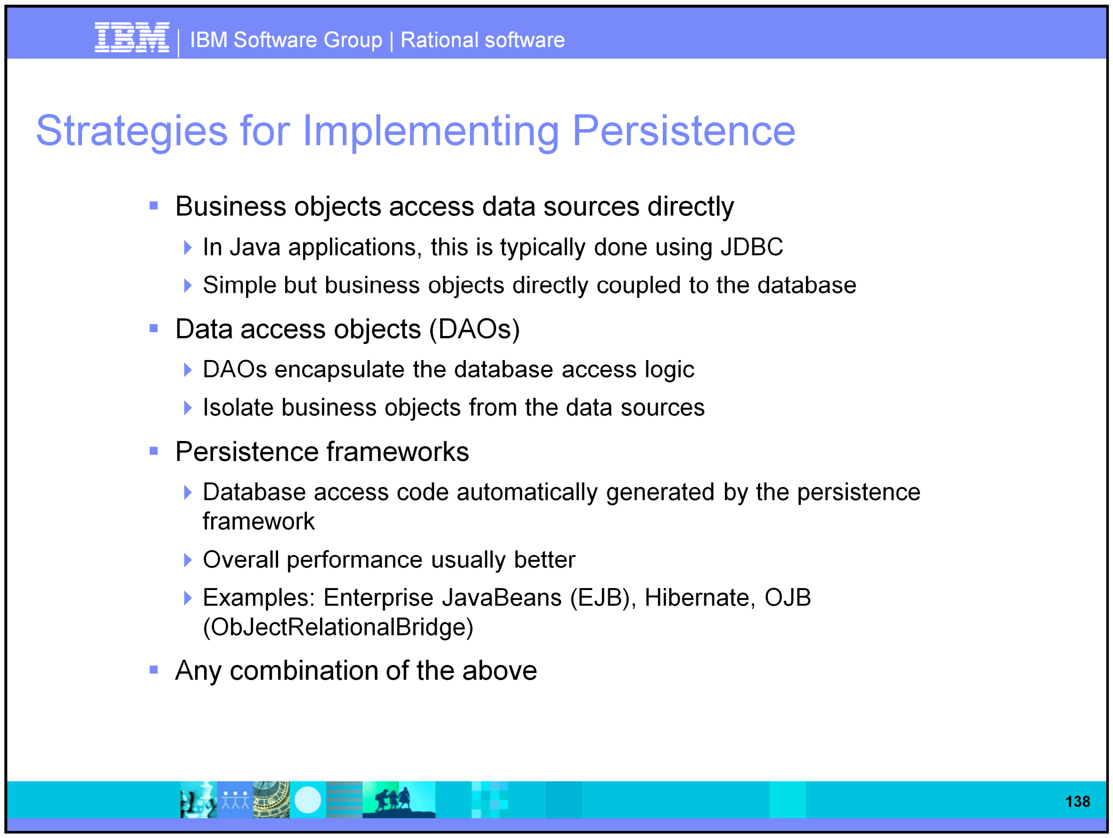


IBM Software Group | Rational software

Where Are We?

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

137

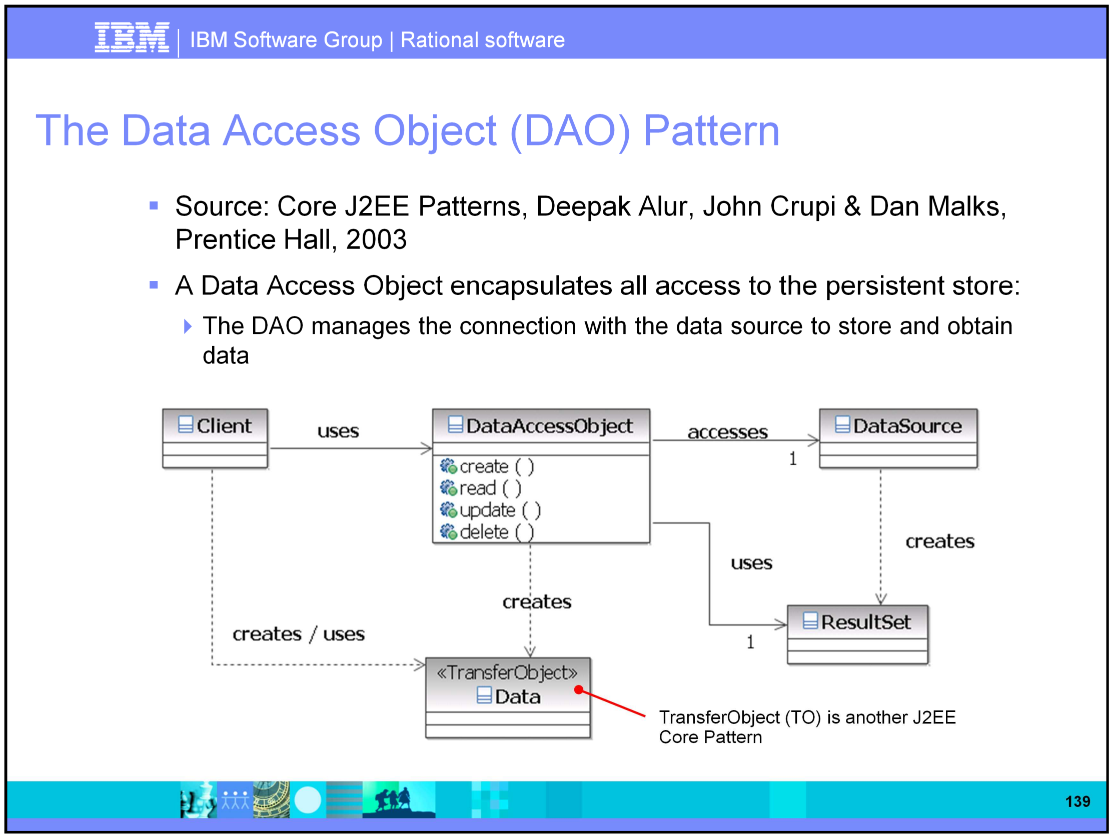


IBM Software Group | Rational software

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

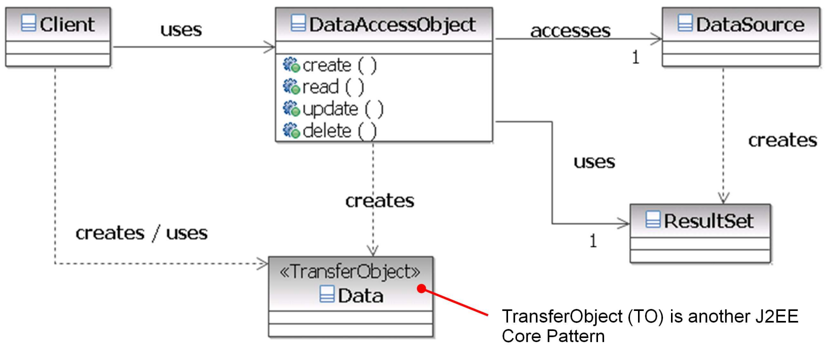
138



IBM | IBM Software Group | Rational software

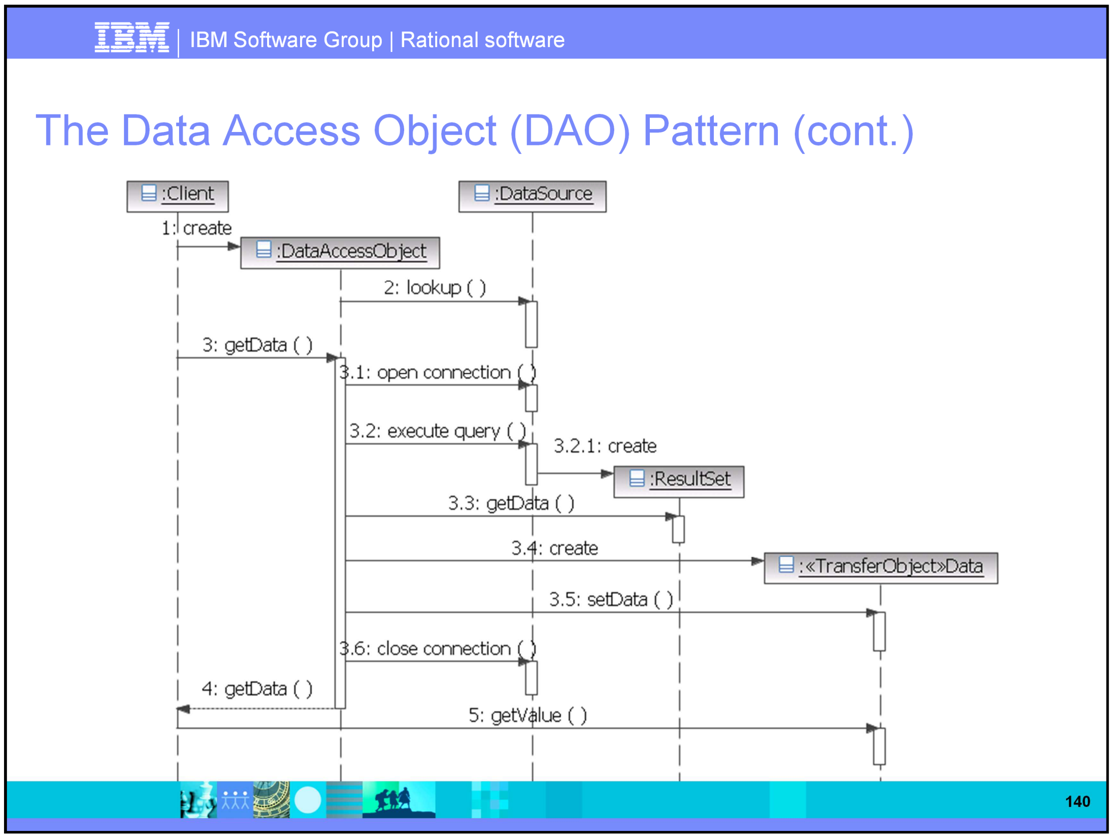
The Data Access Object (DAO) Pattern

- 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



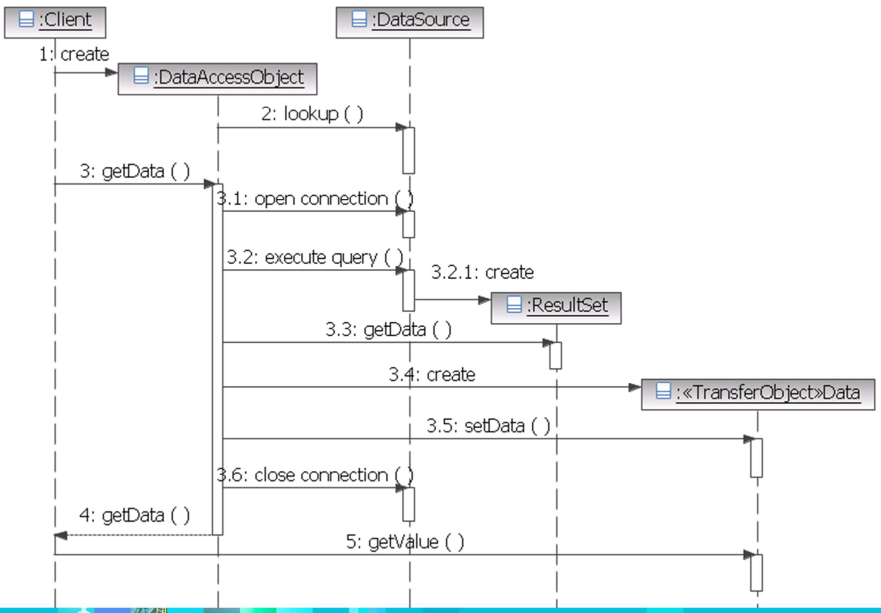
TransferObject (TO) is another J2EE Core Pattern

139

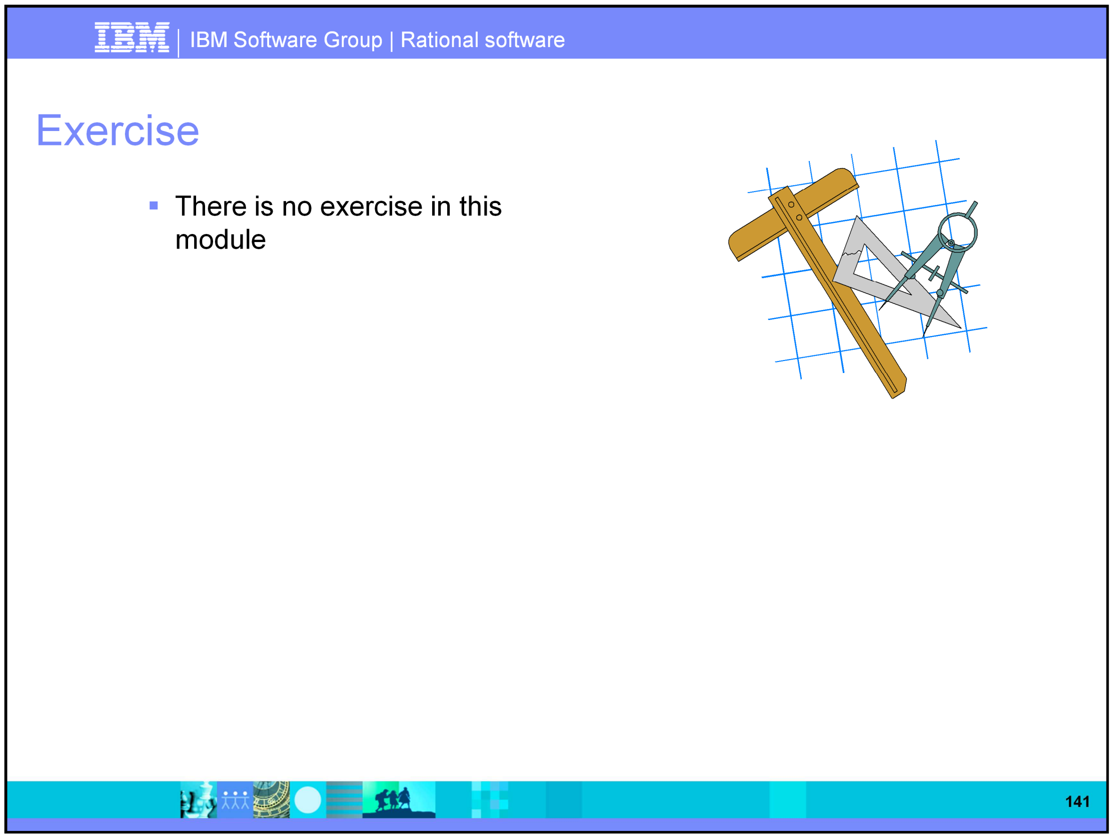


IBM Software Group | Rational software

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



140



Exercise

- There is no exercise in this module