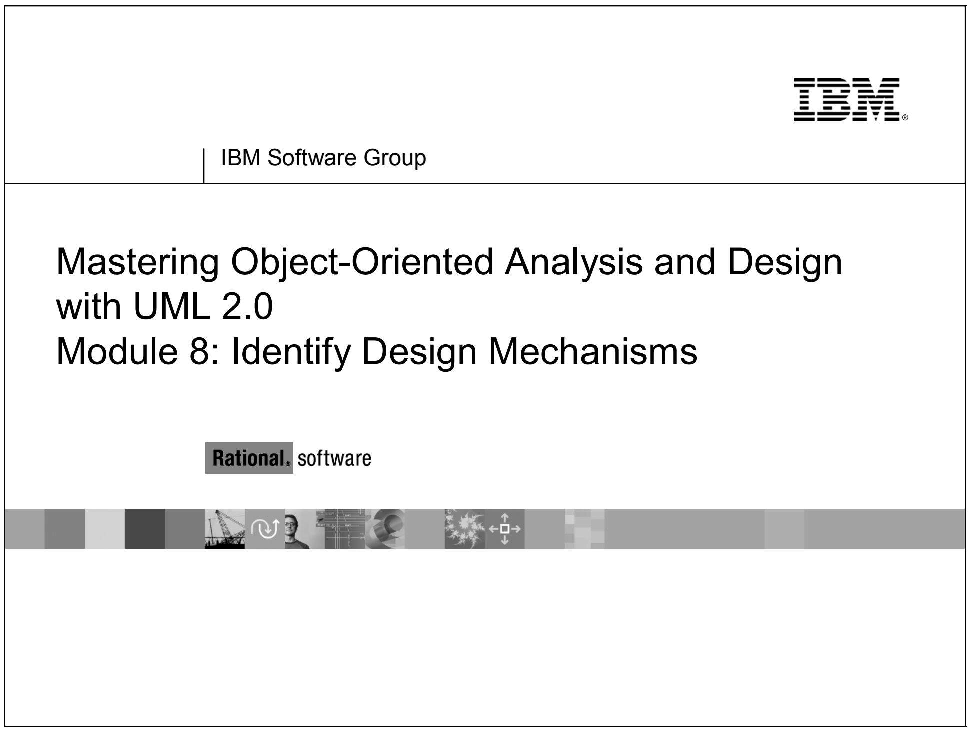
Module 8
Identify Design Mechanisms

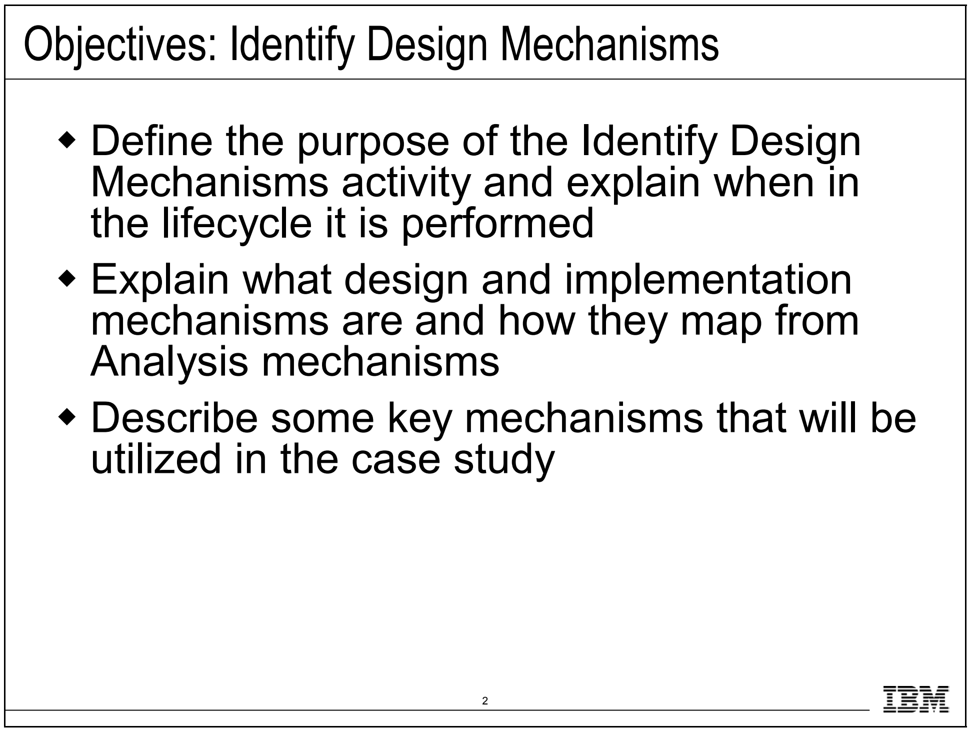

IBM Software Group

Mastering Object-Oriented Analysis and Design with UML 2.0
Module 8: Identify Design Mechanisms

Rational. software

Topics

Objectives: Identify Design Mechanisms



Objectives: Identify Design Mechanisms

- Define the purpose of the Identify Design Mechanisms activity and explain when in the lifecycle it is performed
- Explain what design and implementation mechanisms are and how they map from Analysis mechanisms
- Describe some key mechanisms that will be utilized in the case study

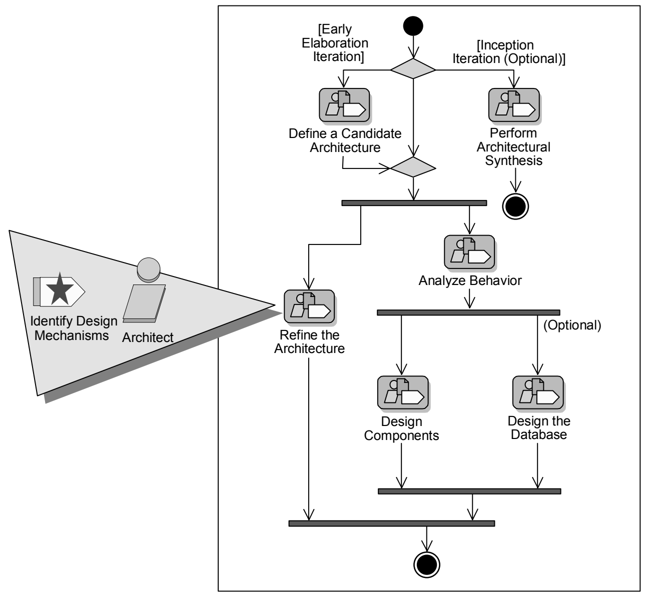
2

In this module, we will describe WHAT is performed in **Identify Design Mechanisms**, but will not describe *how* to do it. Such a discussion is the purpose of an architecture course, which this course is not.

Understanding the rationale and considerations that support the architectural decisions is needed in order to understand the architecture, which is the framework in which designs must be developed.

Identify Design Mechanisms in Context

Identify Design Mechanisms in Context

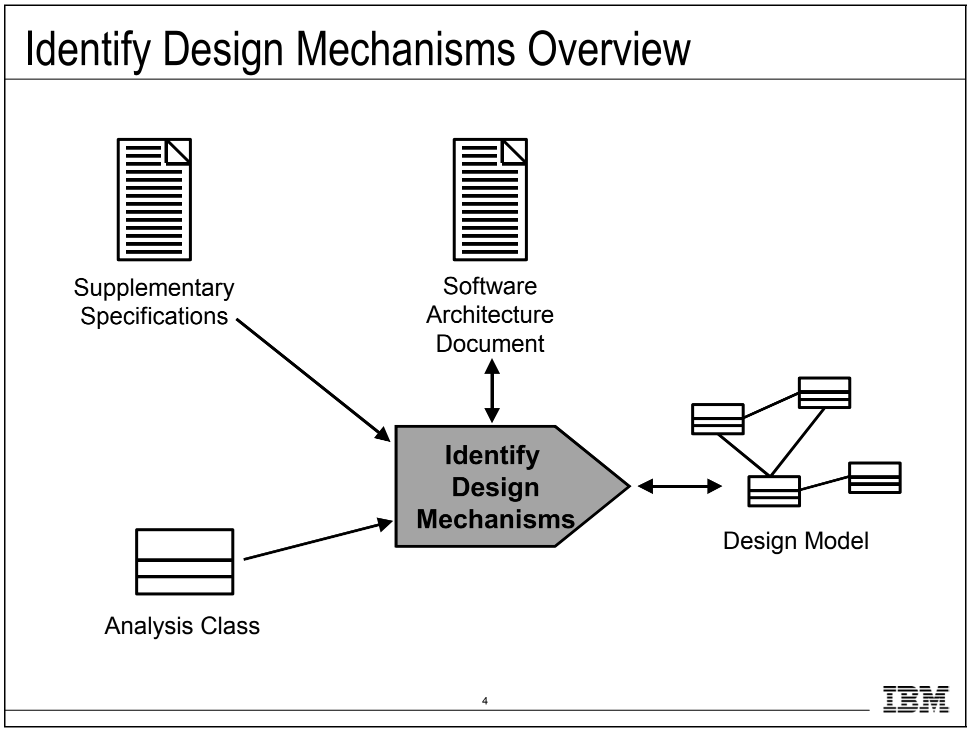

[Early Elaboration Iteration]
[Inception Iteration (Optional)]
Define a Candidate Architecture
Perform Architectural Synthesis
Analyze Behavior
Identify Design Mechanisms
Architect
Refine the Architecture
(Optional)
Design Components
Design the Database

3

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

Identify Design Mechanisms is where we refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment.

Identify Design Mechanisms Overview


Identify Design Mechanisms Overview
Supplementary Specifications
Software Architecture Document
Identify Design Mechanisms
Design Model
Analysis Class
4
IBM

Identify Design Mechanisms is performed by the architect, once per iteration.

Purpose

- To refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment.

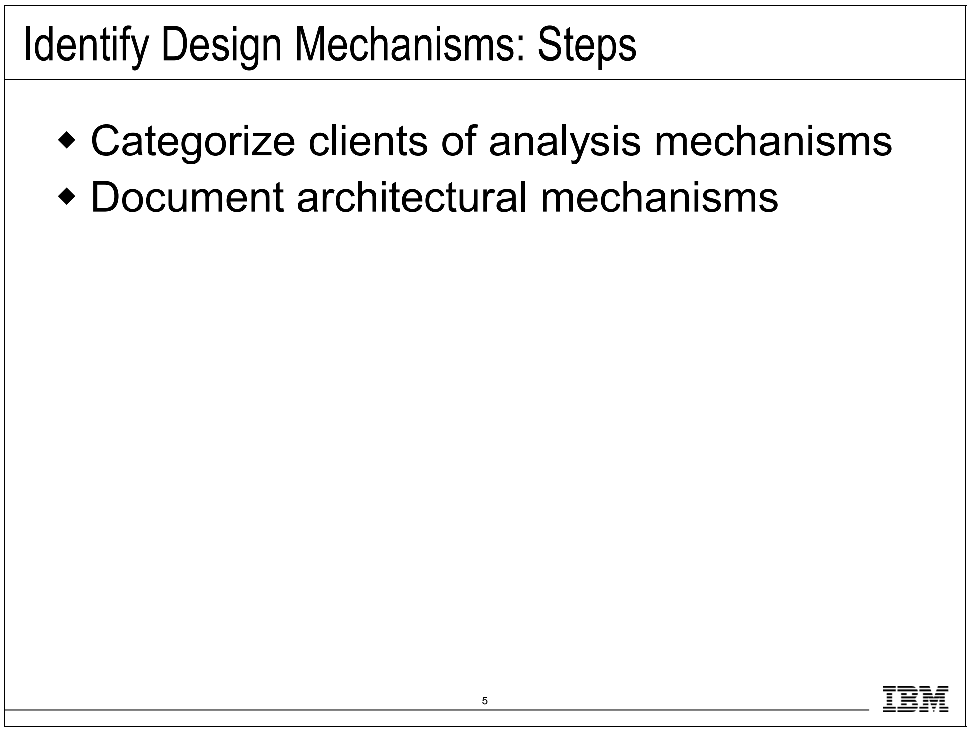
Input Artifacts

- Supplementary Specifications
- Software Architecture Document
- Analysis Class
- Design Model

Resulting Artifacts

- Design Model elements
    - Classes
    - Packages
    - Subsystems
- Software Architecture Document

Identify Design Mechanisms: Steps



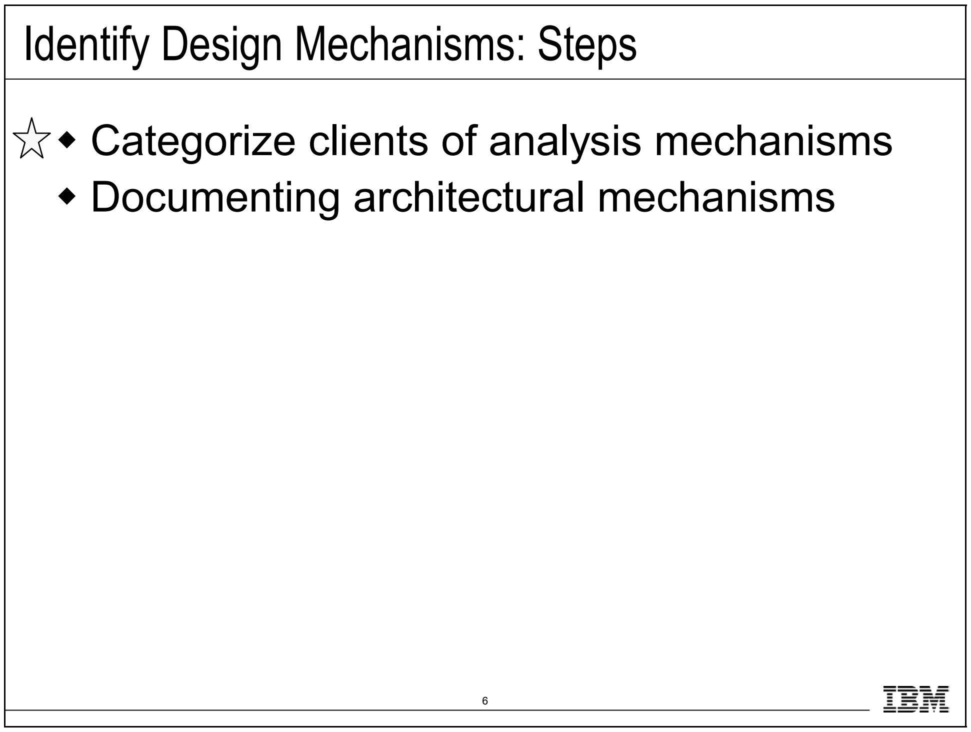
Identify Design Mechanisms: Steps

- Categorize clients of analysis mechanisms
- Document architectural mechanisms

5

The above are the topics we will be discussing within the Identify Design Elements module. Unlike the Designer activity modules, we will not be discussing each step of the activity, as the objective of this module is to understand the important architectural concepts and not to learn *how* to create an architecture.

Identify Design Mechanisms: Steps



Identify Design Mechanisms: Steps

☆ ◆ Categorize clients of analysis mechanisms

◆ Documenting architectural mechanisms

6

Analysis mechanisms provide conceptual sets of services that are used by analysis objects. They offer a convenient shorthand for fairly complex behaviors that will ultimately have to be worried about, but that are out of scope for the analysis effort.Their main purpose is to allow us to capture the requirements of these yet-to-be designed services of the system without having to be concerned about the details of the service provider itself.

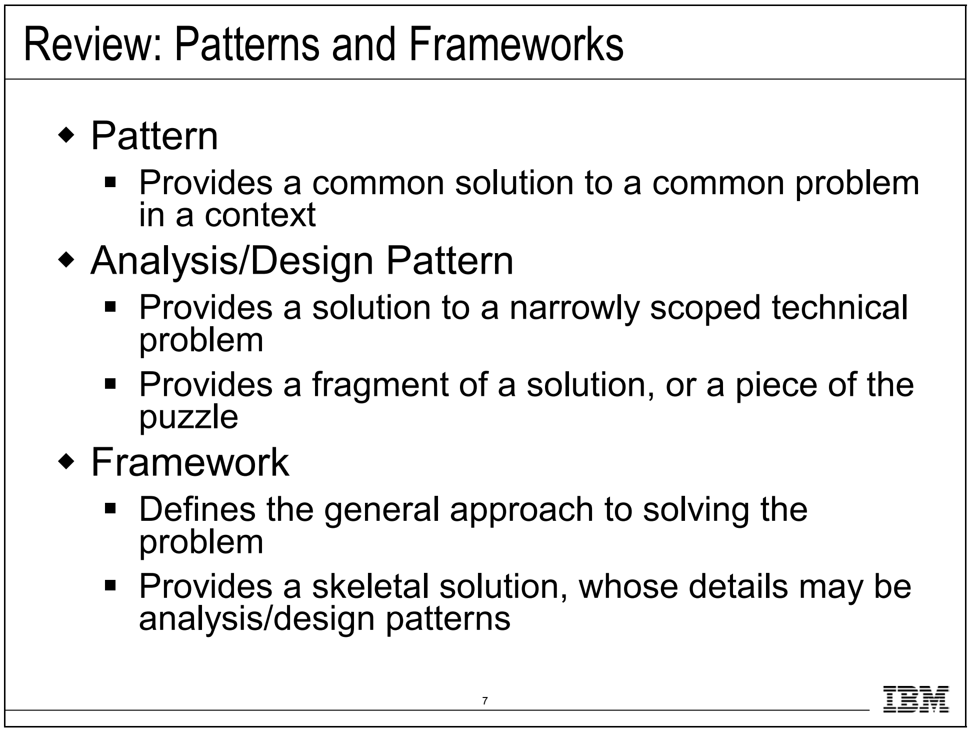
Now we must begin to refine the information gathered on the analysis mechanisms.

We discussed analysis mechanisms in Architectural Analysis. Now we will look at design and implementation mechanisms.

In this section, we will define what design and implementation mechanisms are and how they map from analysis mechanisms. We will provide abstract patterns of behavior for the mechanisms that we will utilize in the later Design activities.

The goal is *not* to teach you how to identify and design the presented mechanisms, but to be able to produce designs that incorporate those mechanisms.

Review: Patterns and Frameworks



Review: Patterns and Frameworks

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

7

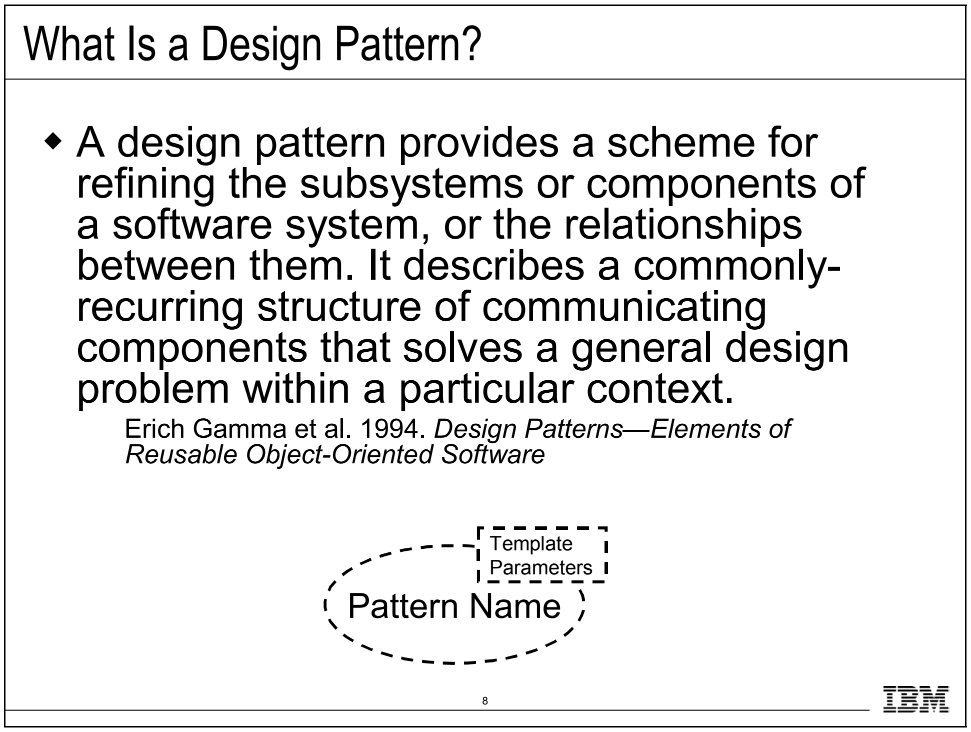
A **pattern** codifies specific knowledge collected from experience. Patterns provide examples of how good modeling solves real problems, whether you come up with the patterns yourself or you reuse someone else's. Design patterns are discussed in more detail on the next slide.

Frameworks differ from analysis and design patterns in their scale and scope. Frameworks describe a skeletal solution to a particular problem that may lack many of the details. These details can be filled in by applying various analysis and design patterns.

A framework is a micro-architecture that provides an incomplete template for applications within a specific domain. Architectural frameworks provide the context in which the components run. They provide the infrastructure (plumbing, if you will) that allows the components to co-exist and perform in predictable ways. These frameworks can provide communication mechanisms, distribution mechanisms, error processing capabilities, transaction support, and so forth.

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

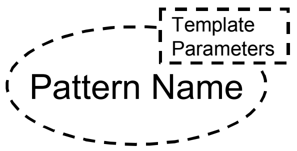
What Is a Design Pattern?



What Is a Design Pattern?

- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

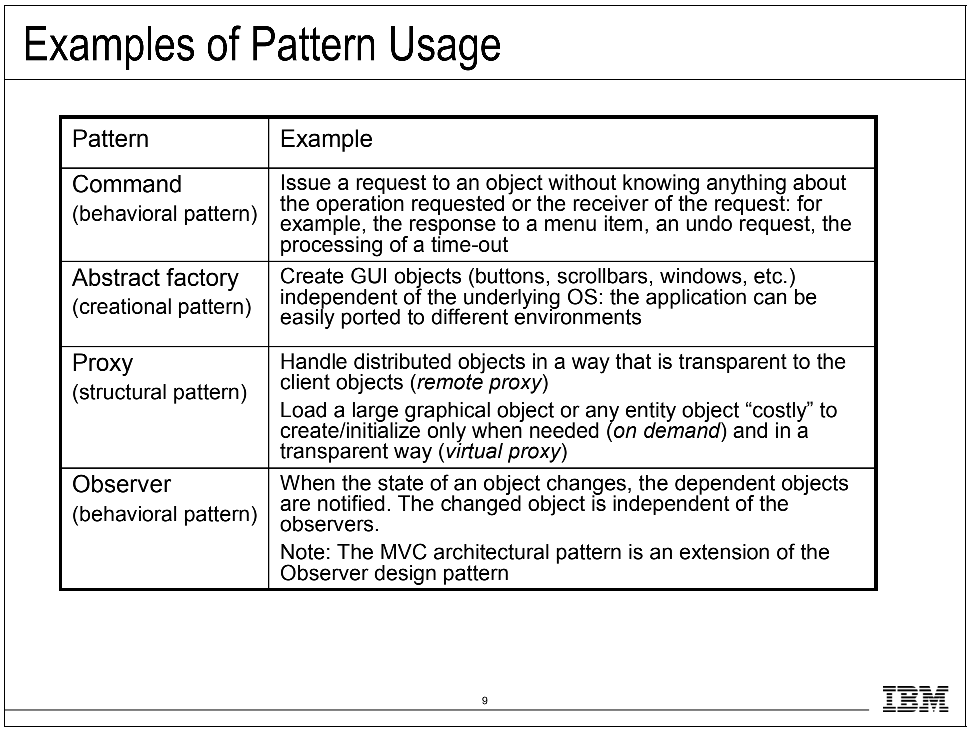
  Erich Gamma et al. 1994. *Design Patterns—Elements of Reusable Object-Oriented Software*



8

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.

Examples of Pattern Usage



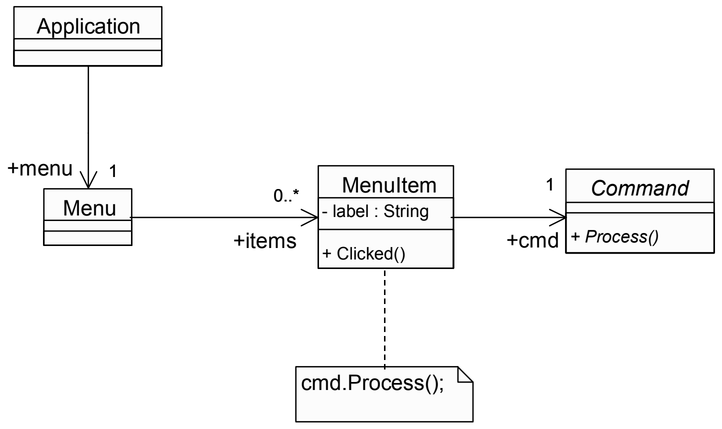
Examples of Pattern Usage

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. Note: The MVC architectural pattern is an extension of the Observer design pattern

9

Detailing the Command Pattern

Detailing the Command Pattern

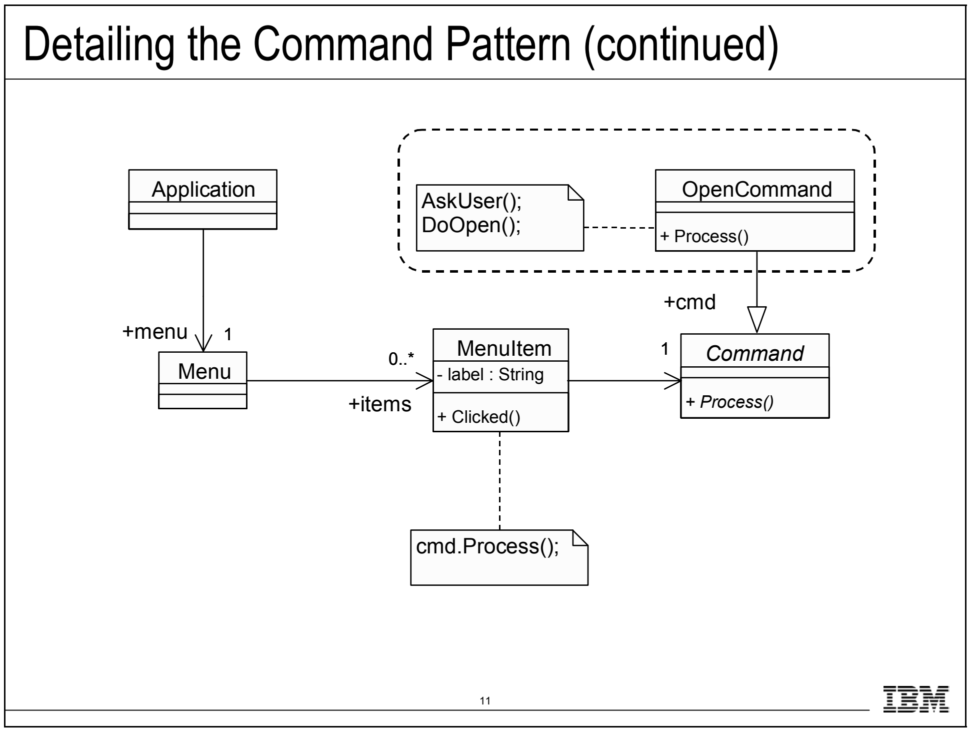


10

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).

- *Application*: it is the client class, it “simulates” the application.
- *Menu*: to simplify, we will make the assumption that our “application” has only one menu represented by an association with a multiplicity of 1 and a role name *menu*.
- *MenuItem*: a menu is composed of menu items.
- *Command*: this is an abstract class. It has one operation called *Process*. It is an abstract operation, which means it must be overridden by subclasses of *Command*.
- We will now make the assumption that the underlying windowing system requires that we define in *MenuItem* an operation called *Clicked* that will be automatically invoked when the user selects the corresponding menu during execution time. The code for *Clicked* is (using a Java-like pseudo-code):
  ```
  cmd.Process();
  ```

Detailing the Command Pattern (continued)



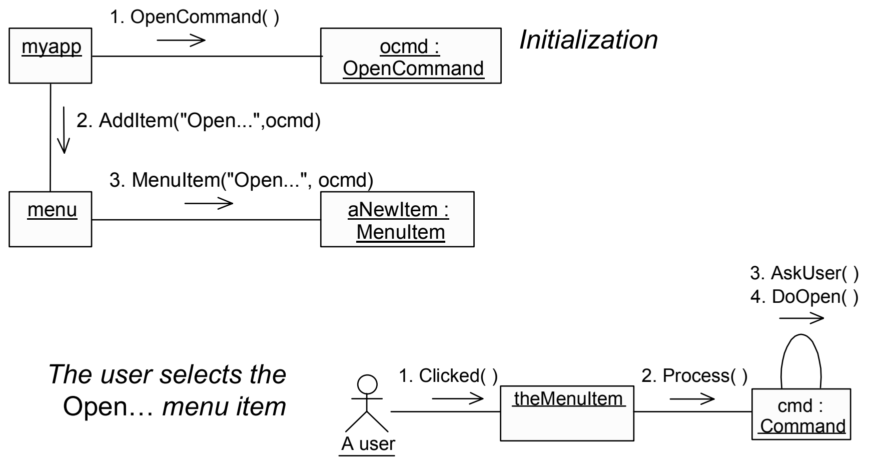
Detailing the Command Pattern (continued)

Suppose now that you want to implement the menu command *Open…* Create a new class called *OpenCommand* that inherits from *Command*. This class overrides the operation *Process* to prompt the user to for the file to open and to open it:

```
AskUser();
DoOpen();
```

Detailing the Command Pattern (continued)

Detailing the Command Pattern (continued)



12

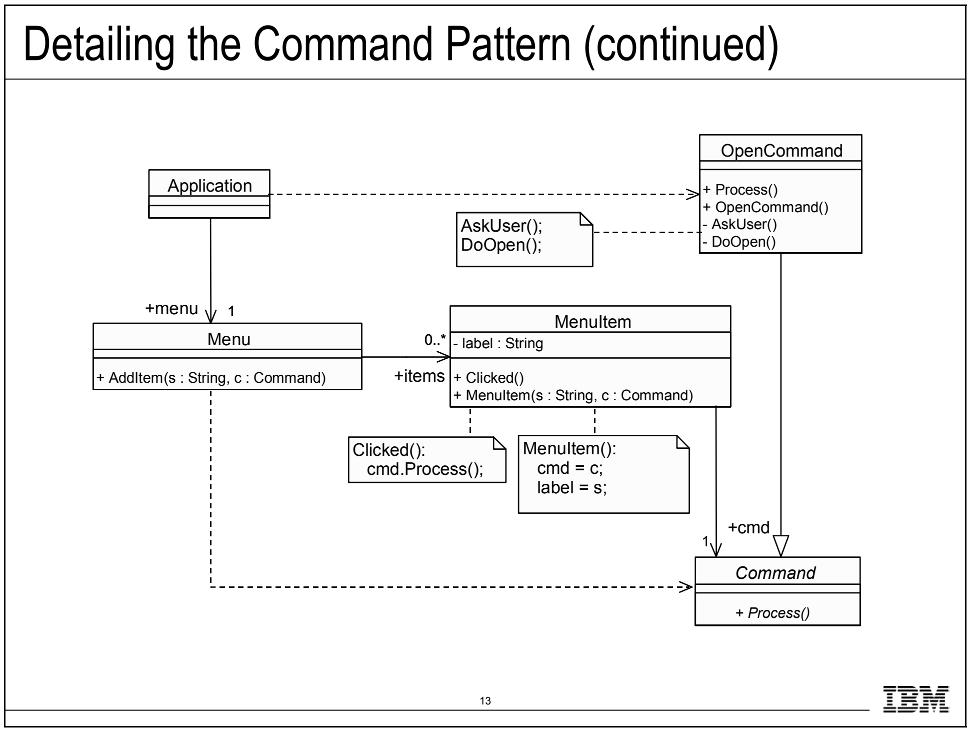
- Look at the dynamic behavior of the system: first, create an interaction diagram showing the initialization of the system. When the system started up, the objects *myapp:Application* and *menu:Menu* were created. Then *myapp* creates the object *ocmd:OpenCommand* (message 1). Then it invokes a new operation from *menu* called *AddItem* that takes two arguments: *s* of type *String* (the label of the menu item to create) and *c* of type *Command*. Note that *myapp* passes to *AddItem,* a subclass of *Command* (message 2).
- *Menu* creates a new menu item (message 3). The arguments of the constructor for *MenuItem* are the same as *AddItem*. The code of the constructor is straightforward:

```
label = s;
cmd = c;
```

It is very important to note that *cmd* is initialized with a subclass of *Command* but that MenuItem "thinks" it is a *Command* object! This is all the magic of polymorphism!

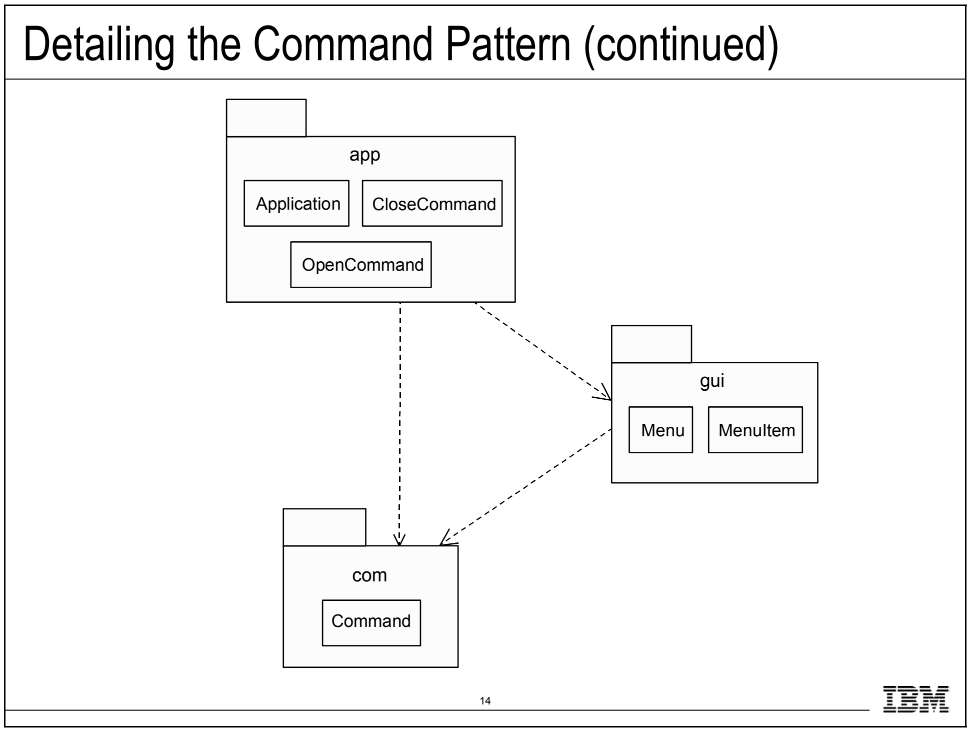
Now, draw a second interaction diagram that will show what happens when the user selects the menu item "Open…". The *Clicked* operation is invoked (message 1). *Clicked* simply executes the *Process* operation of the associated *cmd* object. In this case, the *cmd* object is actually the *OpenCommand* object (although this is transparent to the *MenuItem* object).

Detailing the Command Pattern (continued)


Detailing the Command Pattern (continued)

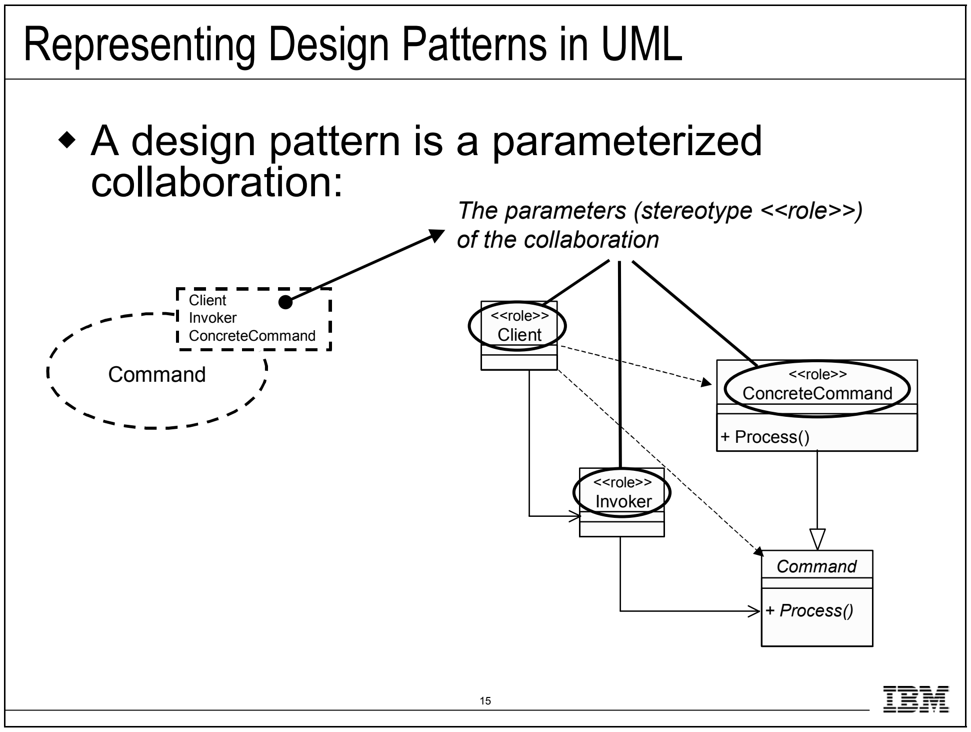
To complete the class diagram, there should be a Uses relationship from *Application* to *OpenCommand* to account for the fact that the former creates the latter, and there should be another Uses relationship from *Menu* to *Command* as the *AddItem* operation takes one argument of type *Command*.

Detailing the Command Pattern (continued)


Detailing the Command Pattern (continued)

So what about reuse? Assign the classes to packages: *OpenCommand* and *Application* to *app* (you have also added a new class, *CloseCommand*), *Menu* and *MenuItem* to *gui* and *Command* to *com*, now add the appropriate dependencies. You have created a reusable set of components (*gui* and *com*) independent of the application packages using them! *MenuItem* can be made an Implementation class only! Only *Menu* is exported.

Representing Design Patterns in UML

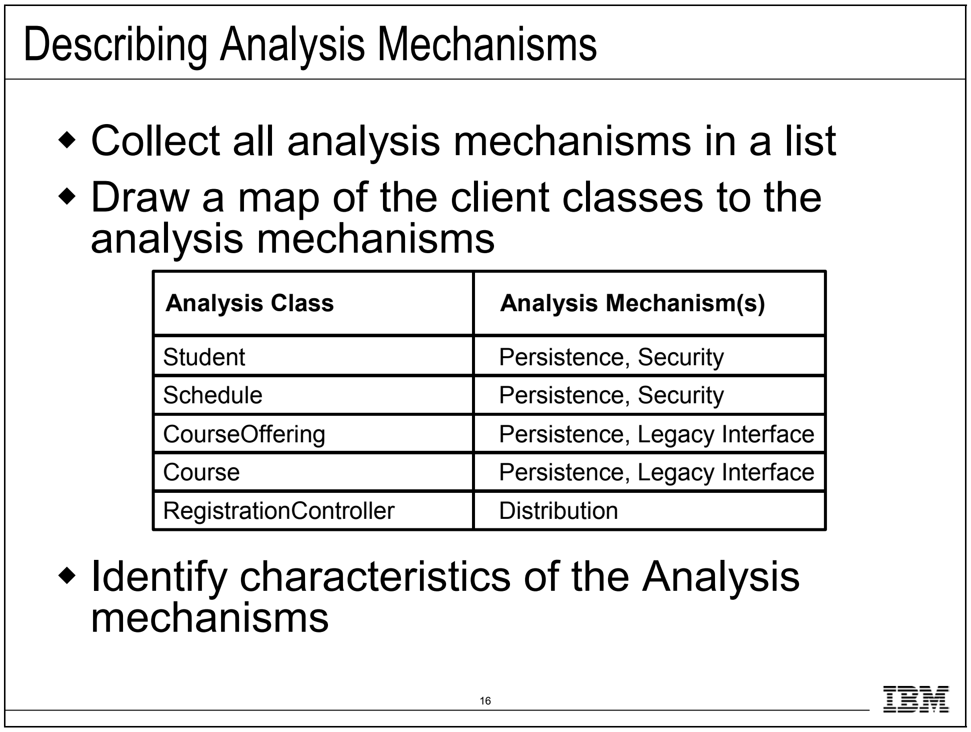


Representing Design Patterns in UML

- A design pattern is a parameterized collaboration:

In the example on the previous slides, the *Client* role was played by the classes *Application* and *Menu*, the *Invoker* was *MenuItem,* and the *ConcreteCommand* was *OpenCommand*.

Describing Analysis Mechanisms



Describing Analysis Mechanisms

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

Analysis Class	Analysis Mechanism(s)
Student	Persistence, Security
Schedule	Persistence, Security
CourseOffering	Persistence, Legacy Interface
Course	Persistence, Legacy Interface
RegistrationController	Distribution

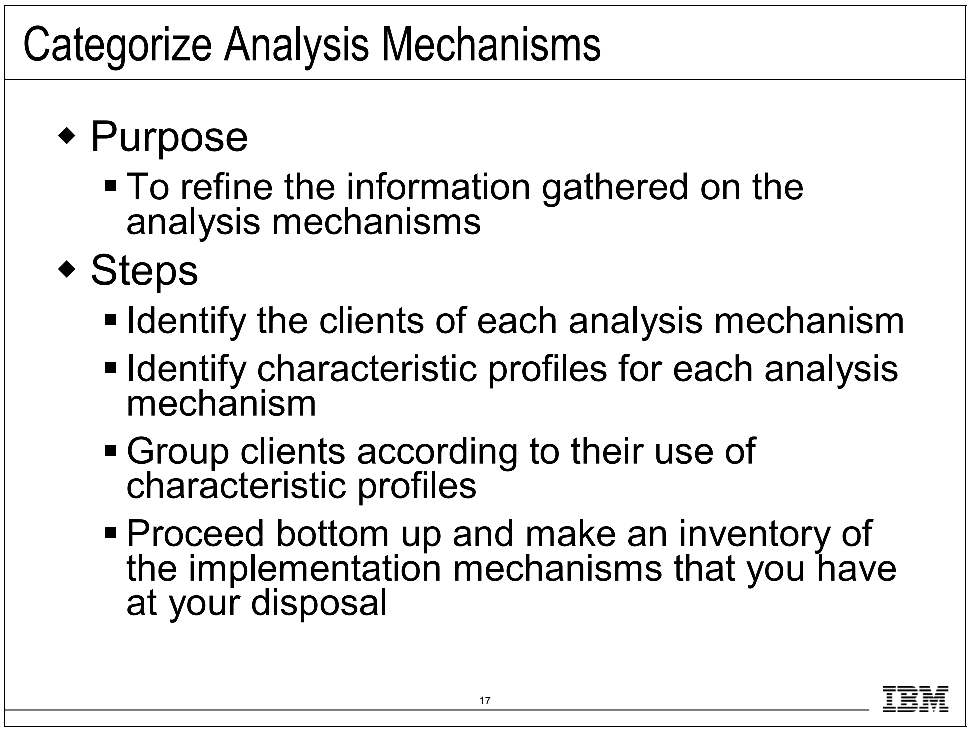
- Identify characteristics of the Analysis mechanisms

16

As we discussed in module 5, analysis mechanisms represent a pattern that constitutes a common solution to a common problem. They can show patterns of structure, patterns of behavior, or both.

These mechanisms might show patterns of structure, patterns of behavior, or both. They are used during analysis to reduce the complexity of analysis, and to improve its consistency by providing designers with a shorthand representation for complex behavior. By using mechanisms as “placeholders” in the architecture, the architecting effort is less likely to become distracted by the details of mechanism behavior.

Categorize Analysis Mechanisms



Categorize Analysis Mechanisms

- Purpose
  - To refine the information gathered on the analysis mechanisms
- Steps
  - 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 implementation mechanisms that you have at your disposal

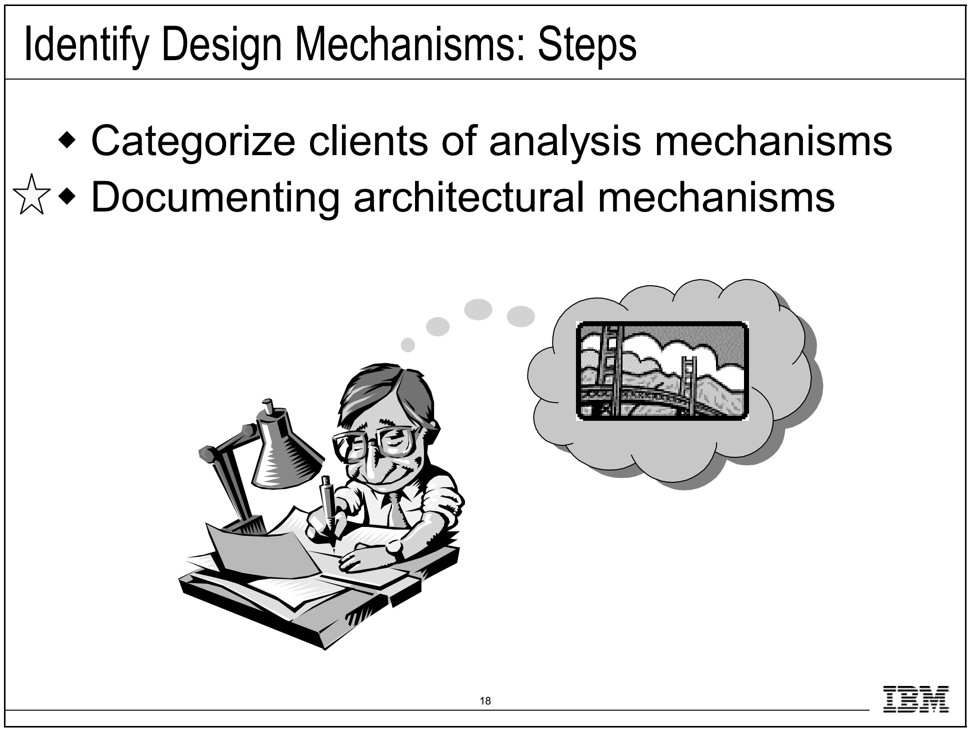
17

Identify the clients of each analysis mechanism. Scan all clients of a given analysis mechanism, looking at the characteristics they require for that mechanism. For example, a number of analysis objects might make use of a persistence mechanism, but their requirements on this can widely vary: A class that has one thousand persistent instances has significantly different persistence requirements than a class that has four million persistent instances. Similarly, a class whose instances must provide sub-millisecond response to instance data requires a different approach than a class whose instance data is accessed through batch applications.

Identify characteristic profiles for each analysis mechanism. There may be widely varying characteristics profiles, providing varying degrees of performance, footprint, security, economic cost, and so forth. Each analysis mechanism is different – so different characteristics will apply to each. Many mechanisms require estimates of the number and size of instances to be managed. The movement of large amounts of data through any system creates tremendous performance issues that must be dealt with.

Group clients according to their use of characteristic profiles. Identify a design mechanism for groups of clients that seem to share a need for an analysis mechanism with a similar characteristics profile. These groupings provide an initial cut at the design mechanisms. An example analysis mechanism, "inter-process communication," might map onto a design mechanism "object request broker." Different characteristic profiles will lead to different design mechanisms that emerge from the same analysis mechanism. The simple persistence mechanism in analysis will give rise to a number of persistence mechanisms in design: in-memory persistence, file-based, database-based, distributed, and so forth.

Identify Design Mechanisms: Steps



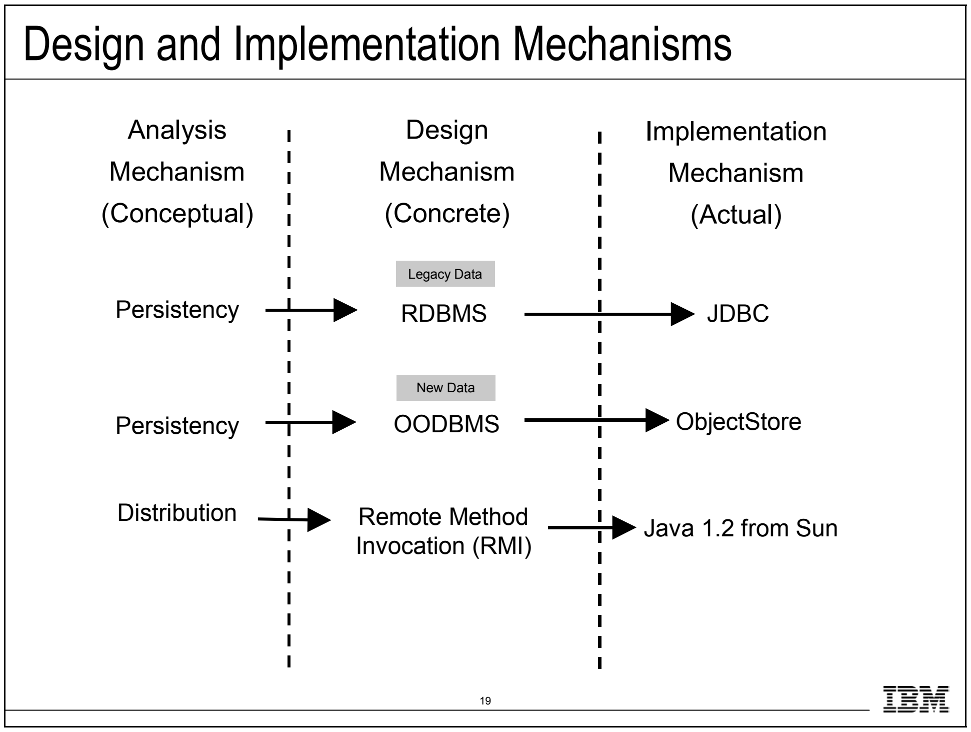
Identify Design Mechanisms: Steps

- Categorize clients of analysis mechanisms
- ☆ Documenting architectural mechanisms

18

Design mechanisms provide an abstraction of the implementation mechanisms, bridging the gap between analysis mechanisms and implementation mechanisms. The use of abstract architectural mechanisms during Design allows us to consider how we are going to provide architectural mechanisms without obscuring the problem-at-hand with the details of a particular mechanism. It also allows us to potentially substitute one specific implementation mechanism for another without adversely affecting the design.

Design and Implementation Mechanisms


Design and Implementation Mechanisms
Analysis Mechanism (Conceptual)
Design Mechanism (Concrete)
Implementation Mechanism (Actual)
Legacy Data
Persistency
RDBMS
JDBC
New Data
Persistency
OODBMS
ObjectStore
Distribution
Remote Method Invocation (RMI)
Java 1.2 from Sun
19
IBM

During Architecture Analysis, you identified the key architectural mechanisms that might be required to solve the problem. Now it is time to refine these and incorporate the decisions made about our implementation.

Design mechanism assumes some details of the implementation environment, but it is not tied to a specific implementation (as is an implementation mechanism). Examples of design mechanisms include:

- **Persistency**: RDBMS, OODBMS, flash card, in-memory storage.
- **Inter-process communication** (IPC): Shared memory, function-call-like IPC, semaphore-based IPC.

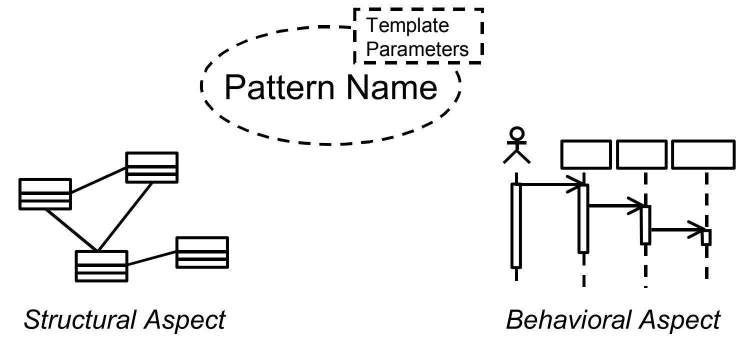
Implementation mechanisms are used during the Implementation process. They are refinements of design mechanisms, and they specify the exact implementation of the mechanism. They are are bound to a certain technology, implementation language, vendor, etc. Some examples of implementation mechanisms include the actual programming language, COTS products, database (Oracle, Sybase), and the inter-process communication/distribution technology in use (COM/DCOM, CORBA).

The above slide shows the architectural mechanism decisions that have been made for each example. For RDBMS persistency (that is, legacy data access), JDBC was chosen. For OODBMS persistency, ObjectStore was chosen, and for Distribution, RMI was chosen. The JDBC mechanism is discussed later in this module. Information on the ObjectStore and RMI mechanisms are provided in the *Additional Information Appendix.*

Review: Documenting Architectural Mechanisms

Review: Documenting Architectural Mechanisms

- Architectural mechanisms can be treated as patterns (i.e., parameterized collaboration)



Documented in Design Guidelines

20

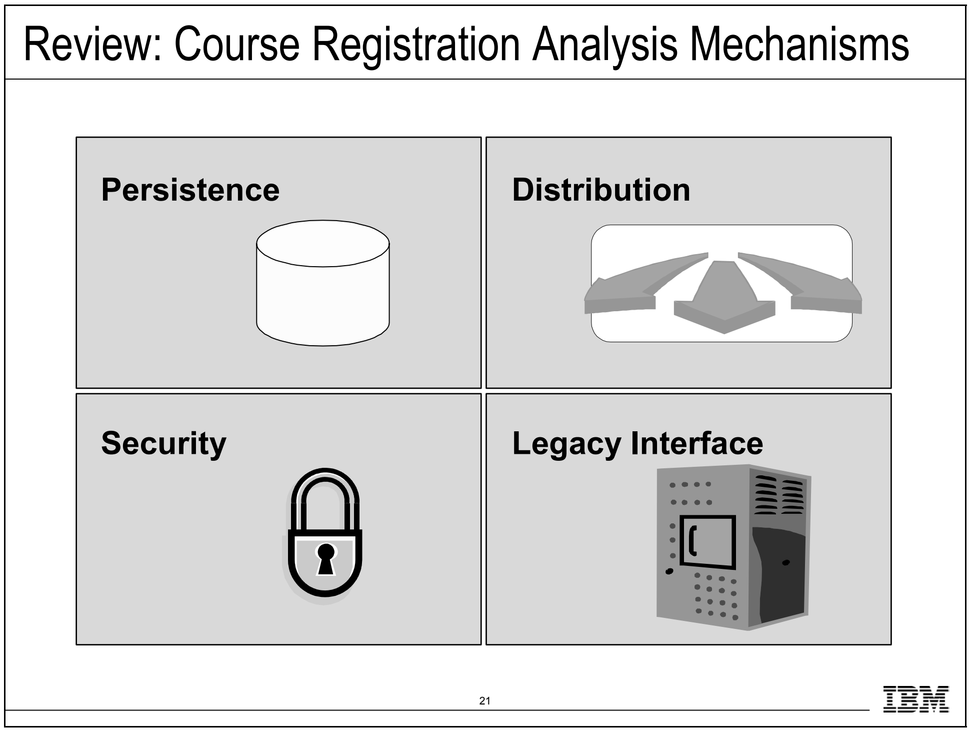
Design patterns as parameterized collaborations were discussed in the Architectural Analysis module. Architectural mechanisms can be treated as patterns and documented the same way (that is, as parameterized collaborations).

Like the patterns discussed in Architectural Analysis, an architectural mechanism parameterized collaboration has a structural aspect and a behavioral aspect. The structural part consists of the classes whose instances implement the mechanism and their relationships (the static view). The behavioral aspect describes how the instances collaborate (that is, send messages to each other) to implement the mechanism (the dynamic view).

The role of the architect is to decide upon and validate mechanisms by building or integrating them, as well as by verifying that they do the job. The architect must then consistently impose the mechanisms upon the rest of the system design. Thus, for each architectural mechanism the architect must provide a static and a dynamic view, accompanied by rules of use.

The mechanisms, the mapping between them, and details regarding their use, must be documented in the Design Guidelines specific to the project, not in the Software Architecture Document (SAD). The SAD captures actual architectural choices made for a system on the basis of nonfunctional requirements and functional requirements. The Design Guidelines document provides for design not yet done. In many organizations, the design guidelines exist as an organizational asset independent of particular projects. It represents the collected reusable design wisdom for that organization in a particular domain. It might then require refinement to suit a project. Therefore, the SAD is the architectural representation (or at least the most significant parts of it). The Design Guidelines cover how to do design, in a very specific, not just conceptual, way.

Review: Course Registration Analysis Mechanisms



Review: Course Registration Analysis Mechanisms

The above slide lists the selected analysis mechanisms for the Course Registration System.

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

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

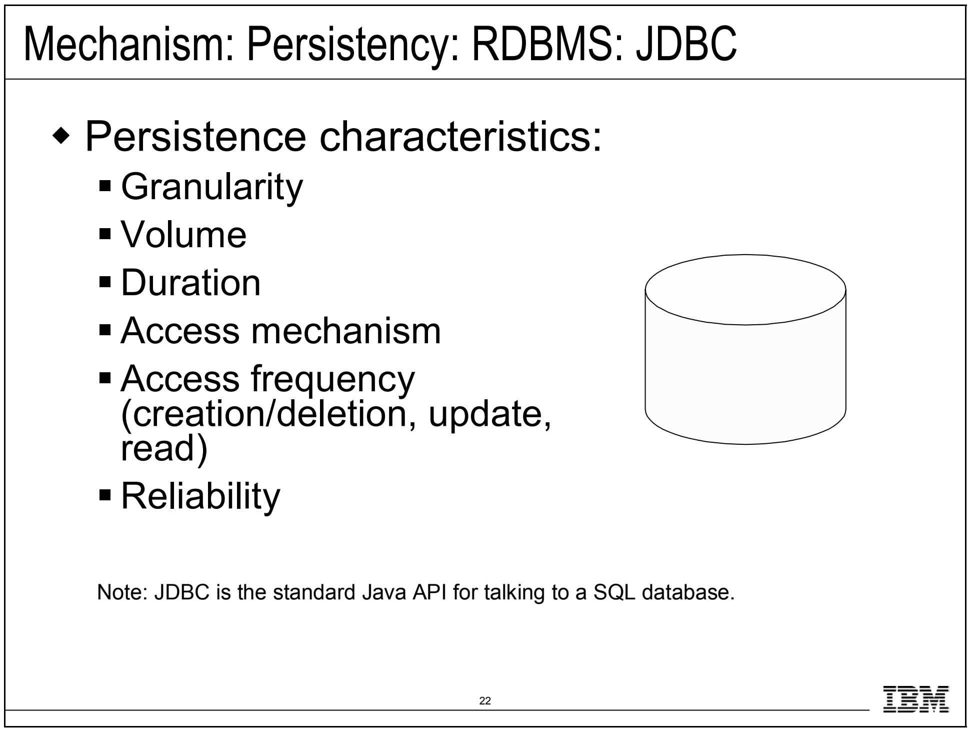
Security: A means to control access to an element.

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

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

Next, we will discuss the design mechanisms for persistence, distribution, and security.

Mechanism: Persistency: RDBMS: JDBC



Mechanism: Persistency: RDBMS: JDBC

- Persistence characteristics:
  - Granularity
  - Volume
  - Duration
  - Access mechanism
  - Access frequency (creation/deletion, update, read)
  - Reliability

Note: JDBC is the standard Java API for talking to a SQL database.

22

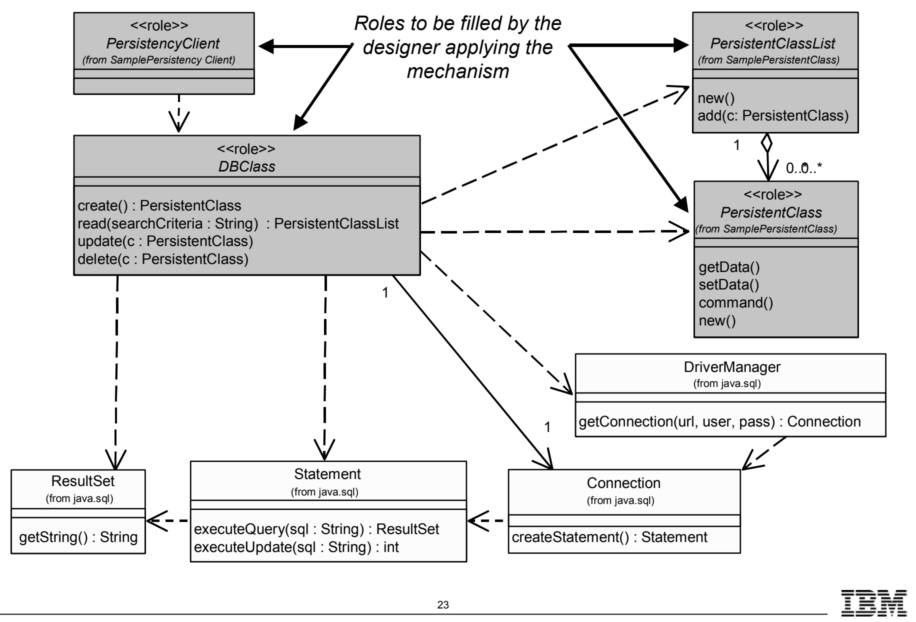
These characteristics were first introduced in Architectural Analysis. When we described the persistency architectural mechanism, we used the following characteristics;

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

- **Granularity**: What is the range of the persistent objects?
- **Volume**: How many objects must be kept persistent?
- **Duration**: How long must the persistent objects be kept?
- **Access mechanism**: How is a given object uniquely identified and retrieved?
- **Access frequency**: Are the objects more or less constant? Are they permanently updated?
- **Reliability**: Can the objects survive a crash of the process, the processor, or the whole system?

Example: Persistency: RDBMS: JDBC

Example: Persistency: RDBMS: JDBC



<<role>> PersistencyClient (from SamplePersistency Client)
Roles to be filled by the designer applying the mechanism
<<role>> PersistentClassList (from SamplePersistentClass)
new()
add(c: PersistentClass)
1
0..*
<<role>> DBClass
create() : PersistentClass
read(searchCriteria : String) : PersistentClassList
update(c : PersistentClass)
delete(c : PersistentClass)
<<role>> PersistentClass (from SamplePersistentClass)
getData()
setData()
command()
new()
1
DriverManager (from java.sql)
getConnection(url, user, pass) : Connection
1
ResultSet (from java.sql)
getString() : String
Statement (from java.sql)
executeQuery(sql : String) : ResultSet
executeUpdate(sql : String) : int
Connection (from java.sql)
createStatement() : Statement
23

The next few slides demonstrate the pattern of use of the persistent mechanism chosen for the RDBMS classes in our example: JDBC. The above diagram is the static view.

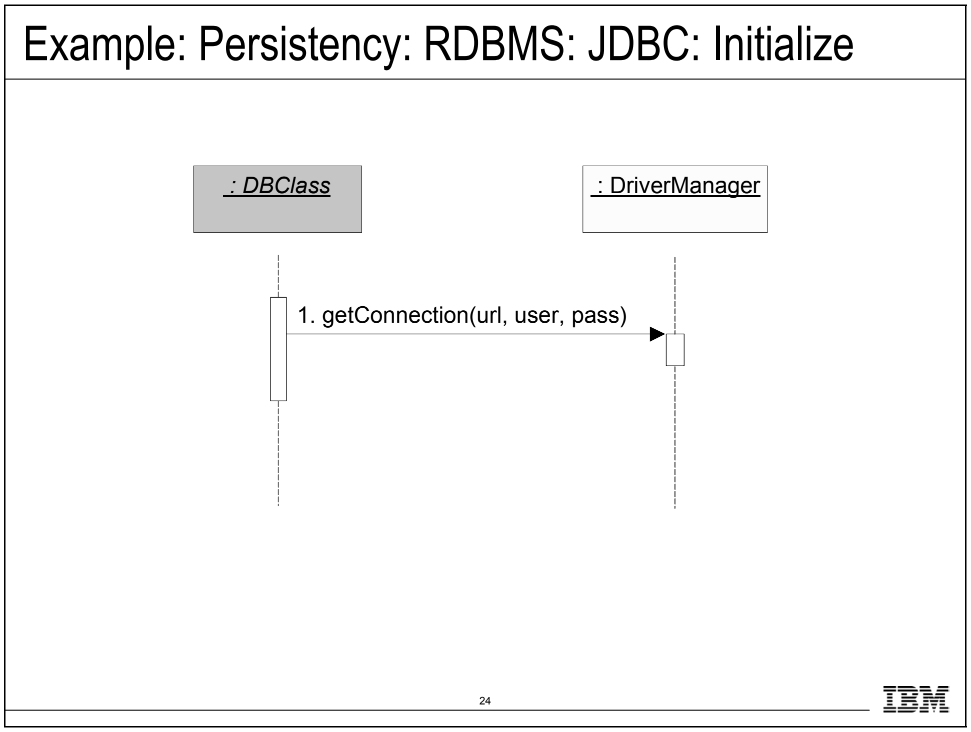
For JDBC, a client works with a **DBClass** to read and write persistent data. The DBClass is responsible for accessing the JDBC database using the **DriverManager** class. Once a database **Connection** is opened, the DBClass can then create SQL statements that will be sent to the underlying RDBMS and 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.

The **DBClass** is the one responsible for making another class instance persistent. It understands the OO-to-RDBMS mapping and has the behavior to interface with the RDBMS. The DBClass 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 DBClass.

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.

Example: Persistency: RDBMS: JDBC: Initialize


Example: Persistency: RDBMS: JDBC: Initialize

Initialization must occur before any persistent class can be accessed.

To initialize the connection to the database, the DBClass must load the appropriate driver by calling the DriverManager getConnection() operation with a URL, user, and password.

getConnection() attempts to establish a connection to the given database URL. The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.

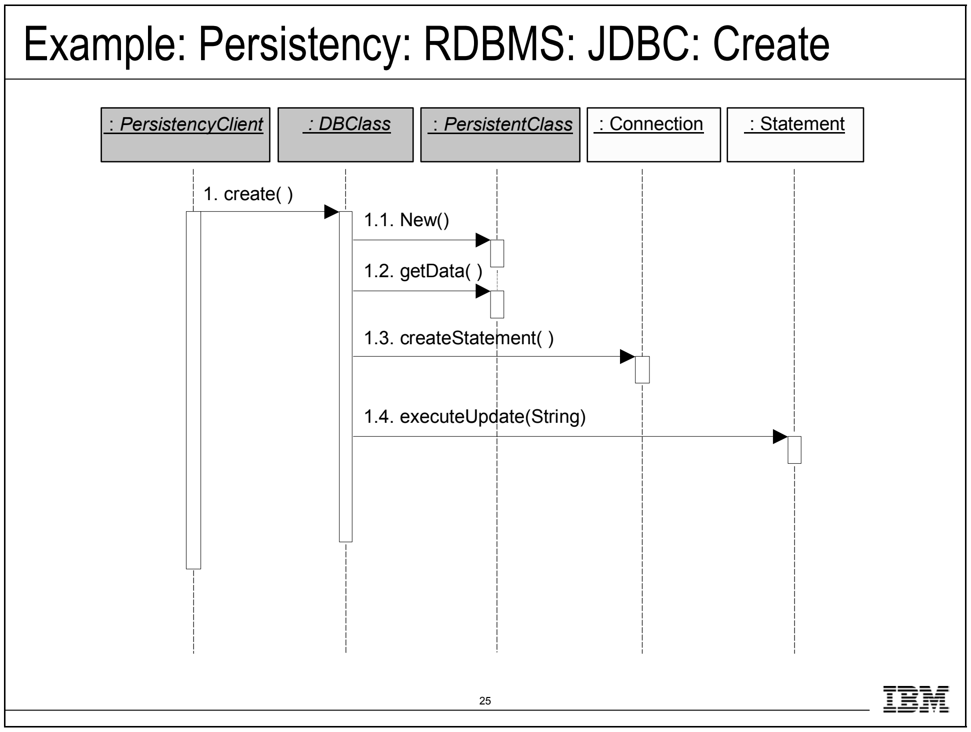
Parameters:

url: A database url of the form jdbc:subprotocol:subname. This URL is used to locate the actual database server. It is not Web-related in this instance.

user: The database user on whose behalf the connection is being made.

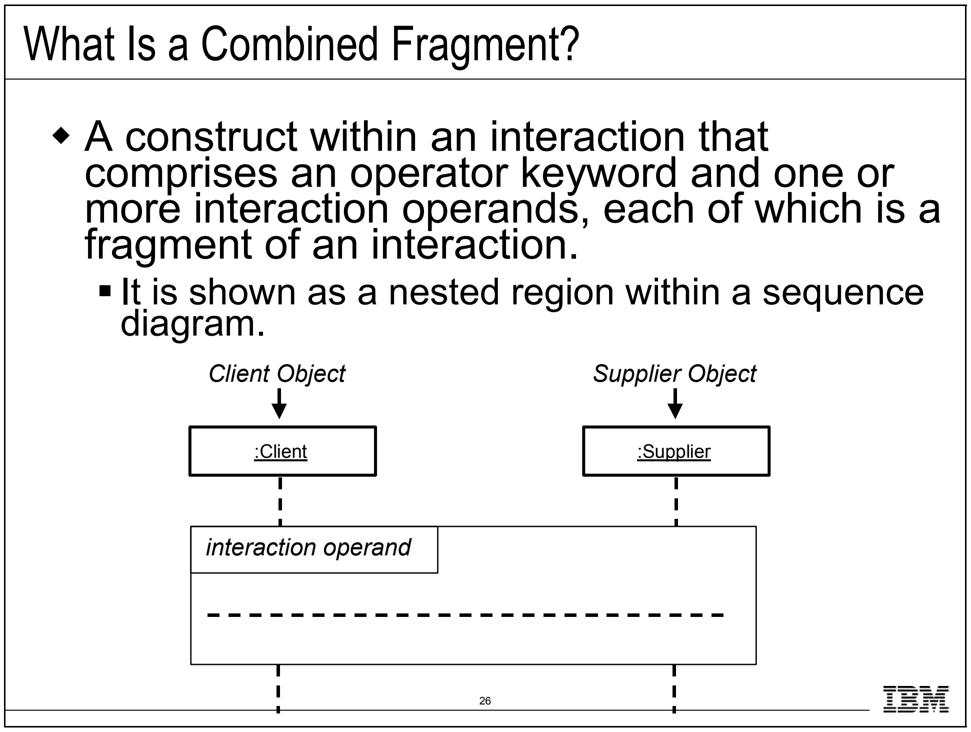
password: The user's password. It returns a connection to the URL.

Example: Persistency: RDBMS: JDBC: Create



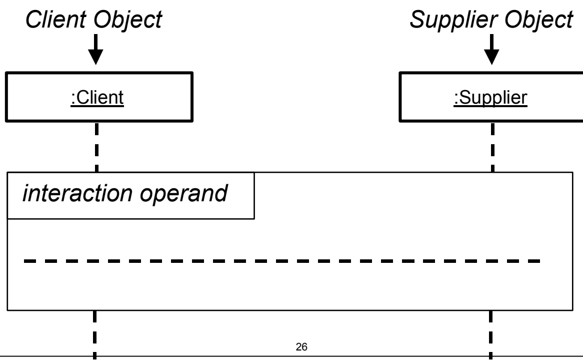
To create a new class, the persistency client asks the DBClass to create the new class. The DBClass creates a new instance of PersistentClass with default values. The DBClass then creates a new Statement using the Connection class createStatement() operation. The Statement is executed, and the data is inserted into the database.

What Is a Combined Fragment?



What Is a Combined Fragment?

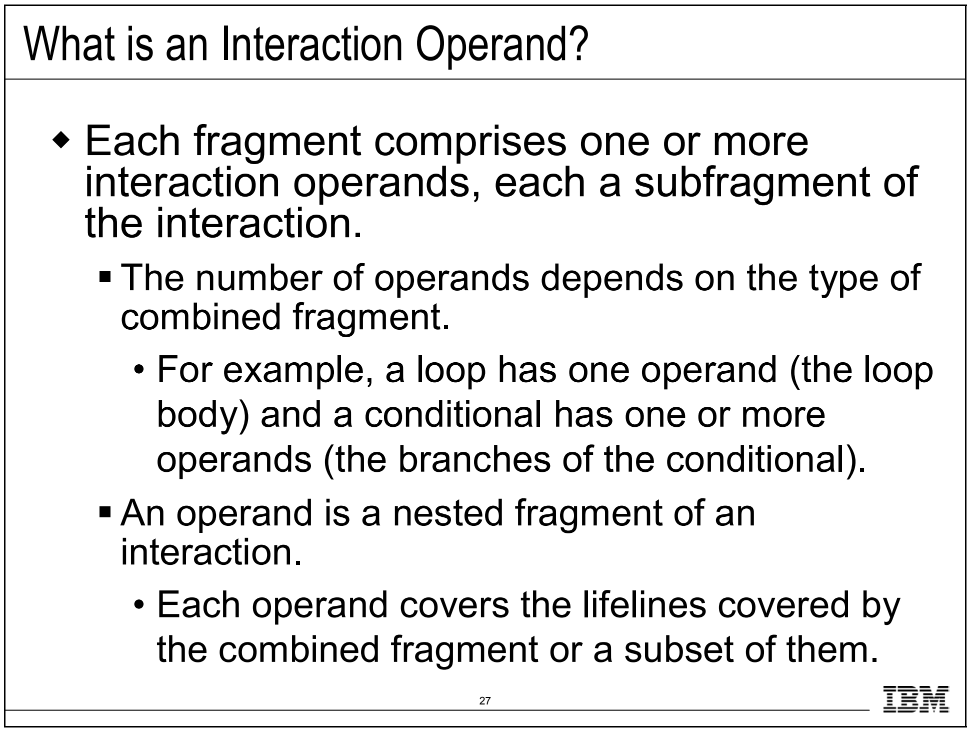
- A construct within an interaction that comprises an operator keyword and one or more interaction operands, each of which is a fragment of an interaction.
  - It is shown as a nested region within a sequence diagram.



26

The general notation for a combined fragment is a rectangle with a small pentagon in the upper left corner containing the interaction operand. The rectangle is nested within its containing fragment or within the sequence diagram as a whole.

What is an Interaction Operand?



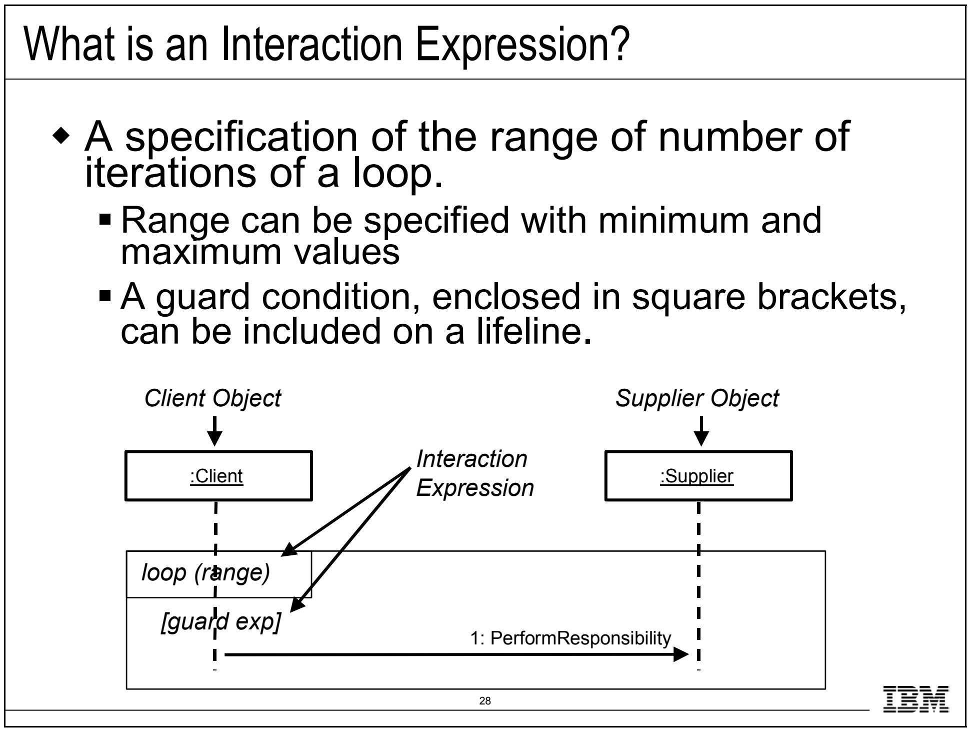
What is an Interaction Operand?

- Each fragment comprises one or more interaction operands, each a subfragment of the interaction.
  - The number of operands depends on the type of combined fragment.
    - For example, a loop has one operand (the loop body) and a conditional has one or more operands (the branches of the conditional).
  - An operand is a nested fragment of an interaction.
    - Each operand covers the lifelines covered by the combined fragment or a subset of them.

27

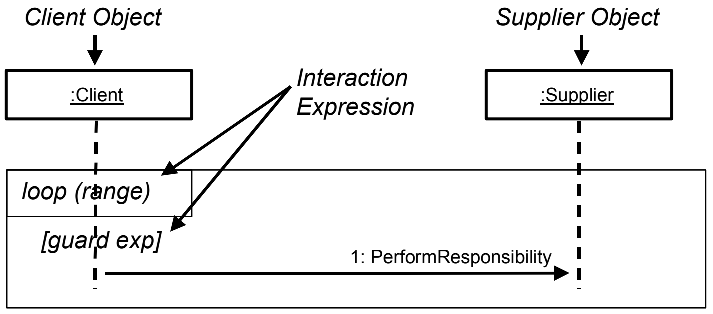
The value of the Interaction Operand is given as text in a small compartment in the upper left corner of the Combined Fragment frame (alt, assert, break, consider, critical, ignore, loop, neg, opt, par, seq, strict). Multiple interaction operands will be separated by a dashed horizontal line and together make up the framed Combined Fragment.

What is an Interaction Expression?



What is an Interaction Expression?

- A specification of the range of number of iterations of a loop.
  - Range can be specified with minimum and maximum values
  - A guard condition, enclosed in square brackets, can be included on a lifeline.



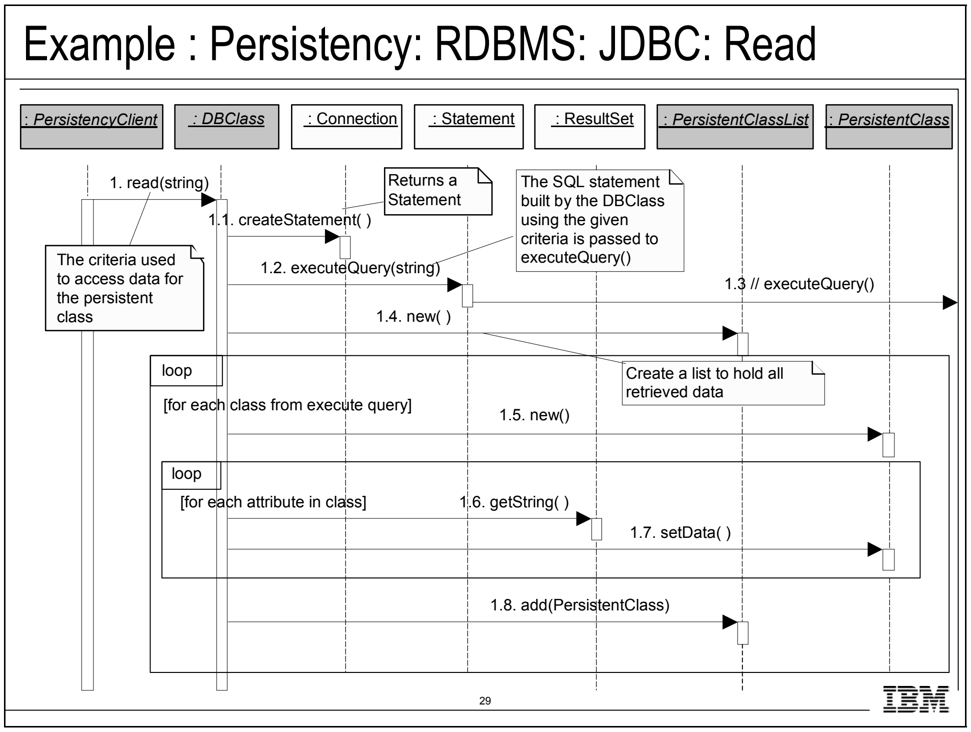
28

The range on the number of iterations of a loop are included in parentheses as part of the tag after the keyword loop:

- loop Minimum = 0, unlimited maximum
- loop (repeat) Minimum = maximum = repeat
- loop (minimum, maximum) Explicit minimum and maximum bounds

In addition to the bounds, a Boolean expression can be included as a guard on a lifeline. As long as the expression is true, the loop will continue to iterate.

Example: Persistency: RDBMS: JDBC: Read



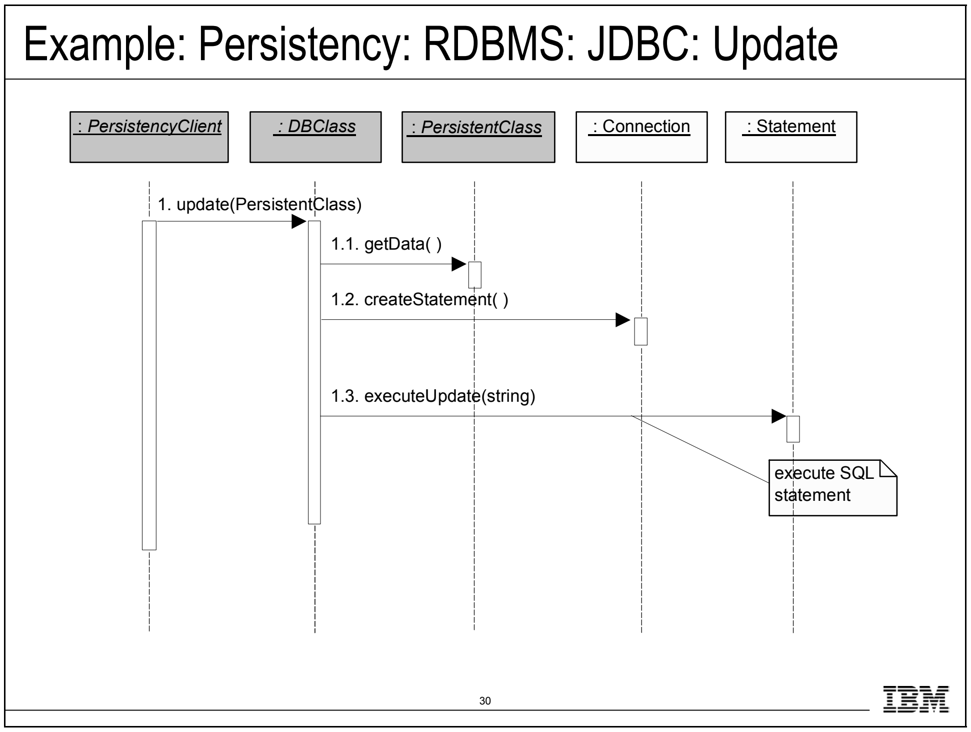
Example : Persistency: RDBMS: JDBC: Read

: PersistencyClient
: DBClass
: Connection
: Statement
: ResultSet
: PersistentClassList
: PersistentClass
1. read(string)
The criteria used to access data for the persistent class
1.1. createStatement( )
Returns a Statement
The SQL statement built by the DBClass using the given criteria is passed to executeQuery()
1.2. executeQuery(string)
1.3 // executeQuery()
1.4. new( )
Create a list to hold all retrieved data
loop
[for each class from execute query]
1.5. new()
loop
[for each attribute in class]
1.6. getString( )
1.7. setData( )
1.8. add(PersistentClass)
29

To read a persistent class, the persistency client asks the DBClass to read. The DBClass creates a new Statement using the Connection class createStatement() operation. The Statement is executed, and the data is returned in a ResultSet object. The DBClass then creates a new instance of the PersistentClass and populates it with the retrieved data. The data is returned in a collection object, an instance of the PersistentClassList class.

Note: The string passed to executeQuery() is *not* the exact same string as the one passed into the read(). The DBClass builds the SQL query to retrieve the persistent data from the database, using the criteria passed into the read(). This is because we do not want the client of the DBClass to have the knowledge of the internals of the database to create a valid query. This knowledge is encapsulated within DBClass.

Example: Persistency: RDBMS: JDBC: Update



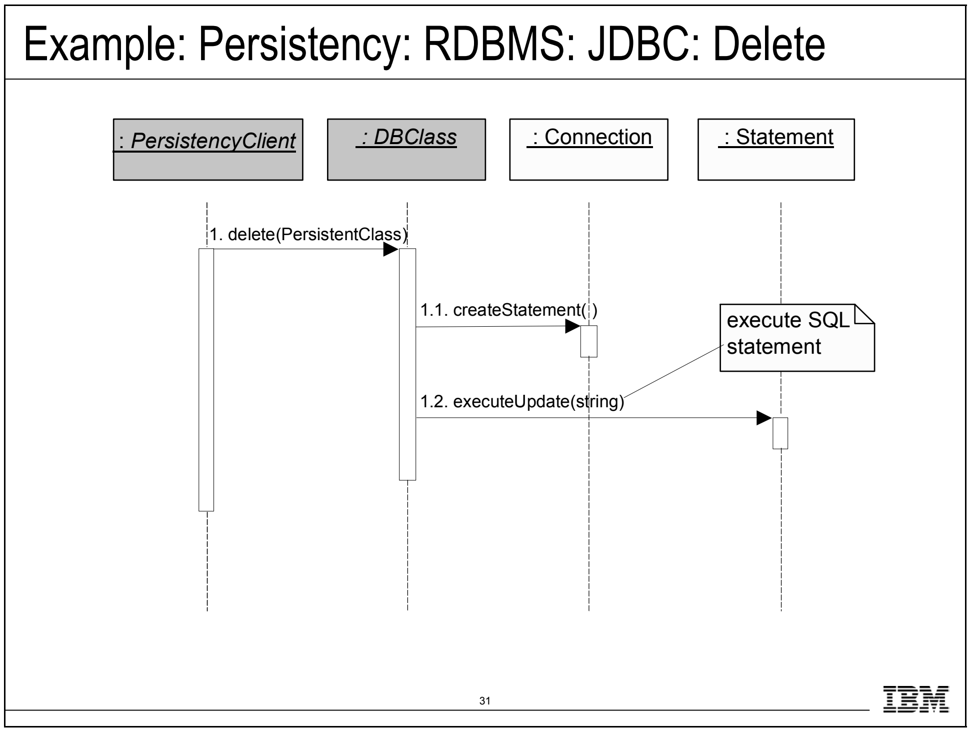
Example: Persistency: RDBMS: JDBC: Update

To update a class, the persistency client asks the DBClass to update. The DBClass retrieves the data from the given PersistentClass object, and creates a new Statement using the Connection class createStatement() operation. Once the Statement is built the update is executed and the database is updated with the new data from the class.

Remember — that it is the DBClass's job to "flatten" the PersistentClass and write it to the database. That is why it must be retrieved from the given PersistentClass before creating the SQL Statement.

Note: In the above mechanism, the PersistentClass must provide access routines for all persistent data so that DBClass can access them. This provides external access to certain persistent attributes that would have otherwise have been private. This is a price you have to pay to pull the persistence knowledge out of the class that encapsulates the data.

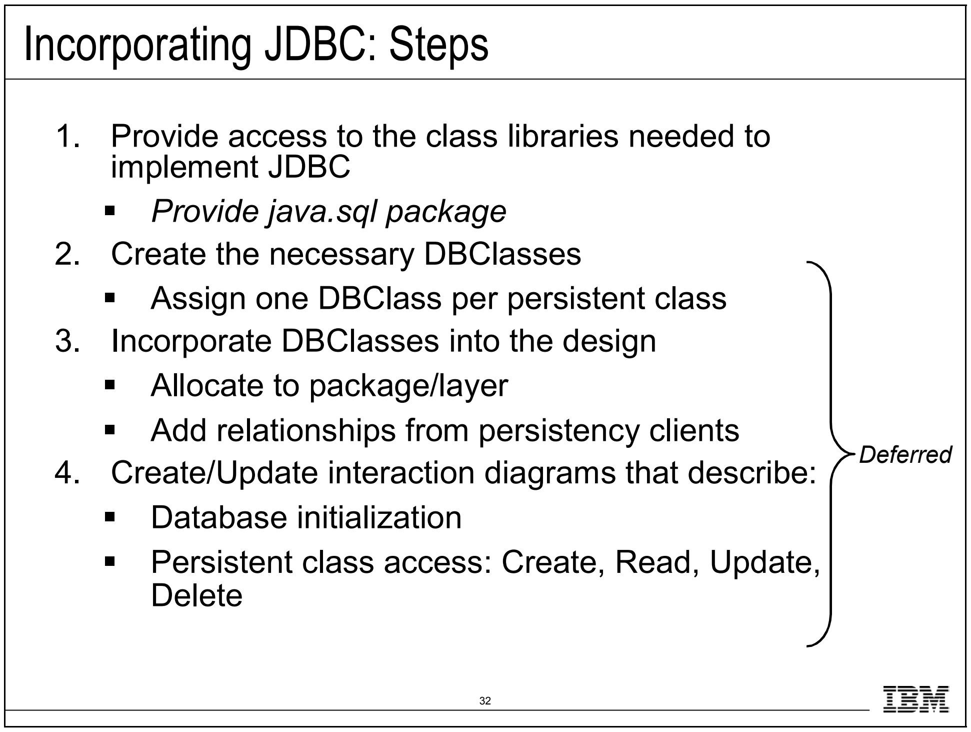
Example: Persistency: RDBMS: JDBC: Delete



Example: Persistency: RDBMS: JDBC: Delete

To delete a class, the persistency client asks the DBClass to delete the PersistentClass. The DBClass creates a new Statement using the Connection class createStatement() operation. The Statement is executed, and the data is removed from the database.

Incorporating JDBC: Steps



Incorporating JDBC: Steps

1. Provide access to the class libraries needed to implement JDBC
   - *Provide java.sql package*
2. Create the necessary DBClasses
   - Assign one DBClass per persistent class
3. Incorporate DBClasses into the design
   - Allocate to package/layer
   - Add relationships from persistency clients
4. Create/Update interaction diagrams that describe:
   - Database initialization
   - Persistent class access: Create, Read, Update, Delete

$\left.\right\}$ *Deferred*

32

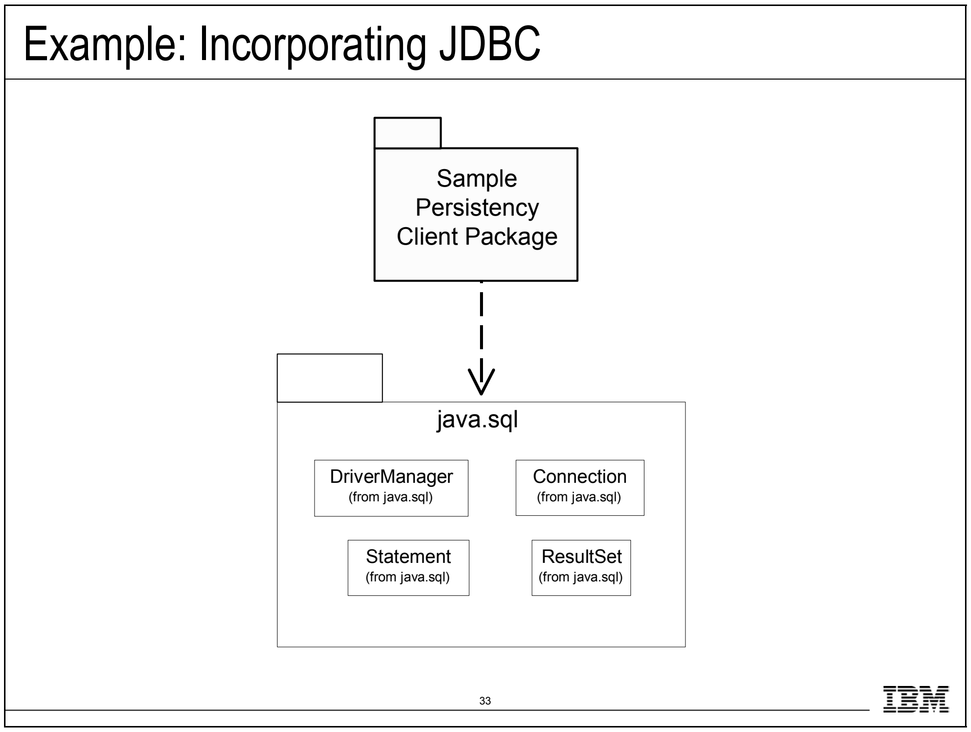
The above is a summary of the steps that can be used to implement the RDBMS persistency mechanism (JDBC). The italicized text on the slide describes the architectural decisions made in regards to JDBC for our Course Registration example. Here are some explanations:

- The java.sql package contains the design elements that support the RDBMS persistency mechanism. It will be depended on by the package(s) in which the DBClasses are placed.
- There is one DBClass per persistent class.
- Once created, the DBClasses must be incorporated into the existing design. They must be allocated to a package/layer.
- Once the DBClasses have been allocated to packages/layers, the relationships to the DBClasses from all classes requiring persistence support will need to be added.

The interaction diagrams provide a means to verify that all required database functionality is supported by the design elements. The sample interaction diagrams provided for the persistency architectural mechanisms during **Identify Design Mechanisms** should serve as starting points for the specific interaction diagrams defined in detailed design.

In **Identify Design Mechanisms**, make sure that the architecture has the necessary infrastructure (that is, that we have access to the class libraries that are needed to implement JDBC). The definition of the actual DBClasses and the development of the detailed interaction diagrams is deferred until detailed design.

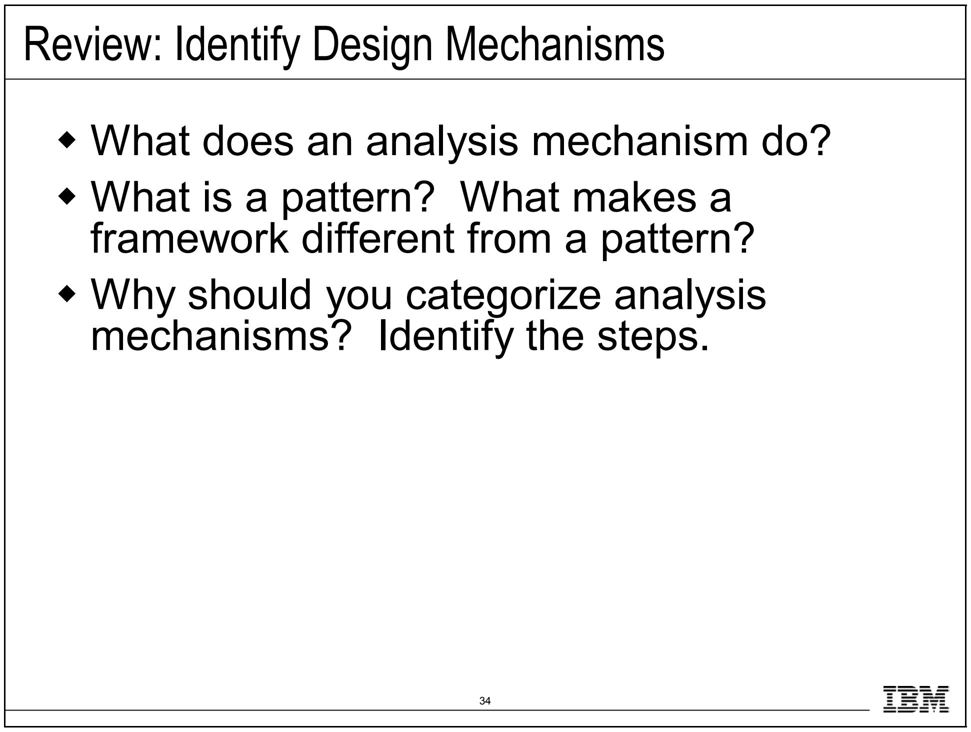
Example: Incorporating JDBC


Example: Incorporating JDBC
Sample Persistency Client Package
java.sql
DriverManager (from java.sql)
Connection (from java.sql)
Statement (from java.sql)
ResultSet (from java.sql)
33
IBM

The following changes must be made to the Course Registration Model to incorporate the JDBC persistency mechanisms:

- Access must be provided to the java.sql package that contains the design elements that support the RDBMS persistency mechanism. The packages where the created DBClasses reside must have a dependency on the java.sql package. Remember, there will be a DBClass for every RDBMS persistent class.
- The creation of the DB classes and the decision as to where they reside in the architecture will be determine during detailed design (for example, Use-Case and Subsystem Design).

Review



Review: Identify Design Mechanisms

- What does an analysis mechanism do?
- What is a pattern? What makes a framework different from a pattern?
- Why should you categorize analysis mechanisms? Identify the steps.

34