J2EE™ Patterns

SL-500

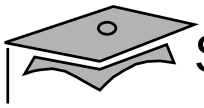
Preface

About This Course

Course Goals

Upon completion of this course, you should be able to:

- Describe the fundamentals of design patterns, including key Gang of Four patterns
- Apply appropriate patterns in the tiers (Integration, Business, and Presentation) of a Java™ 2 Platform, Enterprise Edition (J2EE™ platform) application
- Use patterns to apply best practices in a J2EE platform environment



Course Map

Pattern Fundamentals

Exploring Object-Oriented Design Principles and Design Patterns

Using Gang of Four Behavioral Patterns

Using Gang of Four Creational Patterns

Using Gang of Four Structural Patterns

Patterns In J2EE Application Tiers

Using Architectural Building Blocks

Introducing J2EE™ Patterns

Using Integration Tier Patterns

Using Presentation-to-Business Tier Patterns

Using Intra-Business Tier Patterns

Using Presentation Tier Patterns

More Presentation Tier Patterns

Application of Best Practices In J2EE Environment

Exploring AntiPatterns

Applying J2EE BluePrints Design Guidelines

Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Services:

- Object-oriented analysis and design – Covered in OO-226: *Object-Oriented Analysis and Design Using UML*
- J2EE platform application development – Covered in FJ-310: *Developing J2EE Compliant Applications*
- J2EE system architecture – Covered in SL-425: *Architecting and Designing J2EE Applications*

Refer to the Sun Services catalog for specific information and registration.

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Have you designed systems using object-oriented techniques?
- Have you developed applications using the Java programming language?

How Prepared Are You?

- Can you describe J2EE platform technologies, such as components based on the Enterprise JavaBeans™ specification (EJB™ components), servlet technology components, and pages created with the JavaServer Pages™ (JSP™ pages) technology?
- Can you read a Unified Modeling Language (UML) class diagram and a UML sequence diagram?

Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

Icons

Additional resources

Demonstration

Discussion

Note

Typographical Conventions

- `Courier` is used for the names of commands, files, directories, programming code, programming constructs, and on-screen computer output.
- **`Courier bold`** is used for characters and numbers that you type, and for each line of programming code that is referenced in a textual description.
- *`Courier italics`* is used for variables and command-line placeholders that are replaced with a real name or value.
- ***`Courier italics bold`*** is used to represent variables whose values are to be entered by the student as part of an activity.

Additional Conventions

Java programming language examples use the following additional conventions:

- `Courier` is used for the class names, methods, and keywords.
- Methods are not followed by parentheses unless a formal or actual parameter list is shown.
- Line breaks occur where there are separations, conjunctions, or white space in the code.

Module 1

Exploring Object-Oriented Design Principles and Design Patterns

Objectives

- Describe the fundamental object-oriented design concepts
- Describe the fundamental object-oriented design principles
- Describe the characteristics of design patterns

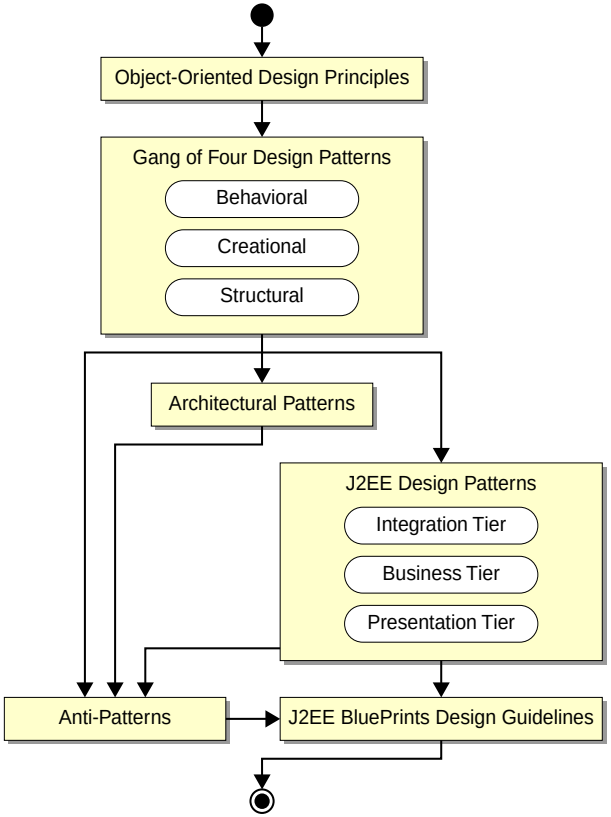
Reviewing Design Goals

This course presents design principles and patterns for reaching the following J2EE™ platform goals:

- Adaptability
- Extensibility
- Maintainability
- Reusability
- Performance
- Scalability
- Reliability
- Development efficiency

Course Flow

The course presents the following topics to reach these goals:



Exploring Object-Oriented Design Concepts

These fundamental object-oriented concepts are the building blocks of the object-oriented and J2EE patterns:

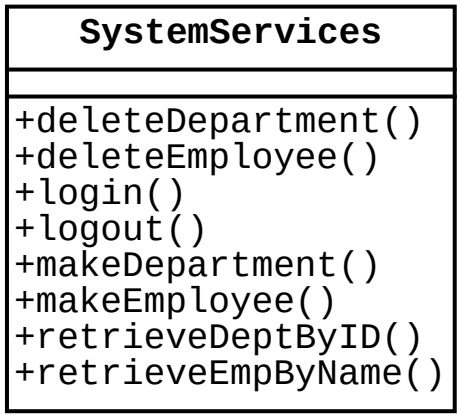
- Cohesion
- Encapsulation
- Coupling
- Implementation inheritance
- Composition
- Interface inheritance
- Polymorphism

Cohesion

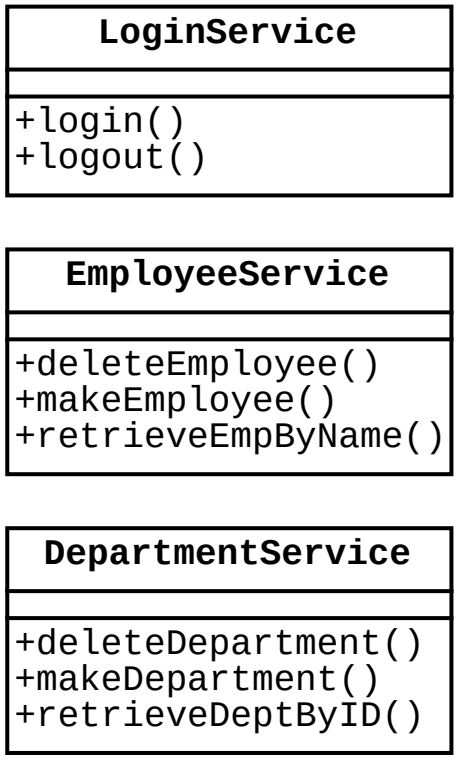
- Cohesion is the degree to which the methods and attributes of a class focus on only one purpose in the system
- Advantages of high cohesion:
  - Avoids the side affects of changing unrelated code within the same class
  - Improves readability by clarifying the class' role
  - Facilitates the creation of small reusable components

Low and High Cohesion Examples

Low Cohesion



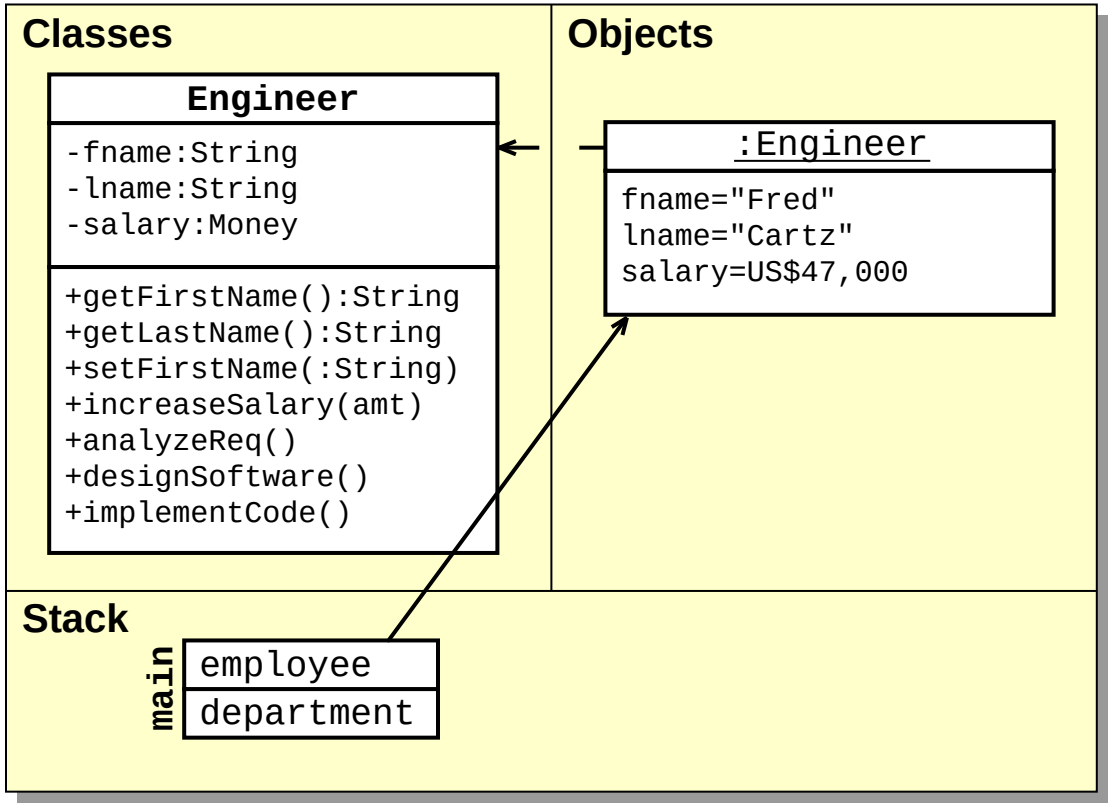
High Cohesion



Encapsulation

- The internal structure and behavior of an object should not be exposed
- Information hiding is key to proper encapsulation:
  - Data should be kept private
  - Accessor and mutator methods should provide an abstract interface to the data
- Advantages:
  - Implementation of a class can vary without changing its interface
  - Developers can use the class without knowing all of its implementation details
  - Inappropriate attribute modifications are prevented

Encapsulation Example



✗ `name = employee.fname;`

✗ `employee.fname = "Samantha";`

✓ `name = employee.getFirstName();`

✓ `employee.setFirstName("Samantha");`

Encapsulation Example

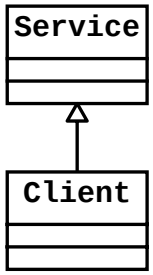
```
1   class Engineer {
2     private String fname;
3     private String lname;
4     private Money salary;
5
6     public String getFirstName() {
7       return fname;
8     }
9
10    public String getLastName() {
11      return lname;
12    }
13
14    public void setFirstName(String fn) {
15      fname = fn;
16    }
17    ...
18  }
```

Coupling

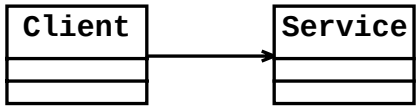
- Coupling is a measure of how dependant classes are on other classes
- To reduce coupling:
  - Hide the implementation of the classes
  - Couple to the abstract interface of a class
  - Reduce the number of methods in the interface of the class
- Consider the coupling of the entire system rather than just between individual classes

Four Levels of Coupling

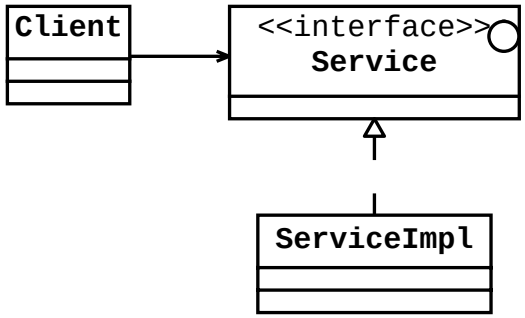
Tight Coupling



Looser Coupling



Looser Abstract Coupling



No Coupling

Implementation Inheritance

Implementation inheritance is inheriting shared attributes and methods from a superclass

Advantages:

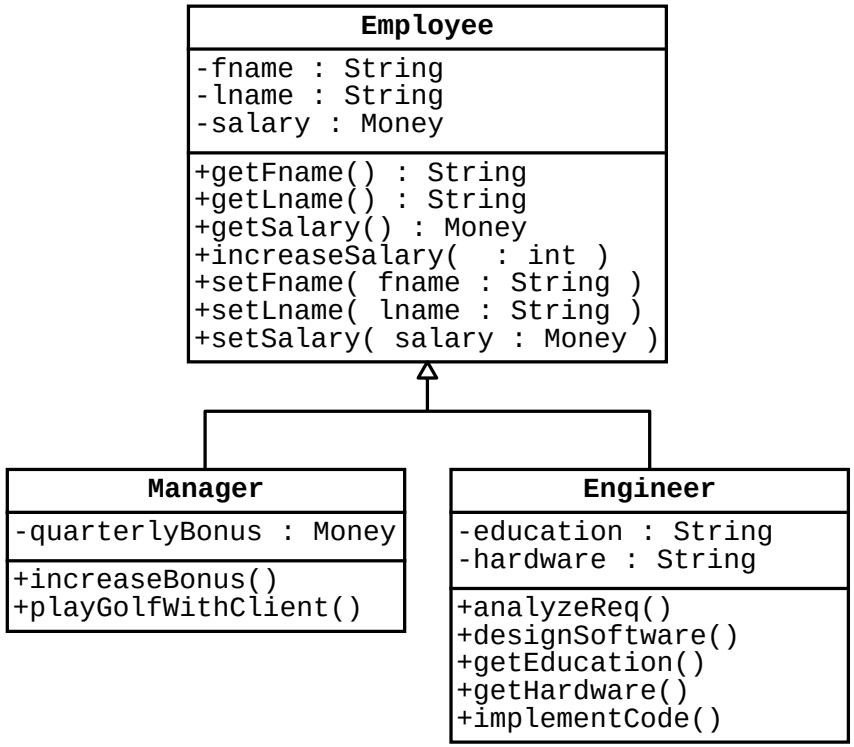
- Avoids duplication of code that is common to subtypes
- Organizes classes according to inheritance

Disadvantages:

- Forces subclass to inherit everything from its superclass
- Changes to the superclass might affect the subclass

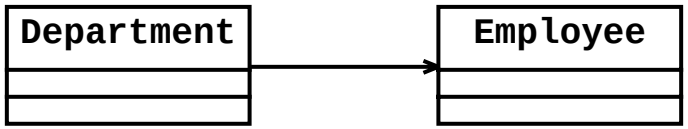
Implementation Inheritance Example

```
1   class Employee
2   {
3   . . .
4   }
5
6   class Manager extends Employee
7   {
8   . . .
9   }
10
11  class Engineer extends Employee
12  {
13  . . .
14  }
```



Composition

- Builds complex objects from simpler objects
- Forms a looser coupling than implementation inheritance



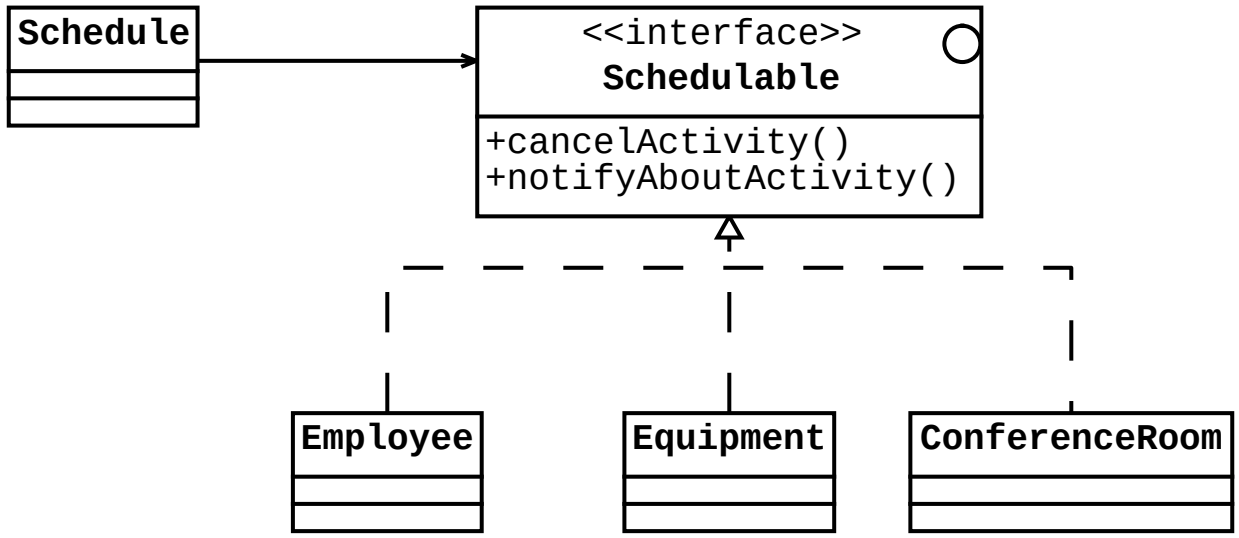

```
1   class Department {
2     private Employee worker;
3     .. .
4   }
```

Interface Inheritance

Interface inheritance is the separation of an interface definition from its implementation:

- Similar to hardware devices that implement a common interface
- Can make a class extensible without being modified
- Java technology-based interfaces (Java interfaces) provide pure interface inheritance; abstract classes allow a mixture of implementation and interface inheritance
- This course uses interfaces whenever possible and leaves it to the attendee to determine whether an abstract class or interface is more appropriate for them

Interface Inheritance Example




```
1   interface Schedulable {
2     public void cancelActivity();
3     public void notifyAboutActivity();
4   }
5   class Equipment implements Scheduable {
6     public void cancelActivity() {//implementation}
7     public void notifyAboutActivity() {//implementation}
8   }
```

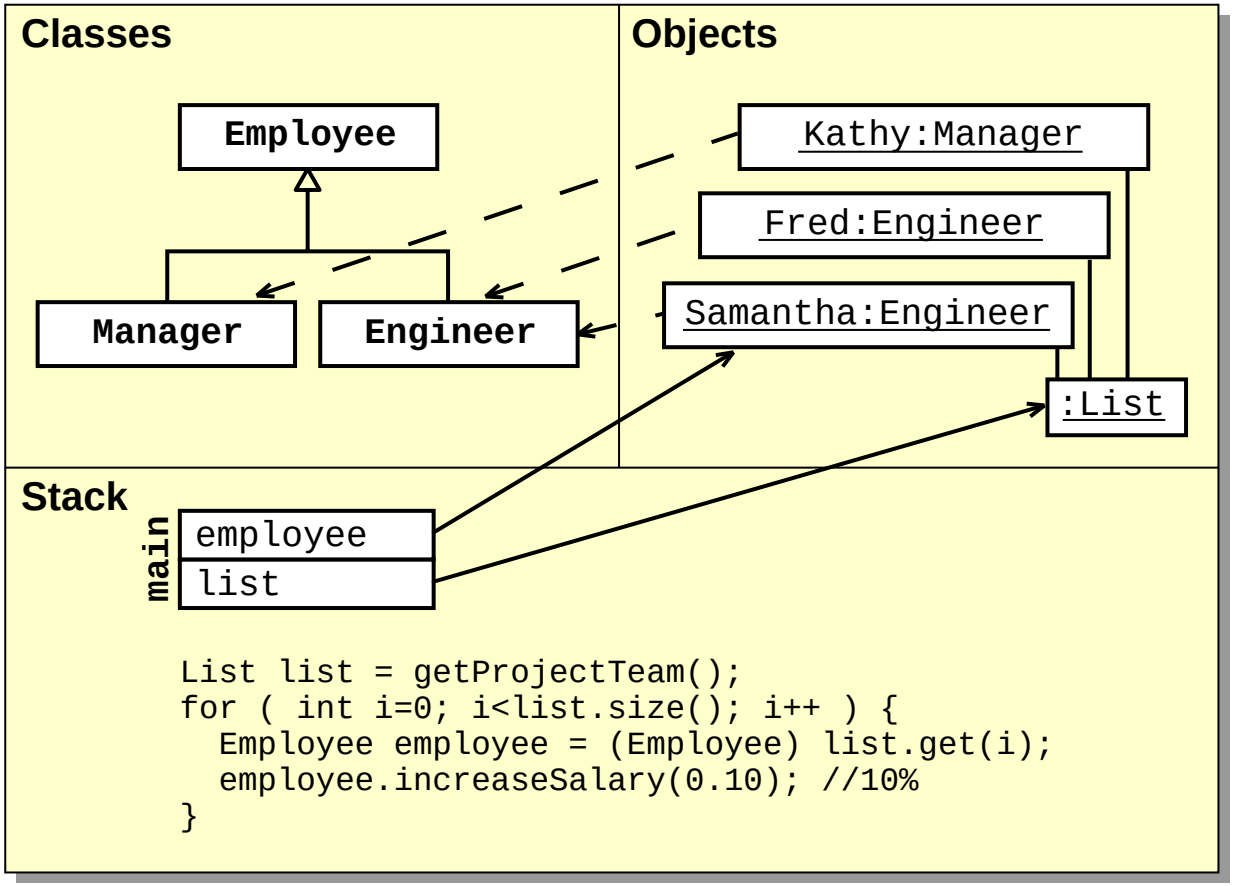
Polymorphism

- Polymorphism allows invocation of operations of specific objects through generic references
- Aspects of polymorphism:
  - A variable can be assigned different types at runtime
  - Method implementation is determined by the object type, not the reference type

Advantages:

- Enables you to write generic code that does not depend upon a specific subclass
- Allows you to code fewer methods because a supertype can be specified as the parameter type

Polymorphism Example



Exploring Object-Oriented Design Principles

The Gang of Four book outlines three object-oriented design principles:

- Favoring composition
- Programming to an interface
- Designing for change

Favoring Composition

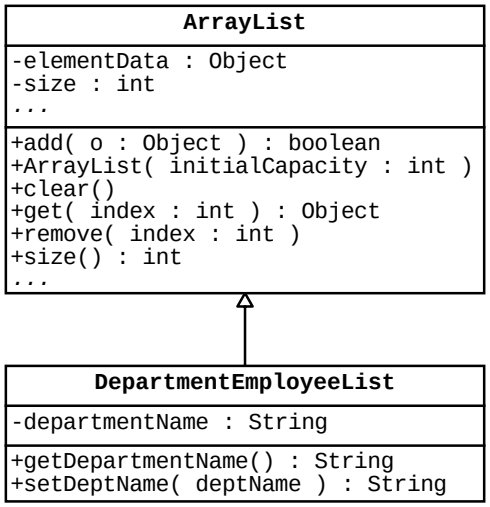
> "Favor object composition over [implementation] inheritance." (Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*)

Reuse functionality through composition instead of implementation inheritance:

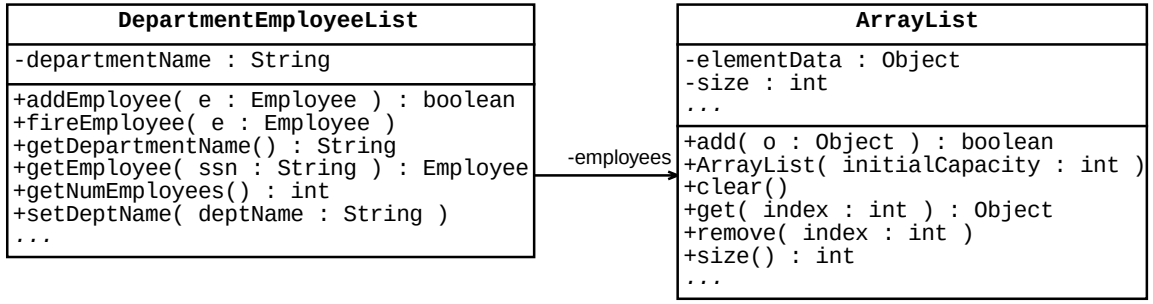
- Implementation inheritance is white-box reuse
- Composition is black-box reuse

Favoring Composition

Implementation Inheritance

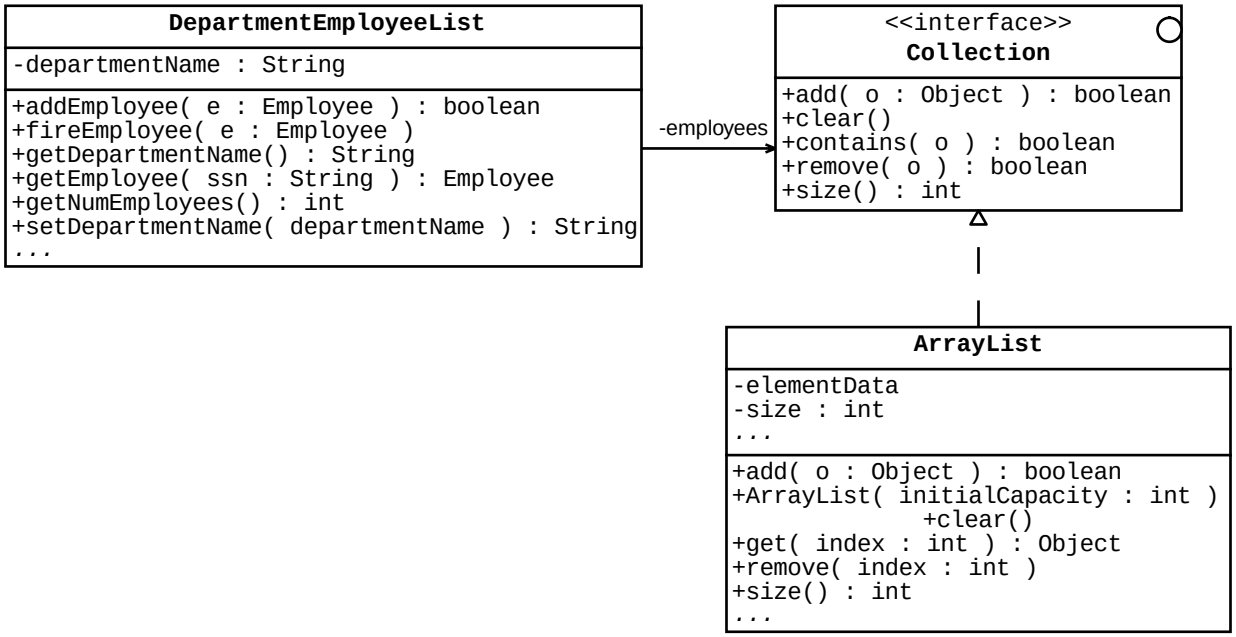


Composition



Programming to an Interface

"Program to an interface, not an implementation." (Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*)



Programming to an Interface

Programming to the `ArrayList` Implementation:

```
1   public class DepartmentEmployeeList {
2     private ArrayList employees = new ArrayList();
3     . . .
4   }
```

Programming to the `Collection` Interface:

```
1   public class DepartmentEmployeeList {
2     private Collection employees = new ArrayList();
3     . . .
4   }
```

Designing for Change

The previous two principles are about designing for change:

- Requirements do change, which makes designing for change very important
- This is more formally stated in the Open-Closed principle: Software entities, such as classes, should be open for extension but closed for modification

Introducing Design Patterns

Design patterns:

- Identify and document proven design experience
- Capture recurring solutions and their trade-offs to similar problems in a given context
- Provide a common vocabulary for design and architecture
- Clarify and document existing architectures
- Move design forward and influence design decisions

Origin of the Pattern Concept

Building architects first developed the pattern concept. They stated that:

> "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
> (Christopher Alexander, *A Pattern Language: Towns, Buildings, Construction*)

Design Pattern Catalogs

Pattern catalogs are groups of patterns. There are catalogs of design patterns for analysis, design, architecture, building architecture, and other topics.

This course focuses on two pattern catalogs:

- Gang of Four Design Pattern catalog
- J2EE Pattern catalog

Gang of Four (GoF) Patterns

The book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides:

- Represented the first publication of several design patterns organized together
- Is often called the Gang of Four or GoF book
- Provided common object-oriented solutions to common object-oriented design problems
- Includes 23 design patterns
- Influenced many J2EE patterns that were built upon the GoF patterns

Gang of Four (GoF) Patterns: Description

"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use."
(Gamma, Helm, Johnson, and Vlissides, *Design Patterns*)

Gang of Four (GoF) Patterns: Groups

The GoF patterns are organized into three groups:

- **Behavioral patterns** – Describe how objects interact and distribute responsibility
- **Creational patterns** – Provide more robust ways to create objects
- **Structural patterns** – Discuss how objects are wired together

Design Pattern Selection

There is no one best pattern, rather there are specific patterns that apply to particular problems:

- Pattern selection should be based on similarities between the context, the problem, and the forces of the pattern and a design problem
- Multiple patterns might be used to solve a particular problem

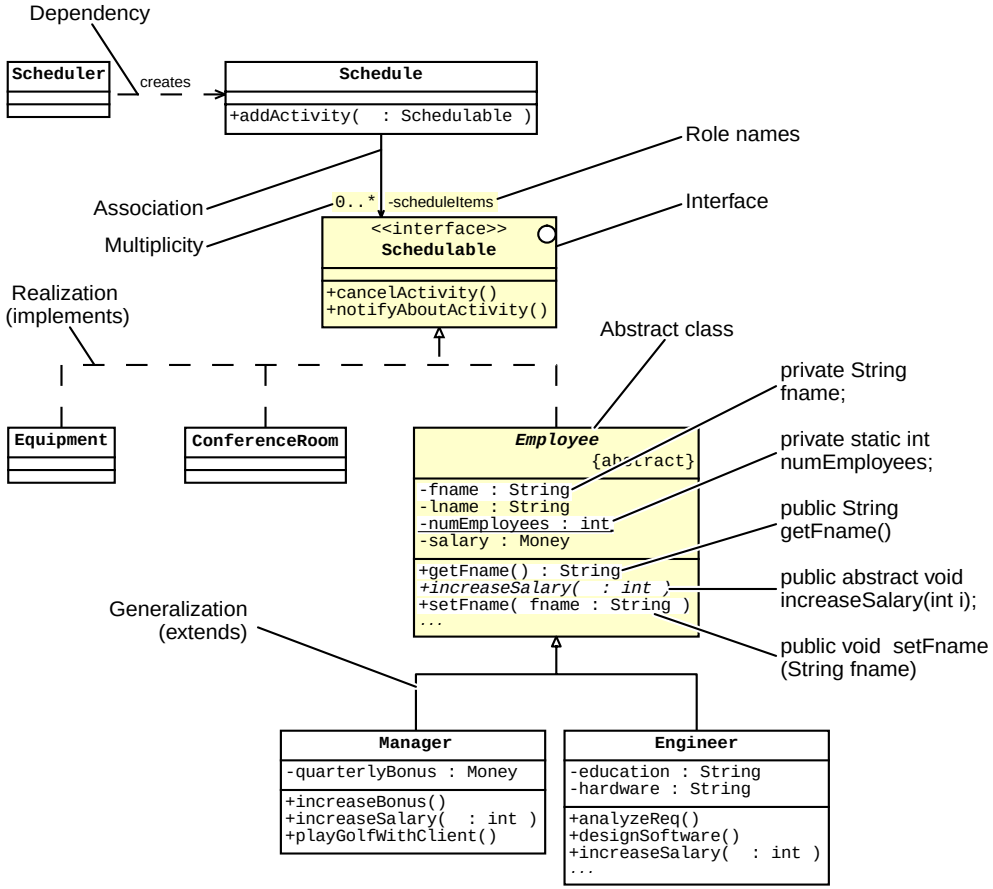
Design Pattern Elements

Design patterns include the following elements:

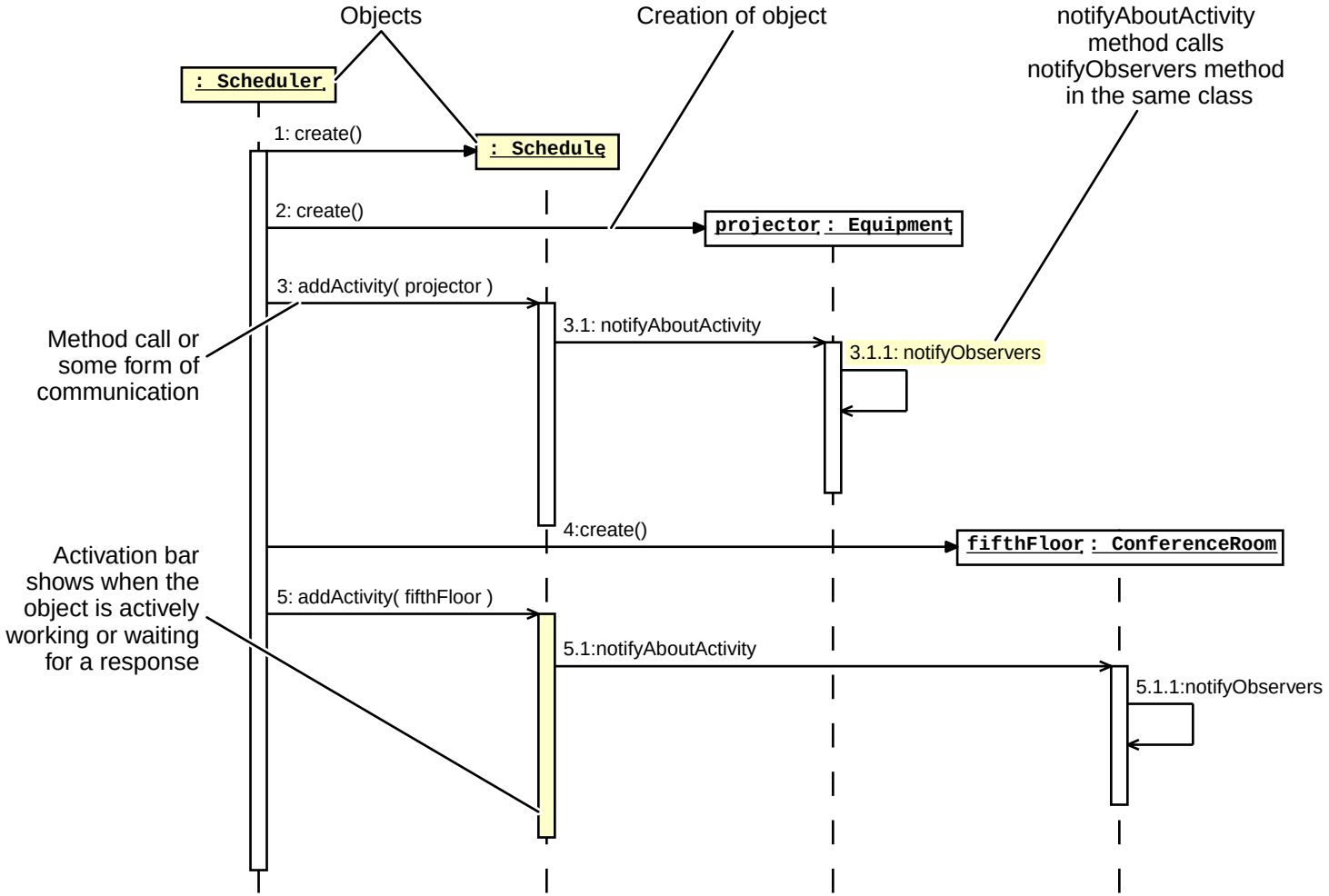
- **Context** – Situation in which the problem being addressed occurs
- **Problem** – Design issue that the pattern addresses
- **Forces** – Requirements that the solution must satisfy
- **Solution and Structure** – Solution to the problem
- **Consequences** – Advantages and disadvantages of using the pattern
- **Related Patterns** – Patterns that are similar or are used to build this pattern

Design Pattern Notation

This course uses UML class and sequence diagrams.



Design Pattern Notation



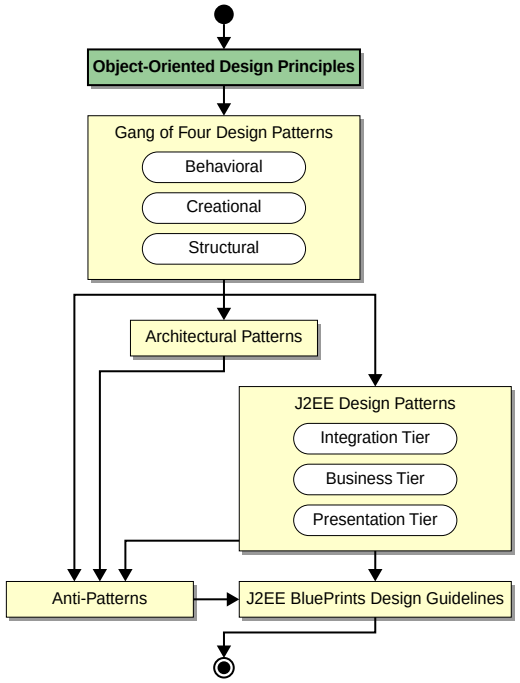
When and How to Apply Patterns

- Some literature suggests that patterns be proactively applied to designs
- Some literature suggests using fewer patterns and only refactoring them into a design
- Either way, you should look for a design balance of flexibility and simplicity
- After taking a design pattern course, you might be inclined to try and solve every problem with a design pattern
- Design patterns are not a golden hammer to solve every problem

Summary

The following object-oriented concepts can help you focus on improving J2EE designs:

- Cohesion
- Encapsulation
- Implementation inheritance
- Interface inheritance
- Polymorphism
- Coupling
- Composition



Summary

The following GoF principles can help you achieve the J2EE platform design goals:

- Favoring composition
- Programming to an interface
- Designing for change

Design patterns will help you reuse the experience of other designers to improve your J2EE platform software

Module 2

Using Gang of Four Behavioral Patterns

Objectives

- Describe the basic characteristics of the Behavioral patterns
- Apply the Strategy pattern
- Apply the Command pattern
- Apply the Iterator pattern
- Apply the Observer pattern

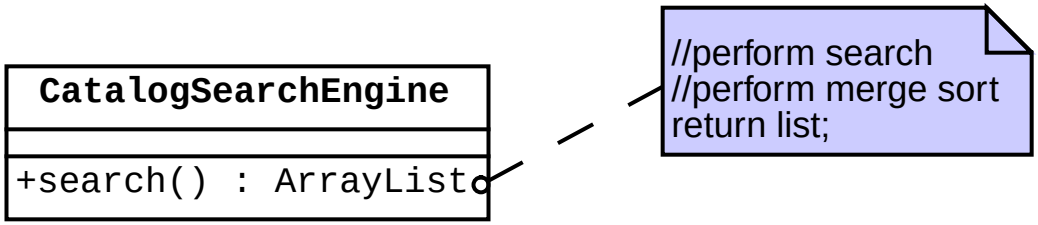
Introducing Behavioral Patterns

Behavioral patterns focus on algorithms and the distribution of responsibility between classes

Pattern	**Primary Function**
Strategy	Encapsulates a family of algorithms for interchangeable use
Command	Encapsulates requests in objects for execution by another object
Iterator	Traverses any collection in a loosely coupled way
Observer	Provides notification of events without polling

Strategy Pattern Example Problem

- The library system's `search` method could use multiple algorithms to sort the results
- You need a way to easily exchange the sorting strategy



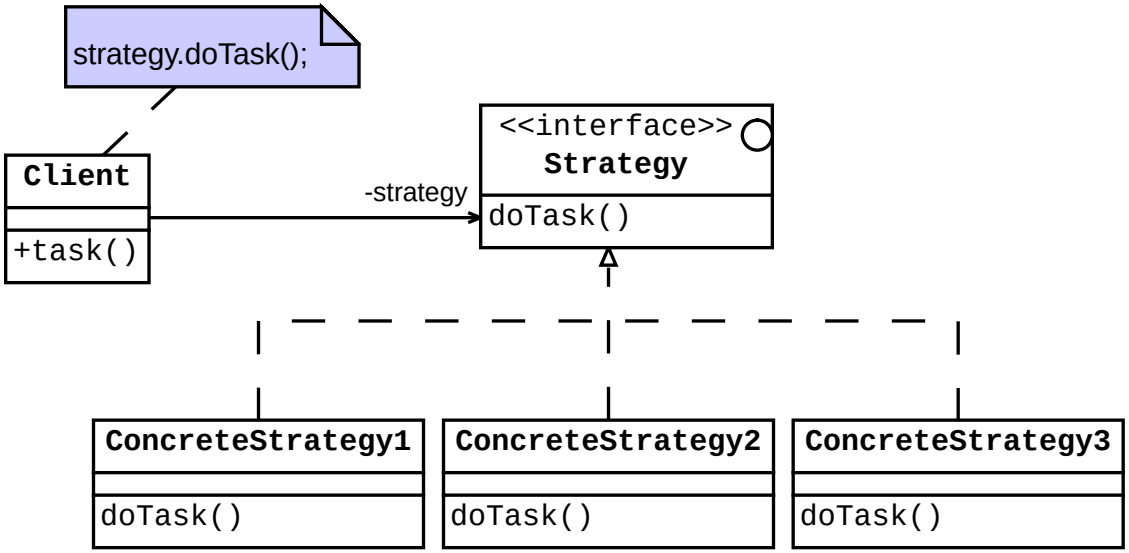
Applying the Strategy Pattern: Problem Forces

- Classes often need to choose from multiple algorithms
- Details of algorithms should be hidden from client classes
- Algorithms implemented in large series of conditional statements are difficult to maintain

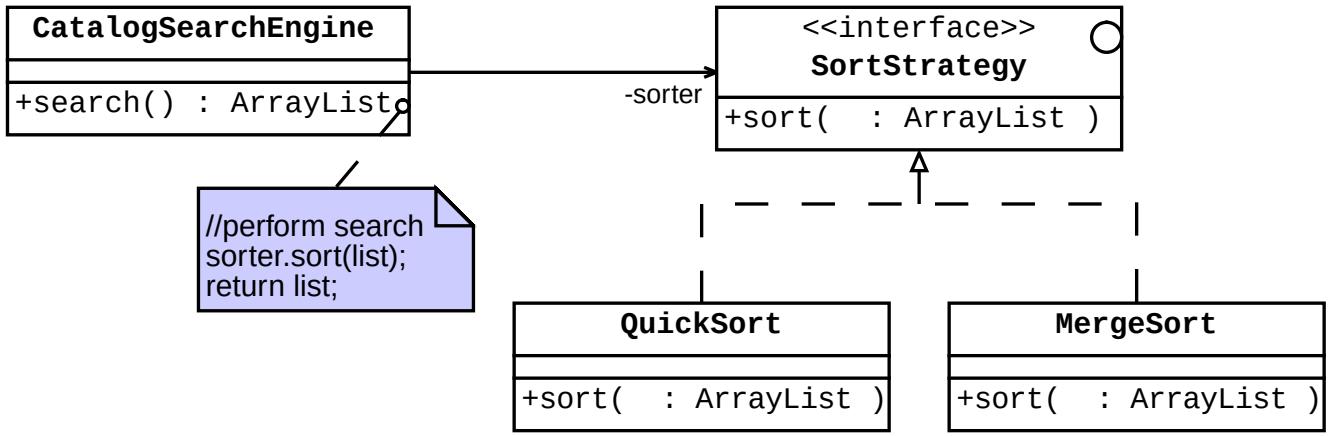
Applying the Strategy Pattern: Solution

- Each algorithm is implemented in a separate class that implements the `Strategy` interface
- The `Client` object has a reference to a `Strategy` object
- When a `Client` object operation is invoked, it forwards the request to that strategy
- The `Client` object can specify which subtype of the `Strategy` interface should be used

Strategy Pattern Structure



Strategy Pattern Example Solution



Strategy Pattern Example Solution

Strategy pattern client:

```
1   public class CatalogSearchEngine {
2
3     private SortStrategy sorter;
4
5     public CatalogSearchEngine(SortStrategy ss) {
6       sorter = ss;
7     }
8     public ArrayList search() {
9       ArrayList list = //perform search
10      sorter.sort(list);
11      return list;
12    }
13  }
```

Strategy Pattern Example Solution

Strategy pattern interface:

```
1   public interface SortStrategy {
2     public void sort(ArrayList al);
3   }
```

First strategy implementation:

```
1   public class QuickSort implements SortStrategy {
2     public void sort(ArrayList al) {
3       //implement Quick sort code
4     }
5   }
```

Applying the Strategy Pattern

Advantages:

- Allows multiple implementations of algorithms without subclassing the client class
- Allows multiple implementations of algorithms without placing them in conditional statements
- Allows easy additions of new algorithms or changes to existing algorithms

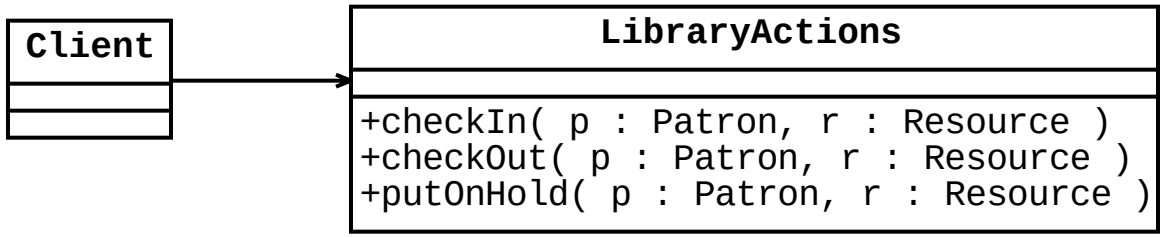
Applying the Strategy Pattern

Disadvantages:

- To select a strategy, clients must be aware of the strategies
- There can be a slight increase in overhead

Command Pattern Example Problem

- A library system has many actions that can be executed
- The system requires undo and redo capabilities
- You need a simple way to structure actions and their invocations



Applying the Command Pattern: Problem Forces

Useful when the client class needs to be separated from how that request is executed:

- There needs to be an easy way to dynamically add new actions and modify existing actions
- The client should be simple
- The client should not need to know the mechanics of handling a specific requests
- The interface to carry out requests should be simple

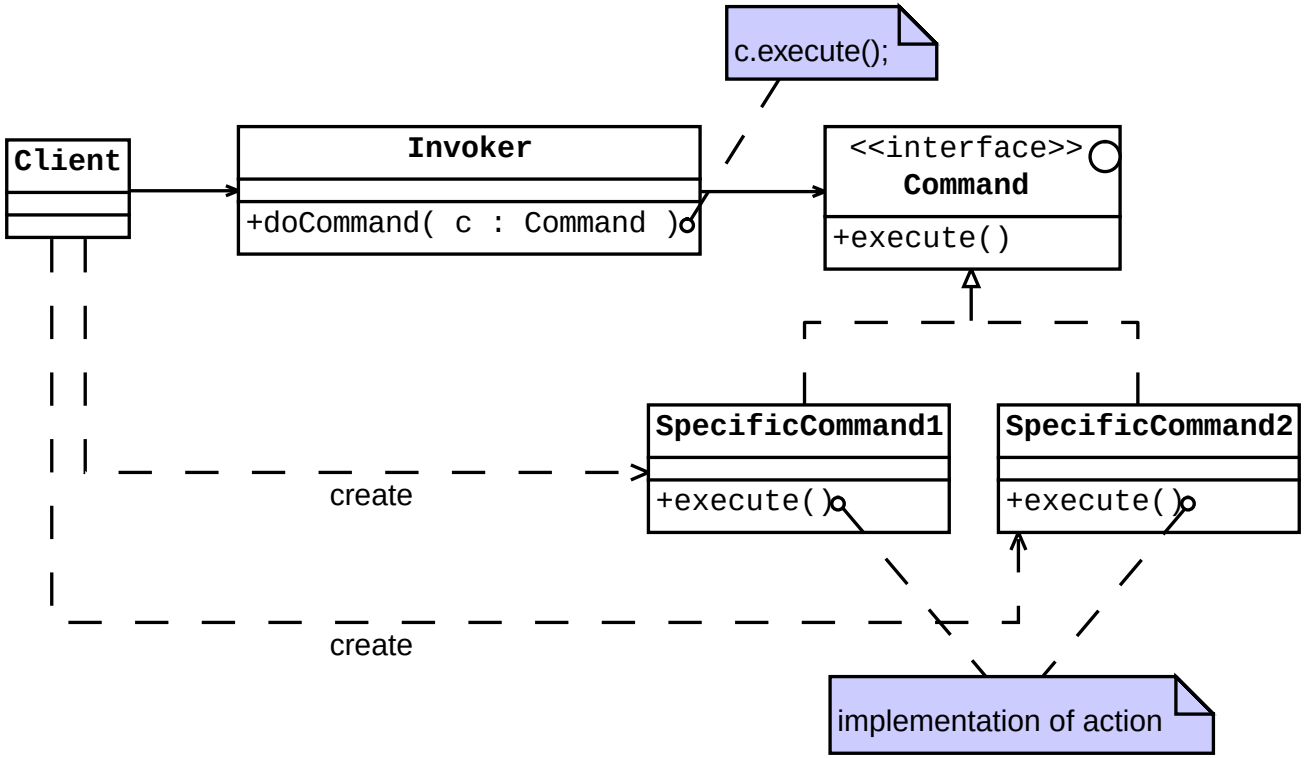
Applying the Command Pattern: Problem Forces

- Undo and redo previously executed actions
- Execute commands asynchronously, remotely, or in a different thread

Applying the Command Pattern: Solution

- Each action is implemented in the `execute` method of its own class that implements the `Command` interface
- The client creates the necessary `SpecificCommand` object and passes it to an `Invoker` object
- The `Invoker` object has no knowledge of what `SpecificCommand` does, it simply invokes the `execute` method

Command Pattern Structure

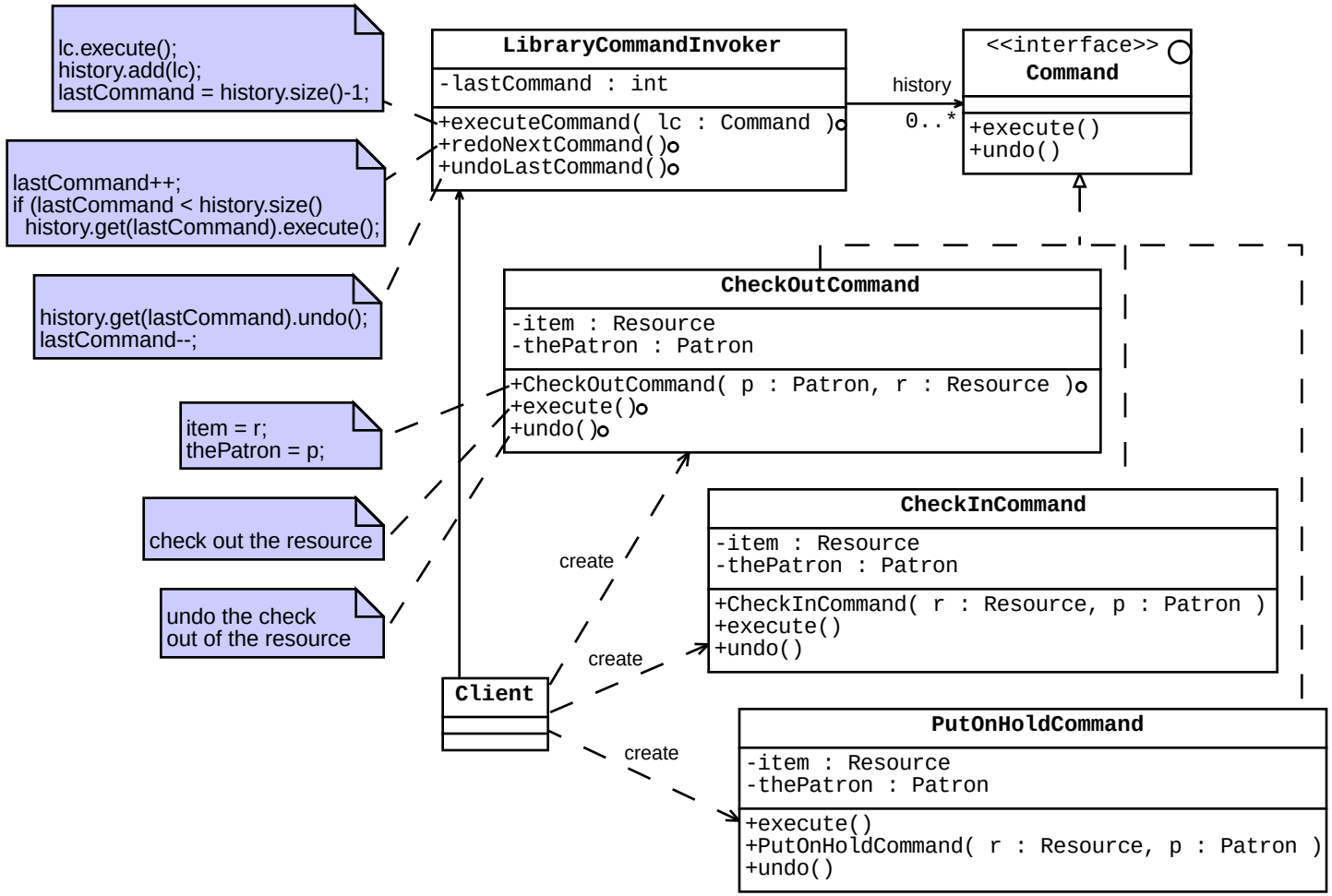


Applying the Command Pattern: Solution

Implementation strategies include:

- The `SpecificCommand` can also contain state data relating to the command
- The `SpecificCommand` can either implement the action itself or call other objects to do the work
- The `Command` interface can also include an `undo` method
- The `Invoker` can be in a separate thread, process, or server from the `Client`
- The `Invoker` can queue the actions and execute them asynchronously

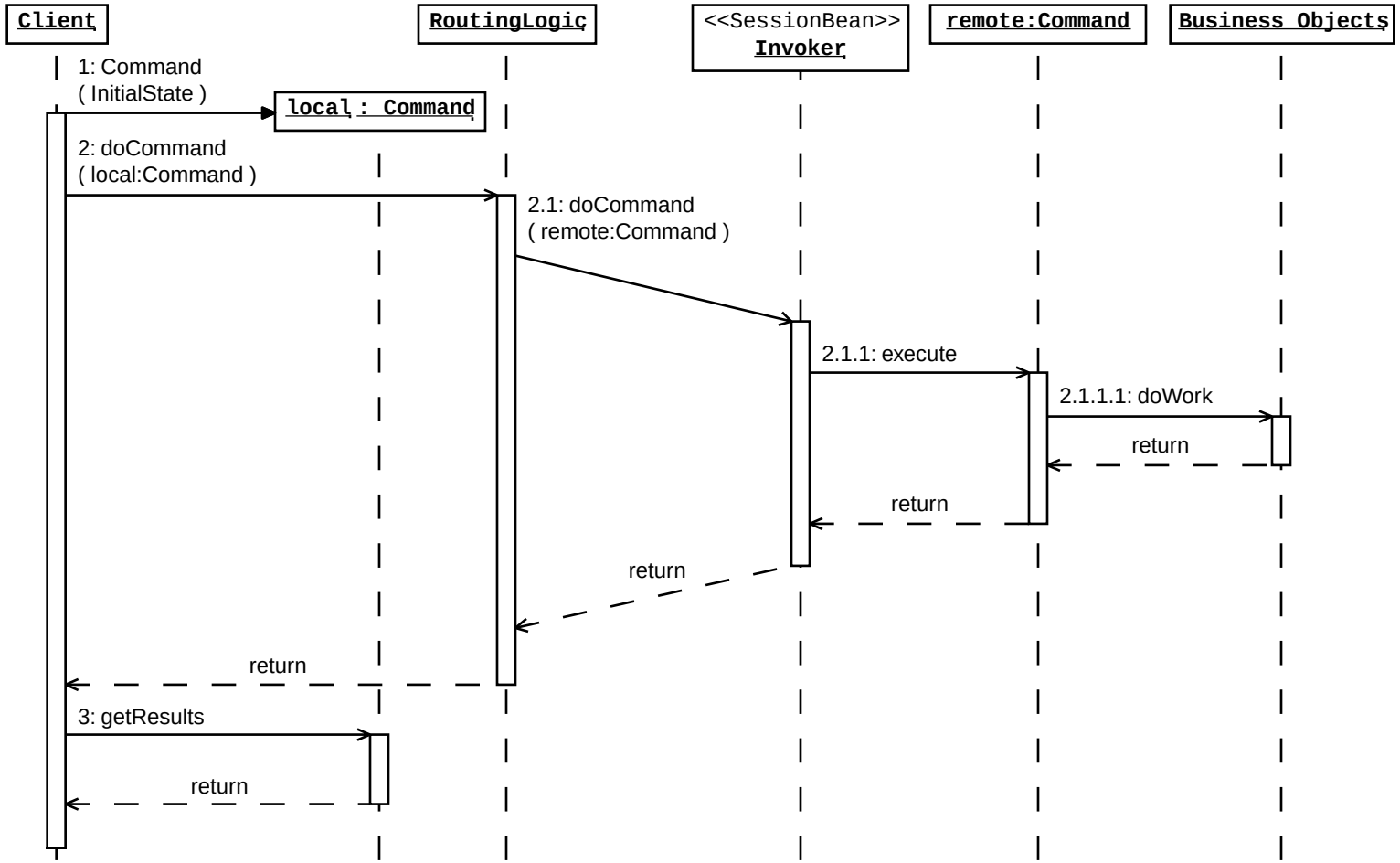
Command Pattern Example Solution



Applying the Command Pattern: Use in the Java™ Programming Language

- GoF Command pattern is used in various enterprise frameworks, such as Struts
- *EJB Design Patterns*, by Floyd Marinescu documents the EJB Command pattern
- EJB Command pattern decouples the client from the business tier and simplifies changing the business logic
- The major distinction between the GoF Command and EJB Command patterns is that `Invoker` is a stateless session bean in the EJB Command pattern

EJB Command Pattern Dynamic Structure



Applying the Command Pattern: Consequences

Advantages:

- Reduces complexity of decision logic
- Provides easy extensibility
- Can readily provide for *undo* operations
- Can place multiple commands on a queue
- Can execute commands in separate threads or remotely

Applying the Command Pattern: Consequences

Disadvantages:

- Might increase overhead in object creation, destruction, and use
- Makes the structure of an application more complex

Applying the Iterator Pattern: Example Problem

- The library system needs to read catalog data from several data sources including a relational database, a proprietary legacy database, and a file of test data
- Each of the data source classes provides various methods for traversing the data
- The traversal logic complicates the data source classes and exposes clients to the implementations of the data source classes

Applying the Iterator Pattern: Problem Forces

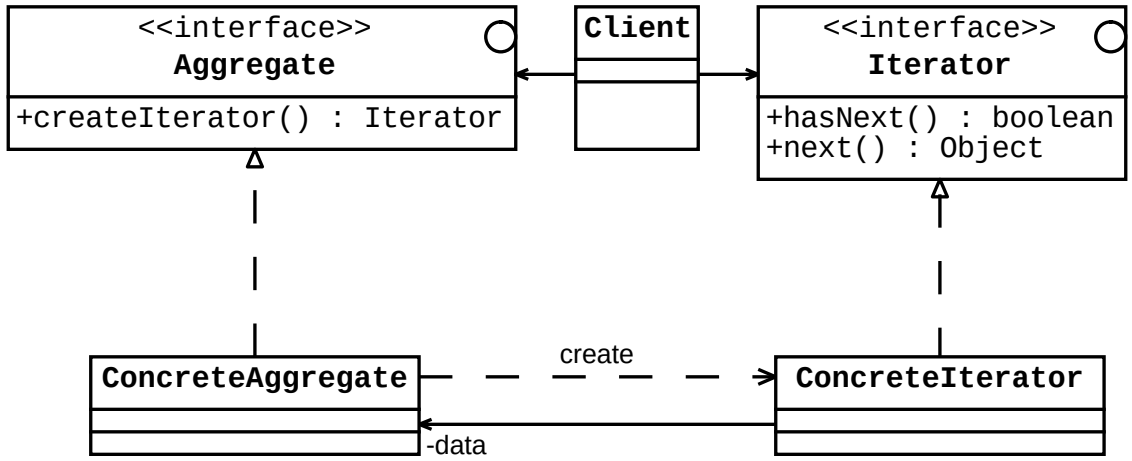
Placing traversal logic directly in a collection or data source is not always desirable:

- Callers become needlessly complex and tightly coupled to the collection if they must use custom traversal logic to access that collection
- Accessing the elements should be easy without a time-consuming lookup
- Multiple traversals should be supported
- For uniformity, different collections should support the same model for traversal

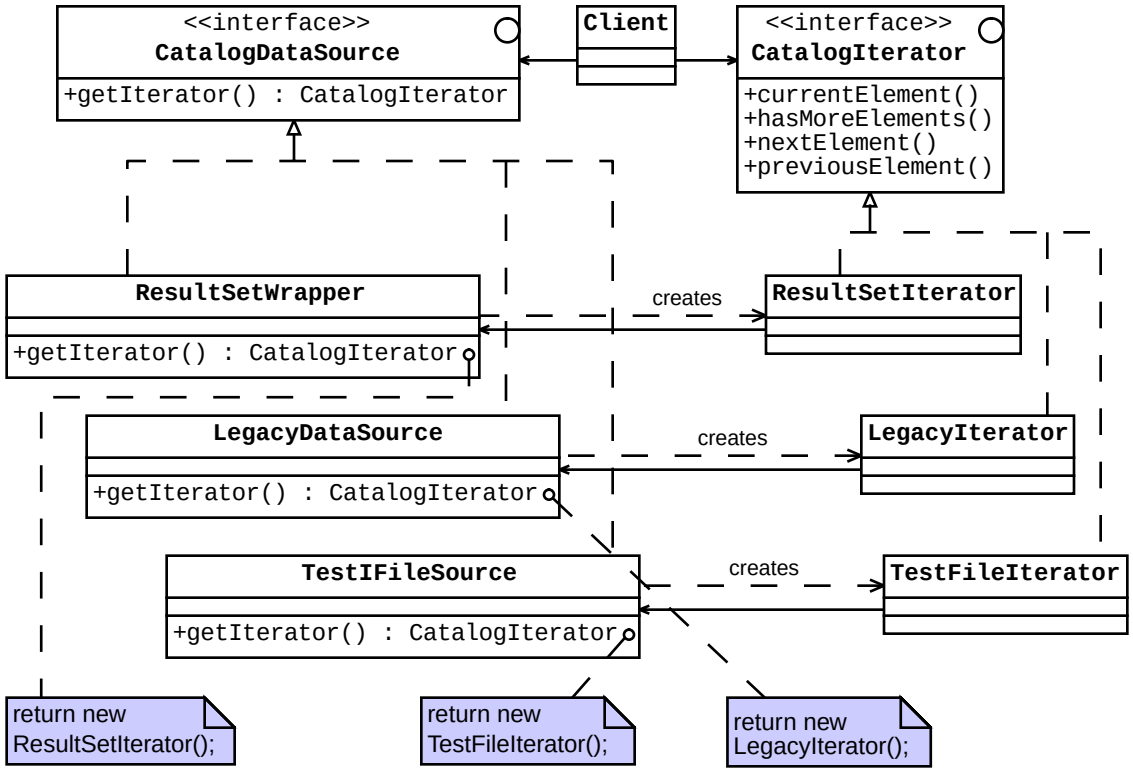
Applying the Iterator Pattern: Solution

- Place the `Aggregate` object's access code in an `Iterator` object
- The `Client` object requests an `Iterator` subtype from the `Aggregate` and uses `Iterator` to advance through the elements of `Aggregate`
- Iterators do not need to iterate over real collections of objects (for example, an input stream can be an iterator source)

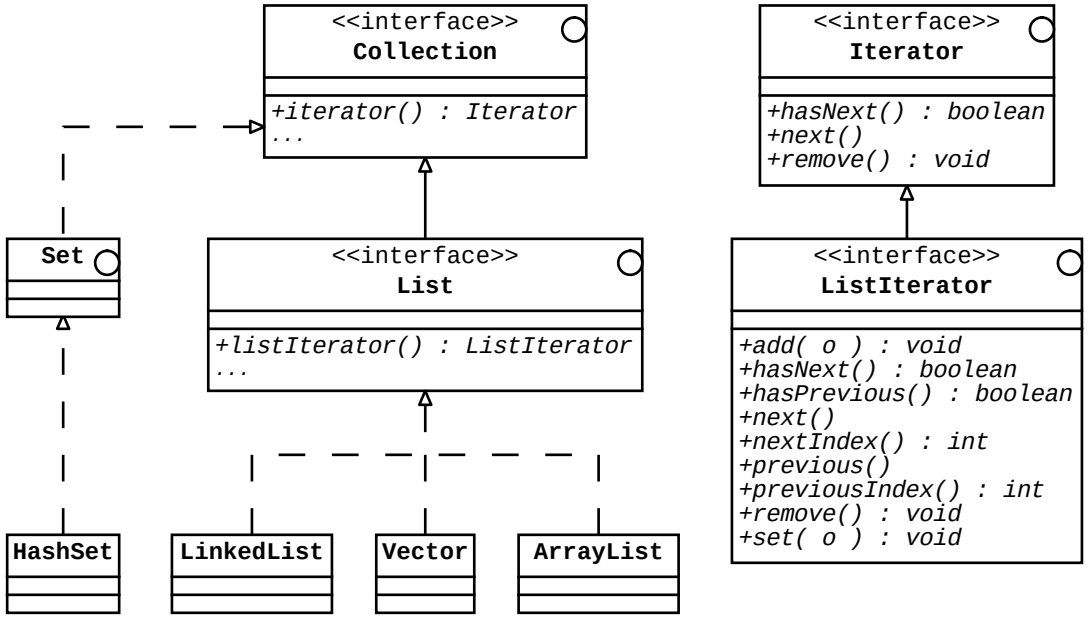
Iterator Pattern Structure



Iterator Pattern Example Solution



Applying the Iterator Pattern: Use in the Java Programming Language



Applying the Iterator Pattern: Use in the Java Programming Language

Collection subtype:

```
1     public void printData(Collection data) {
2       Iterator dataI = data.iterator();
3       while (dataI.hasNext()) {
4         System.out.println(dataI.next());
5       }
6     }
```

Applying the Iterator Pattern: Consequences

Advantages:

- Simplifies the collection interface, lifting out traversal-related methods
- Supports uniform traversal on different collections
- Supports different traversal algorithms on the same collection
- Supports simultaneous multiple traversals

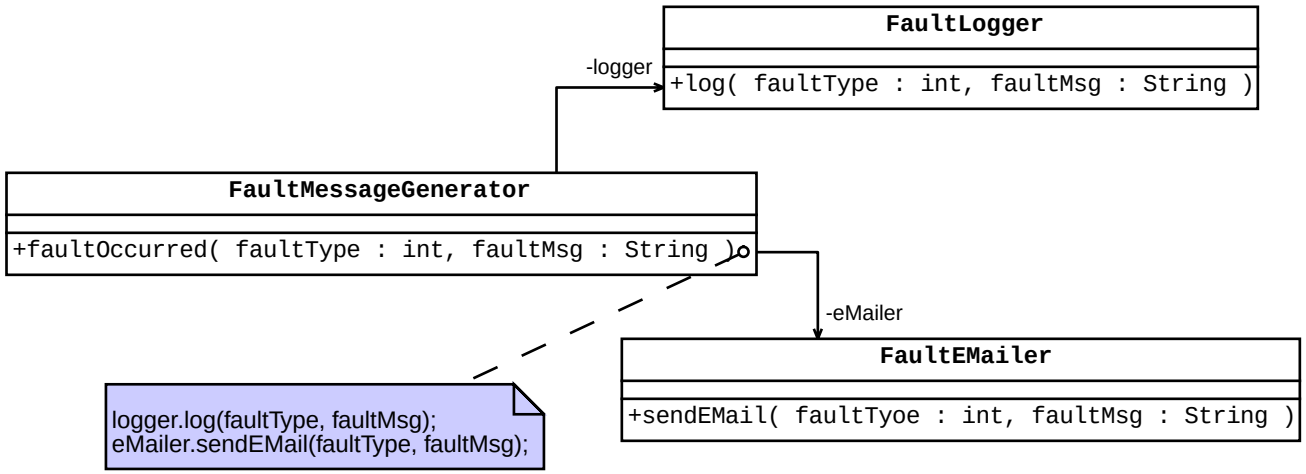
Applying the Iterator Pattern: Consequences

Disadvantages:

- Introduces additional objects at runtime
- Not inherently thread-safe

Applying the Observer Pattern: Example Problem

- When a serious fault occurs in the hotel application, the `faultOccurred` method calls the `log` and `sendEMail` methods in the `FaultLogger` and `FaultEMailer` classes
- You need a way to flexibly add more fault handlers



Applying the Observer Pattern: Problem Forces

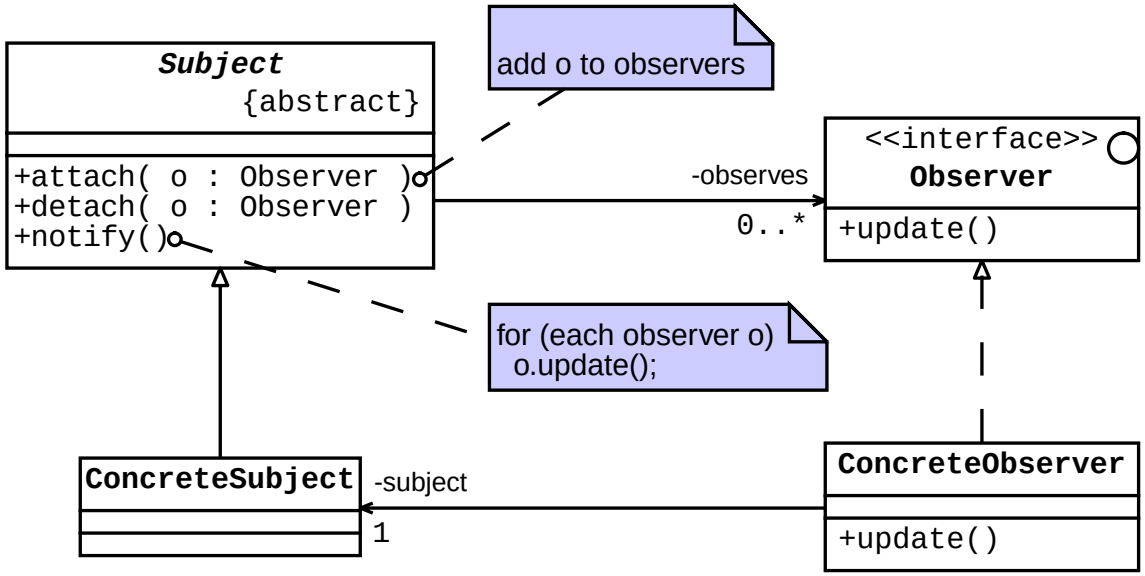
Sometimes several classes need to be aware of changes in another class:

- Polling an object for its state leads to excessive memory and the central processing unit (CPU) resource usage
- Multiple object queries of a single object can lead to synchronization problems
- The observer objects must be aware of the observable object; for notification, the observable object must be aware of its observers
- The observable object should not be coupled to the types of observer objects

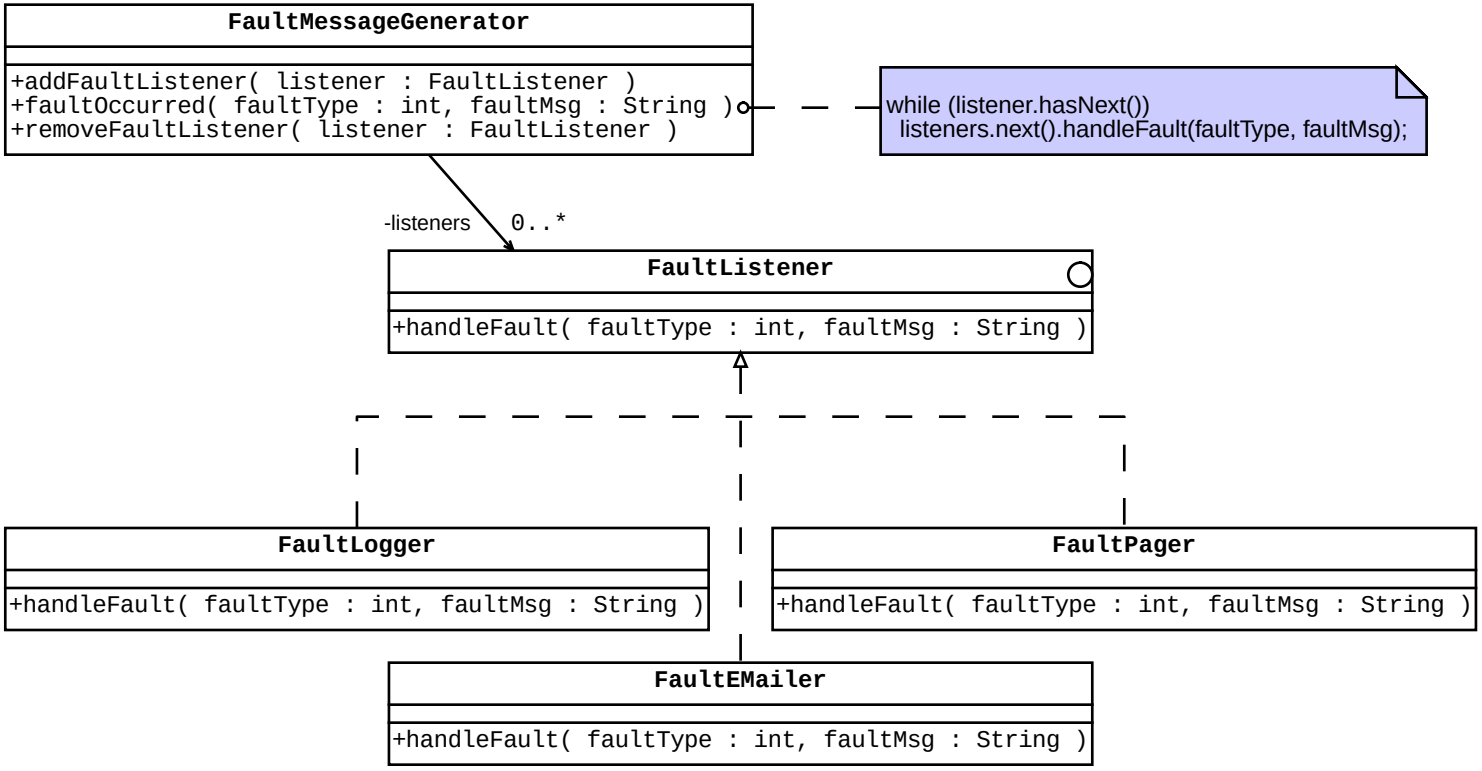
Applying the Observer Pattern: Solution

- An interface is provided through which observers can be called, irrespective of their concrete class types
- The `Subject` object has a collection of `Observer` objects referenced by an `Observer` interface reference
- The `Subject` object has methods that allow an `Observer` object to register or de-register with the `Subject`
- When the `Subject` object changes, it calls a method in each `Observer` object in the `observers` collection
- The Observer pattern functionality of the `ConcreteSubject` class can be placed in a superclass so that `Observer` objects might not need to be coupled to the `ConcreteSubject`

Observer Pattern Structure



Observer Pattern Example Solution



Observer Pattern Example Solution

Observable class:

```
1   import java.util.*;
2   public class FaultMessageGenerator {
3     private ArrayList listeners = new ArrayList();
4
5     public void addFaultListener(FaultListener listener) {
6       listeners.add(listener);
7     }
8     public void faultOccurred(int faultType, String faultMsg) {
9       Iterator listenersI = listeners.iterator();
10      while(listenersI.hasNext()) {
11        FaultListener fl = (FaultListener) listenersI.next();
12        fl.handleFault(faultType,faultMsg);
13    } }
14    public void removeFaultListener(FaultListener listener) {
15      listeners.remove(listener);
16    }
17  }
```

Observer Pattern Example Solution

Observer interface:

```
1   public interface FaultListener {
2     public void handleFault(int faultType, String faultMsg);
3   }
```

Observer Pattern Example Solution

Logging observer:

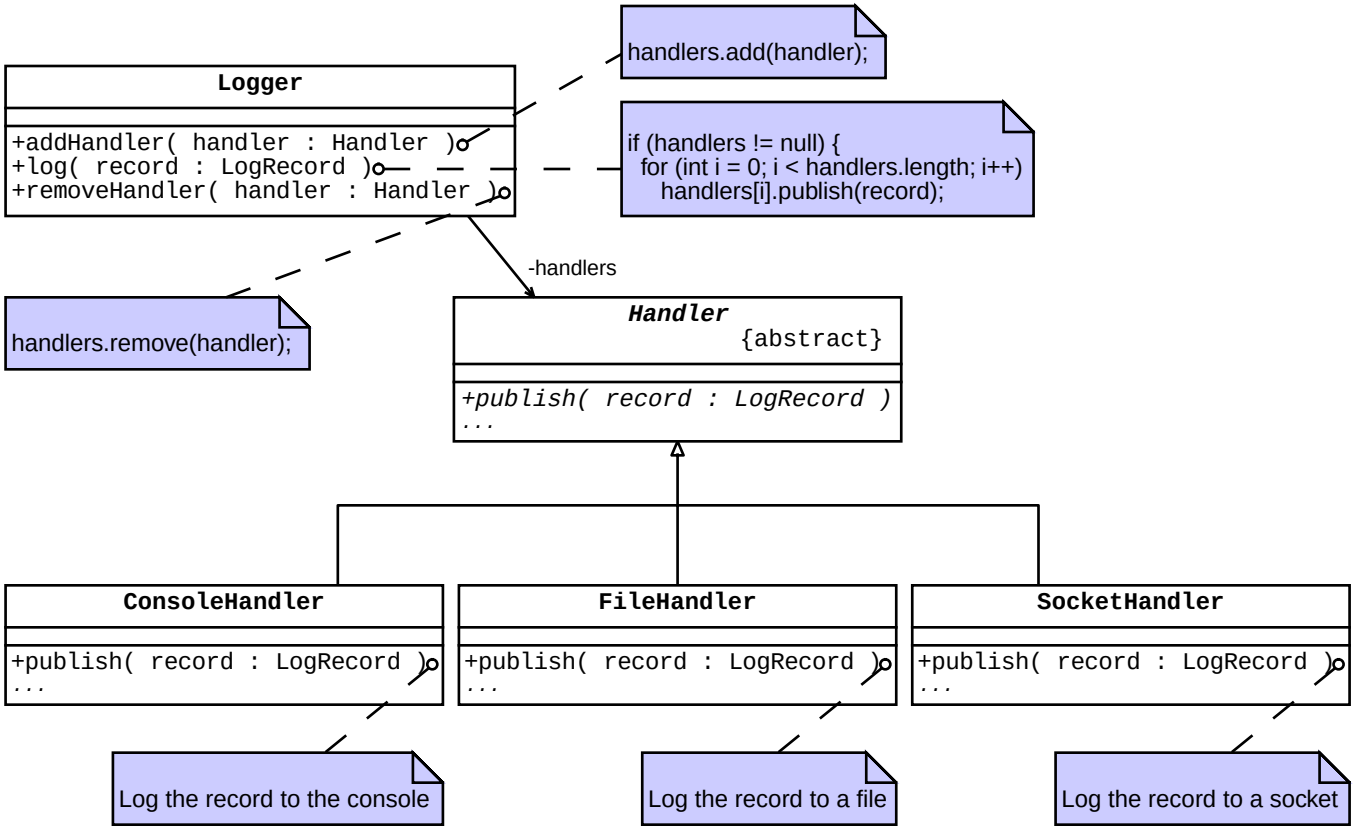
```
1   import java.util.logging.*;
2
3   public class FaultLogger implements FaultListener{
4     private static Logger logger = Logger.getLogger("");
5
6     public FaultLogger(FaultMessageGenerator fmg) {
7       fmg.addFaultListener(this);
8     }
9     public void handleFault(int faultType, String faultMsg) {
10      logger.log(Level.WARNING,
11          "A " + faultType + " occurred: " + faultMsg);
12    }
13  }
```

Observer Pattern Example Solution

Simple test class:

```
1   public class FaultTester {
2     public static void main(String args[]) {
3       FaultMessageGenerator fmg = new FaultMessageGenerator();
4       new FaultLogger(fmg);
5       new FaultPager(fmg);
6       new FaultEMailer(fmg);
7       fmg.faultOccurred(72,"Lost customer data: Peters, V.");
8     }
9   }
```

Applying the Observer Pattern: Use in the Java Programming Language



Applying the Observer Pattern: Consequences

Advantages:

- Subjects and observers can be reused independently
- Coupling is between abstract classes (or interfaces) and not between concrete classes
- Observers are easily changed without changing the subject

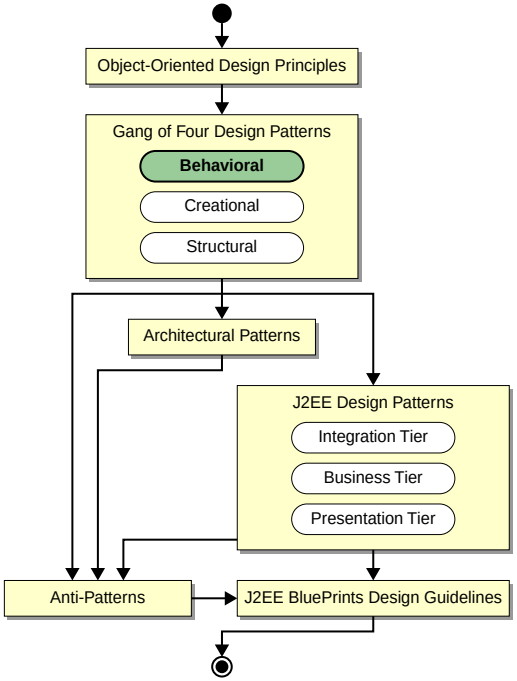
Applying the Observer Pattern: Consequences

Disadvantages:

- An observer system can become flooded with updates
- The observer often calls back to the subject when it receives a notification, which might create a bottleneck

Summary

Behavioral patterns focus on algorithms and the distribution of responsibilities between classes:



- **Strategy** – Encapsulates a family of algorithms for interchangeable use
- **Command** – Encapsulates requests in objects for execution by other objects
- **Iterator** – Traverses any collection in a loosely coupled way
- **Observer** – Provides notification of events without polling

Module 3

Using Gang of Four Creational Patterns

Objectives

- Describe the basic characteristics of the Creational patterns
- Apply the Factory Method pattern
- Apply the Abstract Factory pattern
- Apply the Singleton pattern

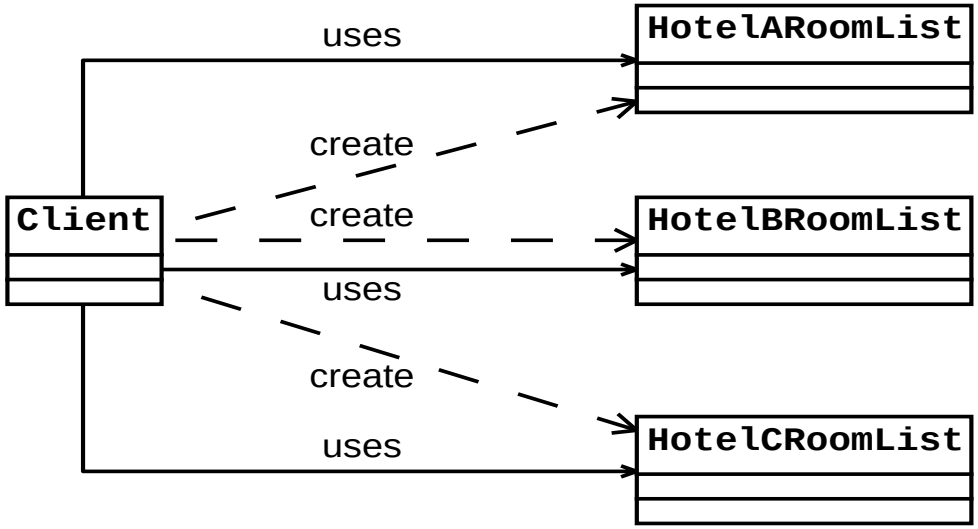
Introducing Creational Patterns

Creational patterns hide the details of object creation.

Pattern	Primary Function
Factory Method	Creates objects of a class or its subclasses through a method call
Abstract Factory	Creates a family of objects through a single interface
Singleton	Restricts a class to one globally accessible instance

Factory Method Pattern Example Problem

- Three hotel systems have been merged
- The `Client` class must directly decide which `HotelRoomList` object to create
- The `Client` is tightly coupled to `HotelRoomList` classes due to the creation process



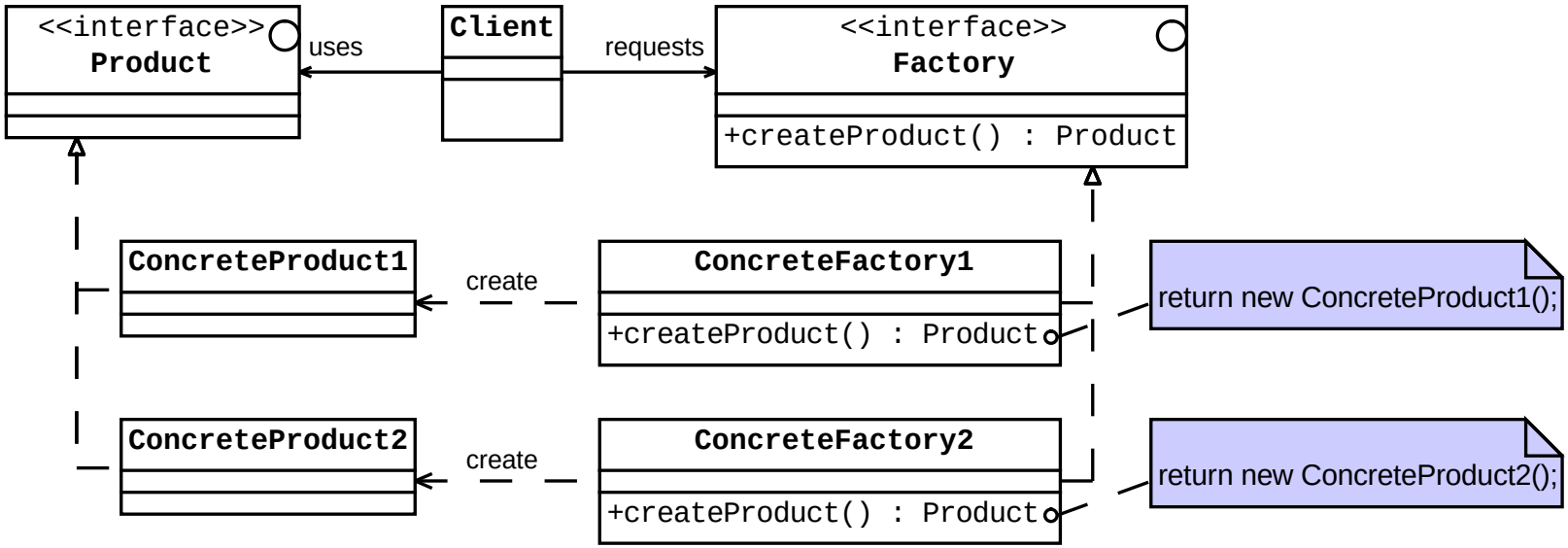
Applying the Factory Method Pattern: Problem Forces

- Clients that call target constructors are tightly coupled to the constructors, limiting flexibility of the client code
- Reusable frameworks must allow you to add new concrete subclasses

Applying the Factory Method Pattern: Solution

- The `Client` class obtains a generic `Factory` reference to a `ConcreteFactory` object
- The `Client` calls the `createProduct` factory method on the `ConcreteFactory`
- The factory method returns a `Product` reference to a new `ConcreteProduct`
- If the `Client` obtains a reference to a different `ConcreteFactory`, it will receive a different `ConcreteProduct`

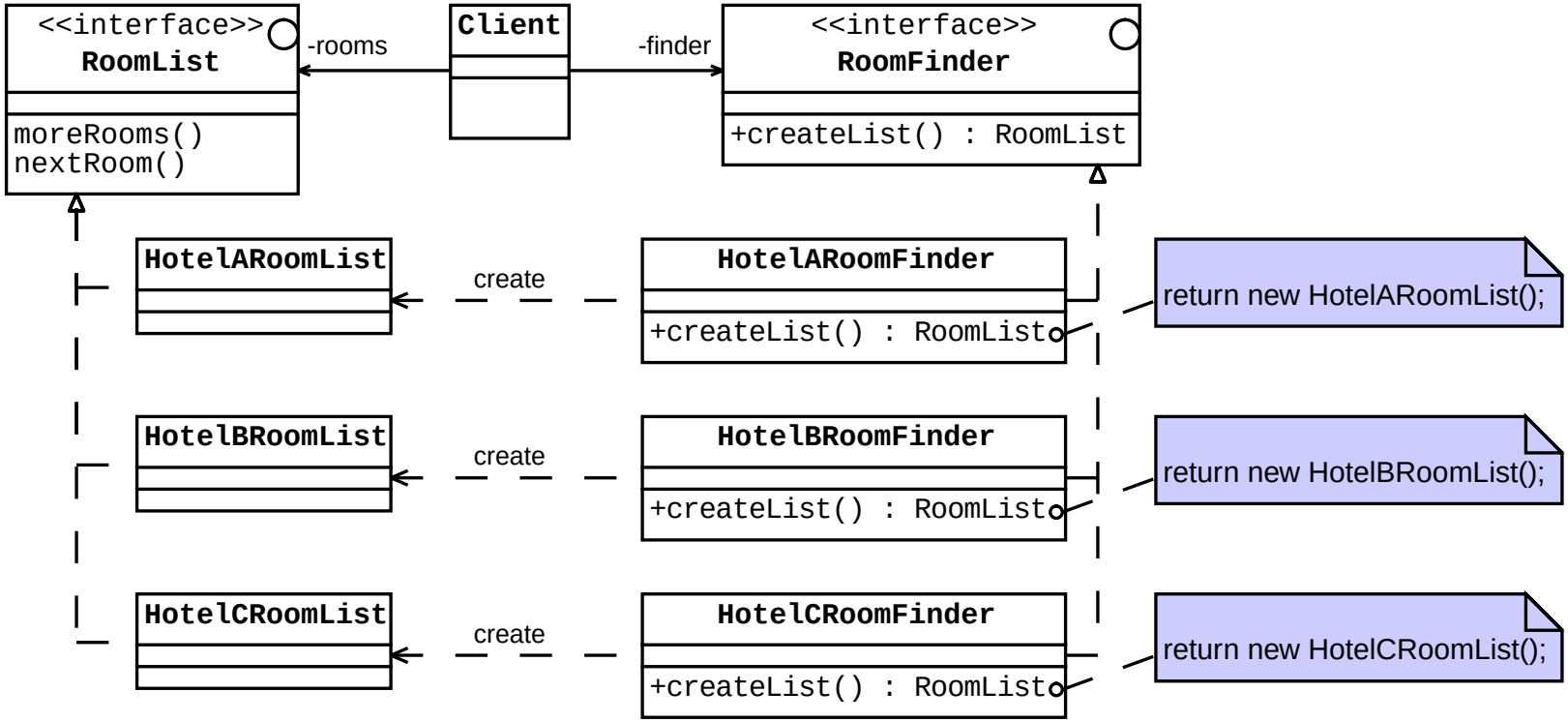
Factory Method Pattern Structure



The client code might look like:

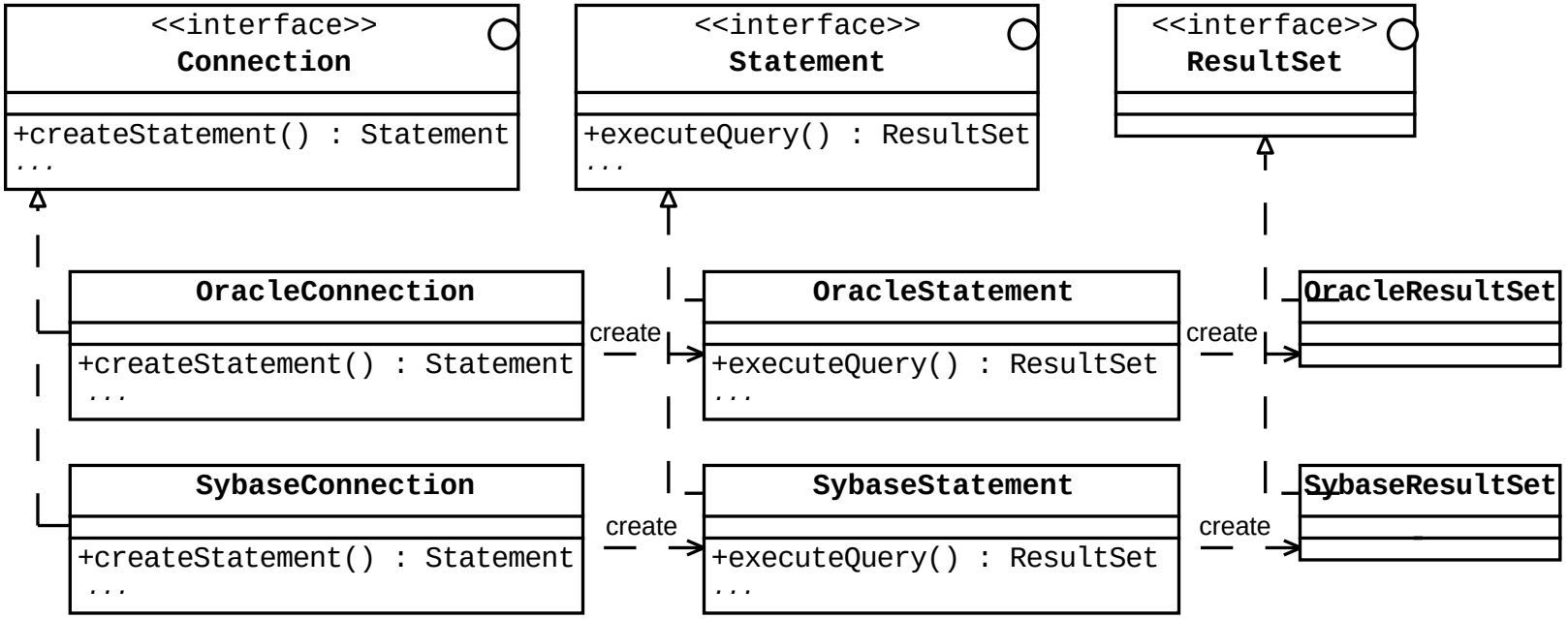
```
Factory f = new ConcreteFactory1();
Product p = f.createProduct();
```

Factory Method Pattern Example Solution



JDBC API Factory Method Scheme Class Diagram

The JDBC API uses two factory methods for creation of `Statement` and `ResultSet` objects



Applying the Factory Method Pattern: Consequences

Advantages:

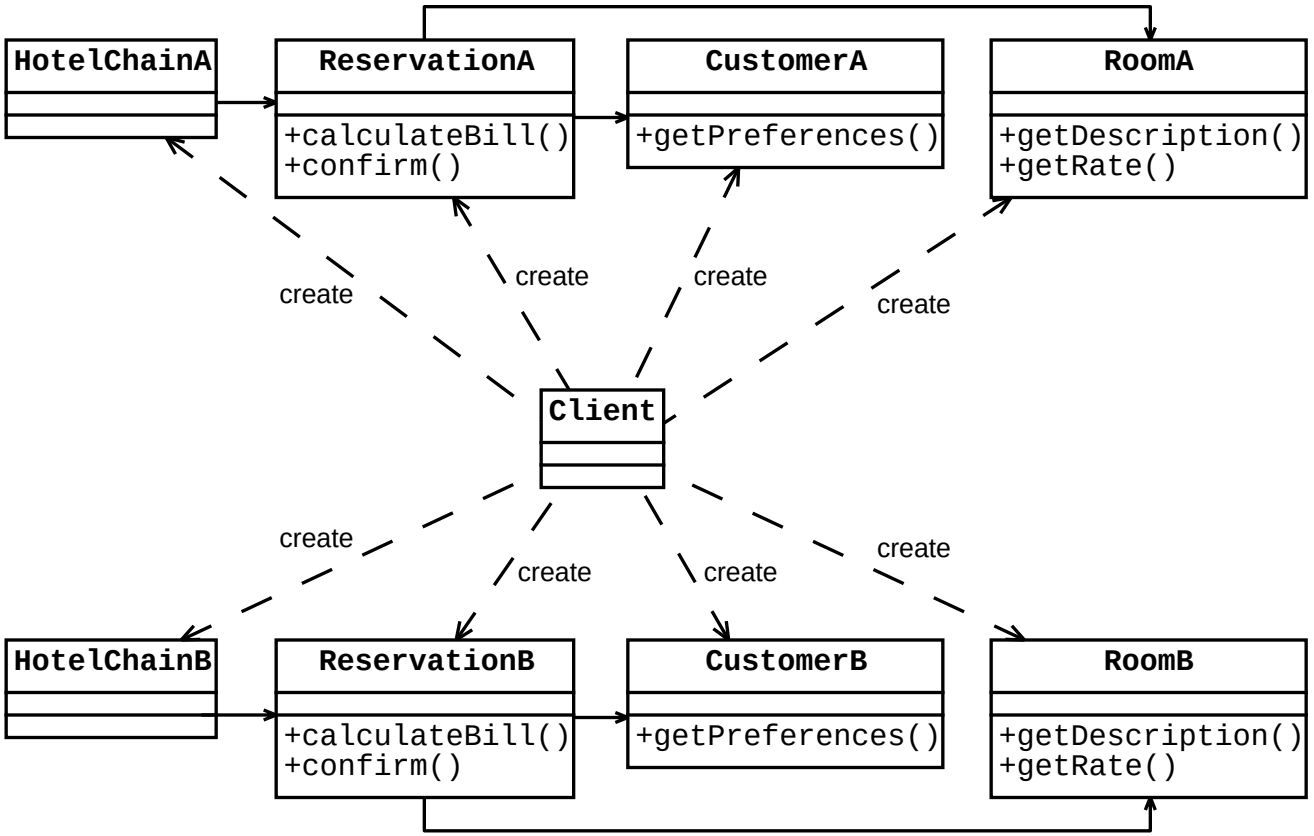
- Factories restore control of object creation to the system; clients merely request objects
- Client code works with any subclass without knowing the name or implementation details

Disadvantage:

You have to subtype the factory interface to create new product types

Abstract Factory Pattern Example Problem

The `Client` class must create a family of hotel objects



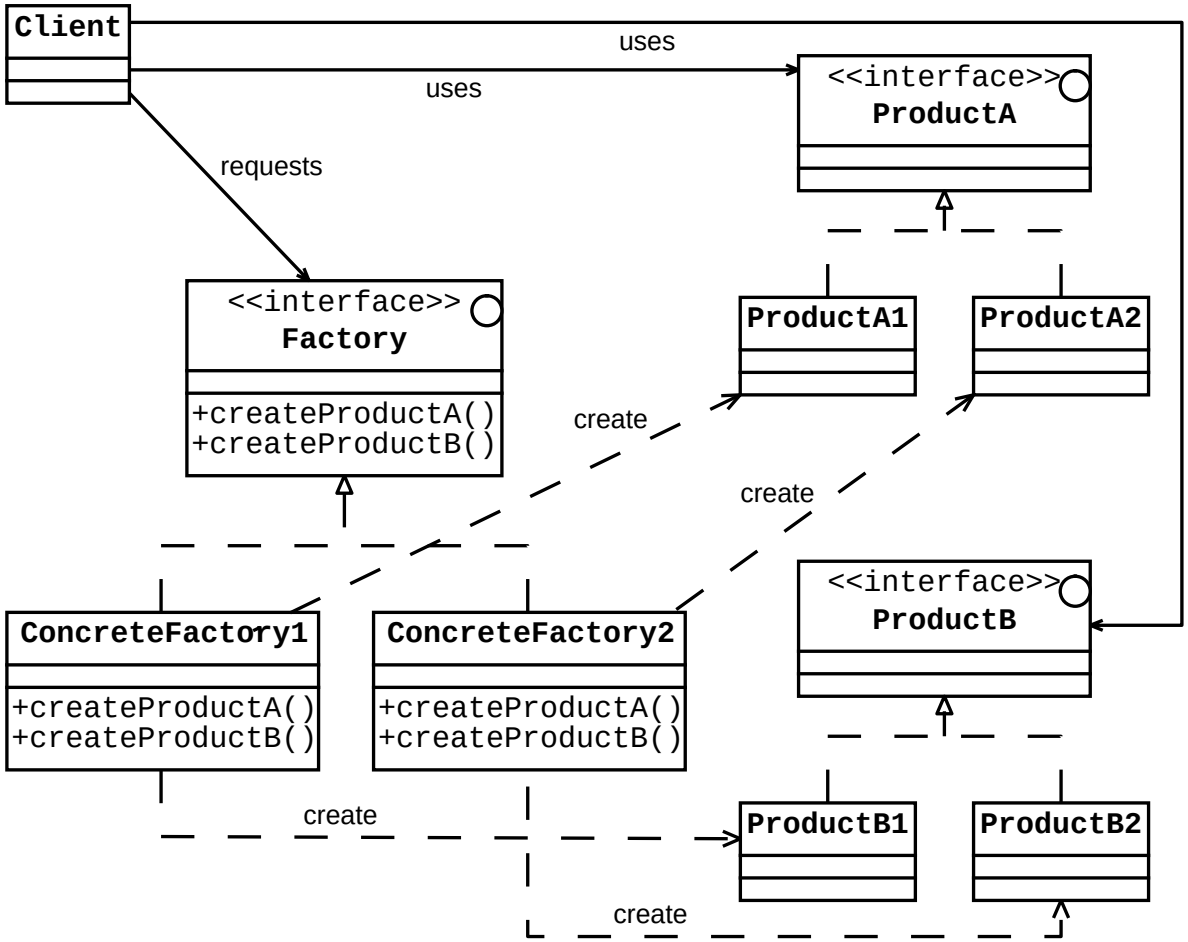
Applying the Abstract Factory Pattern: Problem Forces

- The client does not need to know how to instantiate products
- A client needs to use a family of related objects
- You might not want a client to mix objects from different families
- Client maintainability is sacrificed when you expose details of the related objects to a client

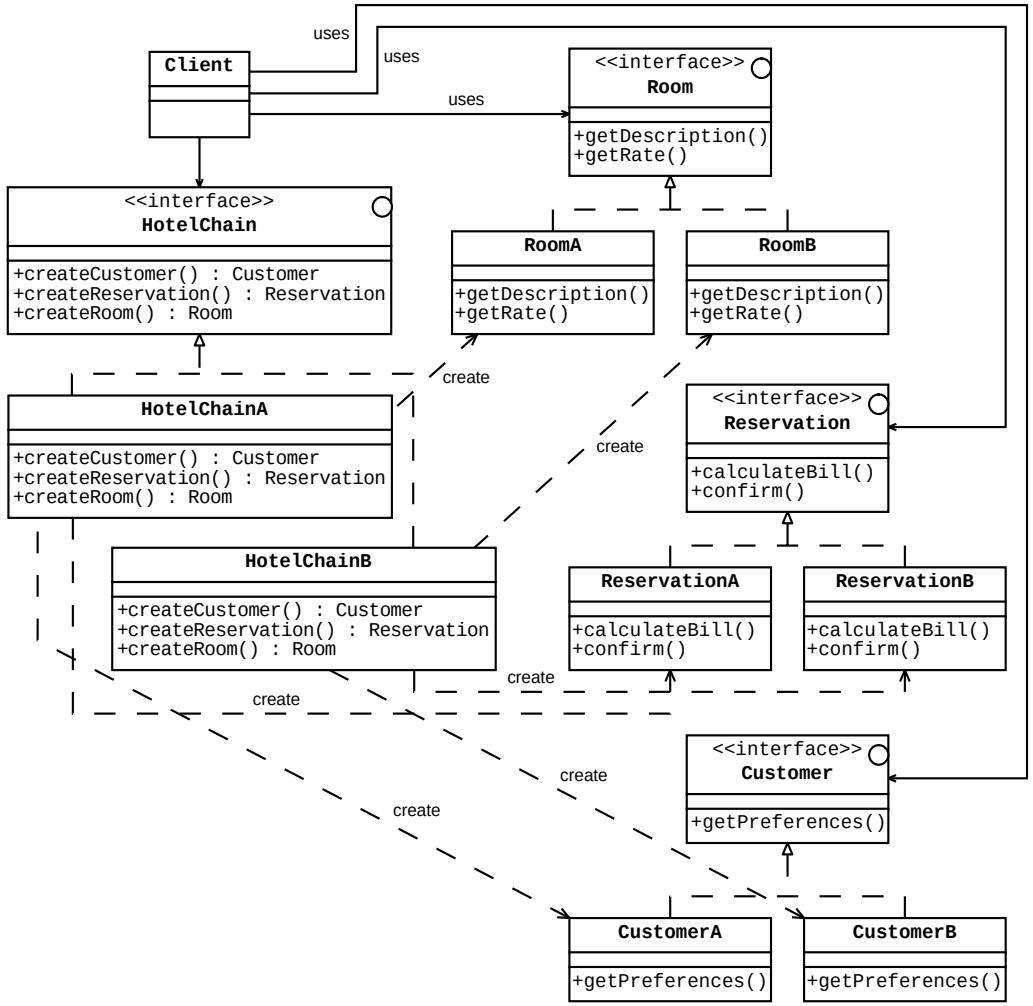
Applying the Abstract Factory Pattern: Solution

- The client obtains an `AbstractFactory` reference to a `ConcreteFactory` object
- The `ConcreteFactory` object has multiple factory methods that produce different objects within a family of products
- By obtaining a different `ConcreteFactory`, the `Client` is switched to a different family of products

Abstract Factory Pattern Structure



Abstract Factory Pattern Example Solution



Applying the Abstract Factory Pattern: Consequences

Advantages:

- Simplifies exchanging one product family for another
- Hides concrete family classes from the client view
- Imposes a consistent naming convention for family members

Disadvantages:

- Developing new families correctly can be time consuming
- Adding family members to an existing scheme can be difficult

Applying the Singleton Pattern: Example Problem

- The hotel system uses a pool of threads to invoke processing
- There should only be one instance of the thread pool so that all parts of the application are borrowing threads from the same pool
- The thread pool must be easily accessible throughout the application

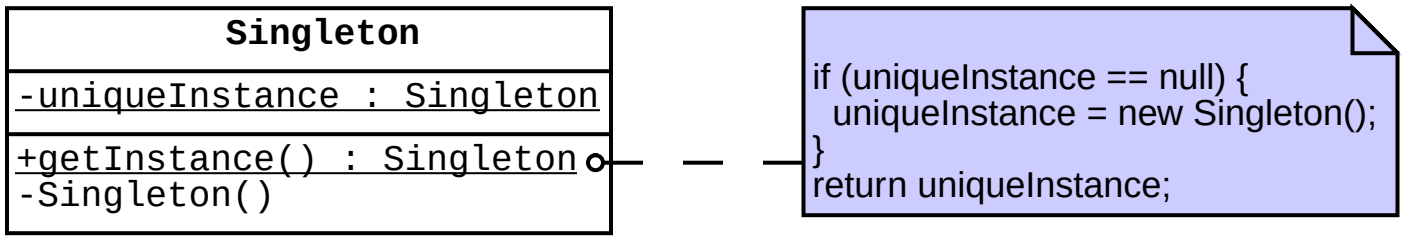
Applying the Singleton Pattern: Problem Forces

- There are some classes that are intended to only have a single instance
- The single instance of a class may need to be easily accessible to different parts of the application
- The only way to secure a constructor is to hide it

Singleton Pattern Structure

Three steps are needed to create a `Singleton` class:

1. The `Singleton` class' constructor is private or protected, hiding it from client view
2. The `Singleton` class has a private static reference to the only instance of itself
3. The `Singleton` class has a public static method that returns the single instance

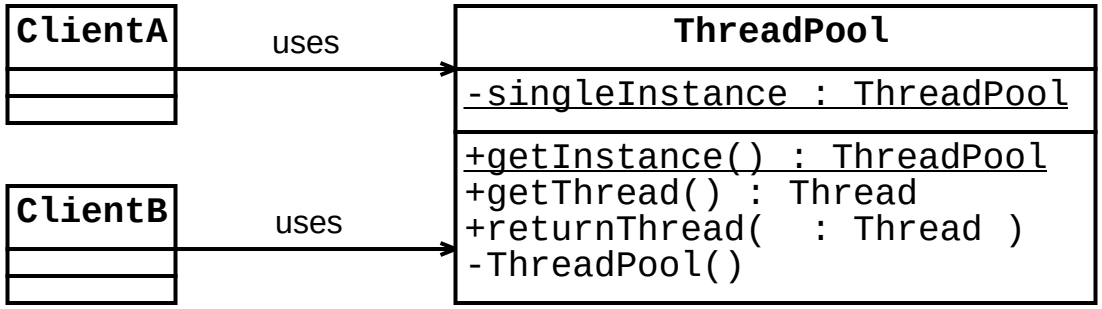


Singleton Pattern Structure

Singleton structure code:

```
1   public class Singleton {
2     private static Singleton uniqueInstance;
3
4     public static Singleton getInstance( ) {
5       if (uniqueInstance == null) {
6         uniqueInstance = new Singleton();
7       }
8       return uniqueInstance;
9     }
10    private Singleton( ) {
11    }
12    //Add attributes and methods for class functionality
13  }
```

Singleton Pattern Example Solution



Applying the Singleton Pattern: Consequences

Advantages:

- A client can easily obtain a reference to the single object
- A singleton can construct the instance at the class loading time or on-demand
- Singleton can guard several instances at once, not just one

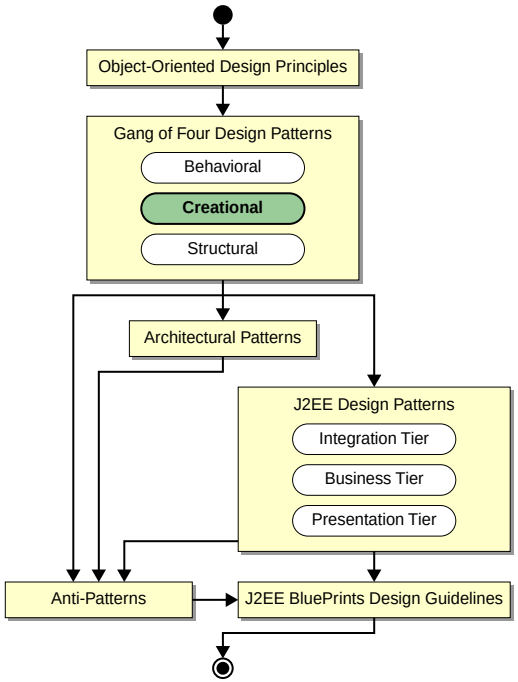
Disadvantages:

- Singleton can replace global variables, but this hint is often misapplied
- Singleton must make on-demand instances thread-safe

Summary

Gang of Four creational patterns hide the details of object creation:

- **Factory Method** – Creates objects of a class or its subclasses through a method call
- **Abstract Factory** – Creates a family of objects through a single interface
- **Singleton** – Restricts object creation to a single globally accessible instance



Module 4

Using Gang of Four Structural Patterns

Objectives

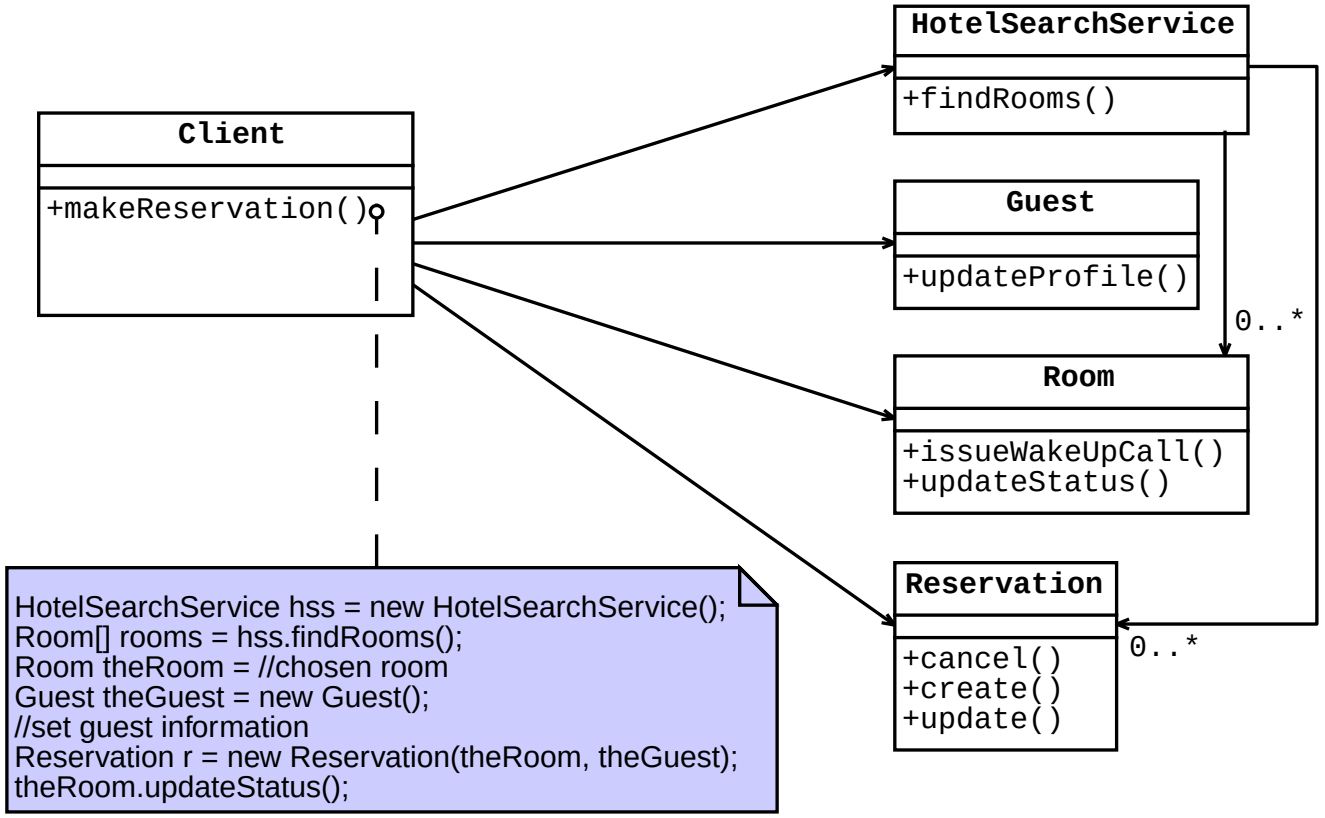
- Describe the basic characteristics of the Structural patterns
- Apply the Façade pattern
- Apply the Proxy pattern
- Apply the Adapter pattern
- Apply the Composite pattern
- Apply the Decorator pattern

Introducing Structural Patterns

Pattern	Primary Function
Façade	Provides a simplified interface to a subsystem.
Proxy	Provides an intermediate object which controls access to another object.
Adapter	Enables a caller to use an object that has an incompatible interface.
Composite	Composes objects into part-whole tree structures.
Decorator	Attaches new functionality to an object dynamically.

Façade Pattern Example Problem

Client is coupled to many classes in reservation subsystem.



Applying the Façade Pattern: Problem Forces

When clients directly link to many classes:

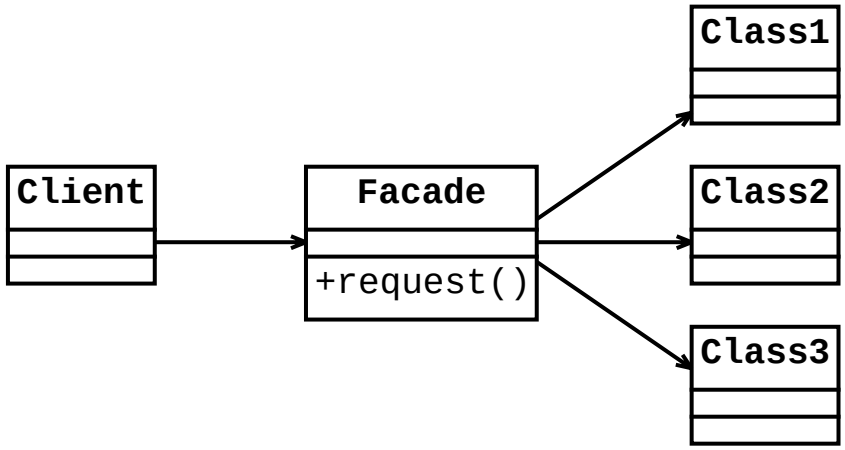
- Client classes are vulnerable to changes in any of those classes
- The programmer must be familiar with the interfaces of many classes
- The client must call the subsystem operations in the correct order and pass state information between the operations
- In a distributed environment, client calls to subsystem objects can increase network overhead

Applying the Façade Pattern: Solution

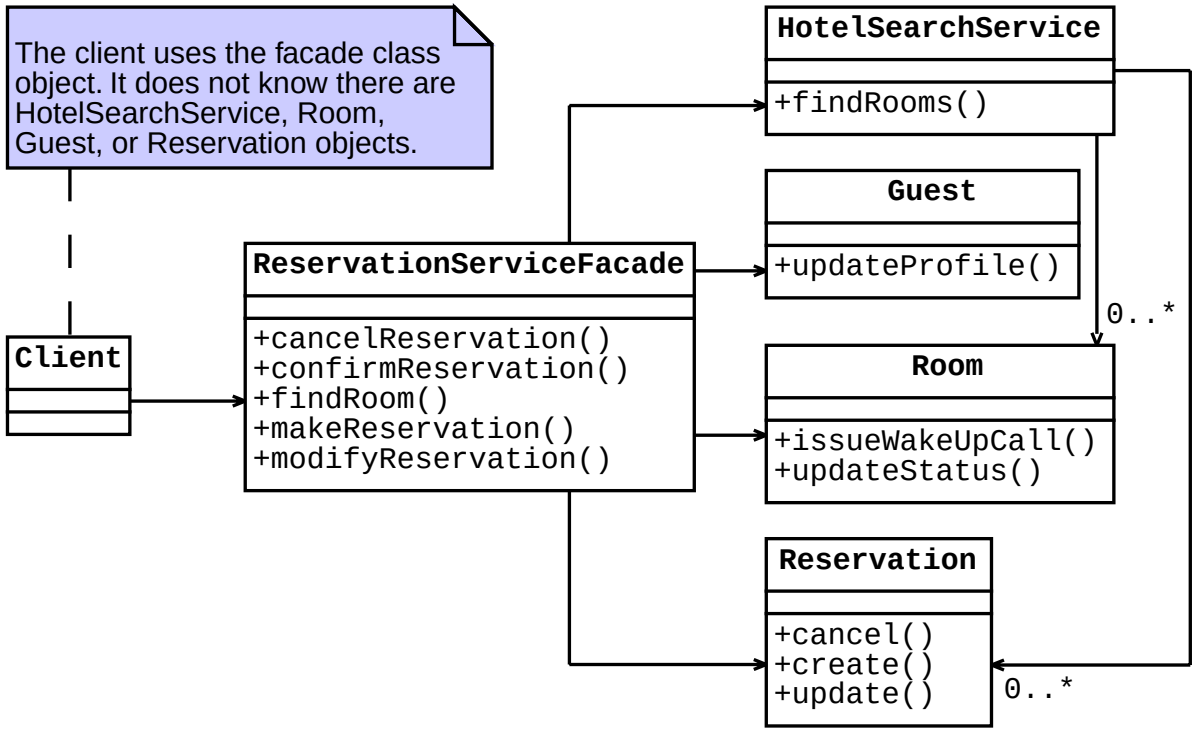
Provide a `Facade` class that has a simple, high-level interface to a subsystem:

- Clients use a `Facade` object instead of communicating with each subsystem object individually
- The `Facade` contains the logic required for sequencing, managing state, and communicating with each subsystem object
- The `Client` may also be allowed to communicate directly with other subsystem classes for additional functionality

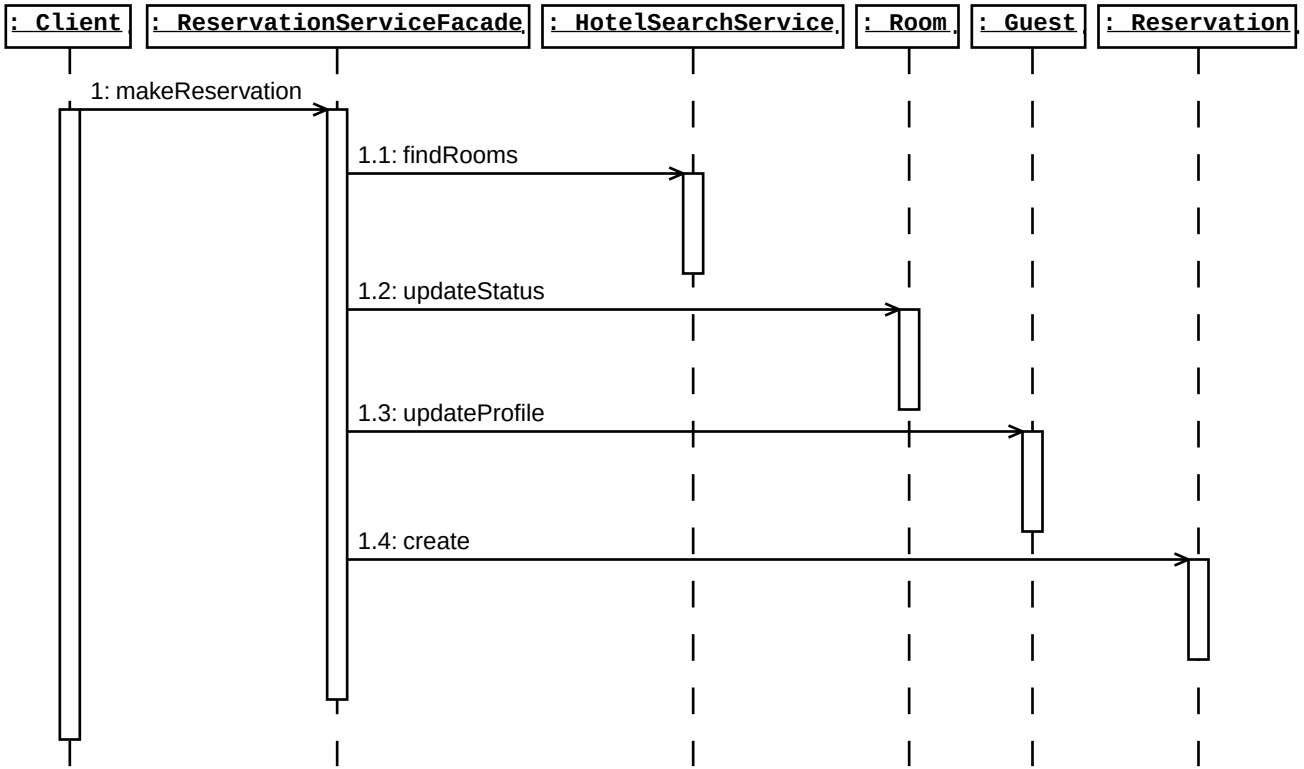
Façade Pattern Structure



Façade Pattern Example Solution



Façade Pattern Example Sequence



Applying the Façade Pattern: Consequences

Advantages:

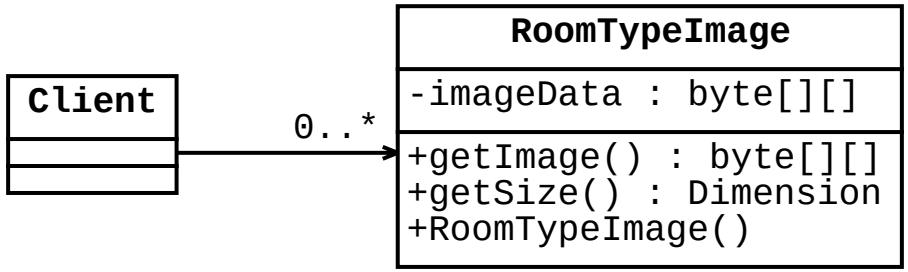
- Clients are shielded from subsystem components, making the clients less brittle
- Network overhead may be decreased in a distributed environment
- Clients are loosely-coupled to subsystems
- Clients can still communicate directly with subsystems

Disadvantage:

Introduces an extra level of indirection, which might impact performance.

Proxy Pattern Example Problem

- `RoomTypeImage` objects are expensive objects that should not be created until they are actually needed
- You might want to avoid having the `Client` class decide when to create expensive `RoomTypeImage` objects



Applying the Proxy Pattern: Problem Forces

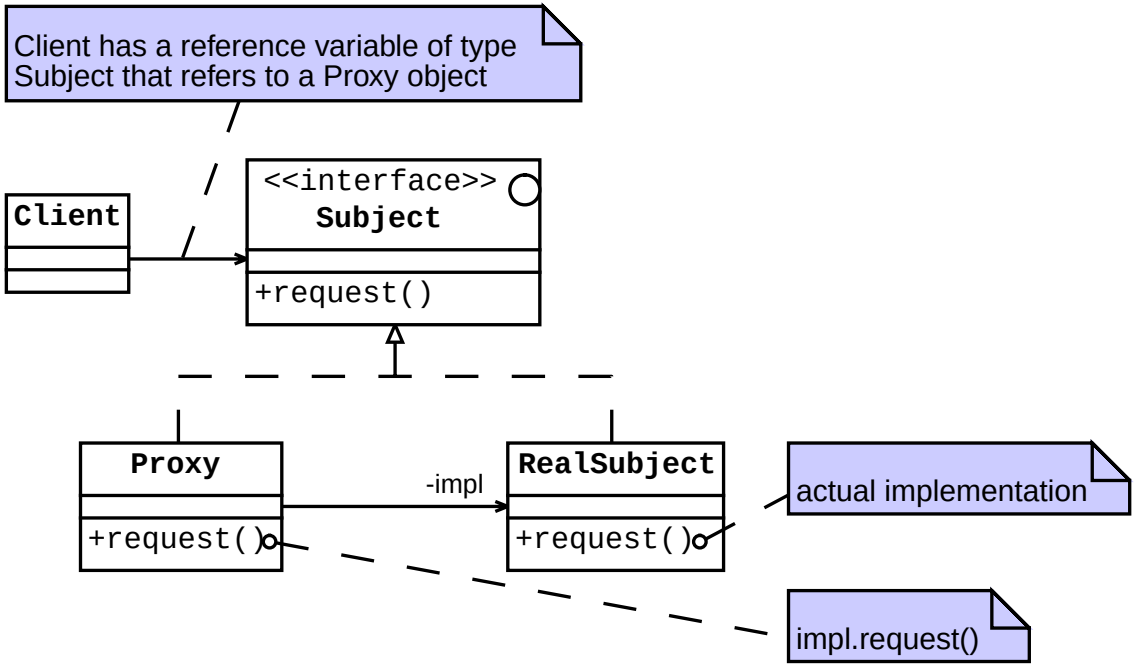
It may be undesirable for a client to directly access these types of classes:

- **Remote objects** – Placing networking code in the client and remote class decreases cohesion
- **Expensive objects** – Creating resource intensive objects before they are actually needed can decrease performance
- **Restricted objects** – Placing security code in a restricted class decreases cohesion

Applying the Proxy Pattern: Solution

- The `Client` uses a `Proxy` object that stands in for the `RealSubject` object
- The `Proxy` and `RealSubject` classes implement the `Subject` interface
- The client has a reference of the `Subject` interface type which only refers to `Proxy` objects
- The `Proxy` object keeps a reference to `RealSubject`
- The `Proxy` object propagates requests to `RealSubject` as appropriate

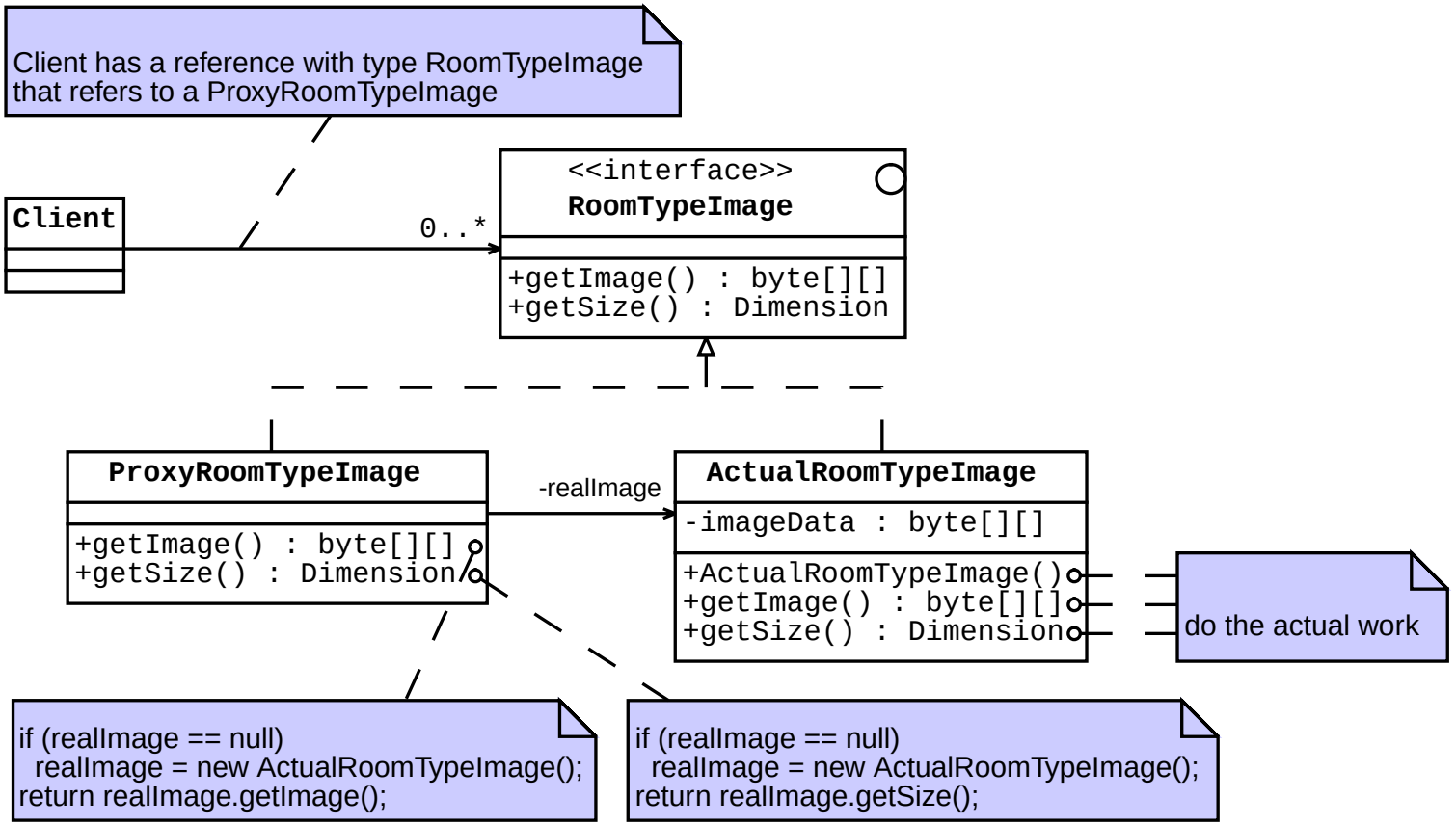
Proxy Pattern Structure



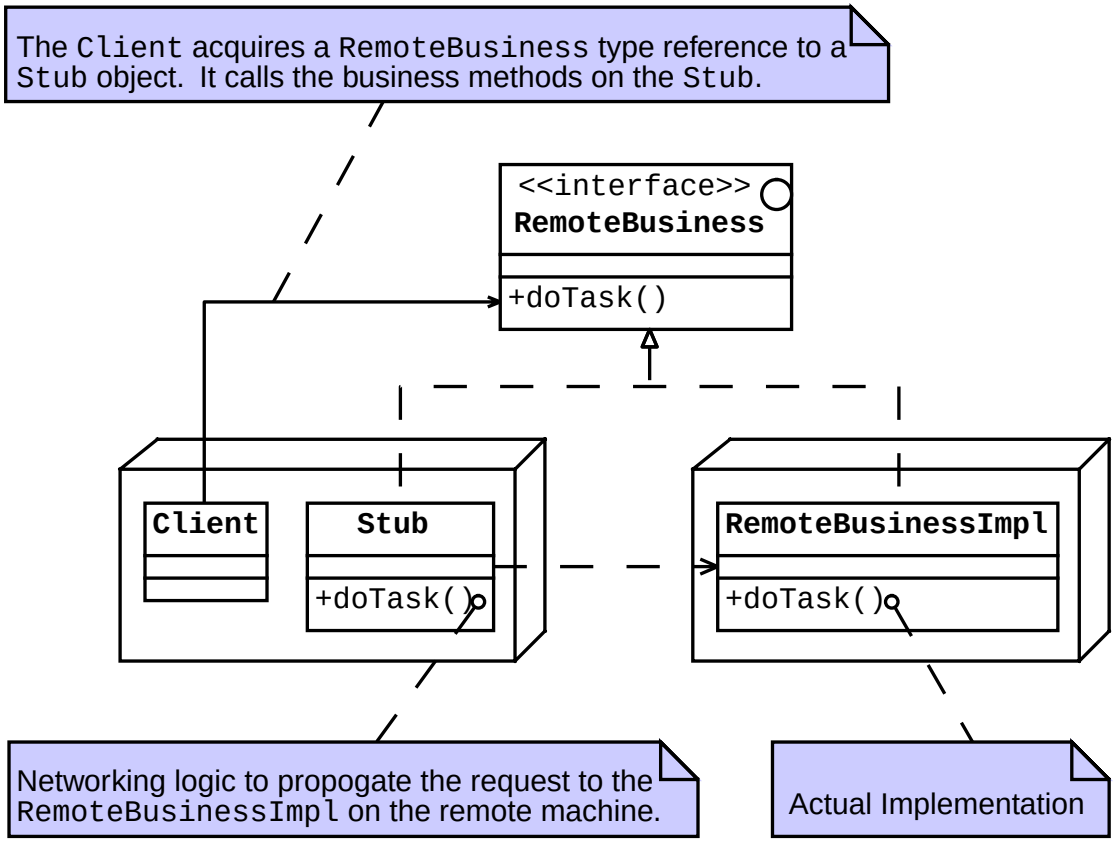
Applying the Proxy Pattern: Solution

- **Remote proxy** – A `Proxy` object that communicates to a remote `RealSubject`
- **Virtual proxy** – A `Proxy` object that creates the resource intensive `RealSubject` after a client makes a request on `Proxy`
- **Restricted proxy** – A `Proxy` object that determines whether an invocation should be propagated to `RealSubject`

Proxy Pattern Example Solution



Applying the Proxy Pattern: Use in the Java Programming Language



Applying the Proxy Pattern: Consequences

Advantages:

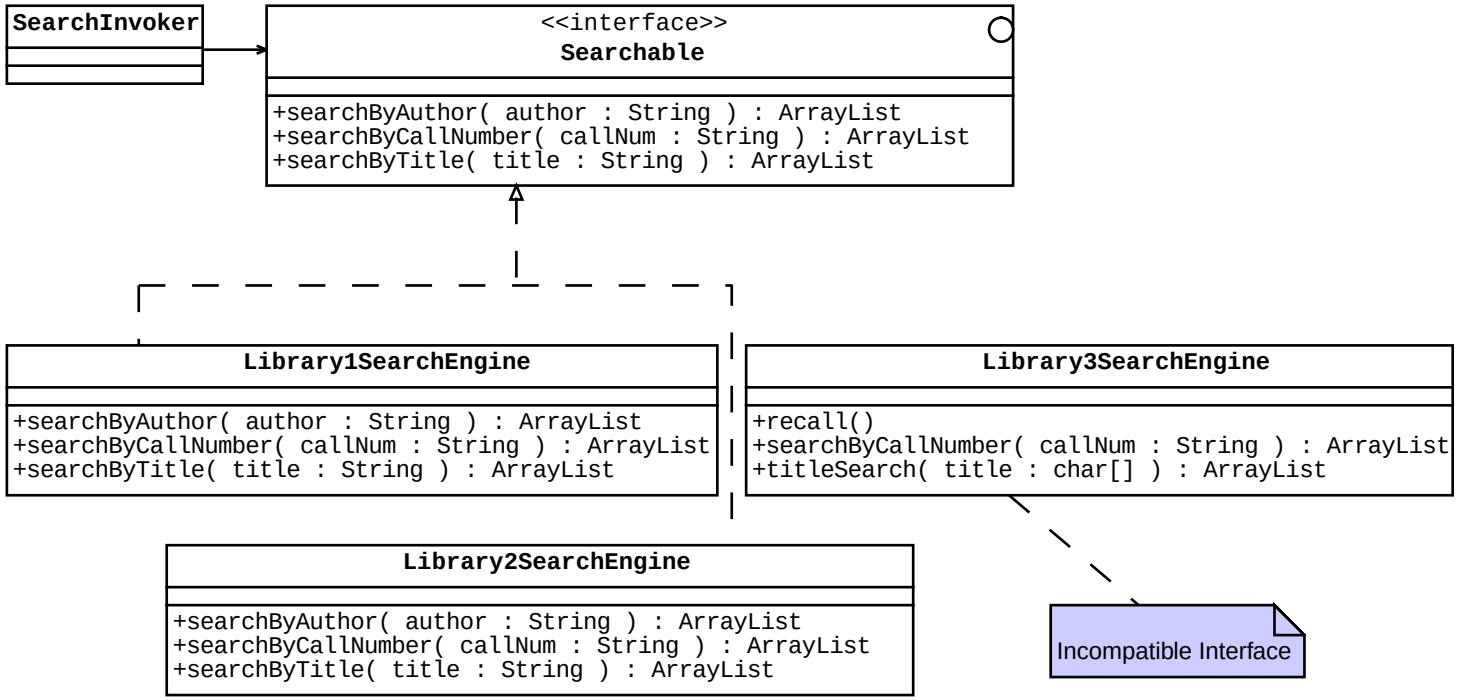
- The proxy is transparent to the client
- The proxy simplifies client access
- The proxy provides a point for extending capabilities
- The proxy can control or limit access to the real object

Disadvantages:

- An extra level of indirection is introduced that might impact performance
- Interaction with the proxy is not always transparent

Adapter Pattern Example Problem

- The `Library3SearchEngine` class needs to be adapted to the `Searchable` interface
- `Library3SearchEngine` should not be modified



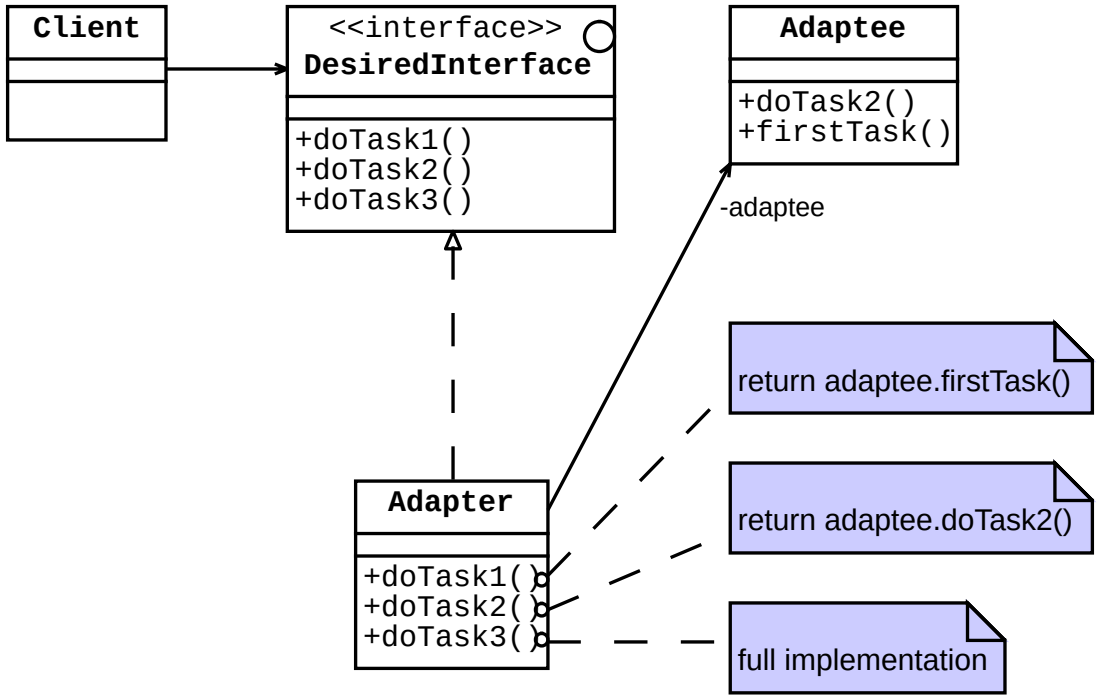
Applying the Adapter Pattern: Problem Forces

- Forcing clients to use classes with incompatible interfaces can complicate code
- Creating different interfaces for the same functionality by copy-and-paste reuse can lead to redundant code

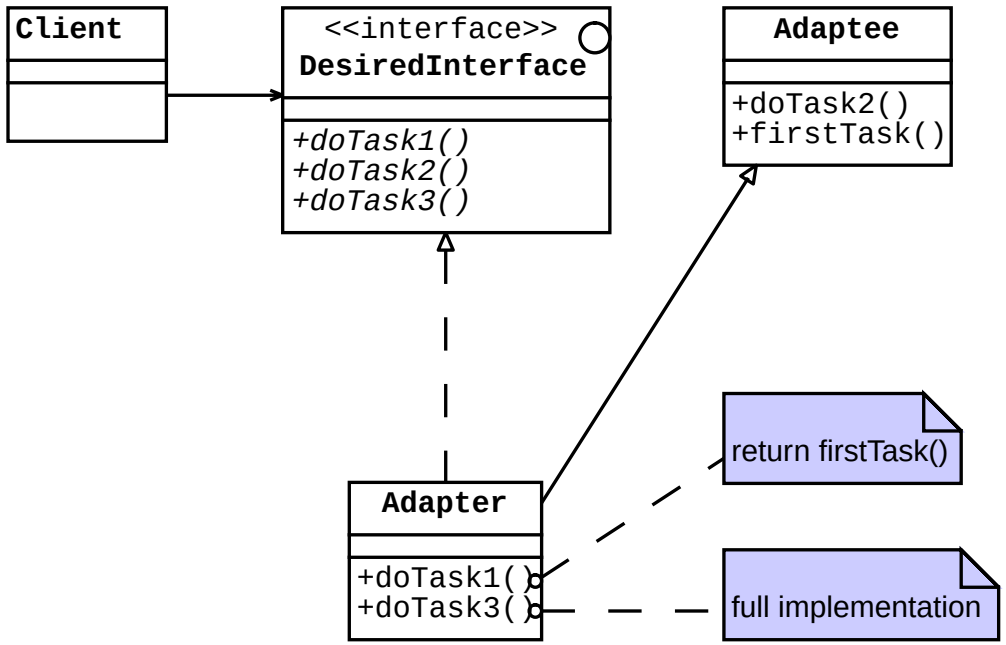
Applying the Adapter Pattern: Solution

- The solution is similar to a hardware adapter
- Create an `Adapter` class with the required interface:
  - The `Client` object invokes methods on `Adapter`
  - The `Adapter` object forwards calls on to the `Adaptee` class for some methods
  - The `Adapter` object can provide more complex or additional methods when required
- In the Object Adapter strategy, `Adapter` has a reference to `Adaptee`
- In the Class Adapter strategy, `Adapter` is a subclass of `Adaptee`

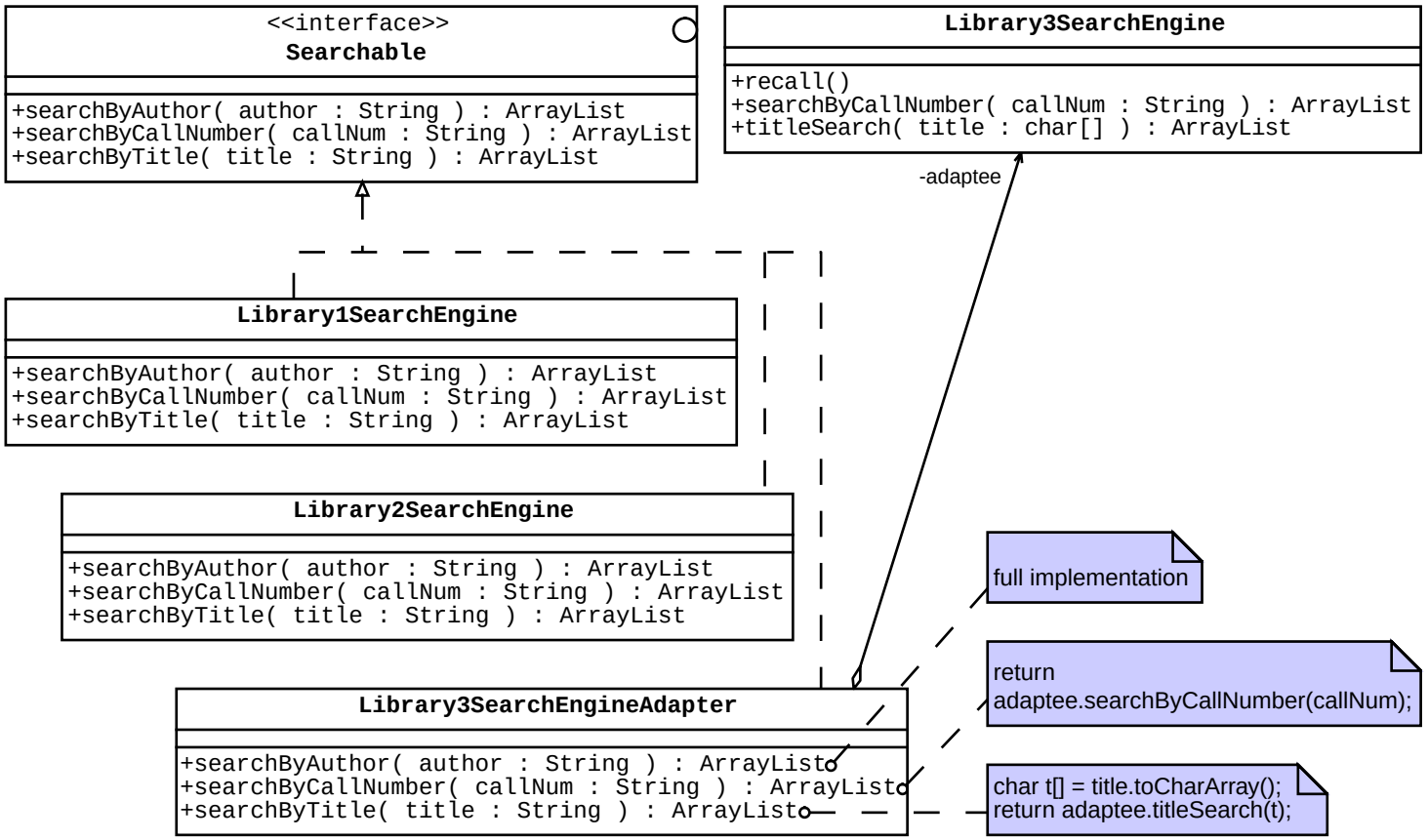
Object Adapter Strategy Structure



Class Adapter Strategy Structure



Object Adapter Strategy Example Solution



Object Adapter Strategy Example Solution

The library adapter class code:

```
1   import java.util.ArrayList;
2   public class Library3SearchEngineAdapter
3           implements Searchable {
4     private Library3SearchEngine adaptee =
5         new Library3SearchEngine();
6     public ArrayList searchByAuthor(String author) {
7       //... full implementation of search by author
8       return results;
9     }
10    public ArrayList searchByCallNumber(String callNum) {
11      return adaptee.searchByCallNumber(callNum);
12    }
13    public ArrayList searchByTitle(String title) {
14      char t[] = title.toCharArray();
15      return adaptee.titleSearch(t);
16    }
17  }
```

Applying the Adapter Pattern: Consequences

Advantages:

- The client class is not complicated by having to use a different interface
- The adaptee class is not modified
- Clients can use the adaptee class with or without the adapter

Applying the Adapter Pattern: Consequences

Object Adapter strategy disadvantages:

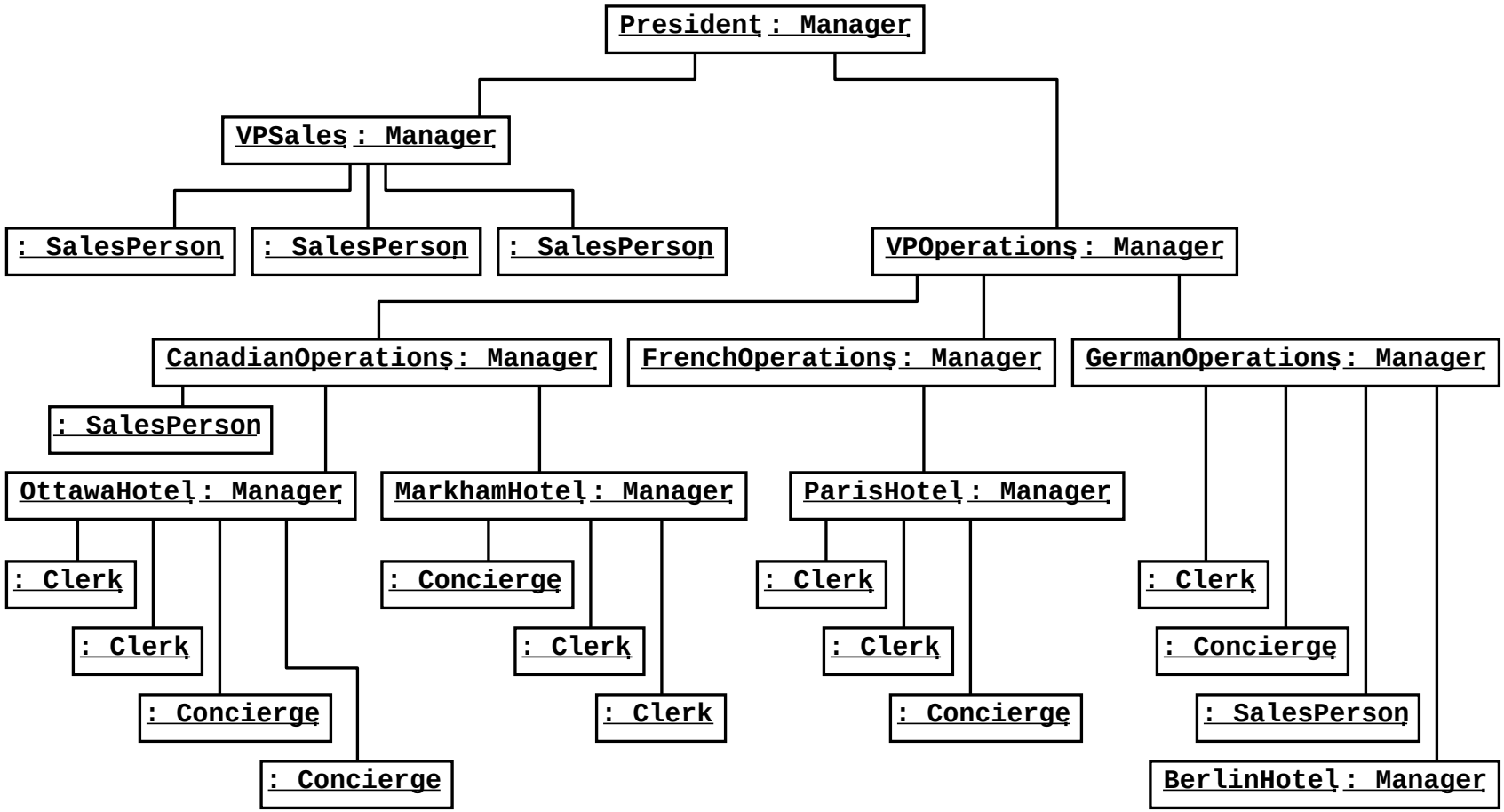
- You must create an additional object
- Requests are forwarded, causing a slight overhead increase

Class Adapter strategy disadvantages:

- The interface of the adapter class also supports the interface of the adaptee class, so the client is coupled to the adaptee class
- The single inheritance slot is used by the adaptee class

Composite Pattern Example Problem

A composite-part employee structure is required



Applying the Composite Pattern: Problem Forces

Objects exist in a composite-part hierarchy:

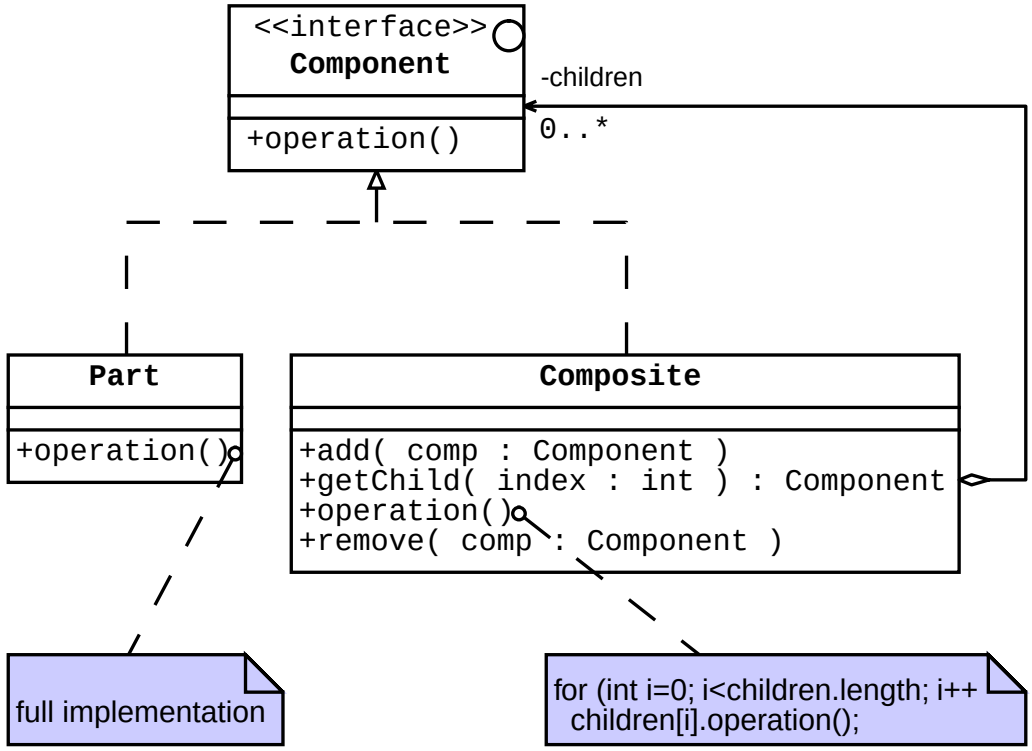
- Variable number of levels of objects that can change dynamically
- Some operations should be propagated down the composite hierarchy
- Need to treat the composite and each of the parts in a uniform way
- It can be inappropriate to handle some of the operations uniformly on composites and parts

Applying the Composite Pattern: Solution

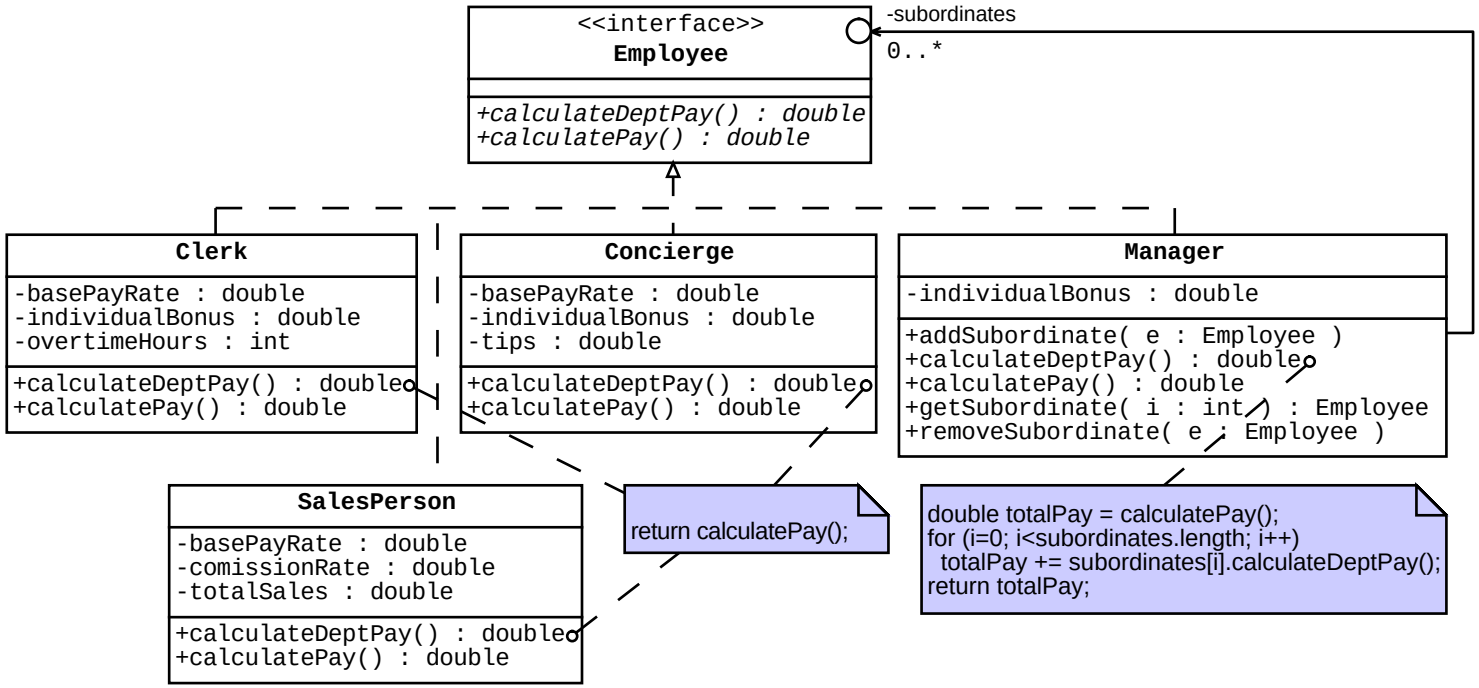
Provide a common interface for the classes of individual part objects and grouped composite objects:

- The `Composite` object has a set of `Component` references that refers to its components, some of which may be other `Composite` objects
- The client manipulates both `Composite` and `Part` objects through a common interface
- Some method calls are propagated down the composition hierarchy

Composite Pattern Structure



Composite Pattern Example Solution



Composite Pattern Example Solution

The `Clerk` class code:

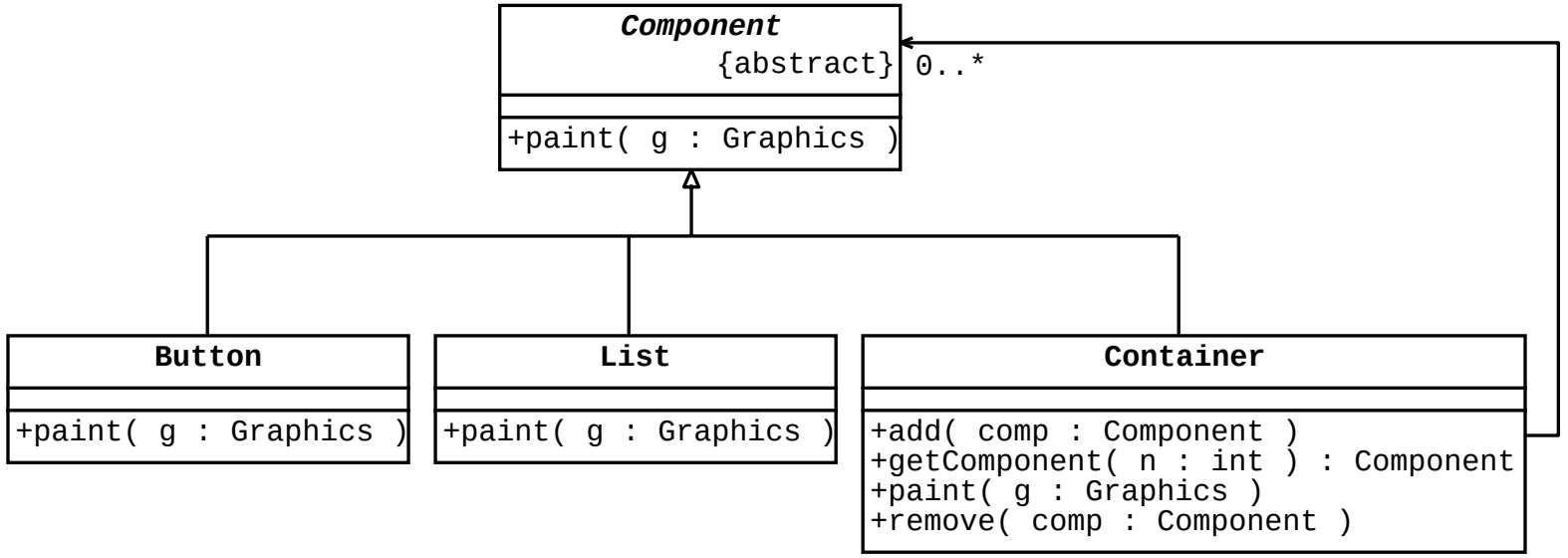
```
1   public class Clerk implements Employee
2   {
3     private double basePayRate;
4     private double individualBonus;
5     private int overtimeHours;
6
7     public double calculateDeptPay() {
8       return calculatePay();
9     }
10    public double calculatePay() {
11      double pay = // ... calculate individual pay
12      return pay;
13    }
14  }
```

Composite Pattern Example Solution

The `Manager` class code:

```
1   public class Manager implements Employee {
2     private Employee subordinates[];
3     public void addSubordinate(Employee e) {
4       // ... add e to the subordinates array
5     }
6     public double calculateDeptPay() {
7       double totalPay = calculatePay();
8       for (int i=0; i<subordinates.length; i++) {
9         totalPay += subordinates[i].calculateDeptPay();
10      }
11      return totalPay;
12    }
13    public double calculatePay() {
14      double pay = // ... calculate individual pay return pay;
15    }
16  . . .
17  }
```

Applying the Composite Pattern: Use in the Java Programming Language



Applying the Composite Pattern: Consequences

Advantages:

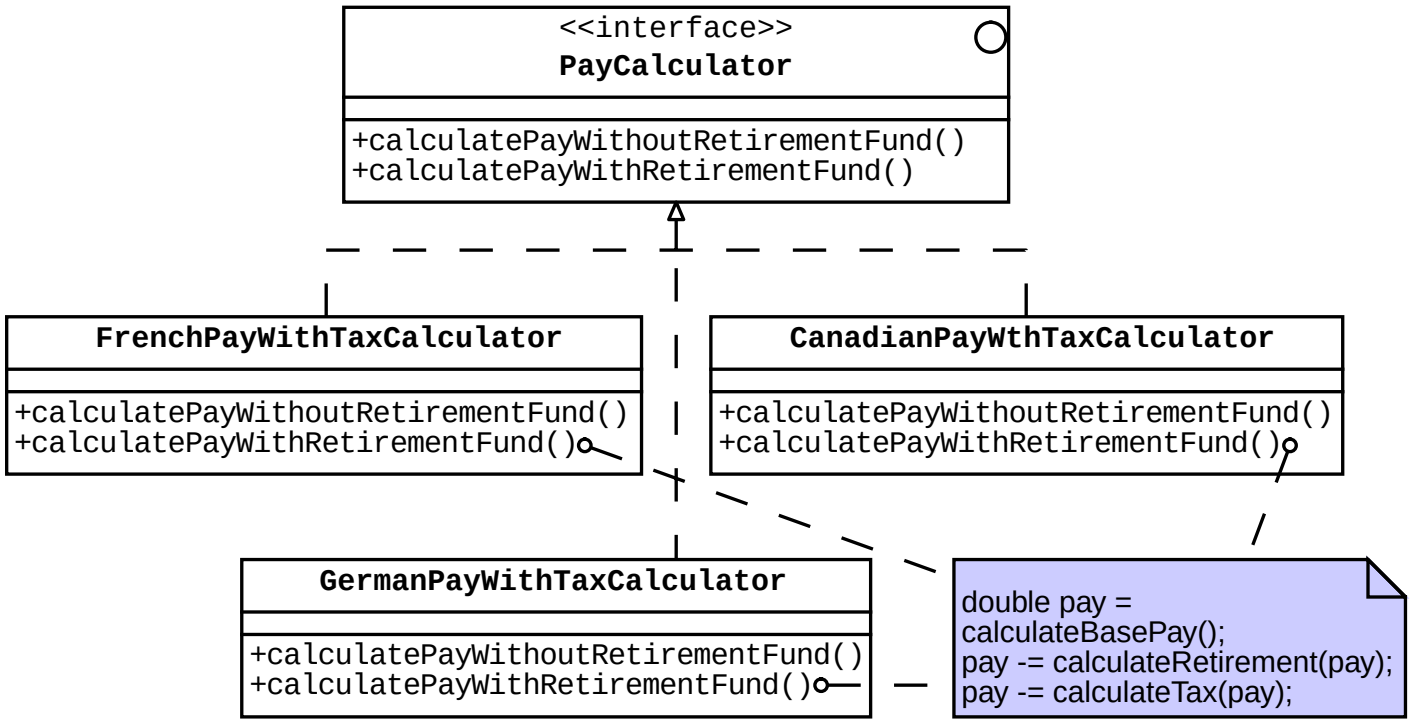
- Composites and parts are treated uniformly
- Composite hierarchy is an arbitrary depth
- Composite hierarchy is highly extensible

Disadvantages:

- The design can become too general
- The component interface might have to provide methods that make sense for composites but not for parts

Decorator Pattern Example Problem

Calculating an employee's pay involves base calculations, various tax calculations, and possibly retirement fund calculations



Applying the Decorator Pattern: Problem Forces

Adding extended functionality through direct subclassing may lead to:

- An explosion of subclasses
- Code duplication in subclasses

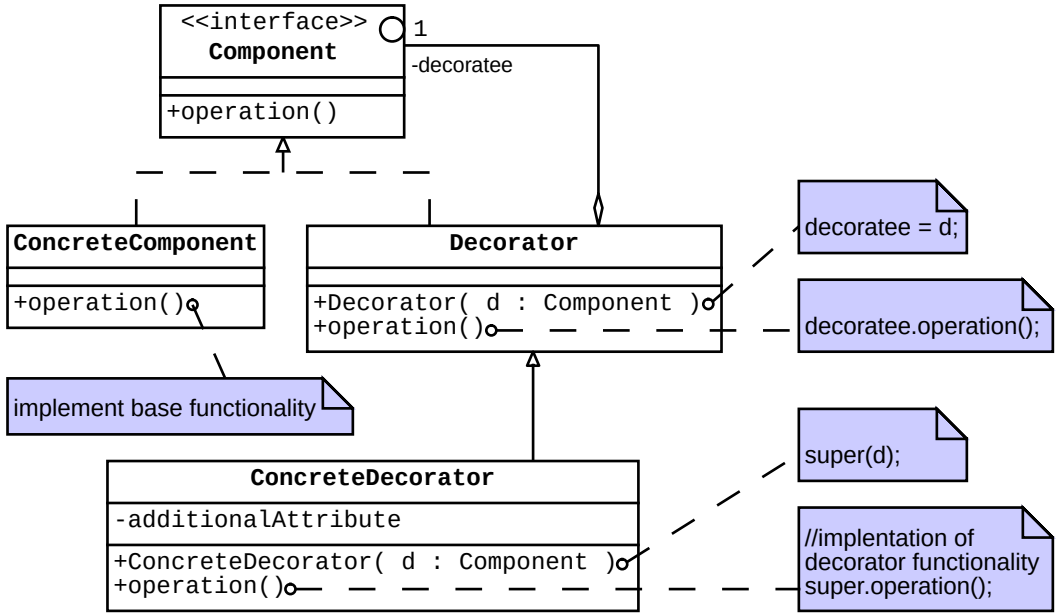
A solution is needed that:

- Allows functionalities to be expanded in a simple manner
- Allows functionalities to be dynamically added or removed
- Maintains high cohesion and minimizes the duplication of code

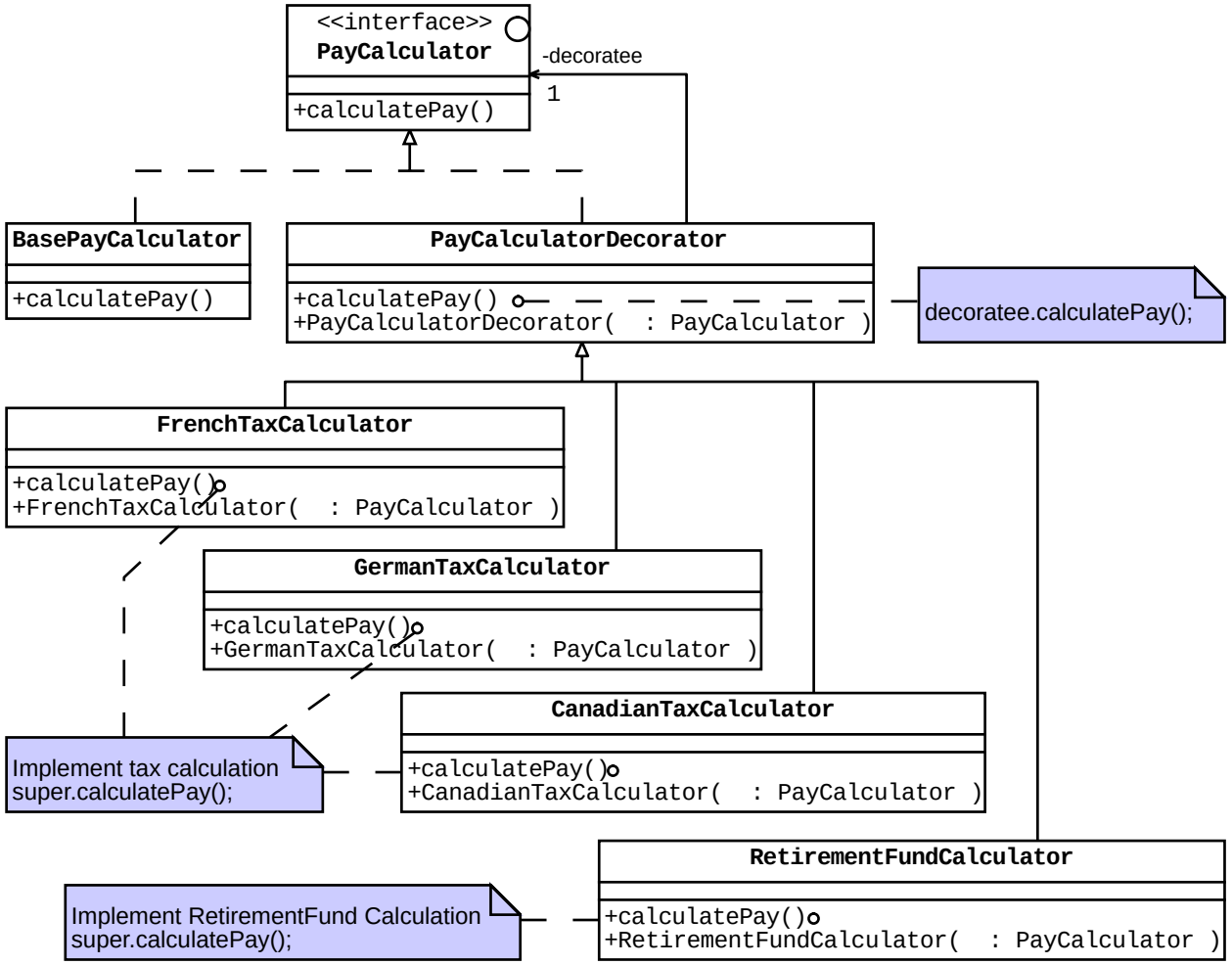
Applying the Decorator Pattern: Solution

- Additional functionalities are moved from the `ConcreteComponent` class into the `Decorator` classes
- The `Decorator` class adds one functionality to another class
- The `Decorator` class keeps a `Component` reference to the `ConcreteComponent` class that it decorates
- The `Decorator` methods implement their functionality and propagate the invocation to the `Component` class they decorate
- The chain of `Decorators` are transparent to the caller when dealing with a decorated class

Decorator Pattern Structure



Decorator Pattern Example Solution

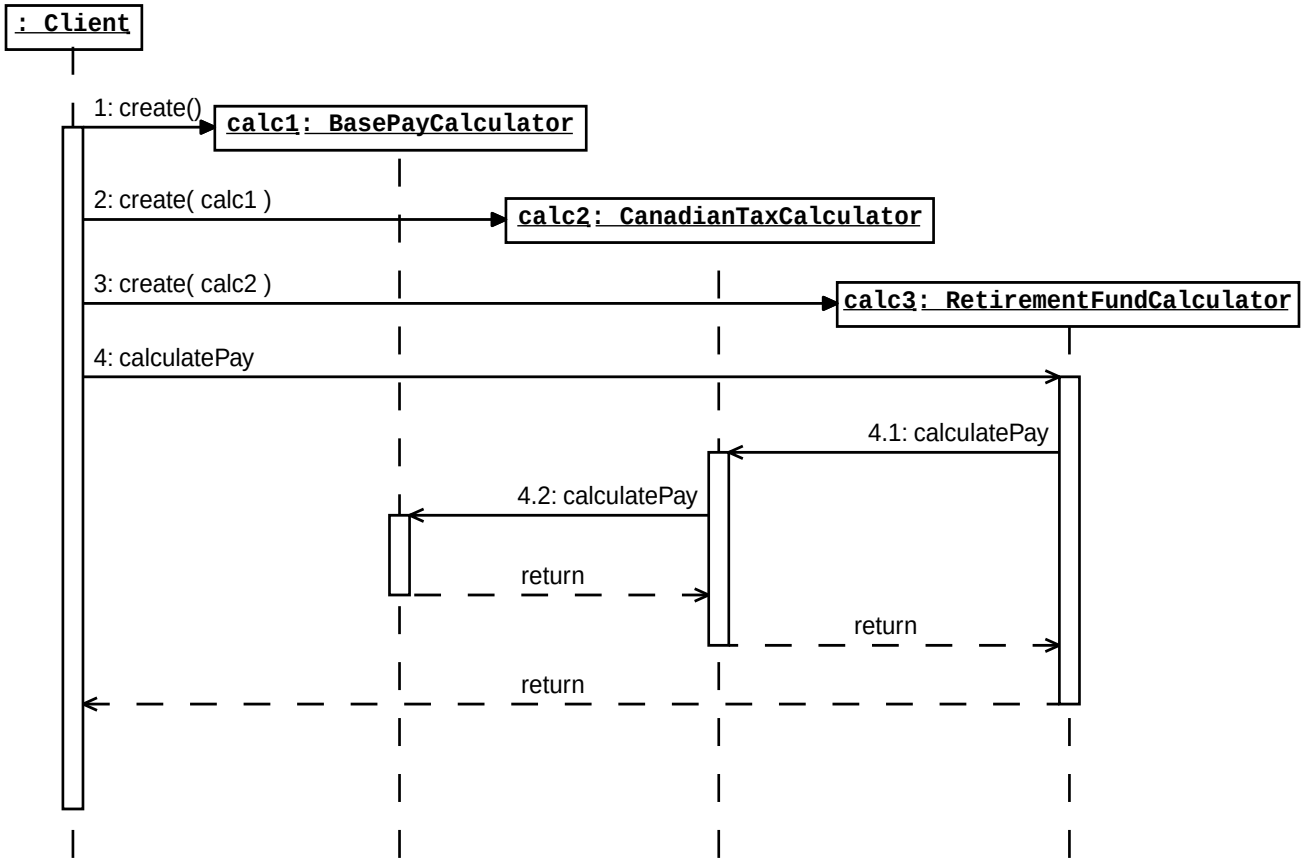


Decorator Pattern Example Solution

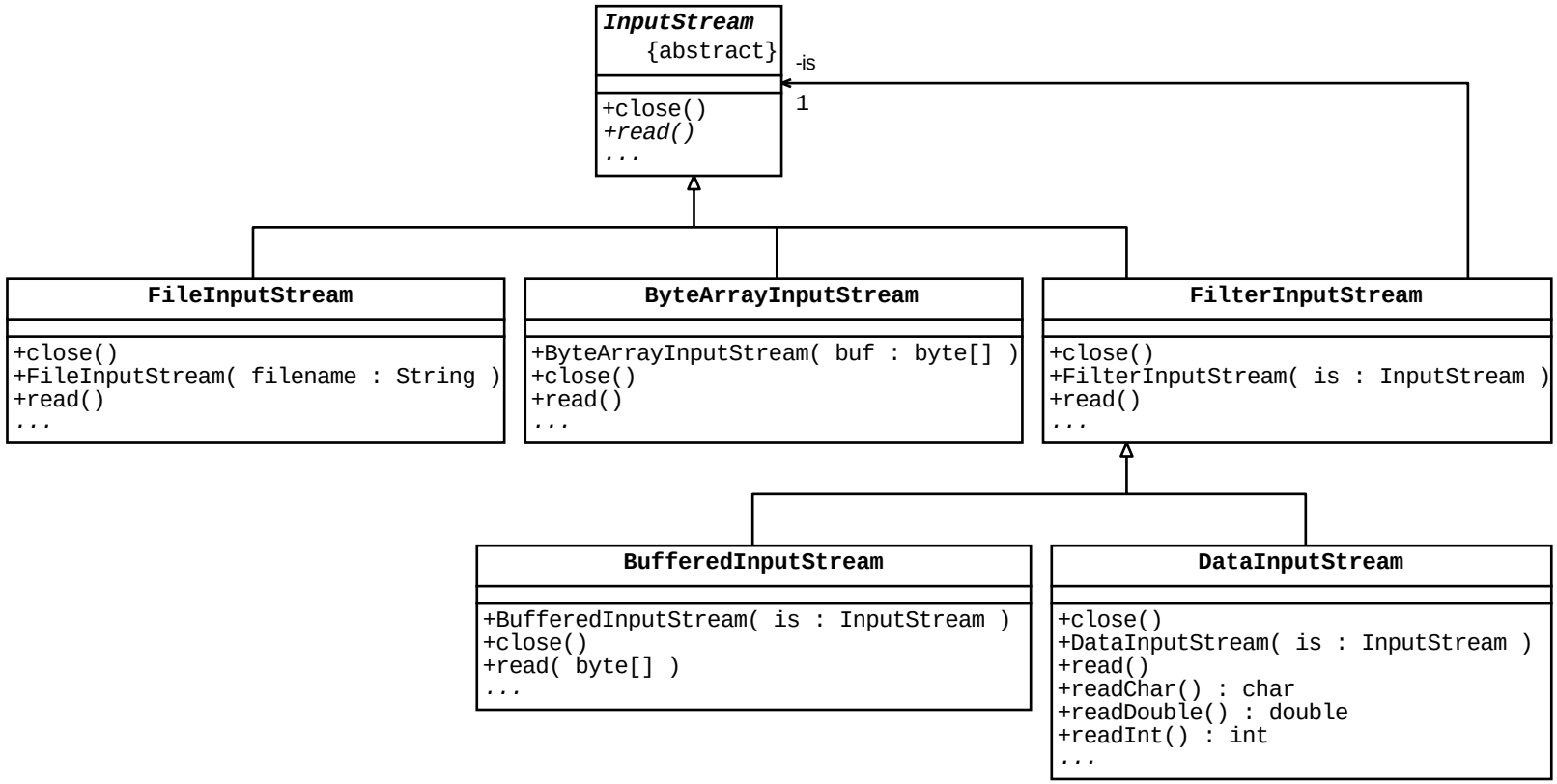
Pay calculation code:

```
1   PayCalculator calc1 = new BasePayCalculator();
2   PayCalculator calc2 = new RetirementFundCalculator(calc1);
3   PayCalculator calc3 = new CanadianTaxCalculator(calc2);
4   calc3.calculatePay();
```

Decorator Pattern Example Sequence



Applying the Decorator Pattern: Use in the Java Programming Language



Applying the Decorator Pattern: Consequences

Advantages:

- Can dynamically modify an object's state and behavior
- Does not modify or break inheritance hierarchies because of an addition to a base class
- Can nest decorators to add new state and behavior

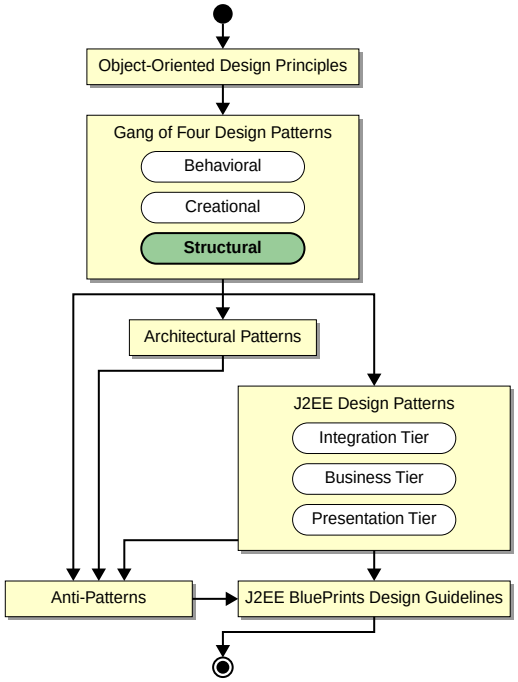
Disadvantages:

- Decorator and decorated object are not the same, so object identity becomes an issue
- Many little objects get created, which can slightly decrease performance

Summary

The GoF structural patterns:

- **Façade** – Provides a simplified interface to a subsystem
- **Proxy** – Provides an intermediate object that controls access to another object
- **Adapter** – Enables a caller to use an incompatible interface
- **Composite** – Composes objects into part-whole tree structures
- **Decorator** – Attaches new functionality to an object dynamically



Module 5

Using Architectural Building Blocks

Objectives

- Compare architectural patterns to design patterns
- Apply the Model View Controller pattern
- Apply the Layers pattern
- Explain tiers and layers in J2EE platform applications

Comparing Architectural Patterns to Design Patterns

- Design patterns concentrate more on the tactical aspects required to solve a low-level problem
- Architectural patterns:
  - Focus on the strategic, high-level composite structure of a solution
  - Usually implemented with multiple design patterns
  - Define the role of each subsystem in the overall solution

Applying the Model View Controller (MVC) Pattern

- Model View Controller (MVC) pattern is an architectural pattern
- Found in many places, including the Java Foundation Classes/Swing (JFC/Swing) API
- MVC separates presentation management from core application logic
- Useful in applications where the user interface might frequently change
- MVC can be adapted to Web applications

Applying the MVC Pattern: Example Problem

- The hotel system needs a non-Web user interface to allow hotel clerks to access reservation information and check guests in and out
- You need a way to divide the system into parts so that:
  - Programmers can specialize on user interface development or business logic
  - The user interface can be changed without changing the business logic
  - The Web reservation system and this non-Web user interface can access the same business logic

Applying the MVC Pattern: Problem Forces

User interfaces and business processing often change independently and some systems have multiple user interface views:

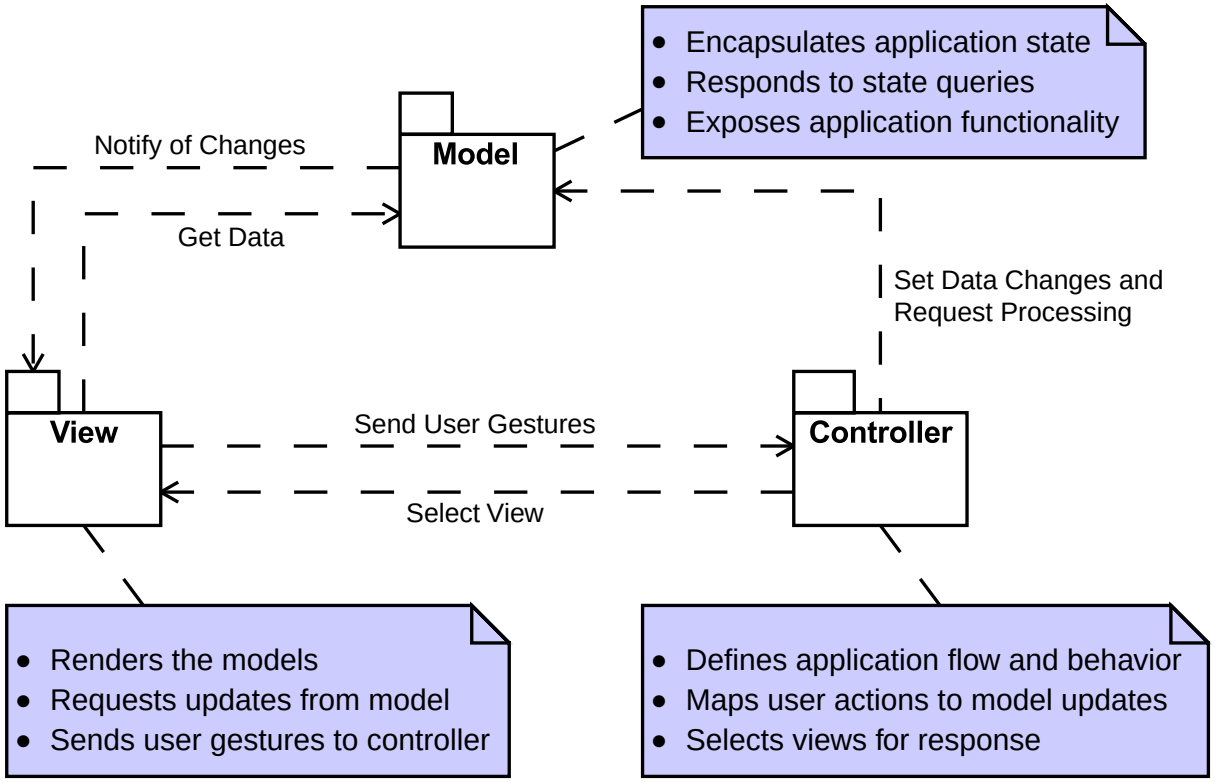
- You can display the same information in different views
- The views might need to immediately reflect changes to the model data
- The model data must respond immediately to changes initiated in the views
- The user interface should be decoupled from the data and processing

Applying the MVC Pattern: Solution

The Model View Controller pattern divides the system into three sets of components:

- **Controller** – Accepts requests from the user, invokes processing on the `Model` components, and determines which `View` component should be displayed
- **View** – Displays a graphical user interface (GUI) containing the `Model` components data to the user and usually passes GUI events to `Controller` components
- **Model** – Contains the business data, processing, and rules. The `Model` should not have any details about the user interface

Model View Controller Pattern Structure

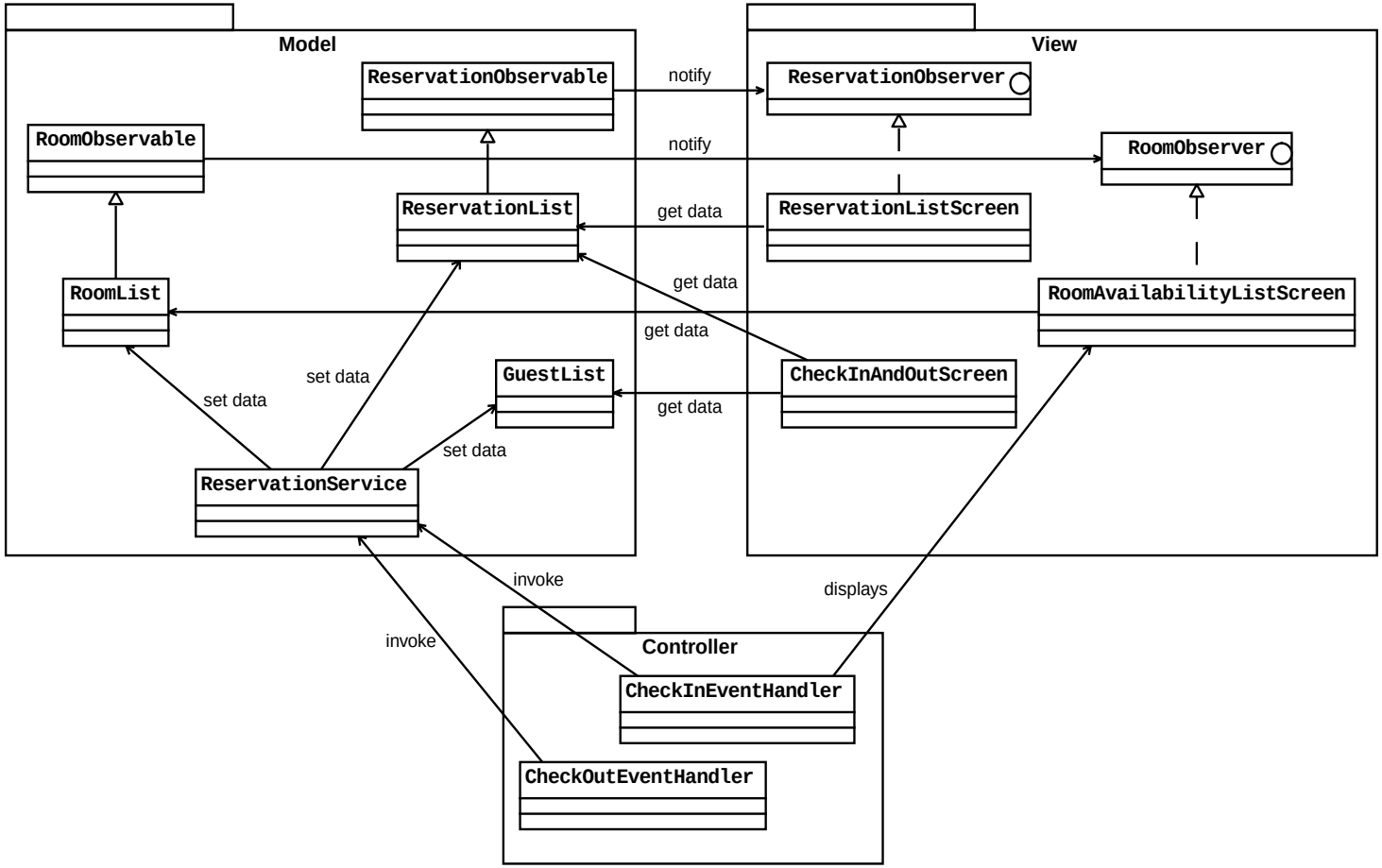


Applying the MVC Pattern: Solution

The MVC architectural pattern may be implemented using several Gang of Four design patterns including:

- **Observer Pattern** – The `View` components can be observers of the `Model` components
- **Composite Pattern** – The `View` components can be a composite of other GUI components

Model View Controller Pattern Example Solution



Applying the MVC Pattern: Consequences

Advantages:

- Multiple views can allow access to the business processing and data from different perspectives
- It is simpler to modify the model, the view, and the controller separately
- It is easier for some developers to focus on the user interface and others to focus on the model

Disadvantage:

Dividing the subsystems can increase communication overhead between the model and the view

Applying the Layers Pattern

"The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction." (Buschmann, Meunier, Rohmert, Sommerlad, and Stal. *Pattern Oriented Software Architecture*):

- The Layers architectural pattern is seen in numerous and diverse systems in computer science
- The principles behind this pattern form the basis for the architecture recommended by the J2EE BluePrints design guidelines and the J2EE Patterns

Applying the Layers Pattern: Example Problem

- A communication protocol is needed to send Web requests between services written in heterogeneous environments
- This protocol must address a variety of low and high level issues
- The low level issues include physically sending the request across the network
- The high level issues include structuring flexible messages between heterogeneous development environments

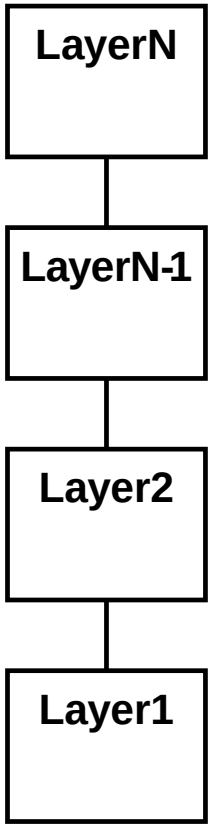
Applying the Layers Pattern: Problem Forces

Complicated systems often have many layers of issues. A layering solution must deal with these forces:

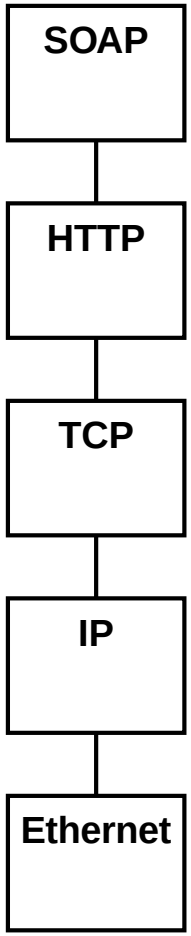
- System functionality changes should not ripple through the system
- Interfaces between components should be stable
- Parts of the system should be able to be exchanged without affecting the rest of the system
- Parts of the system should be reusable
- Data that is passed frequently between subsystems can lead to poor performance

Applying the Layers Pattern: Solution

Each layer can only communicate with the layer below it.



Layers Pattern: Example Solution



Applying the Layers Pattern: Consequences

Advantages:

- Layers may be reusable
- Layers are loosely coupled so that they can be developed and maintained separately
- Layers can be exchangeable without rewriting other layers

Disadvantages:

- Adding a new functionality might affect every layer
- Separating layers can increase communication overhead

Applying Layers and Tiers in J2EE™ Platform Enterprise Applications

- Layering is a J2EE architecture best practice
- The size and complexity of enterprise applications makes layering a natural choice
- The layering approach used by the J2EE Patterns and the J2EE BluePrints design guidelines are described in the SunTone^{SM} Architecture Methodology

SunTone™ Architecture Methodology Layers

The SunTone Architecture Methodology uses a more specific definition of layers than the Layers pattern:

> Layers are "The hardware and software stack that hosts services within a given tier. (layers represent component/container relationships)" (*SunTone Architecture Methodology page 11*)

SunTone Architecture Methodology Layers

- **Application** – Provides a concrete implementation of components to satisfy the functional requirements
- **Virtual Platform** – Provides the APIs that application components implement
- **Upper Platform** – Provides the infrastructure for the application layer components
- **Lower Platform** – Provides the operating systems that support the previous layers
- **Hardware Platform** – Provides the hardware required to support the previous layers

SunTone Architecture Methodology Tiers

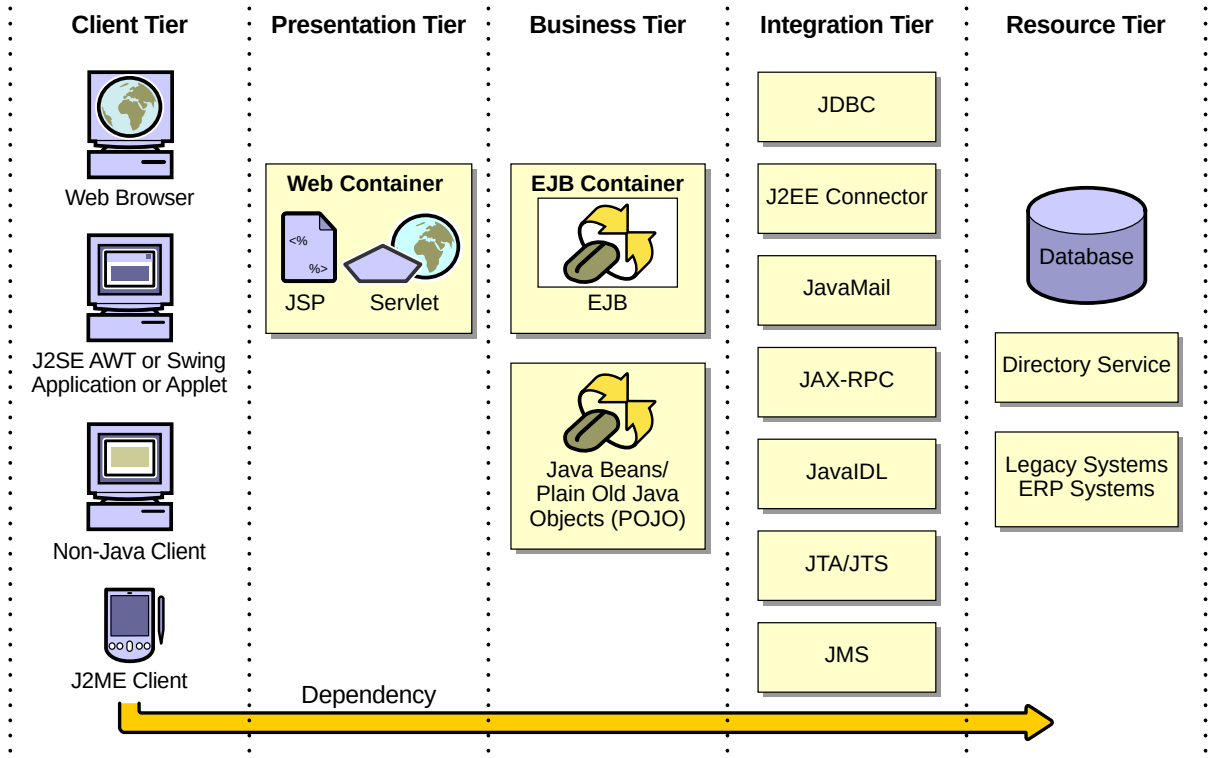
SunTone Architecture Methodology applies the Layers pattern to define tiers

> Tiers are “A logical or physical organization of components into an ordered chain of service providers and consumers.”(*SunTone Architecture Methodology page 10*)

SunTone Architecture Methodology Tiers

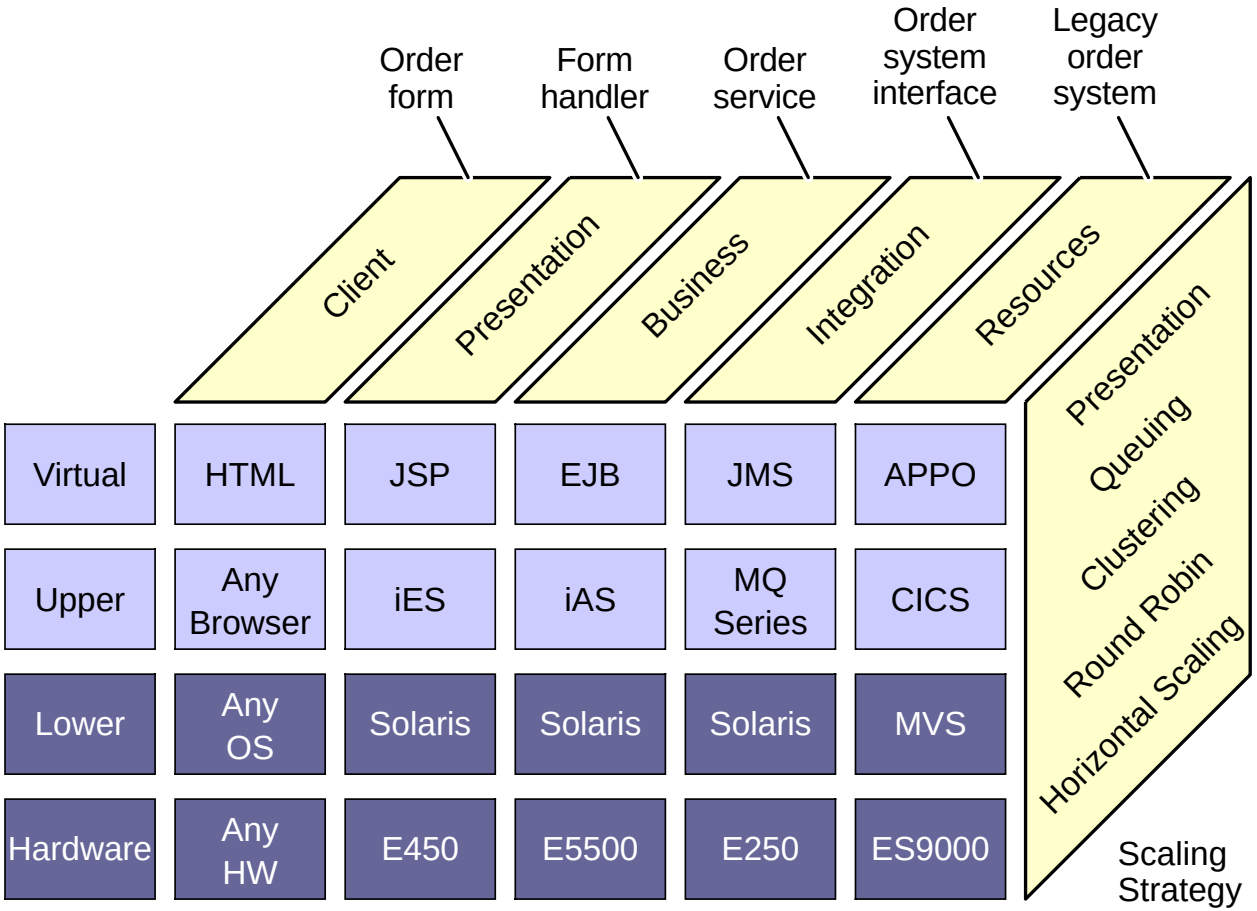
- **Client** – Provides user interaction with the system
- **Presentation** – Provides the hyper-text markup language (HTML) pages and forms that are sent to the Web browser and processes the user's requests
- **Business** – Provides the business services and entities
- **Integration** – Provides components to integrate the Business tier with the Resource tier
- **Resource** – Contains all backend resources, such as a DataBase Management System (DBMS) or Enterprise Information System (EIS)

J2EE Technologies in Each Tier



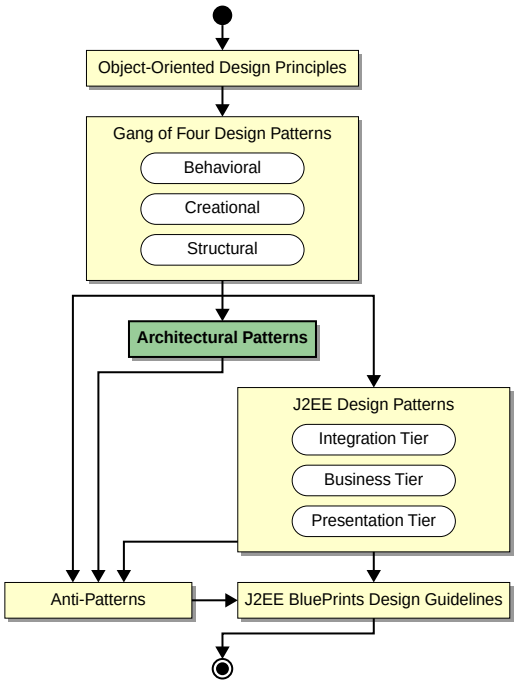
There are no J2EE patterns for the client and resources tiers since those tiers are implemented outside the J2EE platform.

J2EE Platform Example – Layers and Tiers



Summary

- Architectural patterns focus on the strategic, high-level composite structure of a solution
- The MVC pattern separates presentation management from business logic
- Layers are the hardware and software stack within a given tier
- Tiers are the logical or physical organization of components into an ordered chain of providers and consumers



Module 6

Introducing J2EE™ Patterns

Objectives

- Describe the J2EE pattern philosophy
- Describe the J2EE patterns and tiers in the J2EE pattern catalog

Purpose of the J2EE Patterns

- The Java Center^{SM} program, part of Sun Professional Services, has built enterprise solutions for a wide variety of customers
- The J2EE patterns evolved from their experience
- The J2EE patterns solve problems using the J2EE platform technologies
- Some J2EE patterns might already be familiar
- The purpose of the J2EE patterns is to document and recommend best practices rather than develop entirely new techniques

Benefits

The key benefits of the J2EE patterns include:

- Using sound design principles
- Leveraging documented, accepted practices
- Understanding each J2EE technology's primary role
- Reducing coupling and dependencies between components
- Minimizing network traffic
- Reducing costs of network latency

Relation to the J2EE BluePrints Design Guidelines

The J2EE BluePrints design guidelines:

- Describe standard techniques for building complex J2EE platform applications that can scale and change quickly
- Are complemented by the J2EE patterns that provide formally documented solutions to specific problems

Gang of Four and J2EE Patterns

Gang of Four Patterns:

- Abstract solutions that can be implemented in very diverse ways
- Can be used with any object-oriented programming language
- Often build on or expand other GoF patterns

J2EE Patterns:

- Designed for the J2EE platform
- Often build on or expand GoF patterns or other J2EE patterns

Describing the Patterns and Tiers of the J2EE Pattern Catalog

The J2EE patterns catalog is grouped into three tiers:

- Integration tier
- Business tier
- Presentation tier

Integration Tier Patterns

Concerned with the integration of the J2EE platform code with other types of applications and legacy systems:

- **Service Activator** – Allows a client to asynchronously invoke an EJB component, using the Java Message Service (JMS) API
- **Data Access Object** – Isolates database-specific code into classes that expose a business-oriented interface
- **Domain Store** – Creates a robust persistence mechanism that is transparent to the business objects without using entity beans
- **Web Service Broker** – Makes business services available as web services

Business Tier Patterns

Concerned with managing persistence and business processing:

- **Service Locator** – Removes the need for a J2EE platform client to be aware of the Java Naming and Directory Interface™ (JNDI) API when acquiring business components
- **Session Façade** – Provides the presentation tier with a simple interface to access the business tier
- **Business Delegate** – Provides loosely coupled access to business tier components
- **Transfer Object** – Reduces the number of remote method calls by returning multiple values in one object

Business Tier Patterns

- **Application Service** – Centralizes business logic between the service facades and the business objects
- **Business Object** – Separates business data from business logic and workflow logic
- **Transfer Object Assembler** – Assembles transfer object data from multiple business objects
- **Composite Entity** – Wraps a number of related, fine-grained, persistent objects into a single entity, representing a structured organization, containing those objects
- **Value List Handler** – Provides an efficient mechanism for executing queries that might return a large number of objects and browsing through the results

Presentation Tier Patterns

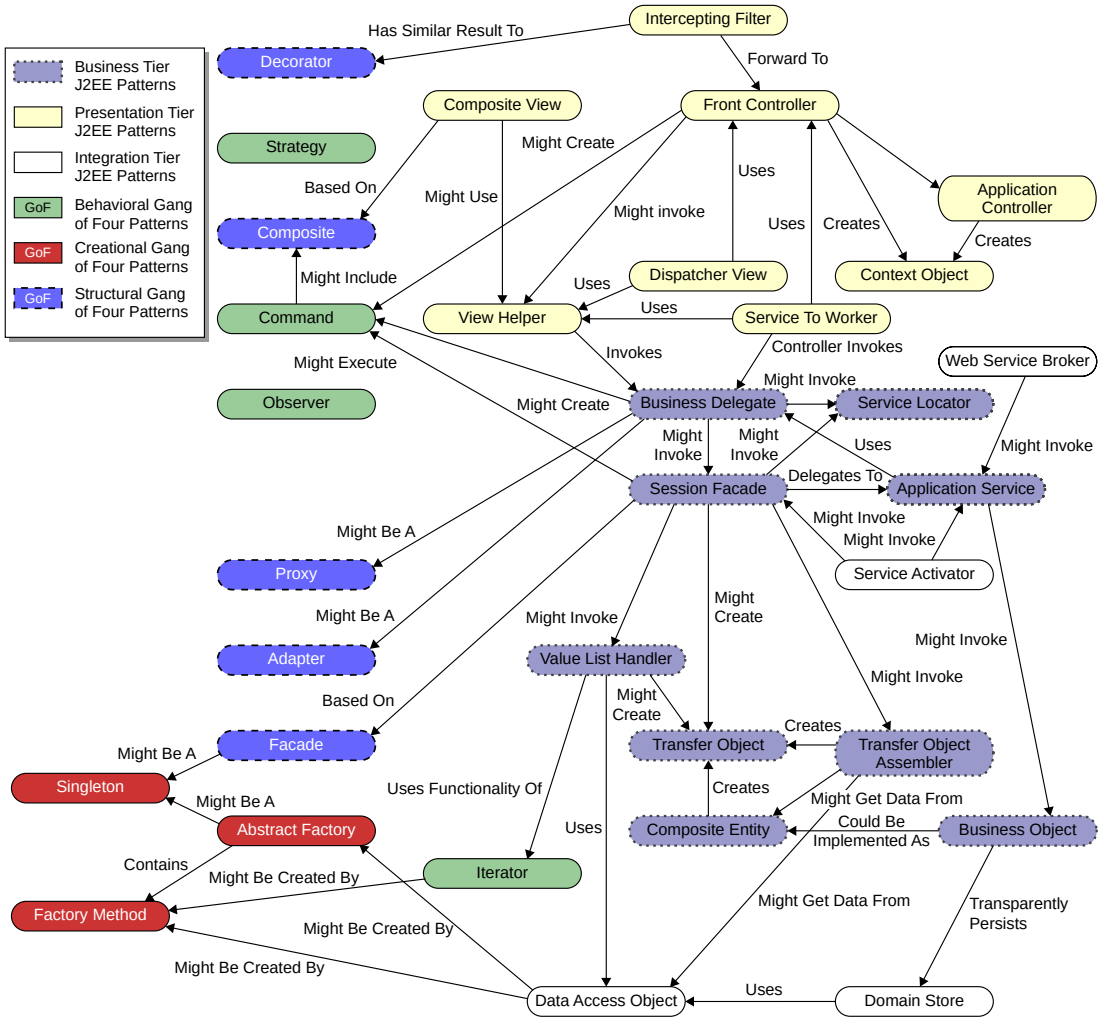
Concerned with organizing the application's presentation components:

- **Intercepting Filter** – Manages pre-processing and post-processing of a client request
- **Front Controller** – Provides a mechanism for centralized management of user requests
- **Application Controller** – Separates the action invocation management and view dispatching management from the front controller component
- **Context Object** – Passes data from context specific objects without passing those objects out of their context

Presentation Tier Patterns

- **View Helper** – Factors out content retrieval for a view from the logic needed to build the view
- **Composite View** – Constructs a view from many different sub-views
- **Dispatcher View** – Combines the Front Controller and View Helper patterns
- **Service to Worker** – Similar to the Dispatcher View pattern, except that the front controller takes more responsibility for view selection and business process invocation

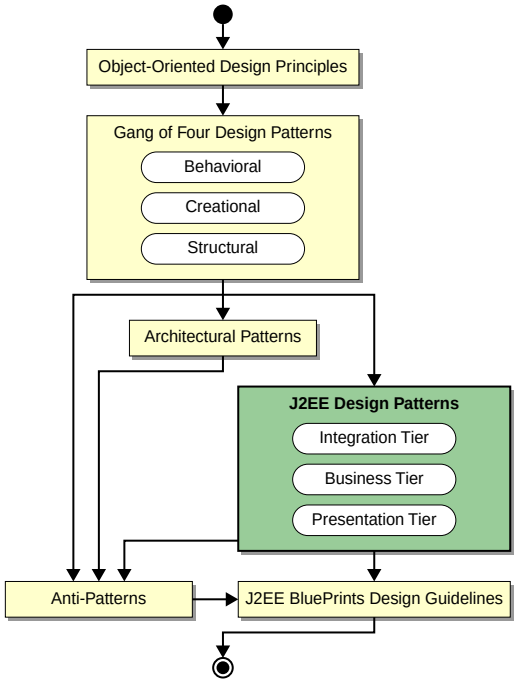
J2EE Pattern Relationships



Summary

J2EE patterns help to improve the internal structure of applications by:

- Consolidating common services
- Minimizing traffic between tiers
- Emphasizing loose coupling of objects



Module 7

Using Integration Tier Patterns

Objectives

- List the features and purpose of the Integration Tier patterns
- Apply the Service Activator pattern
- Apply the Data Access Object (DAO) pattern
- Apply the Domain Store pattern
- Apply the Web Service Broker pattern

Introducing Integration Tier Patterns

- Encapsulate access to the data stores (DAO)
- Provide messaging to the business tier
- Provide integration of web services with business components
- Rely on the JDBC API and JMS API as key technologies

Integration Tier J2EE Patterns Overview

Pattern	Primary Function
Service Activator	Provides asynchronous messaging for business components, including EJB components.
Data Access Object (DAO)	Encapsulates data access logic. Simplifies access to disparate data sources.
Domain Store	Creates a robust persistence mechanism that is transparent to the business objects without using entity beans.
Web Service Broker	Provides a flexible way to make business services available as web services.

All examples in the next five modules tie together as shown in the reference diagram in the student guide.

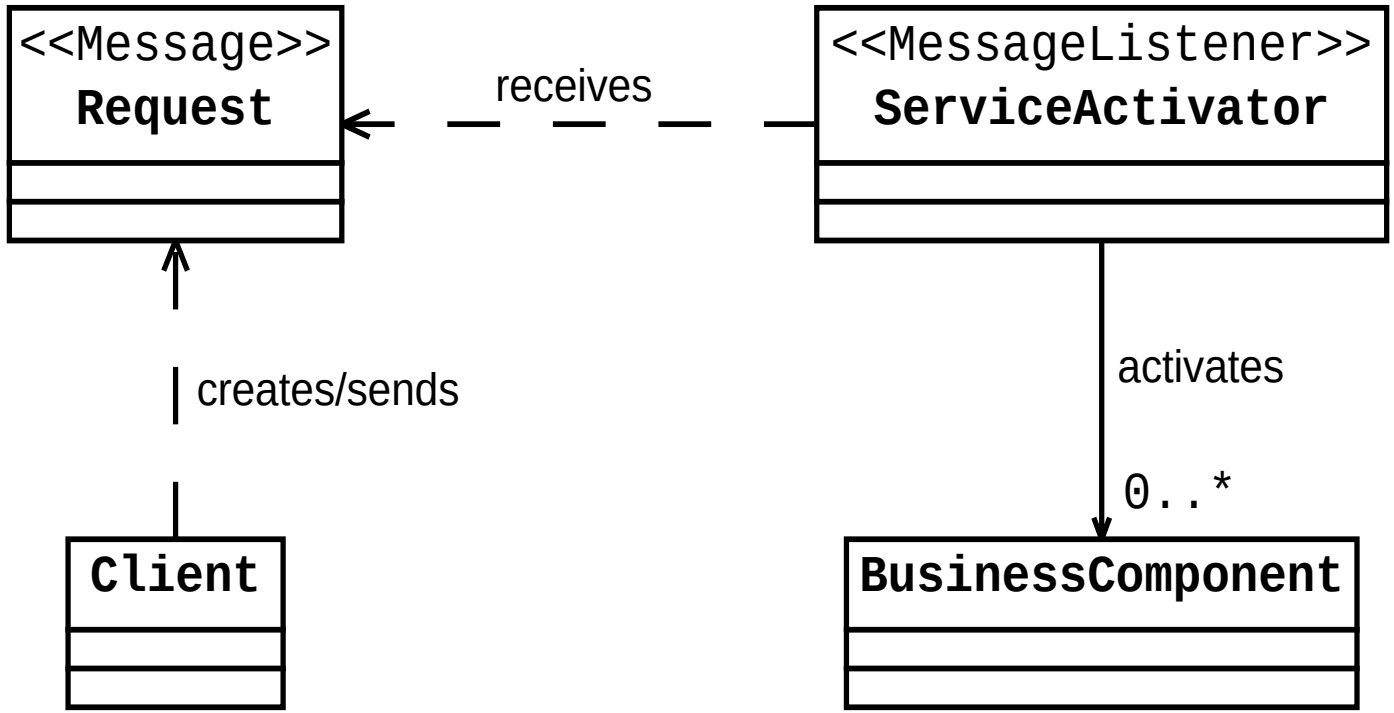
Applying the Service Activator Pattern: Problem Forces

- Invocation through messaging is very useful for:
  - Establishing asynchronous processing
  - Communicating between components with different lifetimes
  - Communicating with loosely coupled components
- A mechanism is needed for EJB 1.x components to be invoked indirectly using messages
- Even with containers based on EJB 2.x architecture (EJB 2.x containers), you should consider how to most flexibly invoke business components using messages

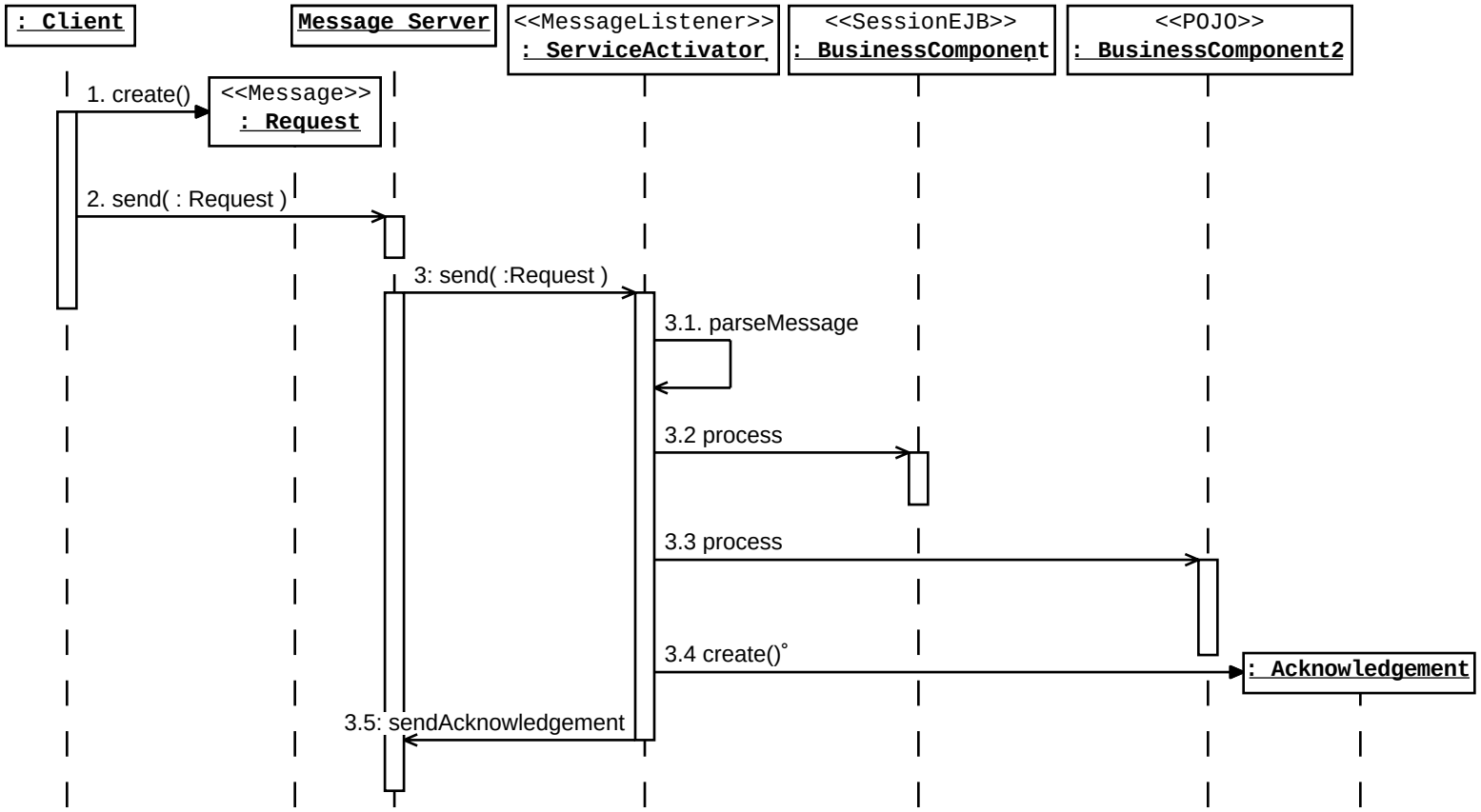
Applying the Service Activator: Solution

- The `ServiceActivator` class is a JMS API `MessageListener` implementation
- For EJB 1.x containers, `ServiceActivator` is a plain old Java object (POJO)
- For EJB 2.x containers, `ServiceActivator` can also be a message-driven bean
- The `ServiceActivator` class receives `Messages` and invokes business components to fulfill the request
- Keeping substantial business logic out of `ServiceActivator` improves maintainability and reusability

Service Activator Pattern Structure



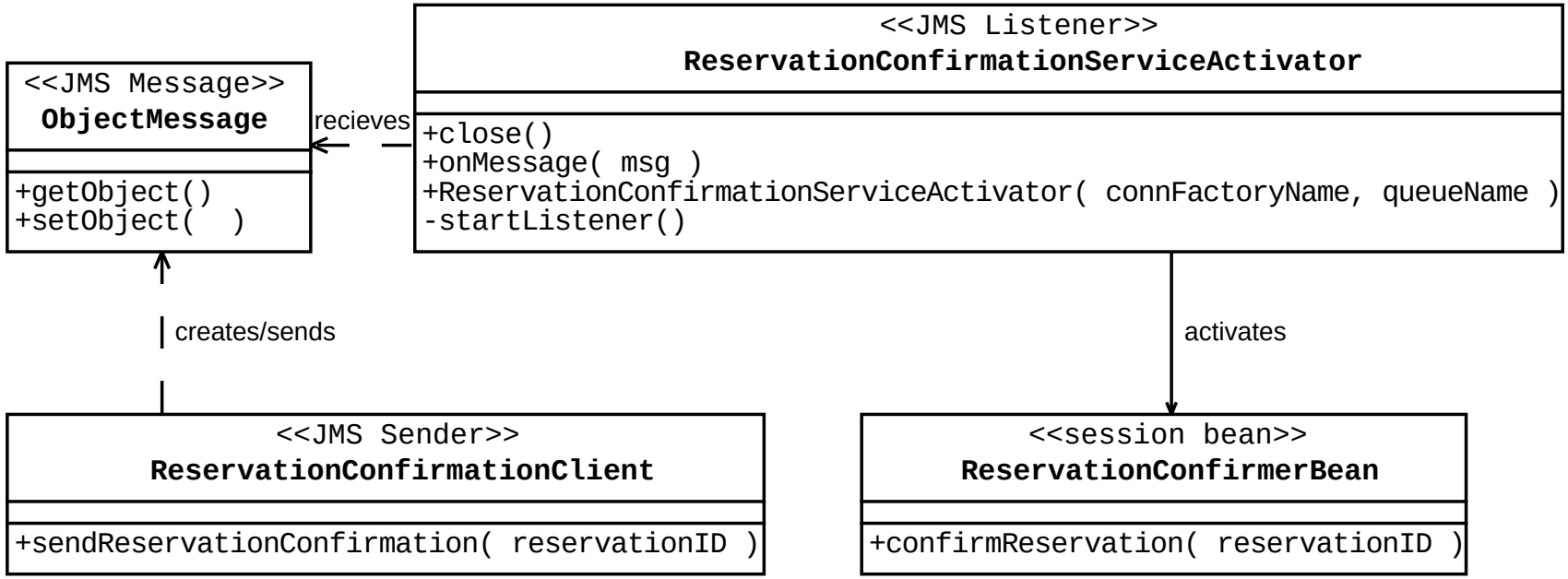
Service Activator Pattern Sequence



Applying the Service Activator Pattern: Strategies

- **POJO Service Activator** – This activator is mostly used with EJB 1.x containers or when no EJB containers are used
- **Message-Driven Bean Service Activator** – This activator is only available with an EJB 2.x container
- **Command Request** – The message can contain a GOF command that is invoked by the service activator
- **Database, Email, or JMS Message Response** – An asynchronous process can place a response to the client in a database, an email message, or another JMS message

Service Activator Pattern Example



Service Activator Pattern Example

```
1   import javax.jms.*;
2   public class ResConfirmServiceActivator
3       implements MessageListener{
4     private QueueSession qSession;
5     private QueueReceiver qReceiver;
6     private String connFactoryName;
7     private String queueName;
8     private JMSServiceLocator serviceLocator;
9     public ResConfirmServiceActivator
10        (String cfn, String qn) {
11      connFactoryName = cfn;
12      this.queueName = qn;
13      startListener();
14    }
15    private void startListener() {
16      try {
17        serviceLocator =
18          new JMSServiceLocator(connFactoryName);
19        QueueConnectionFactory qConnFactory =
20       serviceLocator.getQueueConnectionFactory();
21        QueueConnection qConn =
22          qConnFactory.createQueueConnection();
23        qSession = qConn.createQueueSession
24          (true,Session.AUTO_ACKNOWLEDGE);
25        Queue shipBookQueue =
26          serviceLocator.getQueue(queueName);
27        qReceiver =
28          qSession.createReceiver(shipBookQueue);
29        qReceiver.setMessageListener(this);
```

Service Activator Pattern Example

```
30      }
31      catch (JMSException e) {
32        e.printStackTrace();
33      }
34    }
35    public void onMessage(Message msg) {
36      try {
37        ObjectMessage objMsg = (ObjectMessage) msg;
38        String reservationID =
39          (String) objMsg.getObject();
40        //. . .Lookup and create bean
41        ReservationConfirmerLocal resConfirmer =...
42        resConfirmer.confirmReservation
43          (reservationID);
44      }
45      catch (JMSException e) {
46        e.printStackTrace();
47      }
48    }
49    public void close() {
50      try {
51        qReceiver.setMessageListener(null);
52        qSession.close();
53      }
54      catch(Exception e) {
55        e.printStackTrace();
56      }
57    }
58  }
```

Applying the Service Activator: Consequences

Advantages:

- Allows the invocation of EJB 1.x components from a messaging server
- Provides flexible asynchronous invocations of any type of business component
- Separates business processing from message consuming and parsing

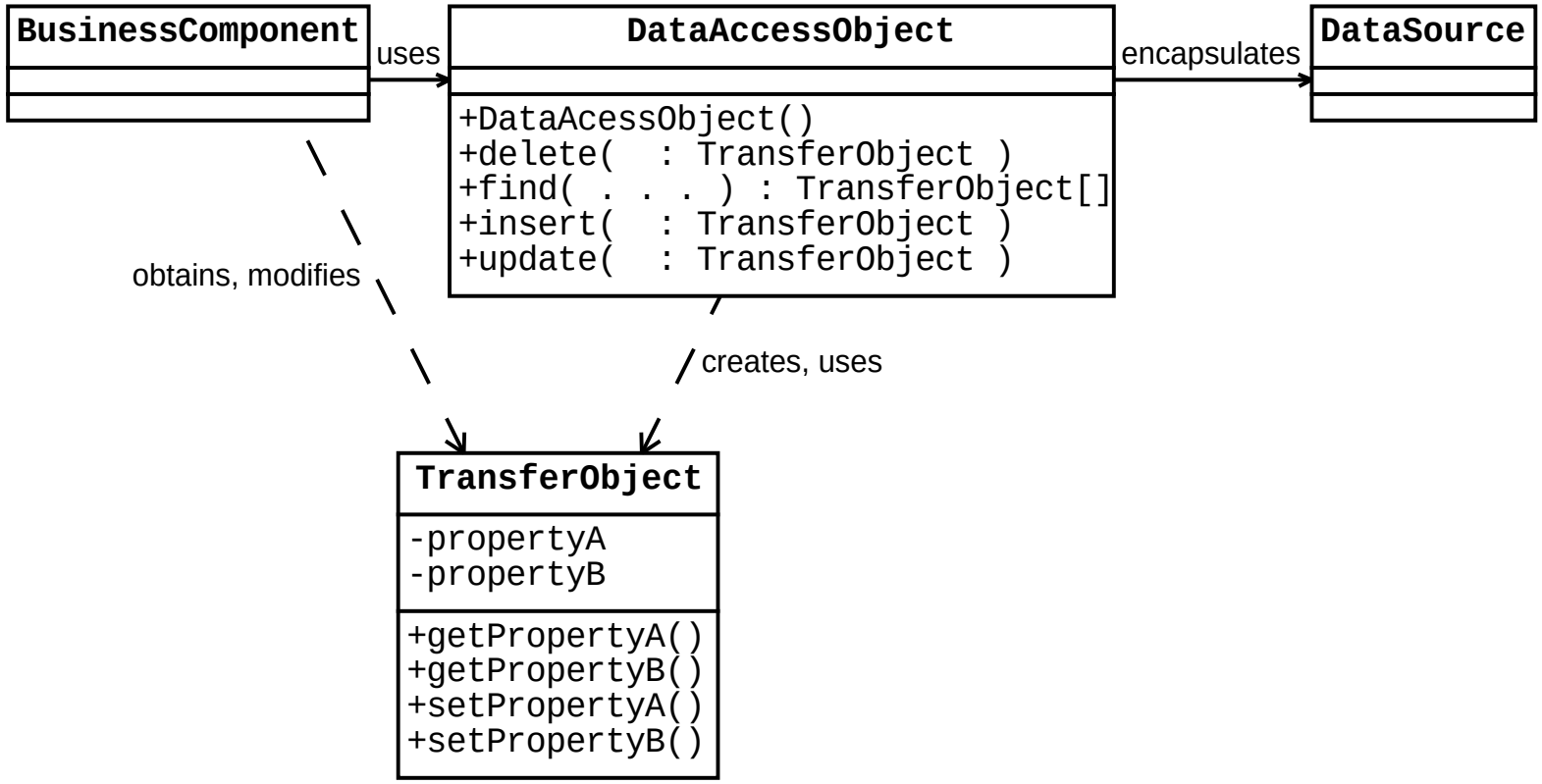
Applying the DAO Pattern: Problem Forces

- Accessing different data sources usually requires different code
- Data source changes should not force rewrites of the business logic
- Data access objects need to be easy to find and create
- Persistence code is more reusable if it is not integrated with business logic

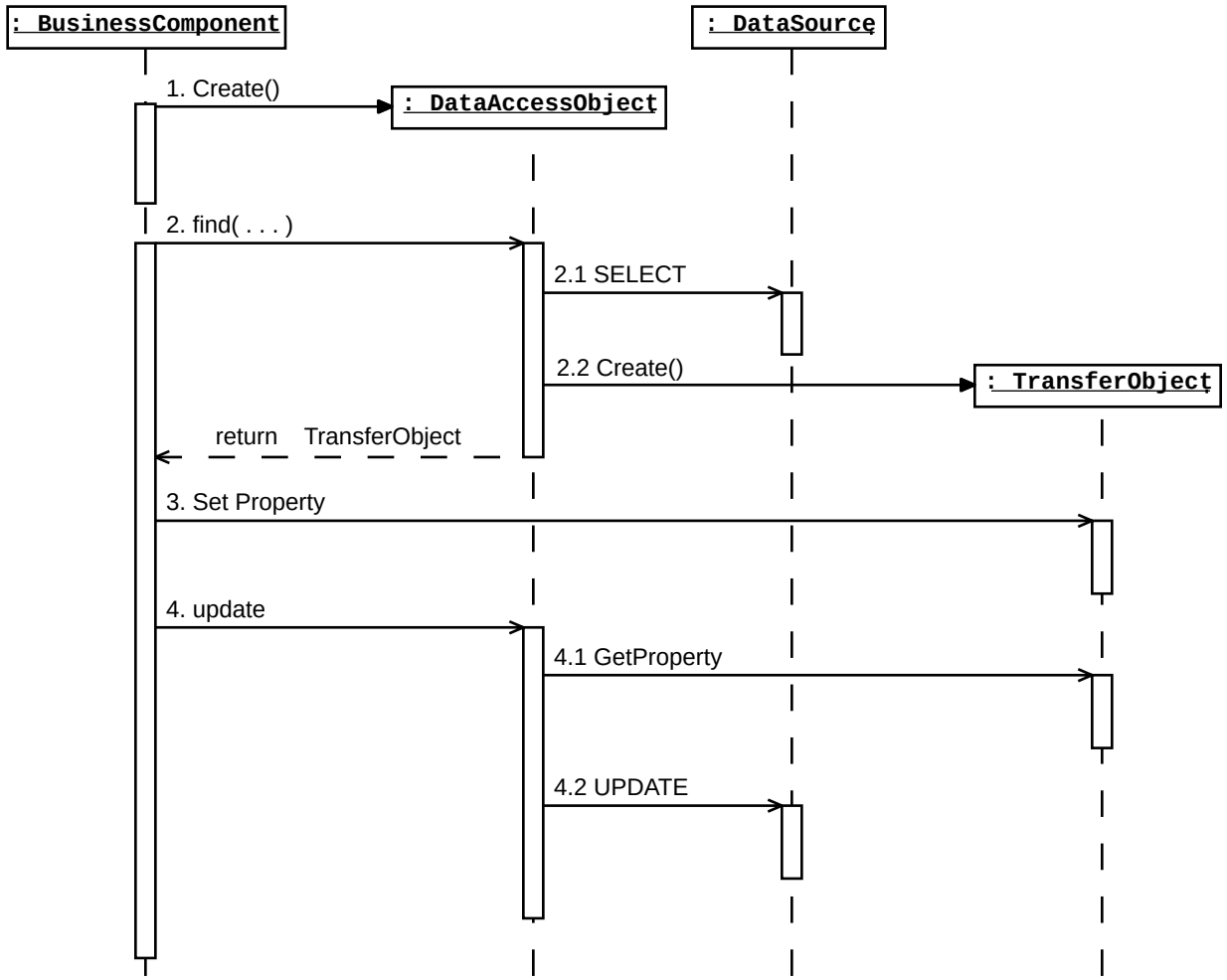
Applying the DAO Pattern: Solution

- Data access code is factored out of the business objects into POJOs called `DataAccessObjects`
- When product changes take place, new implementations of the `DataAccessObject` interface can be added to the system, maintaining transparency to the `BusinessComponent`
- `DataAccessObjects` are usually stateless classes
- `DataAccessObjects` should not expose classes from packages like `java.sql`

DAO Pattern Structure



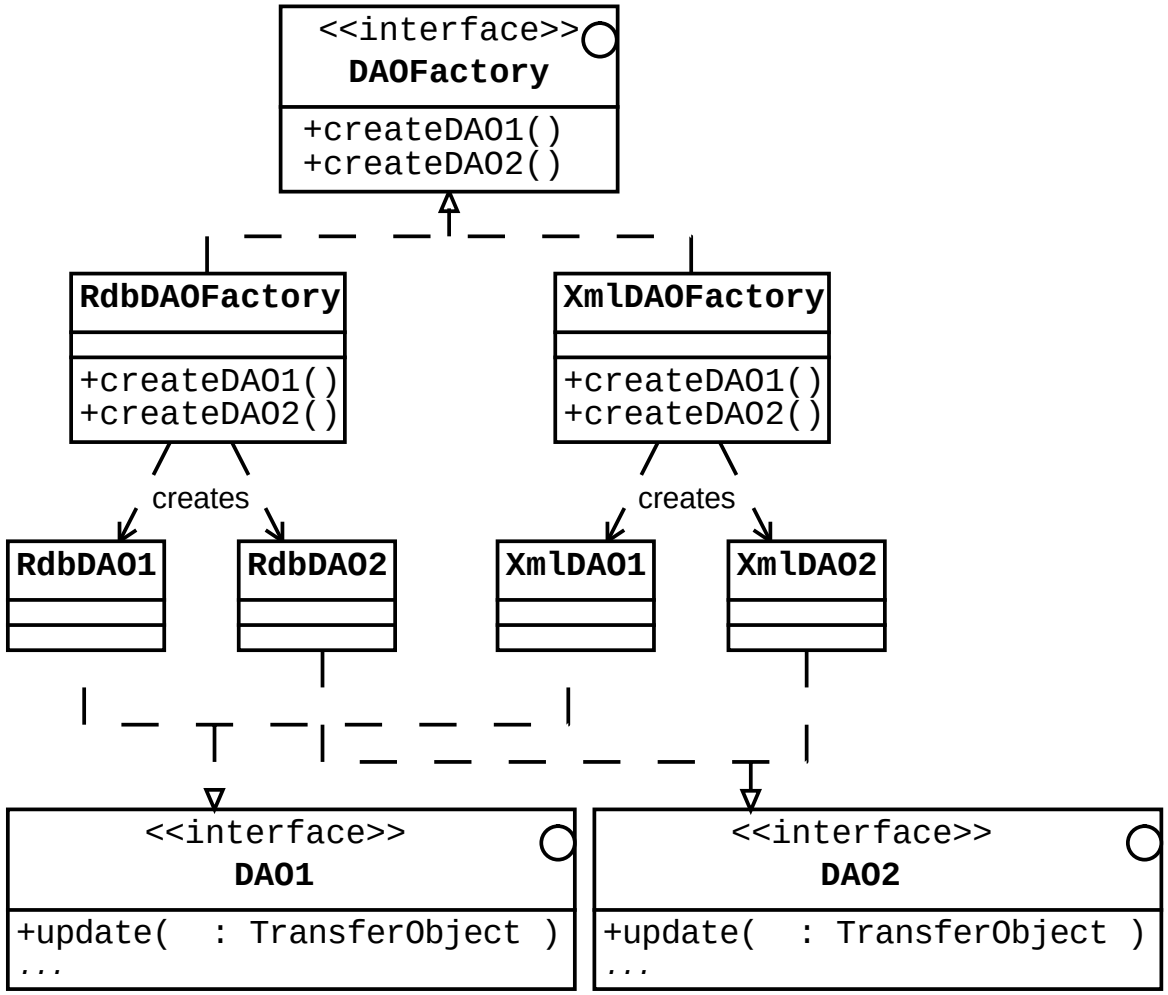
DAO Pattern Sequence



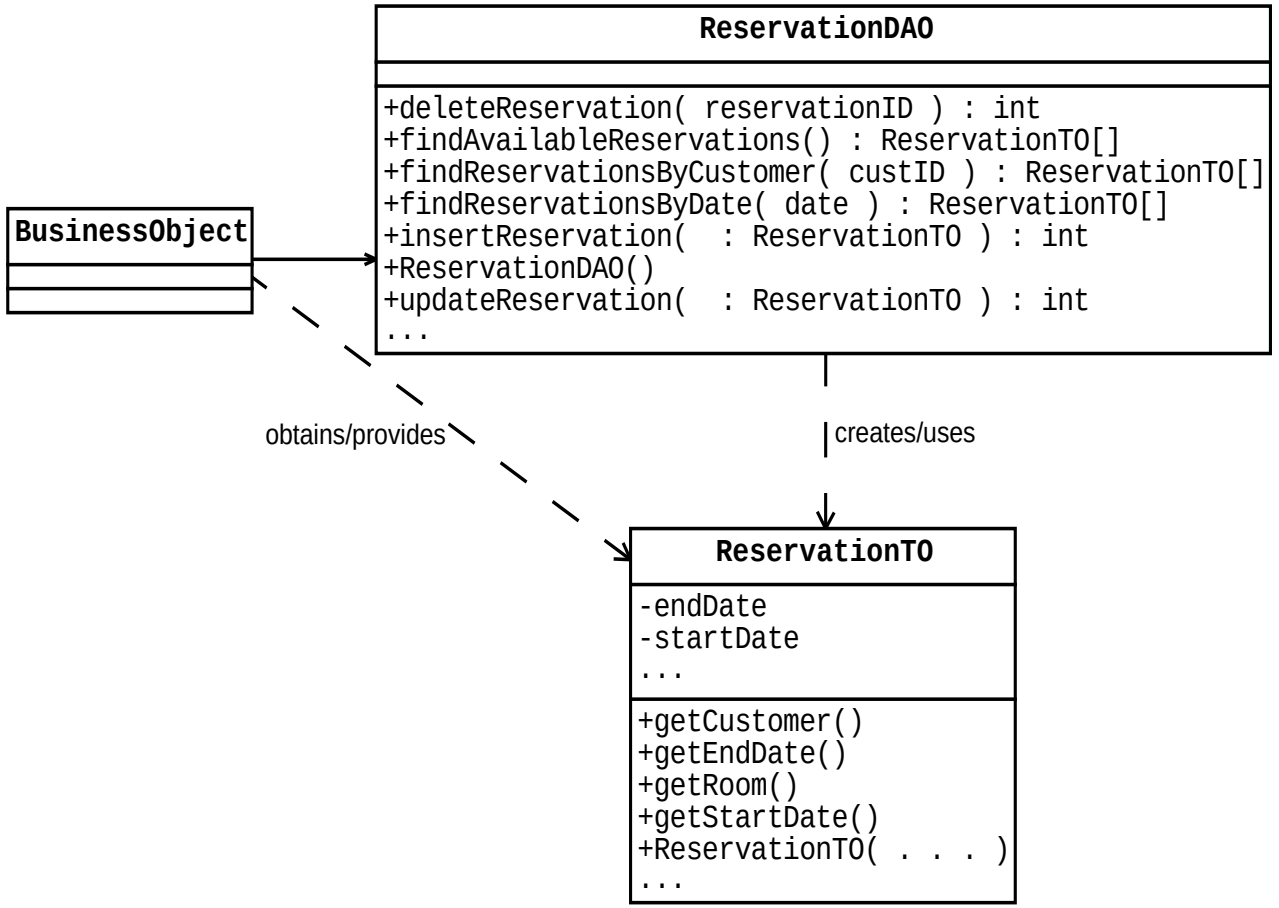
Applying the DAO Pattern: Strategies

- **Transfer Object Collection** – DAO finder methods create simple data structures, called transfer objects, for each row that is returned from the query and placed in a collection
- **Rowset Wrapper List** – Instead of returning transfer objects, the DAO can return a class that is a wrapper around a caching or read-only JDBC technology rowset
- **Factory for the Data Access Object** – The GoF Factory Method and Abstract Factory patterns can be a useful way to create DAO objects

DAO Abstract Factory Strategy Structure



DAO Pattern Example



DAO Pattern Example

```
1   import java.sql.*;
2   import java.util.*;
3   import javax.sql.*;
4   import javax.naming.*;
5
6   public class ReservationDAO {
7     private DataSource ds;
8
9     public ReservationDAO() throws DAOException {
10      try {
11        InitialContext ic = new InitialContext();
12        ds=(DataSource)
13        ic.lookup("java:comp/env/jdbc/hotelDS");
14      } catch (NamingException e) {
15        throw new DAOException
16          ("Error obtaining DataSource", e);
17      }
18    }
19    public ArrayList findReservationByDate
20        (java.util.Date theDate)
21        throws DAOException {
22      ArrayList data = new ArrayList();
23      Connection conn=null;
24      PreparedStatement stmt = null;
25      ResultSet results = null;
26      try {
27        conn = ds.getConnection();
28        String query = "SELECT * FROM RESERVATION“ +
29          “ WHERE DATE = ?";
```

DAO Pattern Example

```
30          stmt = conn.prepareStatement(query);
31          stmt.setDate(1,
32            new java.sql.Date(theDate.getTime()));
33          results = stmt.executeQuery();
34
35          java.util.Date startDate, endDate;
36          while (results.next()) {
37            startDate = results.getDate("startDate");
38            endDate = results.getDate("endDate");
39            data.add(
40              new ReservationTO(startDate, endDate));
41          }
42        }
43        catch (SQLException e) {
44          throw new DAOException
45            ("Error finding reservation", e);
46        }
47        finally {
48          try {
49            if (results != null)
50            results.close();
51            if (stmt != null)
52            stmt.close();
53            if (conn != null)
54            conn.close();
55          } catch (SQLException e) {
56          throw new DAOException
57            ("Error closing connection", e);
58          }
```

DAO Pattern Example

```
59      }
60      return data;
61    }
62    public int deleteReservation
63        (String reservationID) throws DAOException{
64      Connection conn=null;
65      PreparedStatement stmt = null;
66      int count=0;
67      try {
68        conn=ds.getConnection();
69        String query= "DELETE FROM Reservation “
70          + “WHERE ReservationID = ?";
71        stmt=conn.prepareStatement(query);
72        stmt.setString(1, reservationID);
73        count = stmt.executeUpdate();
74      }
75      //. . .implement exception handling
76    }
77    //. . .implement other access methods
78  }
```

Applying the DAO Pattern: Consequences

Advantages:

- Factors data access out of business objects
- Creates a more flexible system for future migration efforts
- Increases cohesiveness in business objects
- Isolates data access logic in a separate tier

Disadvantages:

- An extra layer of infrastructure to be developed is added
- It is not useful with CMP entity beans

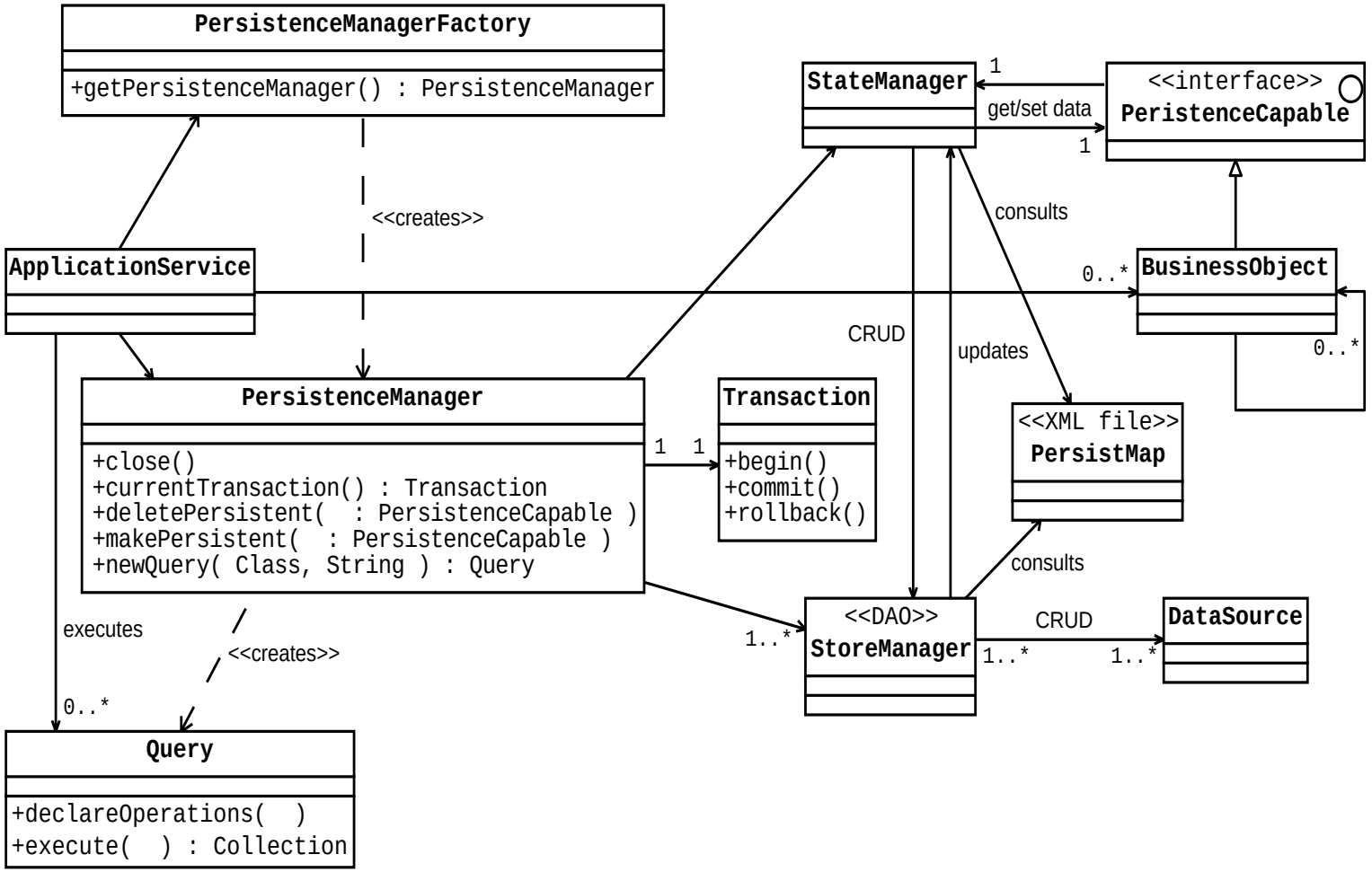
Applying the Domain Store Pattern: Problem Forces

- The DAO pattern separates the persistence code from the business objects, but the pattern does not make the code transparent to the business objects
- Many enterprise applications require sophisticated persistence mechanisms
- Entity beans are not an appropriate solution for some systems:
  - If many of the services of entity beans are not required
  - If the application does not use an EJB container
  - If entity beans constraints, such as no inheritance between business objects, are not acceptable

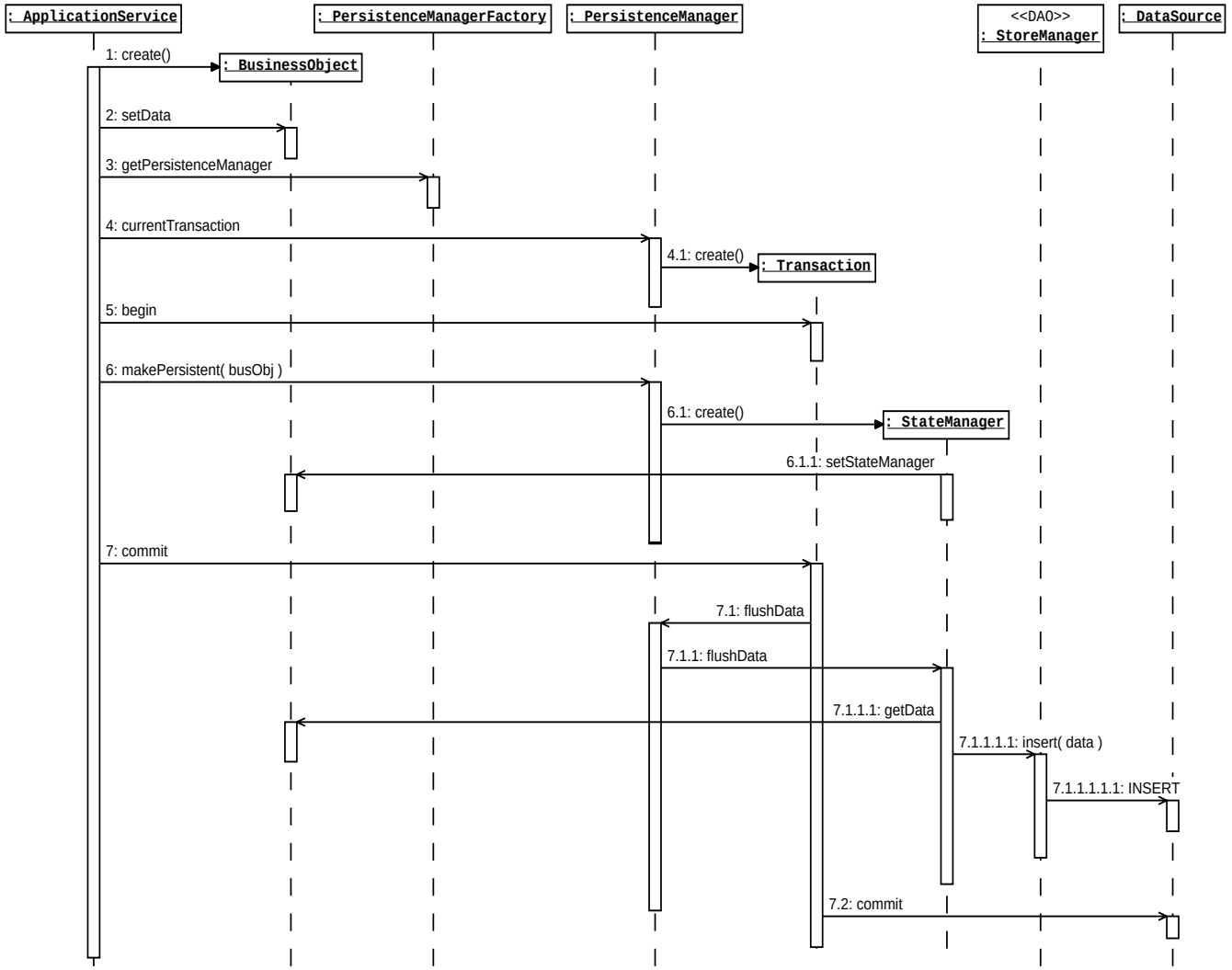
Applying the Domain Store Pattern: Solution

- A domain store is a set of persistence classes that transparently persist business objects
- You can implement a custom domain store implementation, but this is quite a bit of work
- More frequently you might use a persistence product that is based on Java Data Objects (JDO) or a proprietary object relational mapping product
- Although few developers are likely to implement this pattern themselves, learning the pattern will help you learn how these products work

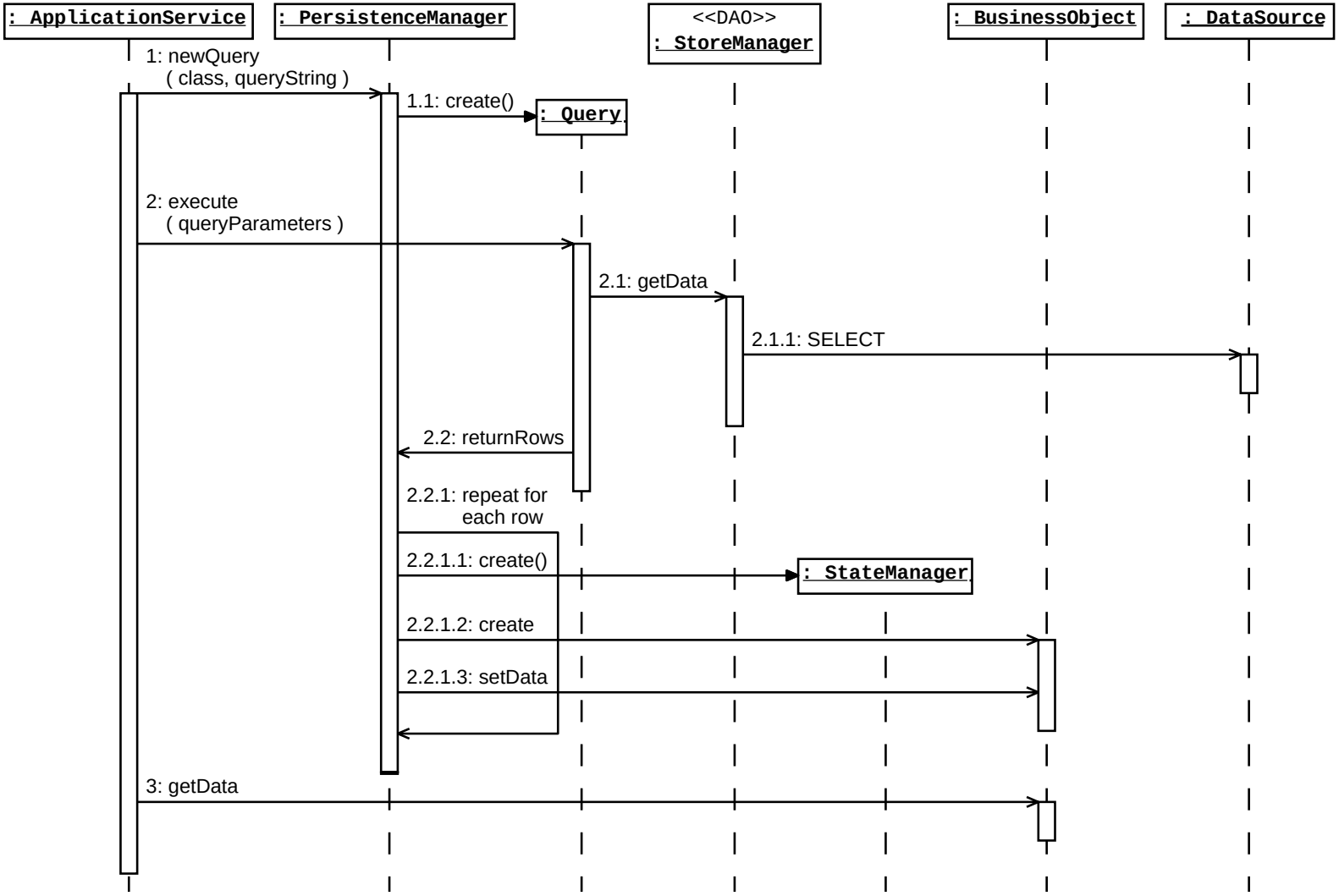
Domain Store Pattern Structure



Domain Store Persistence Sequence



Domain Store Query Sequence



Applying the Domain Store: Example

The `Payment` Business Object:

```
1   package sl500;
2   import java.util.Date;
3   public class Payment {
4     private double amount;
5     private Date datePaid;
6     private PaymentMethod payMethod;
7
8     private Payment() {}
9     public Payment(double a, Date d, PaymentMethod p) {
10      amount = a;
11      datePaid = d;
12      payMethod = p;
13    }
14    //. . .standard plain get/set methods
15  }
```

Applying the Domain Store: Example

The `PaymentMethod` Business Object:

```
1   package sl500;
2
3   import java.util.Date;
4
5   public class PaymentMethod {
6     private String type;
7     private String number;
8     private Date expiration;
9
10    private PaymentMethod() {}
11    public PaymentMethod(String t, String n, Date e) {
12      type = t;
13      number = n;
14      expiration = e;
15    }
16    //. . .standard plain get/set methods
17  }
```

Applying the Domain Store Pattern: Example

JDO Descriptor:

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects
    Metadata 1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
3   <jdo>
4     <package name="sl500">
5       <class name="Payment">
6         <field name="payMethod">
7           <collection element-type="PaymentMethod"/>
8         </field>
9       </class>
10      <class name="PaymentMethod"/>
11    </package>
12  </jdo>
```

Applying the Domain Store Pattern: Example

The `PaymentService` Application Service:

```
1   package sl500;
2   import javax.jdo.*;
3   import java.io.*;
4   import java.util.*;
5
6   public class PaymentService {
7     private PersistenceManager pm;
8
9     public PaymentService() throws IOException {
10      InputStream propertyStream =
11        new FileInputStream("jdo.properties");
12      Properties jdoProps = new Properties();
13      jdoProps.load(propertyStream);
14      PersistenceManagerFactory pmf =
15        JDOHelper.getPersistenceManagerFactory(jdoProps);
```

Applying the Domain Store Pattern: Example

```
16  pm = pmf.getPersistenceManager();
17    }
18    public void makePayment(double amount,
19      String type, String number, Date expiration){
20      Transaction tx = pm.currentTransaction();
21      tx.begin();
22      PaymentMethod method = new PaymentMethod
23      (type, number, expiration);
24     Payment thePayment = new Payment(amount, new Date(), method);
25      pm.makePersistent(thePayment);
26      tx.commit();
27    }
28    public void queryPayments() {
29      Extent e = pm.getExtent(Payment.class, true);
30      Iterator i = e.iterator();
31      while (i.hasNext())
32      {
33        Payment p = (Payment) i.next();
```

Applying the Domain Store Pattern: Example

```
34  System.out.println(p.getAmount() + " - " +
35          p.getPayMethod().getType());
36      }
37      e.close(i);
38    }
39  }
```

Applying the Domain Store Pattern: Strategies

- **Custom Persistence** – You implement the entire domain store pattern from scratch.
- **Java Data Objects** – You write the business objects, the queries, and the application services. The core persistence code is provided by the JDO implementation and is transparent to the developer's code.

Applying the Domain Store Pattern: Consequences

Advantages:

- Allows many benefits of entity beans without the overhead of other entity beans services
- Creates more robust and transparent persistence capabilities than the DAO pattern
- Separates the persistence from the business objects

Applying the Domain Store Pattern: Consequences

Disadvantages:

- You should implement this pattern from scratch only when all other options seem undesirable
- A full persistence framework might be overkill for some applications

Applying the Web Service Broker Pattern: Problem Forces

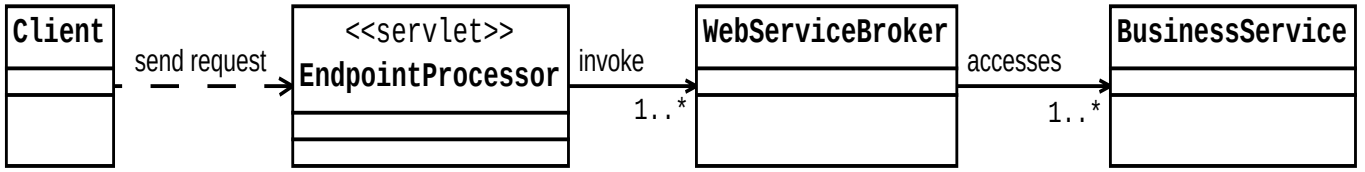
If business service facades are directly exposed as web services the following problems can occur:

- They might contain methods that are insufficiently coarse grained to be effective web services
- They might contain some methods that should not be exposed as web services
- Some of their methods might need to be adapted to make effective web services
- They might require additional monitoring, auditing, logging, or security code

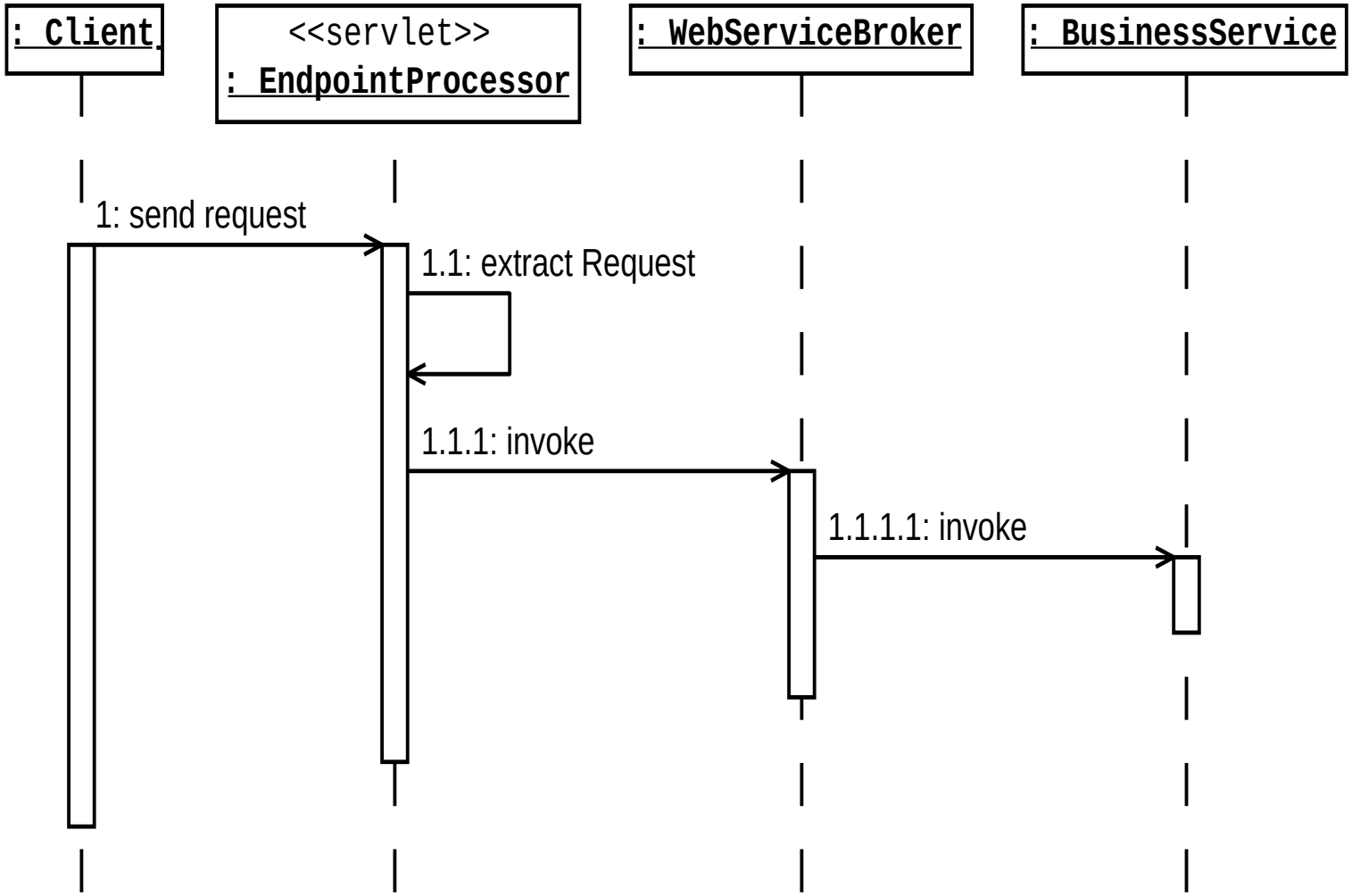
Applying the Web Service Broker Pattern: Solution

- Create a `WebServiceBroker` component that implements any code specific to handling the web service request and then invokes the `BusinessService` component
- The web services request is received by the `EndPointProcessor` component which invokes the `WebServiceBroker`
- The `WebServiceBroker` component might be a EJB 2.1 stateless session bean or a plain old Java object

Web Service Broker Pattern Structure



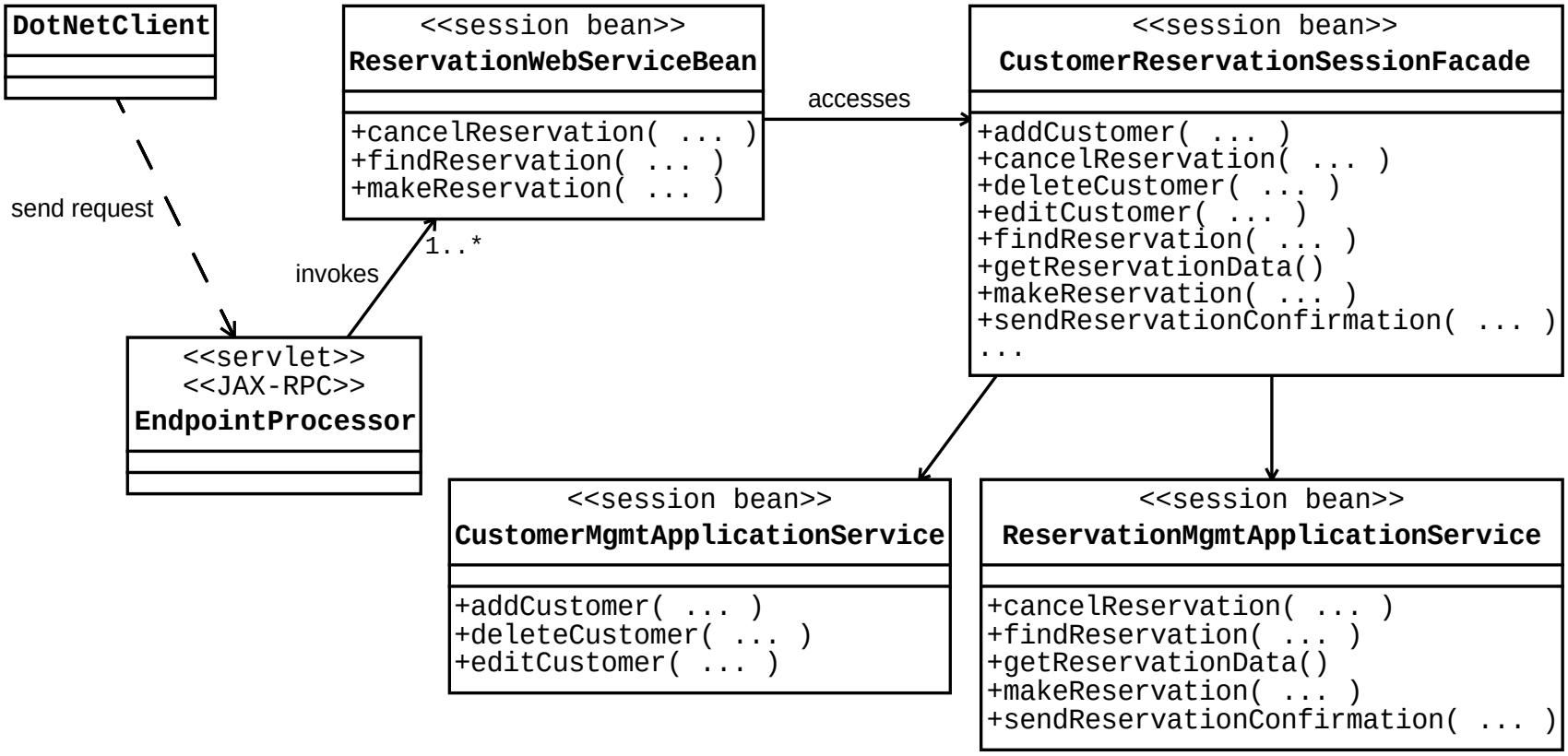
Web Service Broker Pattern Sequence



Applying the Web Service Broker Pattern: Strategies

- **Custom Extensible Markup Language (XML) Messaging** – You write your own custom `EndPointProcessor` component
- **Java™ API for XML-based RPC (JAX-RPC)** – The `EndPointProcessor` servlet is provided by JAX-RPC
- **Java Binder Strategy** – The Java Architecture for XML Binding (JAXB) can be used to process the XML and generate transfer objects based on the XML

Web Service Broker Pattern Example



Web Service Broker Pattern Example

The `ReservationWebService` Interface:

```
1   package sl500;
2
3   import java.rmi.*;
4
5   public interface ReservationWebService extends Remote {
6     public boolean makeReservation(. . .) throws RemoteException;
7     public boolean cancelReservation(. . .) throws RemoteException;
8   }
```

Web Service Broker Pattern Example

The ReservationWebServiceBean Class:

```
1   package sl500;
2   import javax.ejb.*;
3   public class ReservationWebServiceBean implements SessionBean {
4     public void ejbCreate() {
5      //. . . Obtain session facade bean
6     }
7     public boolean makeReservation(. . .) {
8       //. . .Call makeReservation on session facade bean
9     }
10    public boolean cancelReservation(. . .) {
11      //. . .Call cancelReservation on session facade bean
12    }
13    public void ejbRemove() {}
14    public void ejbActivate() {}
15    public void ejbPassivate() {}
16    public void setSessionContext(SessionContext sc) {}
17  }
```

Applying the Web Service Broker Pattern: Consequences

Advantages:

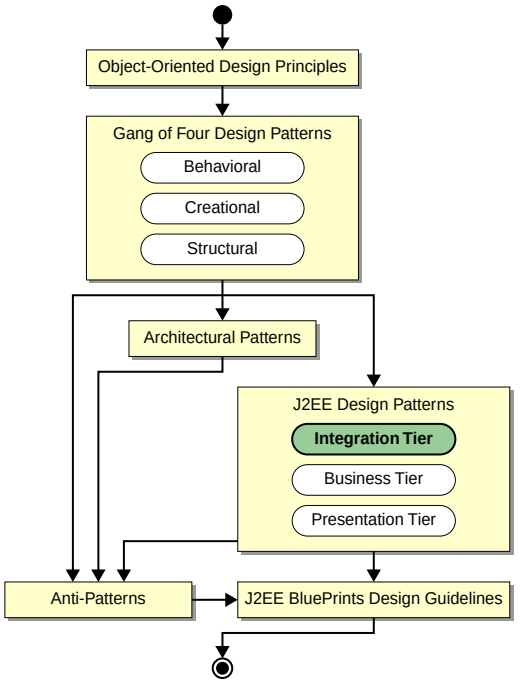
A layer is introduced where web service specific code can be added.

Disadvantages:

- Web services can increase network overhead due to the protocols used.
- The business service component is likely to have a remote interface for access by the presentation tier. A local interface must be added for use by the web service broker component.

Summary

- **Service Activator pattern** – Provides asynchronous messaging for business components, including EJB components
- **Data Access Object pattern** – Encapsulates data store details
- **Domain Store pattern** – Creates a robust persistence mechanism that is transparent to the business objects without using entity beans



- **Web Service Broker pattern** – Provides a flexible way to make business services available as web services

Module 8

Using Presentation-to-Business Tier Patterns

Objectives

- Describe basic characteristics of the business tier J2EE patterns that facilitate communication with the presentation tier
- Apply the Service Locator pattern
- Apply the Session Façade pattern
- Apply the Business Delegate pattern
- Apply the Transfer Object pattern

Introducing Business Tier Patterns

Business Tier patterns:

- Built primarily with EJB components
- Organize business logic
- Reduce coupling
- Reduce network overhead

The patterns in this module focus on structuring the interaction between the presentation and business tiers

J2EE Platform Business Tier Patterns

Pattern	Primary Function
Service Locator	Provides one interface for service lookups.
Session Façade	Hides business component detail with a simpler interface.
Business Delegate	Decouples presentation clients and business services.
Transfer Object	Reduces remote calls by encapsulating return value data into one object.

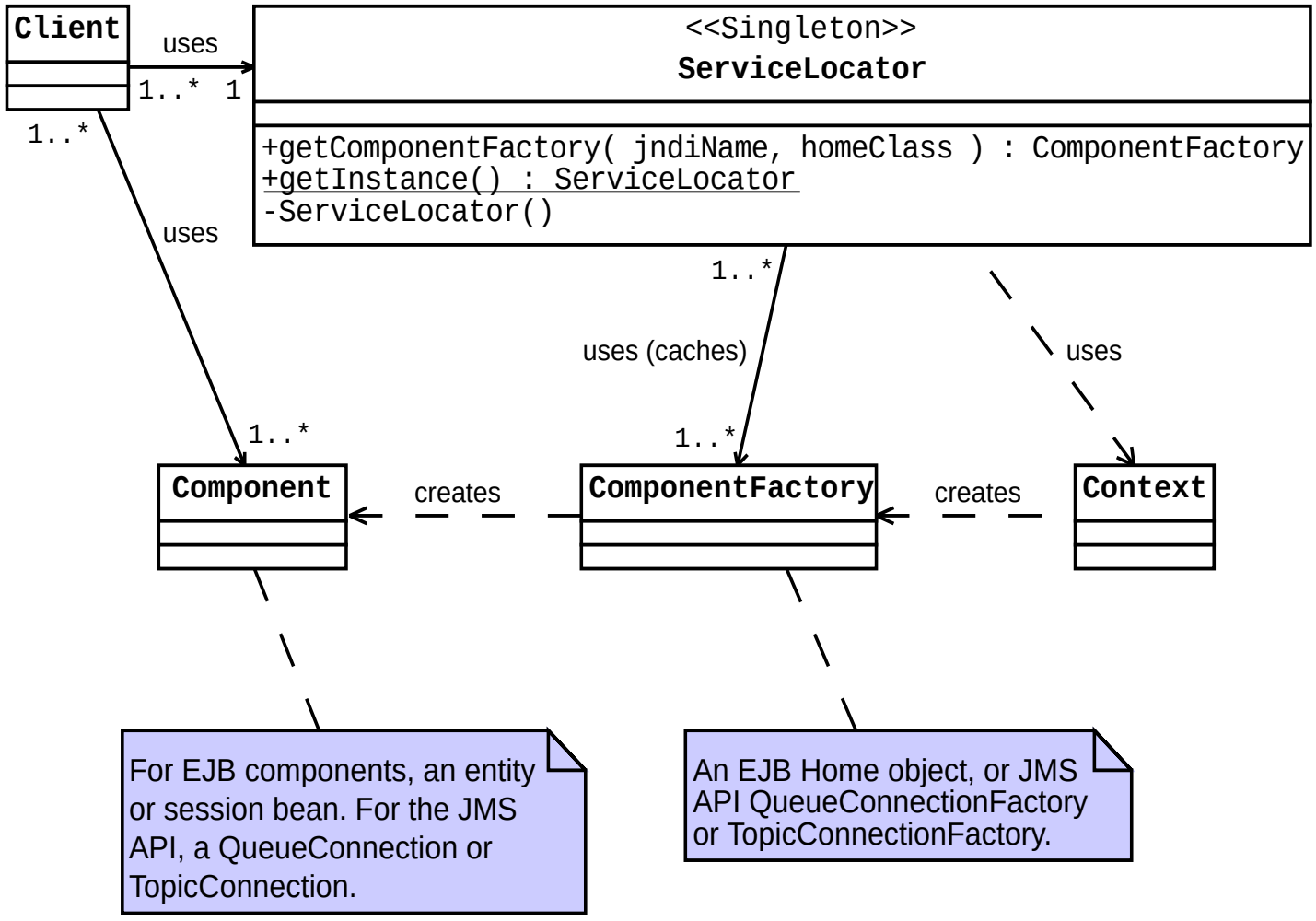
Applying the Service Locator Pattern: Problem Forces

- All session or entity beans clients need the same JNDI API code to locate a home object and to create a bean
- Disadvantages of putting this code in each client:
  - Code is duplicated
  - Client programmer has to know how to write bean client code
  - Reusable home objects are not cached
- The same issues apply to clients looking up JMS `ConnectionFactory` objects

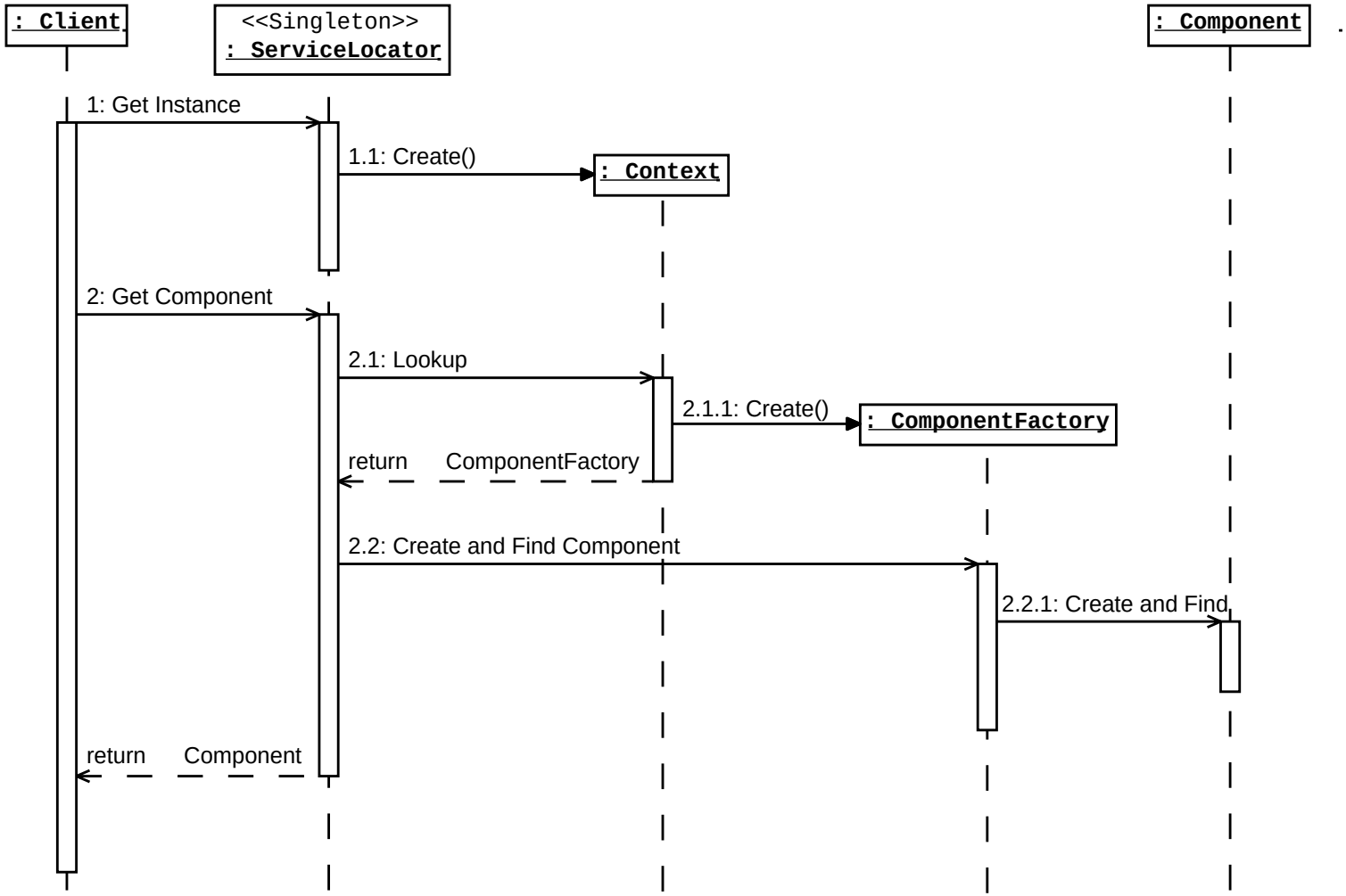
Applying the Service Locator Pattern: Solution

- The `ServiceLocator` class is implemented as a singleton
- The `ServiceLocator` object creates a JNDI `Context` object and does a lookup of the `ComponentFactory` object
- The `ServiceLocator` can either:
  - Return `ComponentFactory` to the `Client` class that uses it to create `Component` instances
  - Use `ComponentFactory` to create the `Component` instances
- The `ServiceLocator` has a cache of previously found `ComponentFactory` instances

Service Locator Pattern Structure



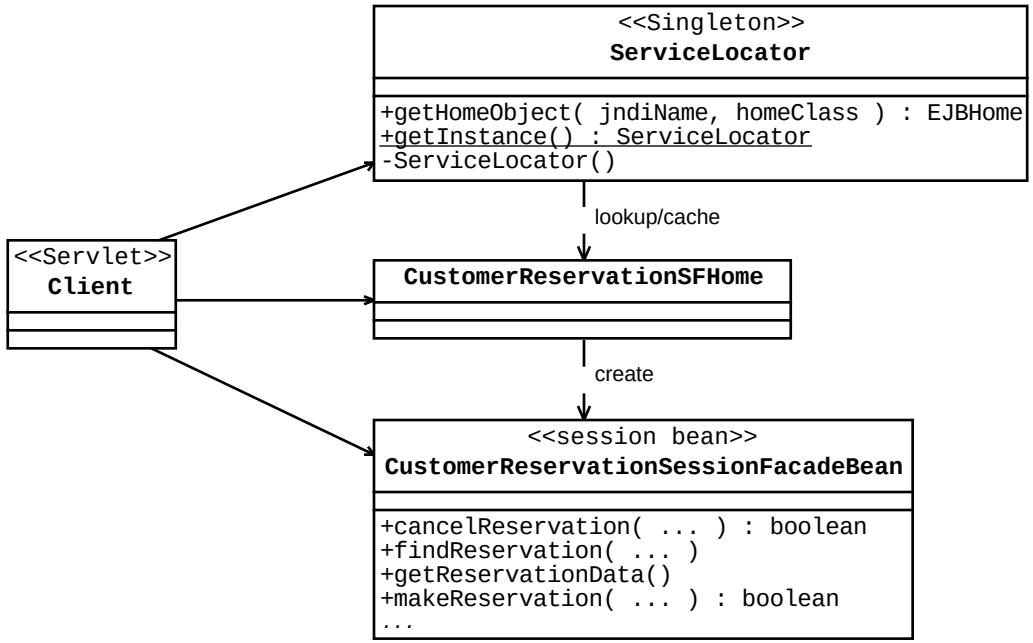
Service Locator Pattern Sequence



Applying the Service Locator Pattern: Strategies

- **EJB Component Service Locator**
- **JMS API Topic and Queue Service Locator**
- **Web Service Locator**
- **Combined Service Locator –** Some combination of the above three strategies
- **Type Checked Service Locator –** The component's JNDI name and the resource class are hard coded in the `ServiceLocator` class and the client passes in a parameter to specify which set of values to use
- **Service Locator Properties –** The same as the last strategy, but the JNDI name and resource class name are specified in a configuration file

Service Locator Pattern Example



Service Locator Pattern Example

```
1   public class ServiceLocator {
2     private HashMap nameHomePairs = null;
3     private Context ctx = null;
4     private static ServiceLocator singleton = null;
5
6     private ServiceLocator() throws NamingException{
7       nameHomePairs = new HashMap();
8       ctx = new InitialContext();
9     }
10    public static ServiceLocator getInstance()
11        throws NamingException {
12      if (singleton == null) {
13        singleton = new ServiceLocator();
14      }
15      return singleton;
16    }
```

Service Locator Pattern Example

```
17    public EJBHome getHomeObject
18          (String jndiName, Class homeClass)
19          throws NamingException {
20      if (nameHomePairs.containsKey(jndiName)) {
21        return (EJBHome) nameHomePairs.get(jndiName);
22      }
23      Object ejbRef = ctx.lookup(jndiName);
24      EJBHome ejbHome = (EJBHome)PortableRemoteObject.
25          narrow(ejbRef, homeClass);
26      nameHomePairs.put(jndiName, ejbHome);
27      return ejbHome;
28    }
29  }
```

Applying the Service Locator Pattern: Consequences

Advantages:

- The complexity of lookup and creation is hidden
- Access to the service is centralized and uniform without duplication of the lookup code
- New business components are easily added to the lookup service
- Performance can be improved by caching previously looked up objects

Applying the Service Locator Pattern: Consequences

Disadvantage:

Even as a Singleton, there is one Service Locator instance per Java virtual machine

Applying the Session Façade Pattern: Problem Forces

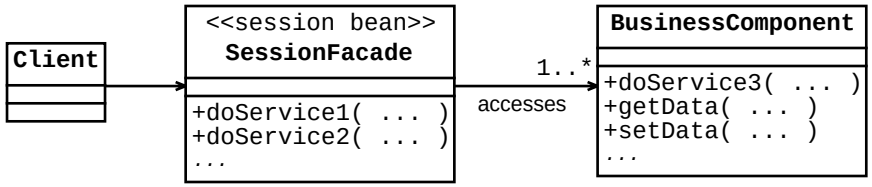
To fulfill a user request, a client might need the services of multiple business components. This causes the following problems:

- When clients interact directly with multiple business components, the tiers become tightly coupled
- Clients that deal with many business components become responsible for managing their interaction
- Fine-grained client access increases network traffic
- Lack of a uniform access strategy to business components leads to ad hoc development and misuse

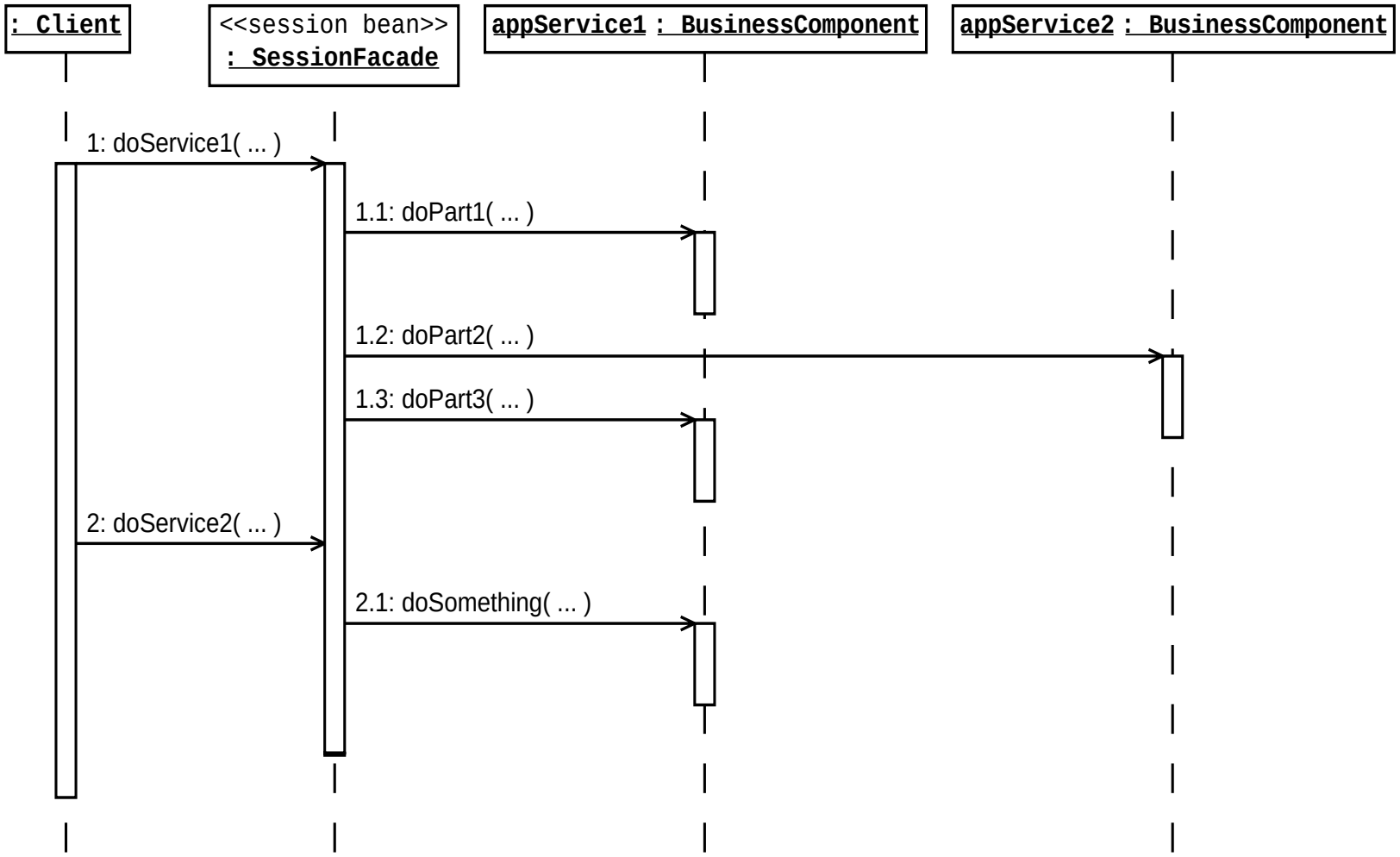
Applying the Session Façade Pattern: Solution

- You apply the GoF Façade pattern with a session bean in front of the other business components
- The client invokes coarse-grained business service methods on the `SessionFacade` component
- The `SessionFacade` component finds, creates, and invokes the necessary `BusinessComponent` instances
- The `SessionFacade` component should usually not contain significant business logic
- Mapping each use case to a session facade will create too many session facades

Session Façade Pattern Structure



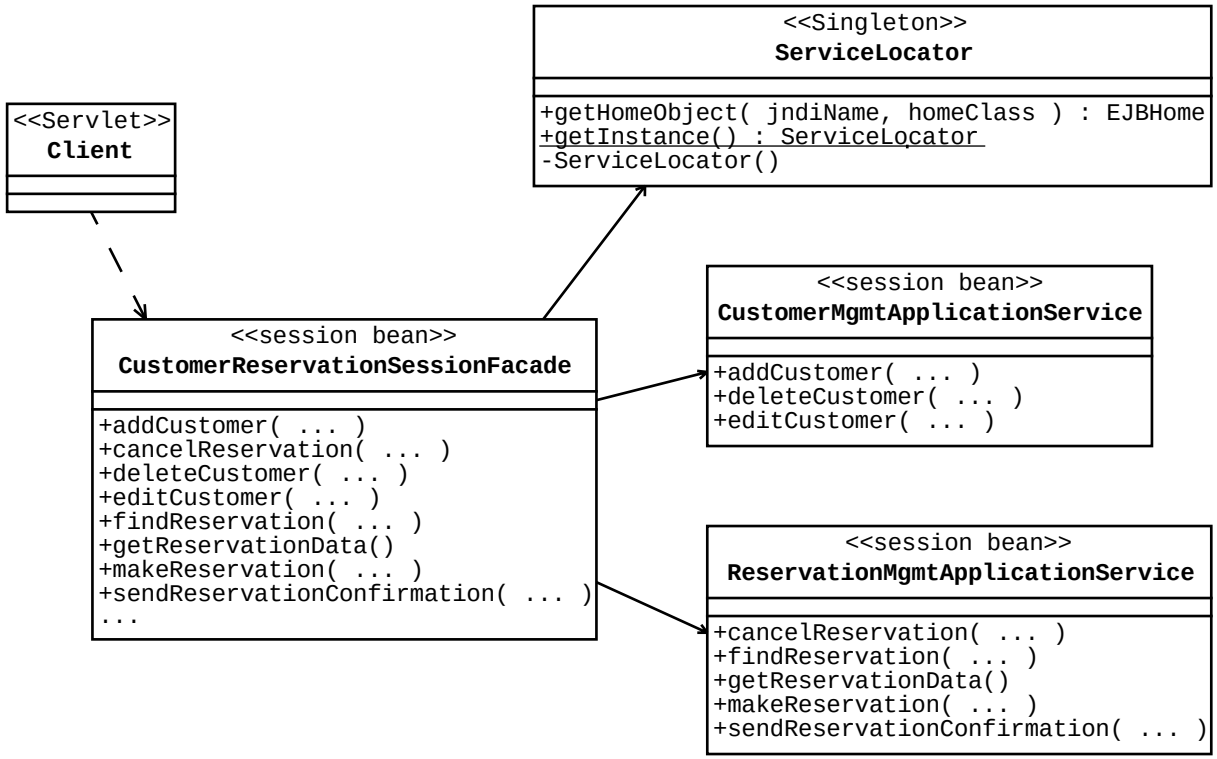
Session Façade Pattern Sequence



Applying the Session Façade Pattern: Strategies

- **Stateless Session Bean** – For single method calls
- **Stateful Session Bean** – For multiple method calls with intermediate state information

Session Façade Pattern Example



Session Façade Pattern Example

```
1   public class CustomerReservationSessionFacade
2       implements SessionBean {
3     private CustMgmtASHome custHome = null;
4     private ResMgmtASHome resHome = null;
5     private CustomerMgmtApplicationService custService = null;
6     private ReservationMgmtApplicationService resService = null;
7
8     public void ejbCreate() throws CreateException {
9       try {
10        ServiceLocator locator = ServiceLocator.getInstance();
11        resHome = (ResMgmtASHome) locator.getHomeObject
12          ("ejb/ResAS", ResMgmtASHome.class);
13        custHome = (CustMgmtASHome) locator.getHomeObject
14          ("ejb/custAS", CustMgmtASHome.class);
15        resService = resHome.create();
16        custService = custHome.create();
17      }
18      catch (Exception e) {
19        System.err.println(e);
```

Session Façade Pattern Example

```
20          throw new CreateException
21            ("Unable to initialize facade");
22        }
23      }
24      public boolean makeReservation(. . .)
25            throws Exception {
26          if (newCustomer==true)
27            custService.addCustomer(. . .);
28          return resService.makeReservation(. . .);
29      }
30    . . .
31    }
```

Applying the Session Façade Pattern: Consequences

Advantages:

- Simplifies client access to the business tier components
- Eliminates a client's direct dependency on multiple business objects
- Reduces network traffic between the service tiers
- Provides a place for centralized security management
- Centralizes transaction control in the session façade
- Provides a uniform interface to the business tier

Applying the Session Façade Pattern: Consequences

Disadvantage:

- Adds an extra layer

Applying the Business Delegate Pattern: Problem Forces

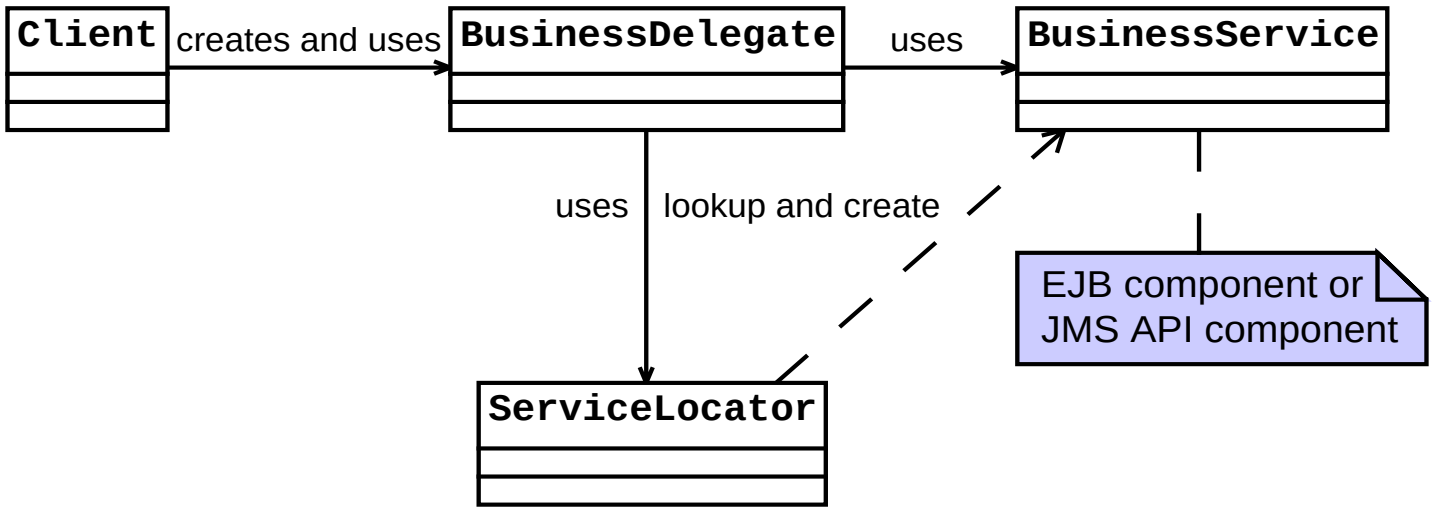
If presentation tier clients interact directly with business services:

- Clients are exposed to the underlying implementation details of the service API
- Business service changes affect the client
- The presentation tier components might make too many remote invocations due to a lack of caching or request aggregation
- Presentation tier programmers needs to know how to invoke the business service components

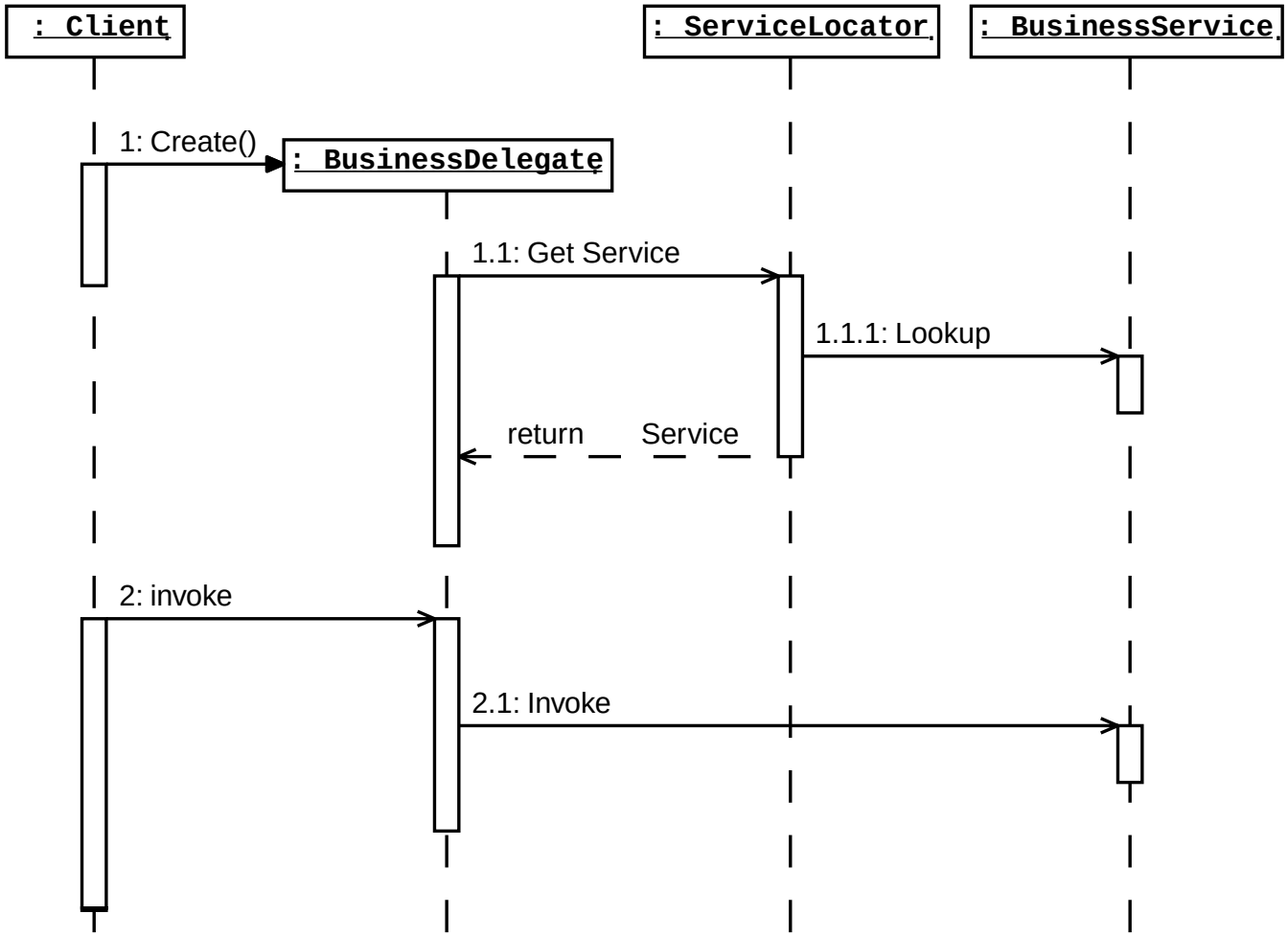
Applying the Business Delegate Pattern: Solution

- On behalf of the `Client` class, the `BusinessDelegate` uses a `ServiceLocator` object to obtain a `BusinessService` component and then invokes business methods on it
- The `BusinessDelegate` might catch technology specific exceptions and throw generic exceptions
- The `BusinessDelegate` might automatically retry failed requests
- The `BusinessDelegate` might cache results

Business Delegate Pattern Structure



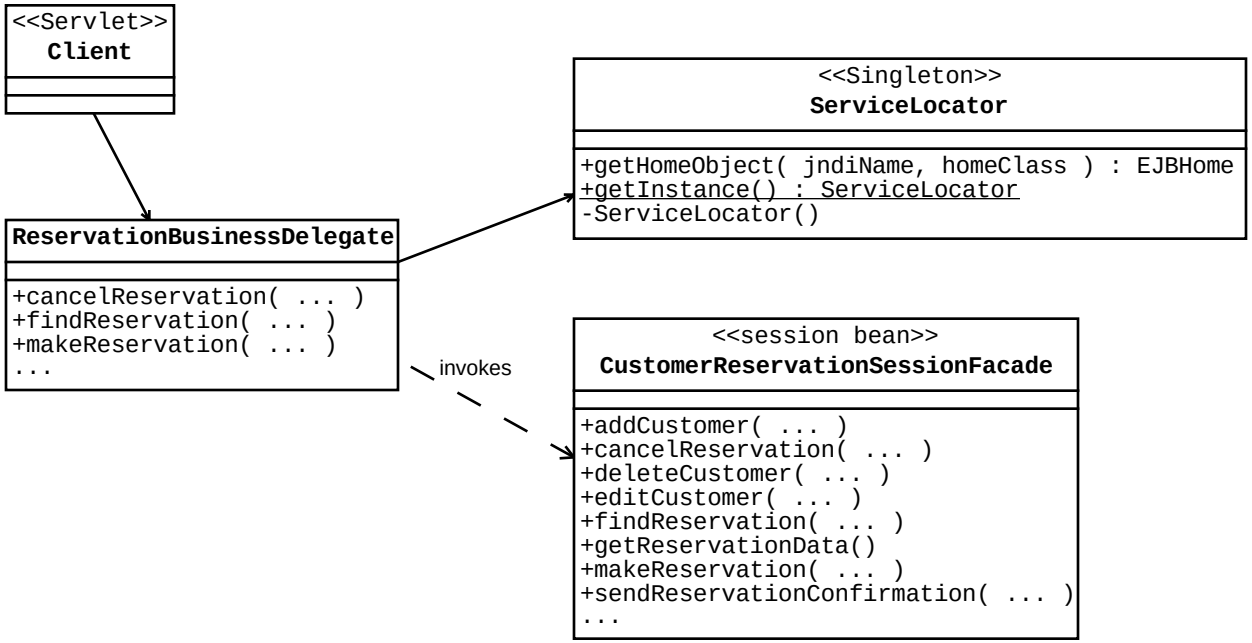
Business Delegate Pattern Sequence



Applying the Business Delegate Pattern: Strategies

- **Delegate Proxy** – The business delegate acts as a proxy that supports approximately the same interface as the business component
- **Delegate Adapter** – Disparate systems can send requests using XML to a business delegate that translates the requests and sends the requests to J2EE components

Business Delegate Pattern Example



Business Delegate Pattern Example

```
1   public class ReservationBusinessDelegate {
2     private ReservationSessionFacade resSF = null;
3     public ReservationBusinessDelegate()
4         throws ReservationException {
5       try {
6         ServiceLocator locator = ServiceLocator.getInstance();
7         ReservationSessionFacadeHome resSFHome =
8           (ReservationSessionFacadeHome)
9           locator.getHomeObject("ejb/ReservationSF",
10          ReservationSessionFacadeHome.class);
11        resSF = resSFHome.create();
12      }
13      catch (NamingException e) {
14        throw new ReservationException();
15      }
16      catch (CreateException e) {
17        throw new ReservationException();
18      }
```

Business Delegate Pattern Example

```
19      catch (RemoteException e) {
20        throw new ReservationException();
21      }
22    }
23    public boolean makeReservation(String resData)
24        throws ReservationException {
25      try {
26        return resSF.makeReservation(resData);
27      }
28      catch (Exception e) {
29        throw new ReservationException();
30      }
31    }
32  }
```

Applying the Business Delegate Pattern: Consequences

Advantages:

- Reducing coupling between the presentation tier and the business tier
- Adding a simple, uniform interface for the clients to use to access business components
- Improving performance made possible by caching requests

Applying the Business Delegate Pattern: Consequences

Disadvantages:

- Adding an additional layer to manage
- Hiding remoteness which can lead developers to ignore the costs of these method invocation

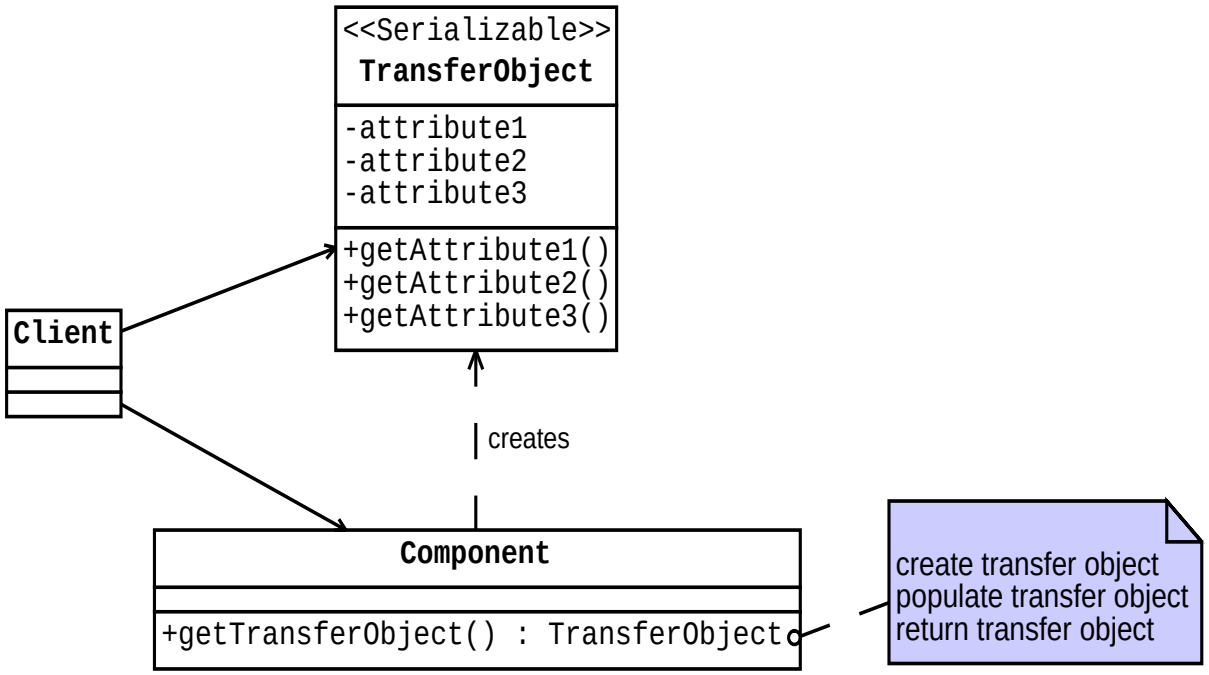
Applying the Transfer Object Pattern: Problem Forces

- Data often needs to be exchanged between remote tiers
- If components only expose fine grained getter methods, the remote client must make one remote method invocation per attribute
- Remote calls should be coarse-grained whenever possible
- Clients typically require more read-only data than updatable data

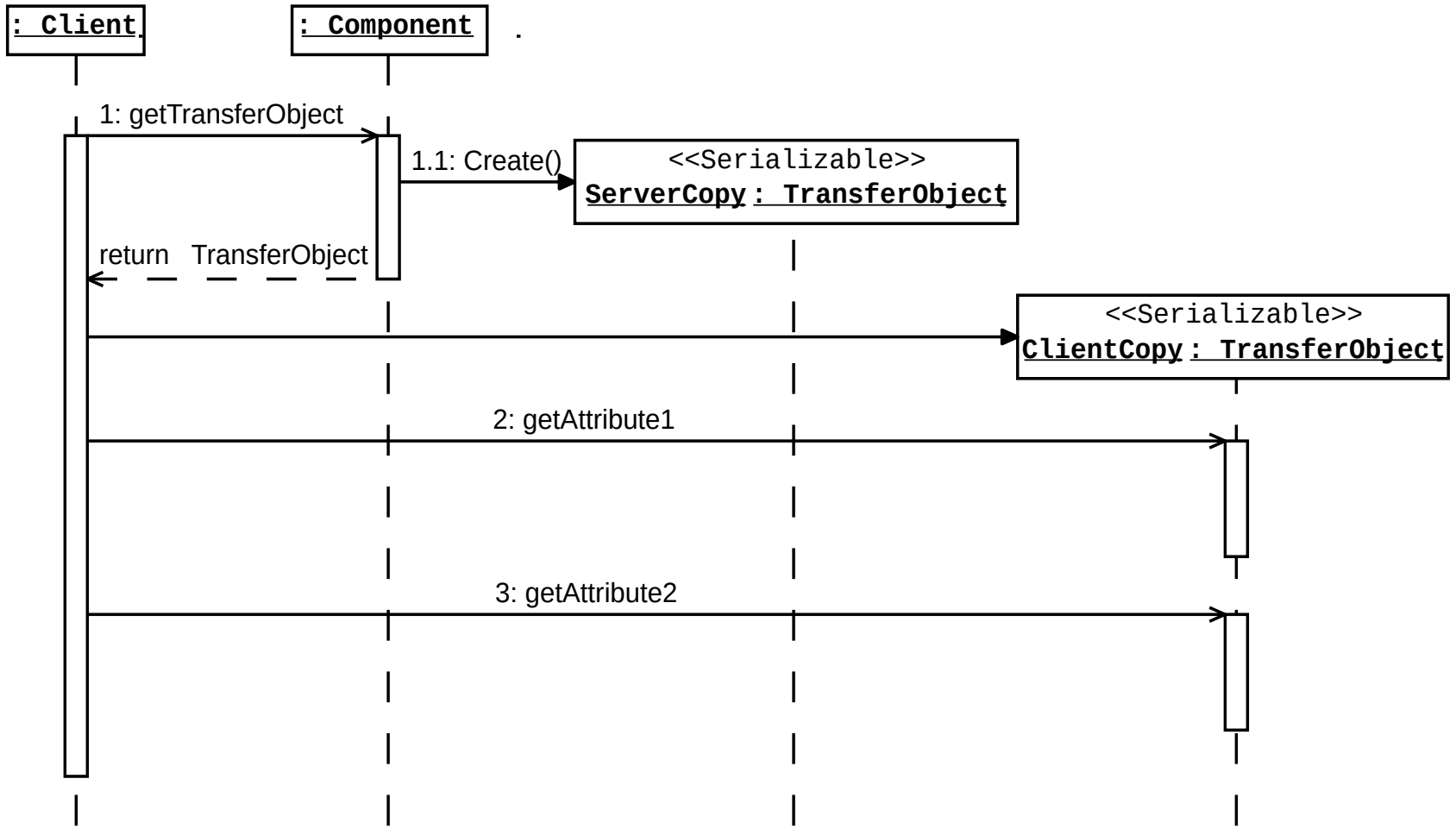
Applying the Transfer Object Pattern: Solution

- A Transfer Object is a serializable JavaBeans component that encapsulates attribute values and provides accessor methods
- The `getTransferObject` method of `Component` creates the `TransferObject`, populates it, and returns it to the `Client` class
- The `Client` class calls `get` methods to get the data from the `TransferObject`
- The network overhead should be low because only the `getTransferObject` method is remote

Transfer Object Pattern Structure



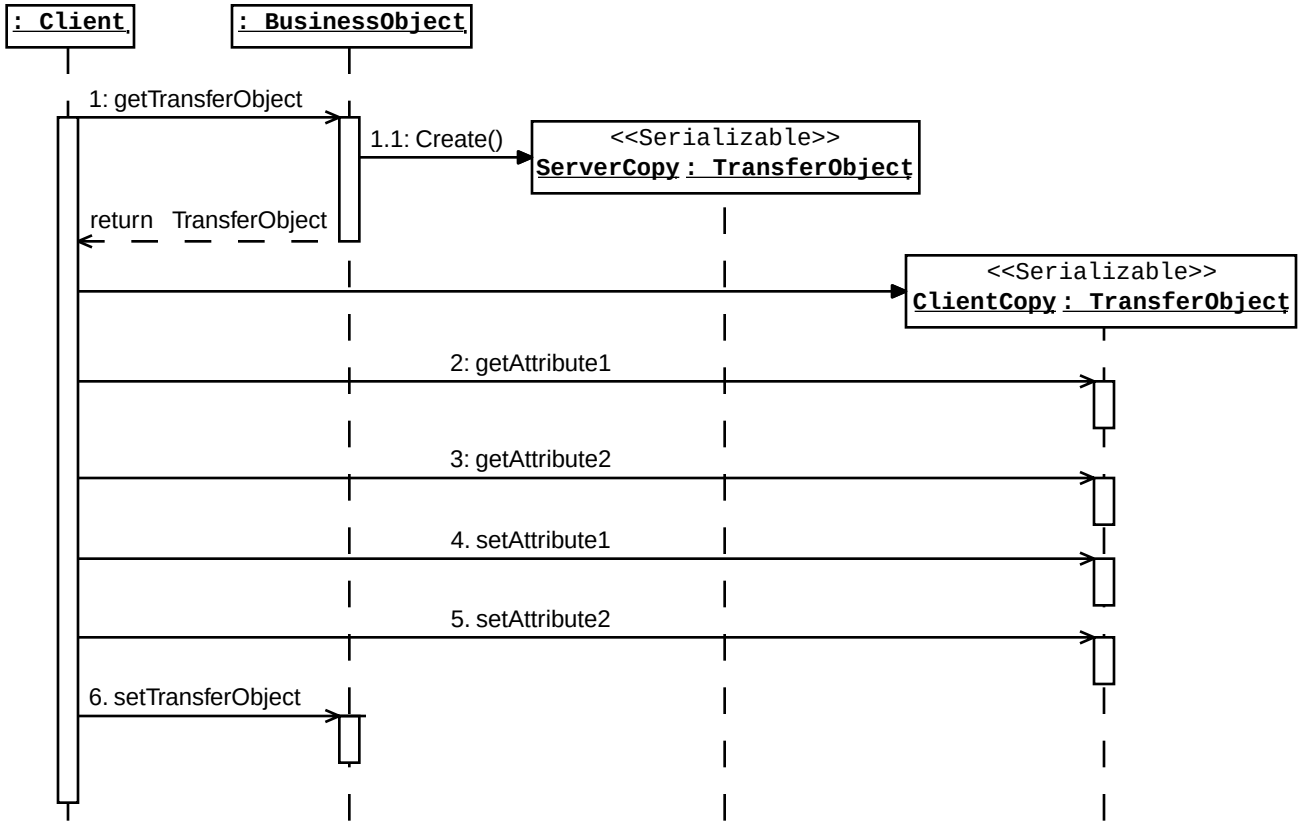
Transfer Object Pattern Sequence



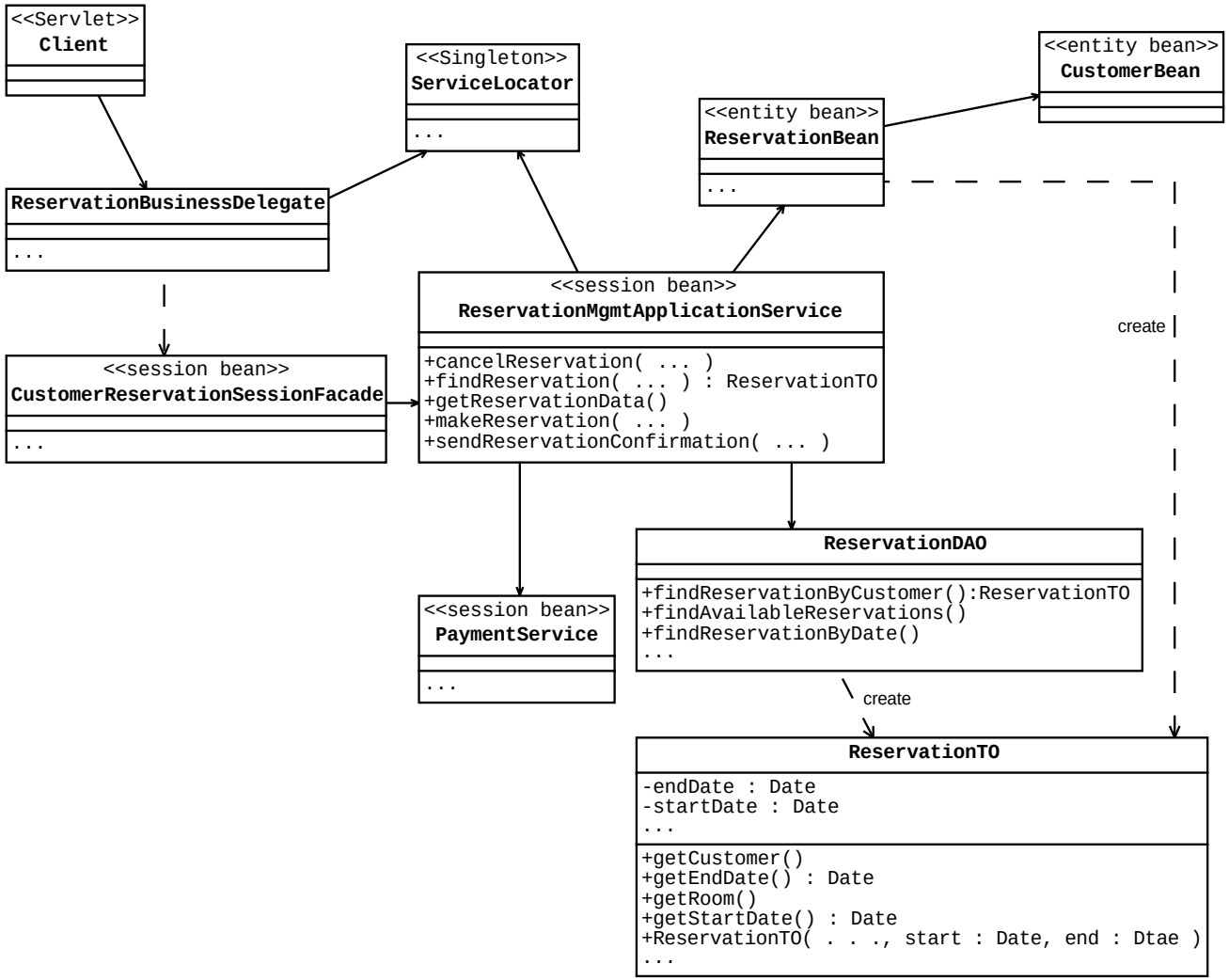
Applying the Transfer Object Pattern: Strategies

- **Multiple Transfer Objects** – The `Component` object provides several `getTransferObject` methods that return different `TransferObject` types
- **Entity Inherits Transfer Object** - The `BusinessObject` is a subclass of the `TransferObject`
- **Updateable (Mutable) Transfer Objects:**
  - A `TransferObject` provides `set` methods
  - A `Component` object's `setTransferObject` method updates the `Component` data
  - Time stamp or version number can be used to protect against overwriting changes made by other clients

Updatable Transfer Objects Sequence



Transfer Object Pattern Example



Transfer Object Pattern Example

```
1   public class ReservationTO implements Serializable
2   {
3     private Date startDate;
4     private Date endDate;
5
6     public ReservationTO() { }
7     public ReservationTO(Date start, Date end) {
8       startDate = start;
9       endDate = end;
10    }
11    public Date getStartDate() {
12      return startDate;
13    }
14    public Date getEndDate() {
15      return endDate;
16    }
17  }
```

Applying the Transfer Object Pattern: Consequences

Advantages:

- Simplifies entity bean and remote interfaces
- Reduces network traffic

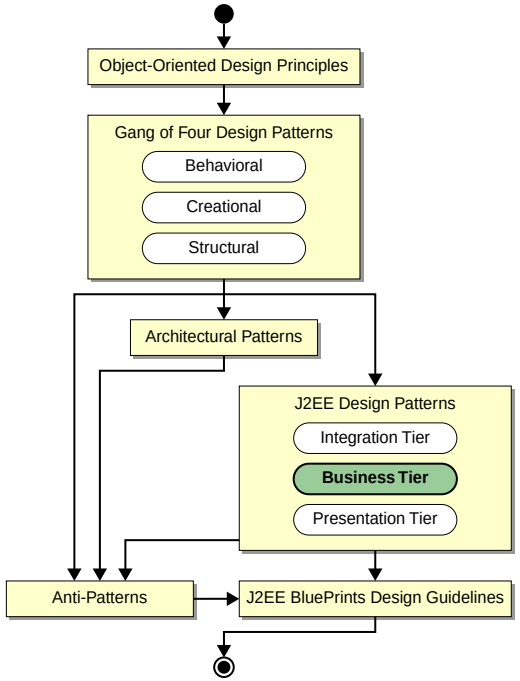
Disadvantages:

- Transfer object data may become stale
- Synchronization logic can complicate the system
- The system can be complicated when using concurrent access and transaction processes

Summary

Presentation to business tier patterns:

- **Service Locator** – Focuses all service lookups through one interface
- **Session Façade** – Provides simpler interfaces to business components
- **Business Delegate** – Decouples presentation tier components from business components
- **Transfer Object** – Improves efficiency of the transfer of data between the tiers



Module 9

Using Intra-Business Tier Patterns

Objectives

- Describe the basic characteristics of the Intra-Business Tier patterns
- Apply the Application Service pattern
- Apply the Business Object pattern
- Apply the Transfer Object Assembler pattern
- Apply the Composite Entity pattern
- Apply the Value List Handler pattern

J2EE Platform Intra-Business Tier Patterns

Pattern	Primary Function
Application Service	Provides a central place to put business logic between the service façades and the business objects
Business Object	Describes how to organize and separate business data from business logic and workflow
Transfer Object Assembler	Builds complex transfer objects
Composite Entity	Organizes related persistence objects into a composite structure
Value List Handler	Manages result sets according to client requirements and limits

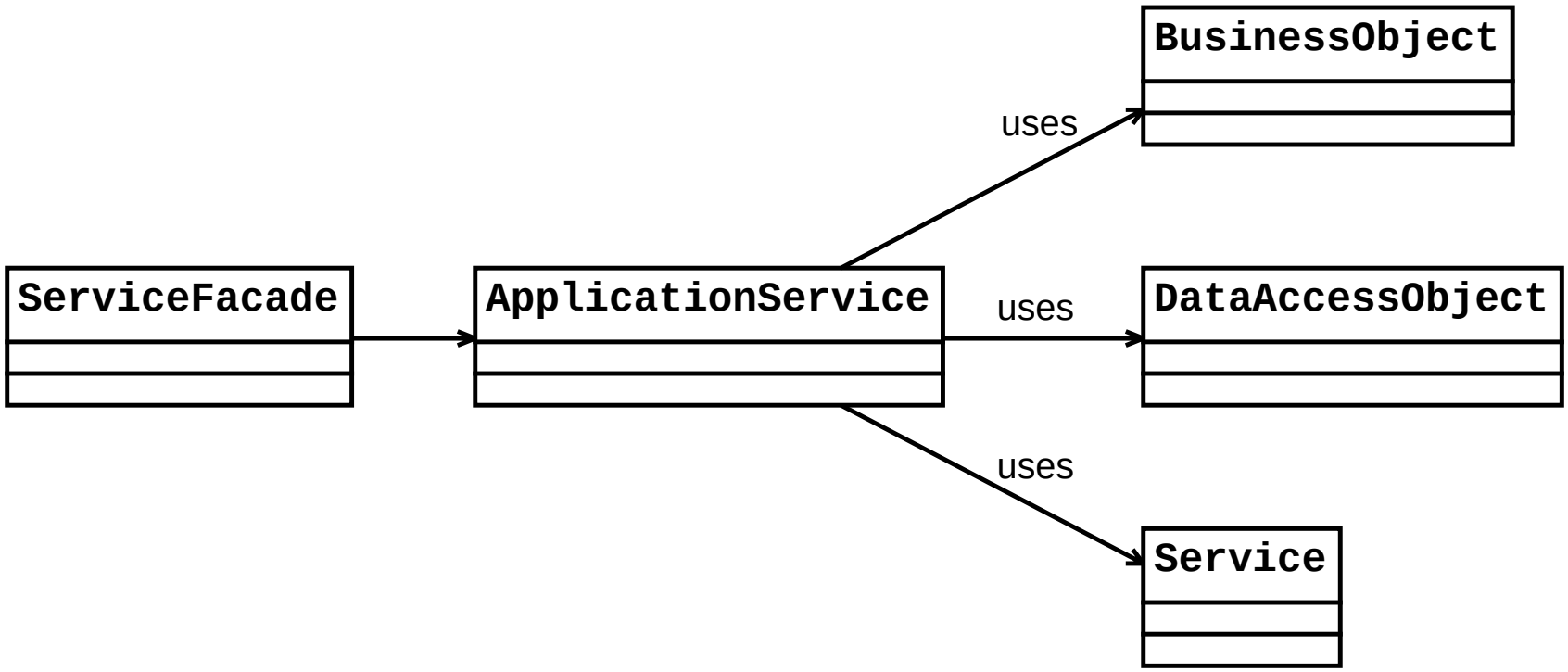
Applying the Application Service Pattern: Problem Forces

- Service façade is a generic term for session façades or POJO façades.
- Business objects represent business data entities.
- Business processing logic should not be put in the business objects. This can lead to a lot of expensive couplings between the business objects.
- Business processing logic should not be placed in the service façades. This can lead to code duplication when the same logic is needed for multiple use cases.
- You need to place business processing logic somewhere in between service façades and business objects.

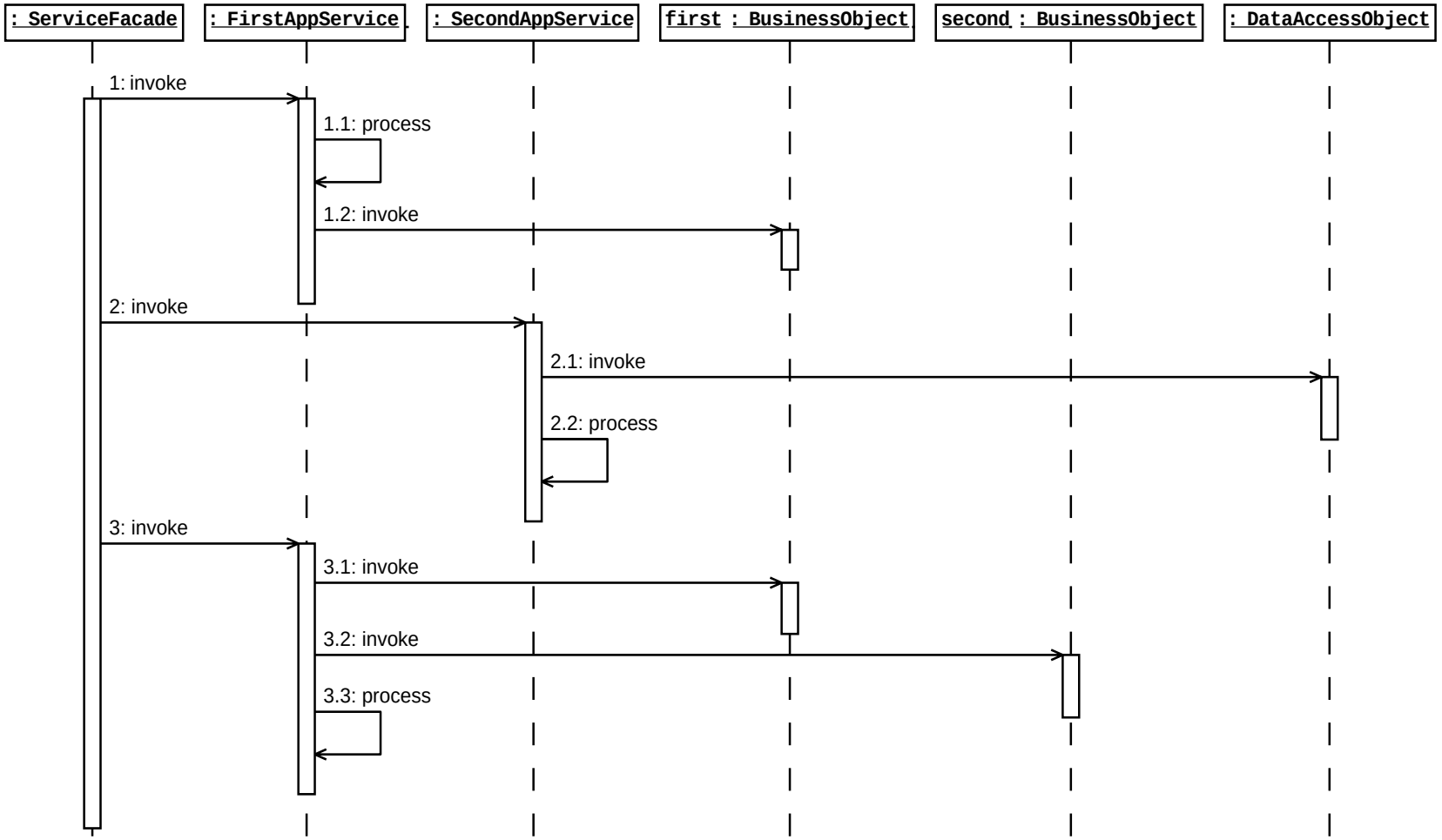
Applying the Application Service Pattern: Solution

- The `ServiceFacade` class has very little code and primarily just invokes the appropriate `ApplicationService` methods
- The `ApplicationService` methods have the business logic that encapsulates the business objects and coordinates their invocations
- The `ApplicationService` components can also use other `ApplicationService` components, ordinary `Service` objects, and `DataAcessObjects`

Application Service Pattern Structure



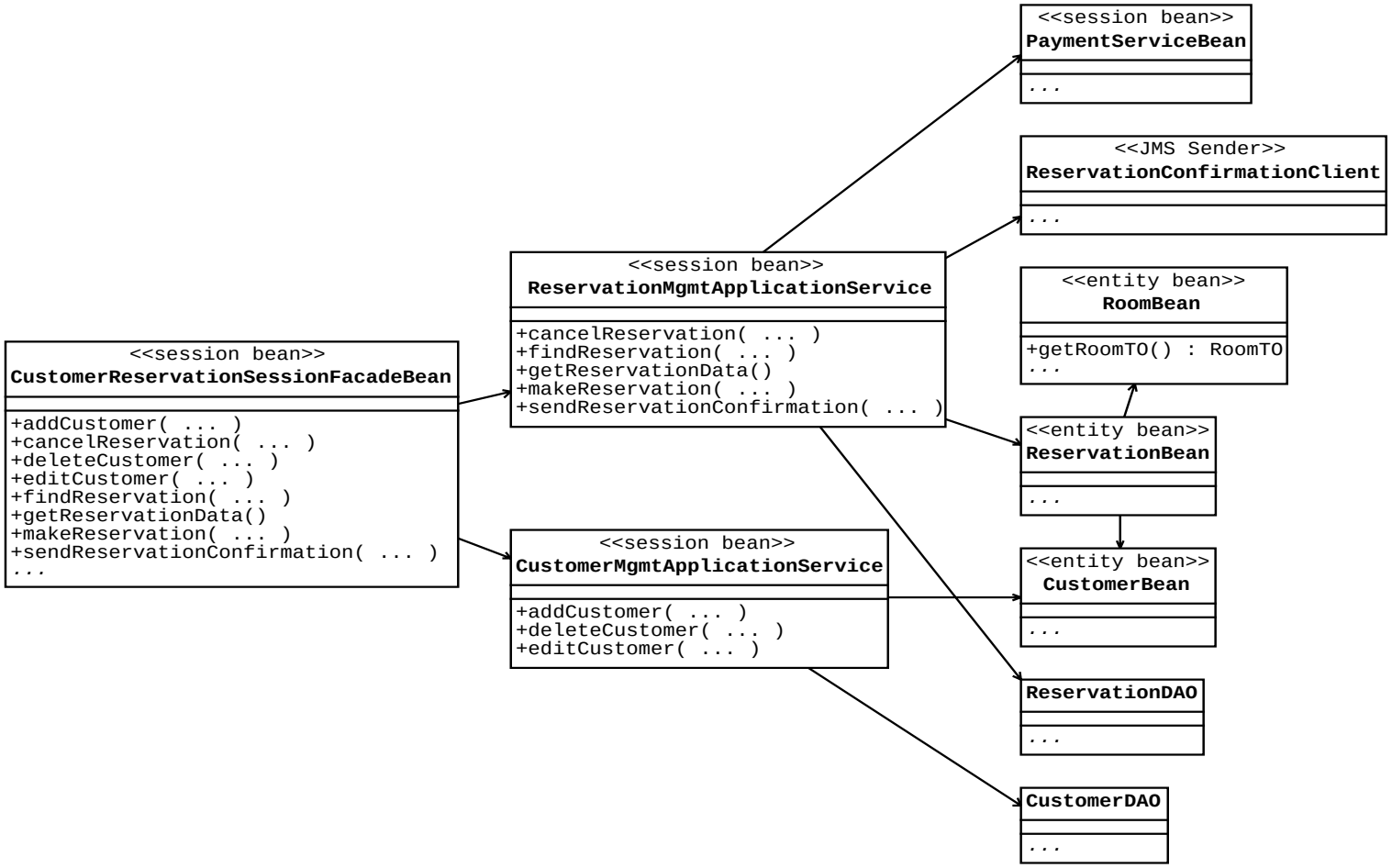
Application Service Pattern Sequence



Applying the Application Service Pattern: Strategies

- **Application Service Command** – An application service might be invoked by a command as specified by the GoF Command pattern. Alternatively, the application service might itself be a command.
- **GoF Strategy for Application Service** – If there are multiple application services that achieve the same result using different algorithms, they might be implemented using a Strategy pattern.
- **Application Service Layer Strategy** – You can have multiple layers of application services that invoke each other. The lower layers are the most client generic layers, and the upper layers are the most client specific.

Application Service Pattern Example



Application Service Pattern Example

```
1   public class ReservationMgmtAS
2       implements SessionBean {
3     private ReservationHome resHome = null;
4     private PaymentHome payHome = null;
5     private PaymentService payService = null;
6     public void ejbCreate() throws CreateException {
7       try {
8         ServiceLocator locator =
9           ServiceLocator.getInstance();
10        resHome = (ReservationHome) locator.
11          getHomeObject("ejb/Reservation",
12          ReservationHome.class);
13        payHome = (PaymentHome) locator.
14          getHomeObject("ejb/Payment",
15          PaymentHome.class);
16        payService = payHome.create();
17      }
18      catch (Exception e) {
19        throw new CreateException
20          ("Unable to initialize AS");
21      }
22    }
23    public boolean makeReservation(. . .)
24    throws Exception {
25      Reservation res = resHome.create(. . .);
26      double cost = res.getCost();
27      return payService.verifyPayment(cost);
28    }
29  . . .
```

Applying the Application Service Pattern: Consequences

Advantages:

- Encourages reusability of business workflow logic
- Avoids duplicating code in service facades

Disadvantage:

Introduces an extra layer that might be unnecessary for smaller applications

Applying the Business Object Pattern: Problem Forces

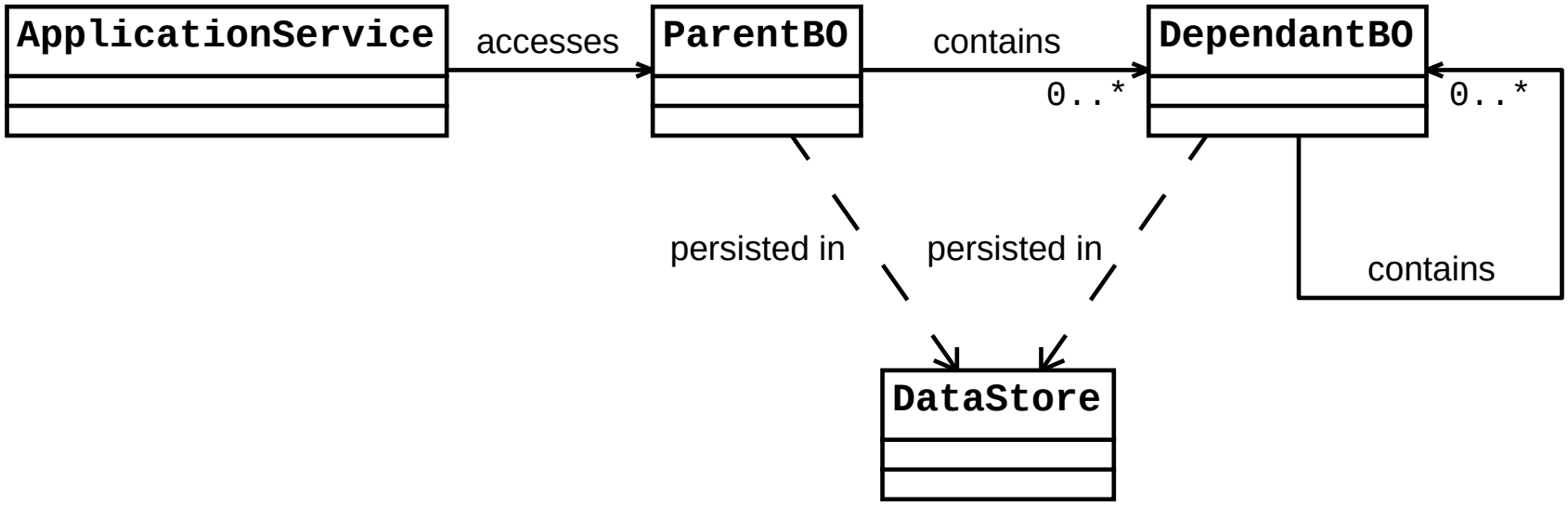
If business service components directly access the data store:

- Changes to the data store or its schema are likely to affect the business services
- The business service components have to be more aware of how to properly communicate with the data store
- Many applications have to deal with potentially complex issues such as transaction handling, data caching, and data synchronization
- Much of the data access and handling code might need to be repeated in multiple business service classes

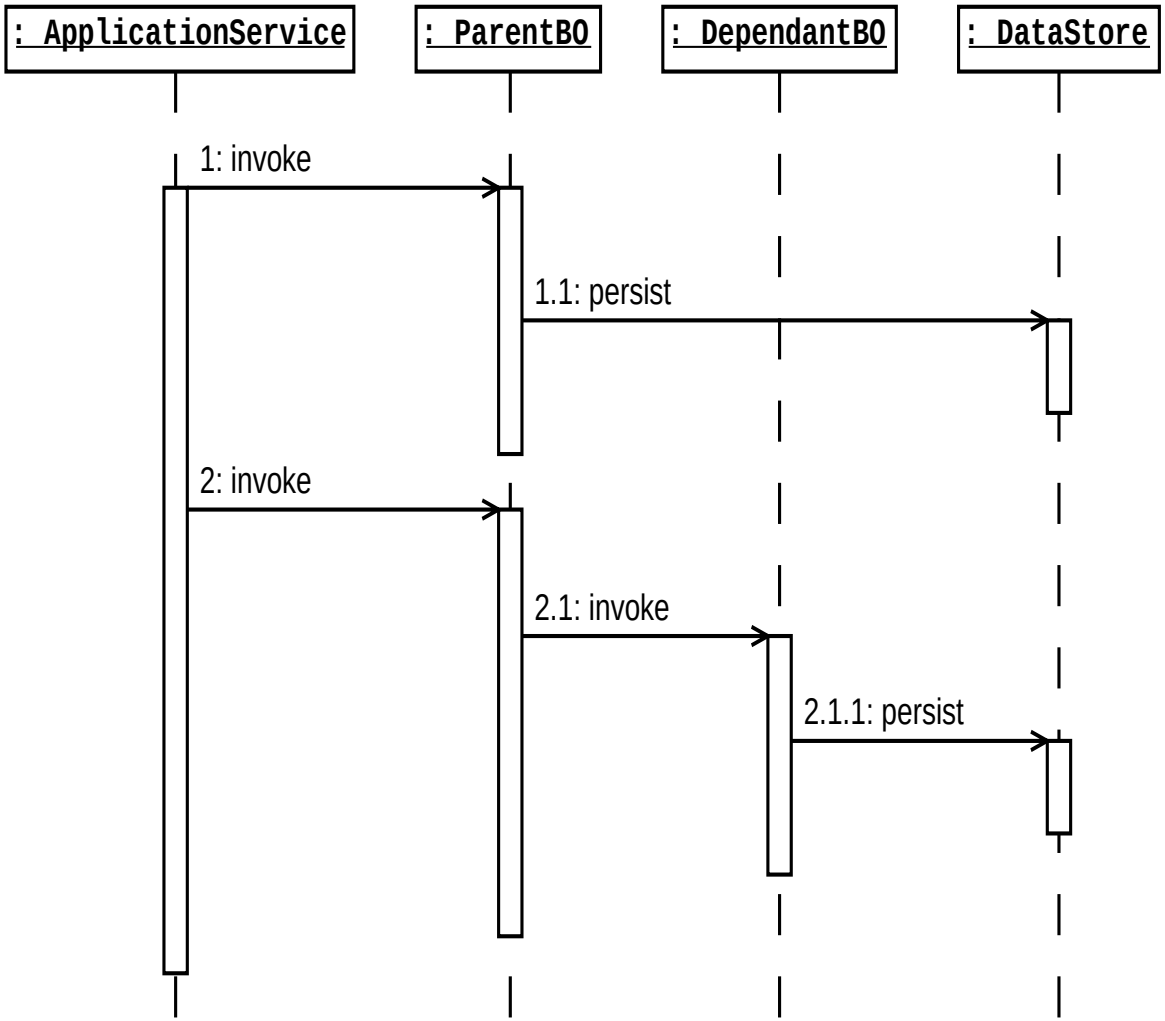
Applying the Business Object Pattern: Solution

- Business entities should be separated into a separate layer of reusable business objects
- Business objects can have some business logic in them, but the business logic should be fairly minimal and not use case specific
- The business logic in business objects should be logic that is very data oriented such as data validation and manipulation
- Data persistence code should be separated into another layer such as described by the Data Access Object pattern or the Domain Store pattern

Business Object Pattern Structure



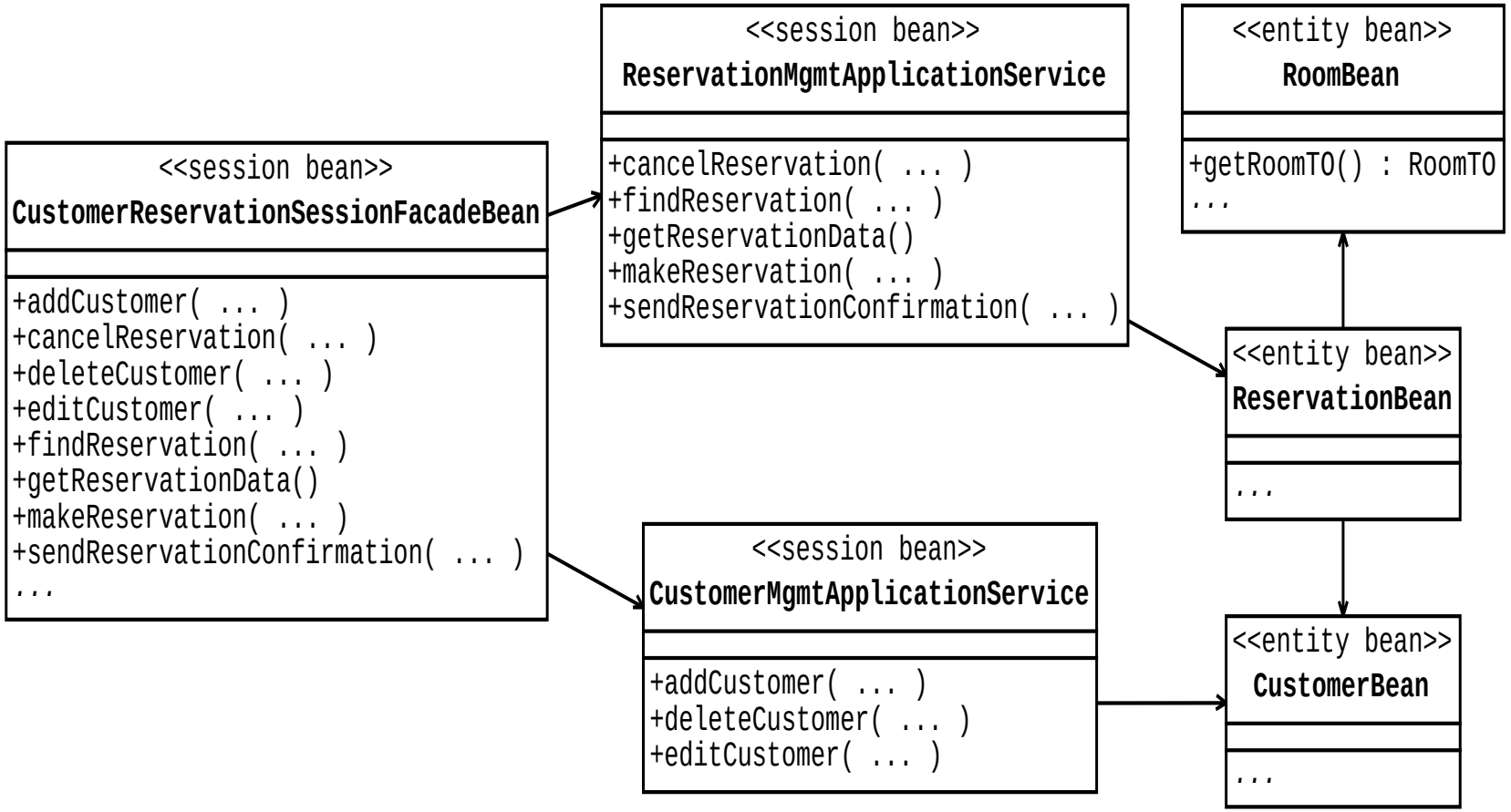
Business Object Pattern Sequence



Applying the Business Object Pattern: Strategies

- **POJO Business Object** – Although POJO business objects might seem faster and simpler, it might be the opposite if you need entity bean services such as transactional and concurrency control
- **Composite Entity Business Object** – Business objects can be implemented as entity beans

Business Object Pattern Example



Applying the Business Object Pattern: Consequences

Advantages:

- Business data can be viewed as object-oriented data
- Data validation and manipulation code is centralized
- The business data logic is separated from the persistence logic
- The business service layer is simplified

Applying the Business Object Pattern: Consequences

Disadvantages:

- Business objects can become bloated if too much business logic is placed in them
- Small applications might not need this extra layer

Applying the Transfer Object Assembler Pattern: Problem Forces

Client classes often need data that can be acquired from numerous business objects, including enterprise beans, data access objects, and transfer objects. If the client gets the needed data directly from several business components:

- The client becomes tightly coupled to the business components
- Many remote invocations might be required
- Client cohesion decreases by having to obtain and construct the data model

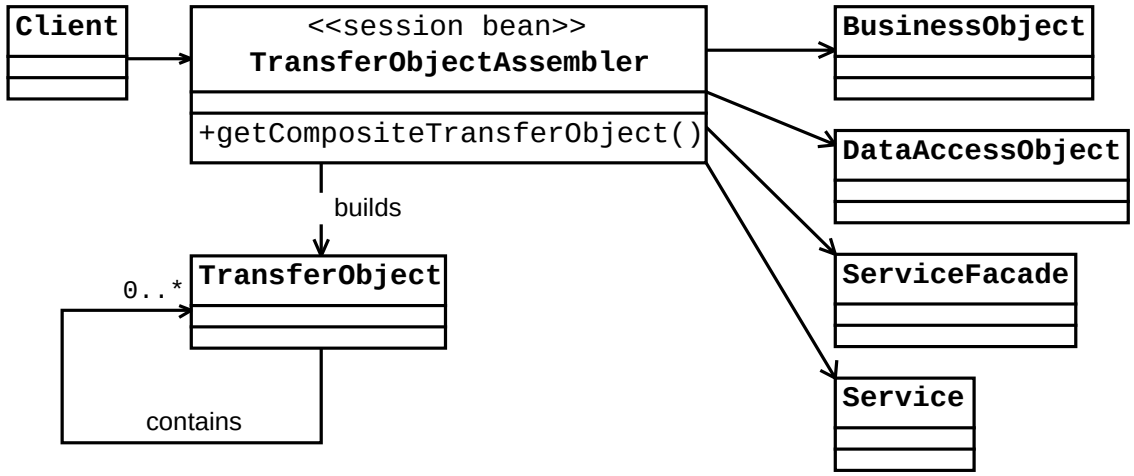
Applying the Transfer Object Assembler Pattern: Solution

- In the business tier, use a session bean that exposes coarse-grained `getTransferObject` methods
- This `getTransferObject` method gathers data from other business tier components, builds a composite `TransferObject`, and returns it to the `client`
- The `BusinessObject` instances that provide the data can be session beans, entity beans, DAOs, or other object types

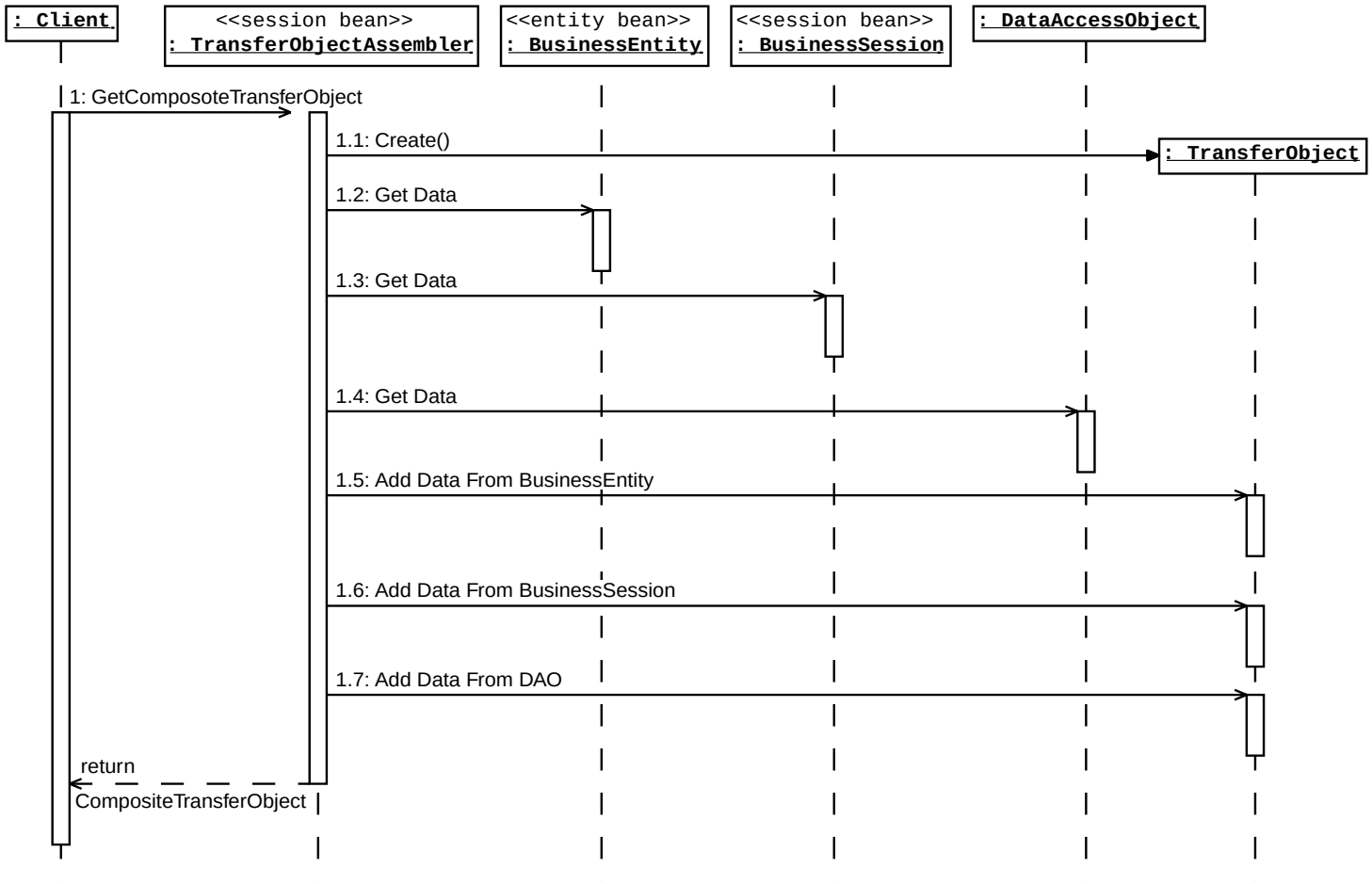
Applying the Transfer Object Assembler Pattern: Solution

- The `TransferObject` instance can be a single object or can be a composite that contains other `TransferObject` instances
- These `TransferObject` instances are immutable

Transfer Object Assembler Pattern Structure



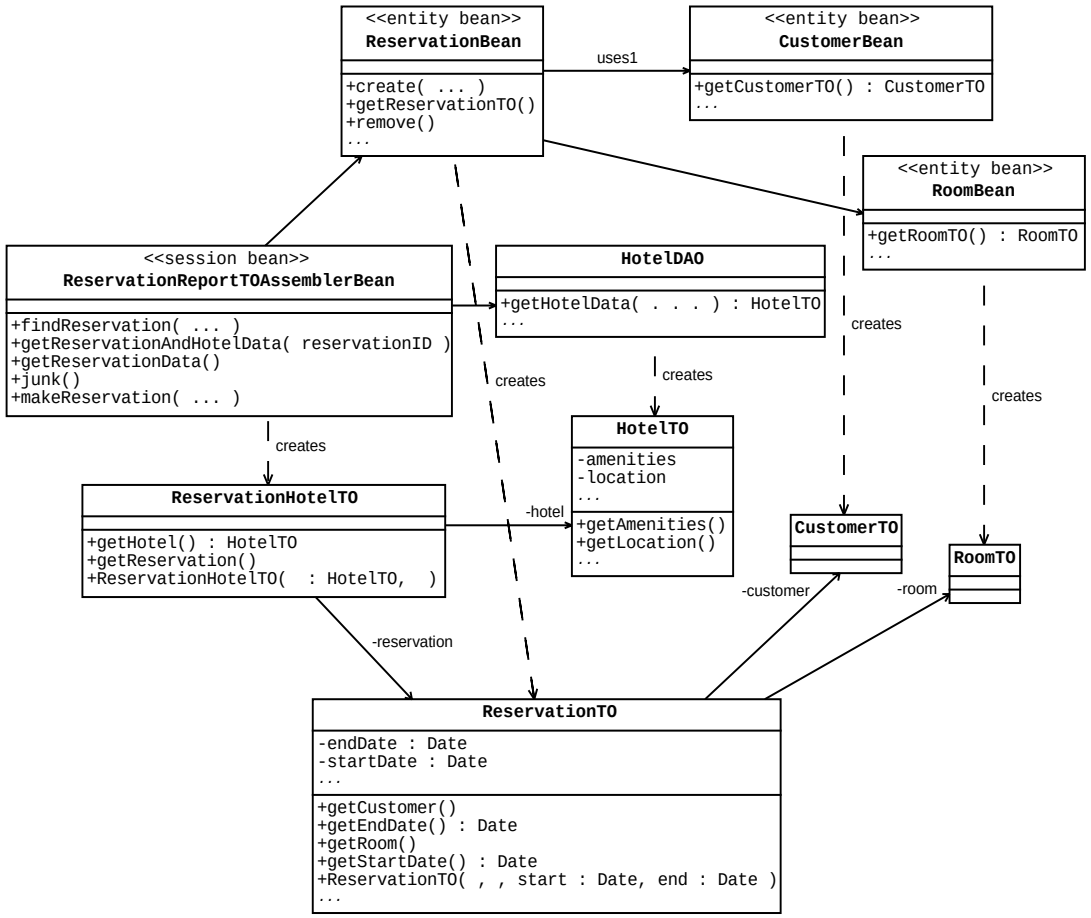
Transfer Object Assembler Pattern Sequence



Applying the Transfer Object Assembler Pattern: Strategies

- **POJO Transfer Object Assembler** – If implemented as a POJO, the transfer object assembler is usually fronted by a session bean such as the session facade component
- **Session Bean Transfer Object Assembler** – If implemented as a session bean, the transfer object assembler is usually a stateless session bean

Transfer Object Assembler Pattern Example



Transfer Object Assembler Pattern Example

Transfer object assembler code:

```
1     public ReservationHotelTO getReservationHotelData
2         (String reservationID) {
3       HotelTO hotelData = hotelDAO.getHotelData();
4       ReservationTO resData = reservation.getReservationTO();
5       return new ReservationHotelTO(hotelData, resData);
```

Applying the Transfer Object Assembler Pattern: Consequences

Advantages:

- Removes business logic from client
- Reduces coupling
- Reduces network traffic
- Improves client performance
- Improves transaction performance

Applying the Transfer Object Assembler Pattern: Consequences

Disadvantages:

- Object creation overhead on the server side increases
- Data in transfer objects might get stale
- Update logic must be handled separately

Applying the Composite Entity Pattern: Problem Forces

When using entity beans you need to make several decisions:

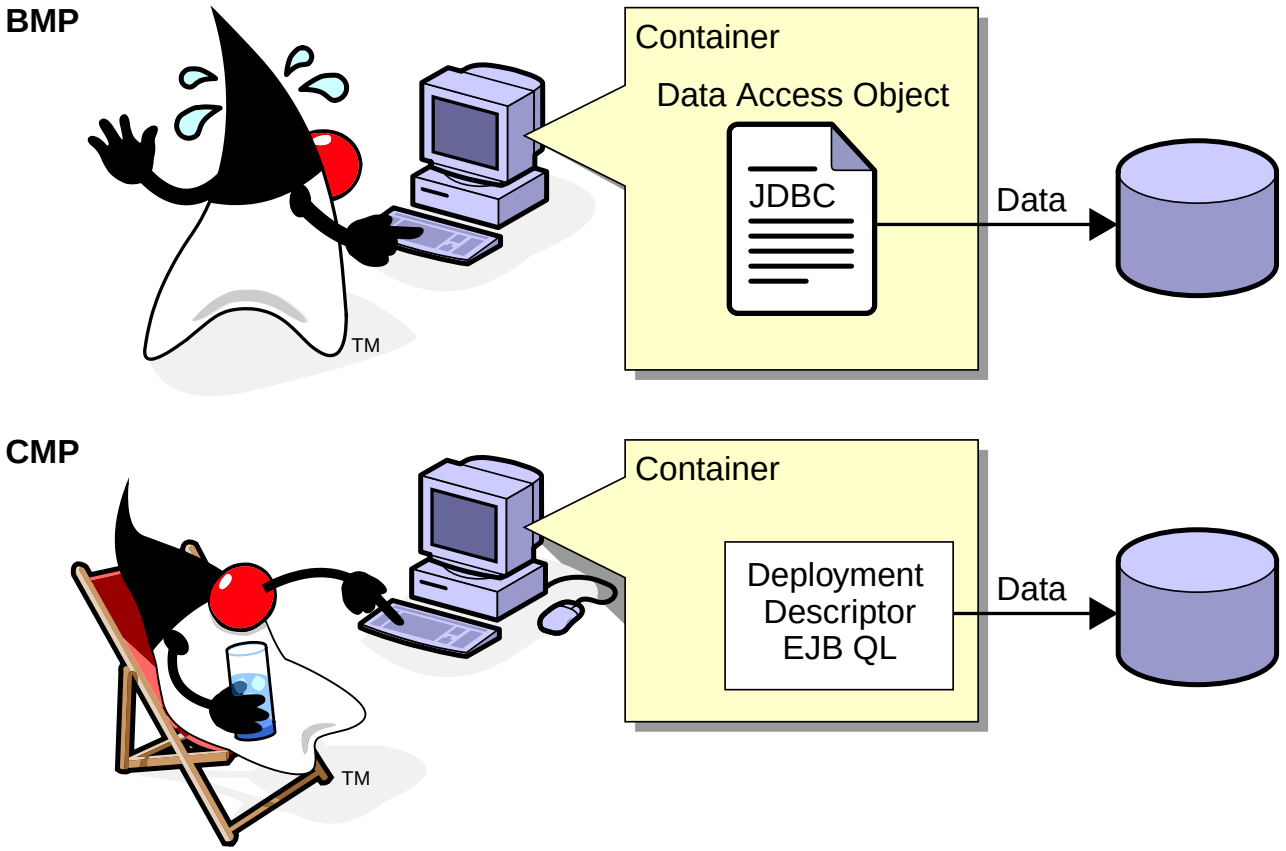
- Local or remote interfaces
- BMP or CMP
- Whether to use CMR

Applying the Composite Entity Pattern: Problem Forces

Consider the following forces:

- Avoid dependencies between remote entity beans
- You might need BMP for custom or legacy persistence
- Entity beans container services are often desirable for business objects
- The database schema should be hidden from the client

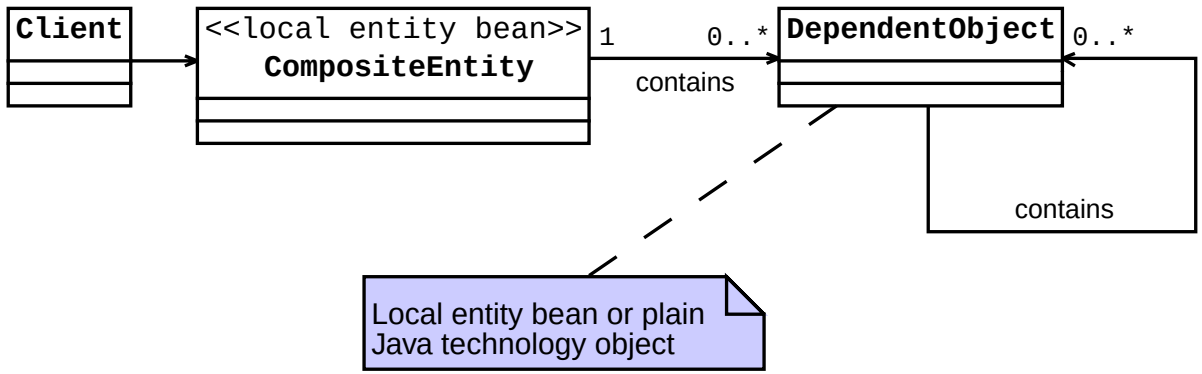
Comparing BMP and CMP



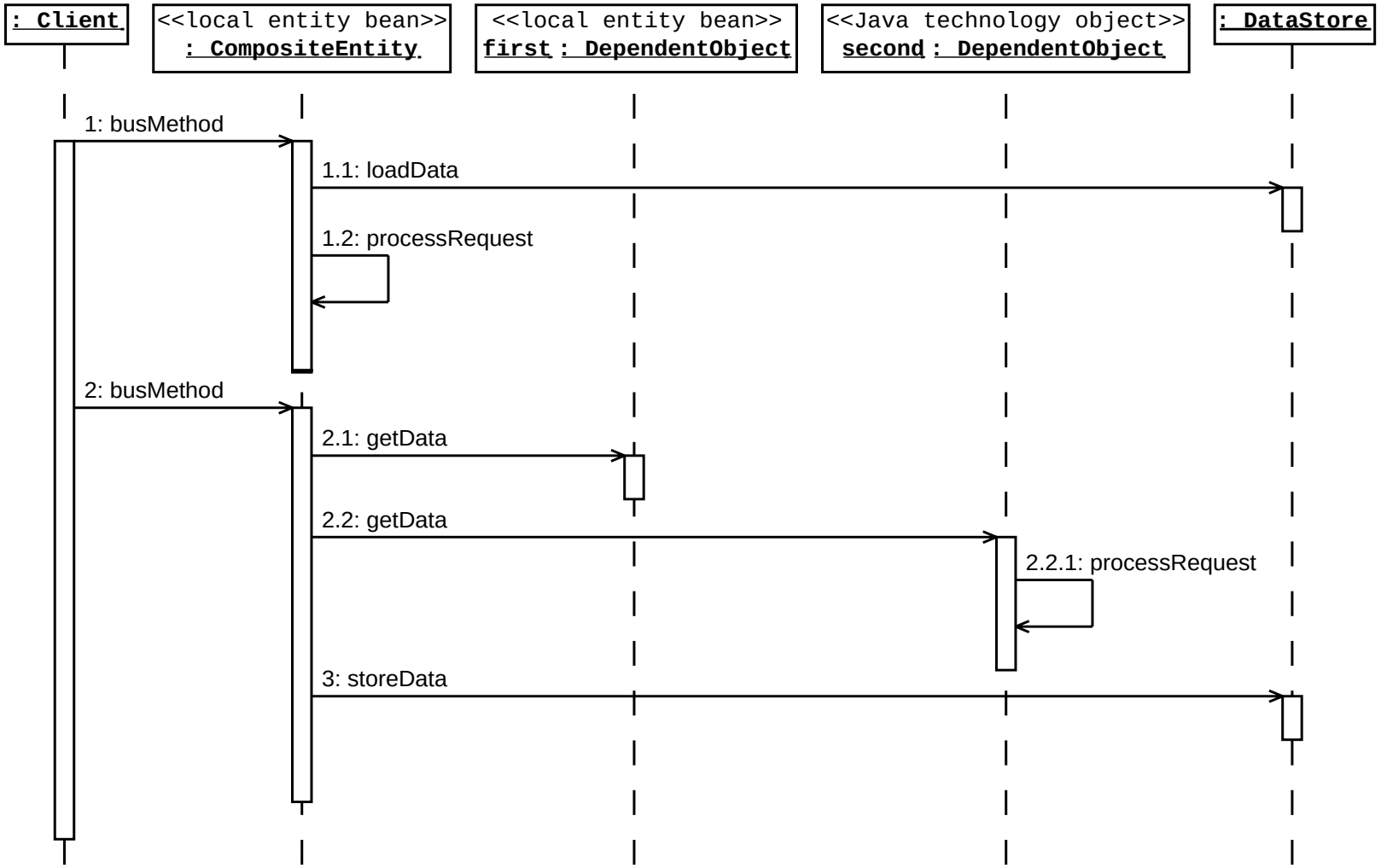
Applying the Composite Entity Pattern: Solution

- Parent objects are reusable, independently deployable, components that manage their own life cycles.
- Dependant objects are managed and accessed only by their parent objects.
- For EJB 1.1 containers, the parent object should be implemented as a remote entity bean. The dependant objects should be implemented as a POJO.
- For EJB 2.0 containers, the parent object should be implemented as a local entity bean. The dependant objects can be implemented as either POJOs or local entity beans. If you are using CMP, CMR can be used.

Composite Entity Pattern Structure



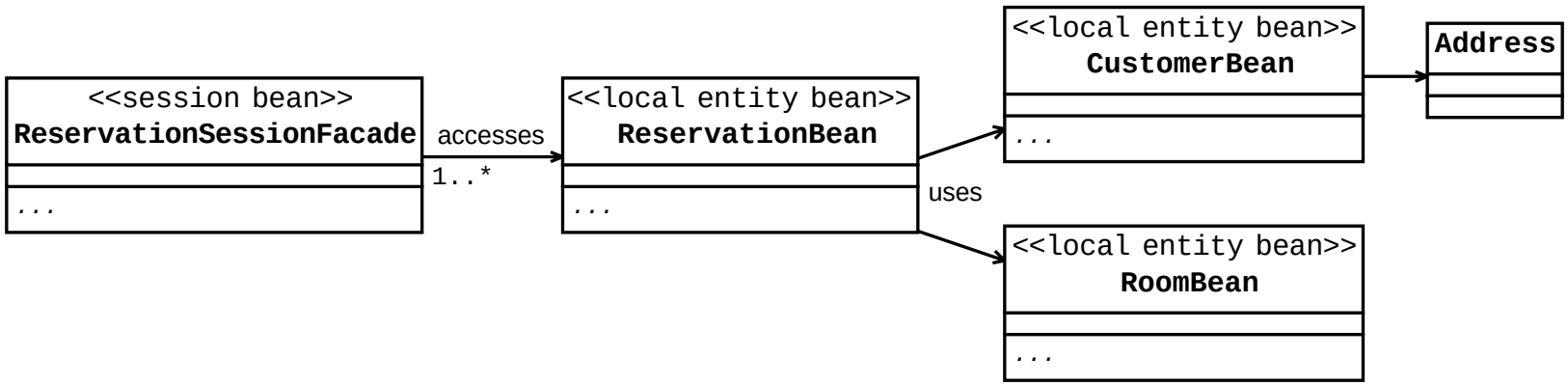
Composite Entity Pattern Sequence



Applying the Composite Entity Pattern: Strategies

- **Composite Entity Remote Façade** – In applications with trivial business logic, a session façade can be replaced by a composite entity that acts as an entity façade. This is dangerous for complex applications.
- **Lazy Loading** – With BMP, `ejbLoad` method only loads some data. The other data is loaded by business methods if and when it is required.
- **Store Optimization (Dirty Marker)** – With BMP, `ejbStore` method only stores data that is marked as dirty.

Composite Entity Pattern Example



Composite Entity Pattern Example

Composite entity code:

```
1   public abstract class ReservationBean
2       implements EntityBean {
3     //virtual persistence fields
4     public abstract String getReservationID();
5     public abstract void
6       setReservationID(String resID);
7     public abstract Date getStartDate();
8     public abstract void setStartDate(Date d);
9     public abstract Date getEndDate();
10    public abstract void setEndDate(Date d);
11    //virtual relationship fields
12    public abstract CustomerLocal getCustomer();
13    public abstract void
14      setCustomer(CustomerLocal c);
15    public abstract RoomLocal getRoom();
16    public abstract void setRoom(RoomLocal r);
17    private EntityContext context;
```

Composite Entity Pattern Example

```
18    public String ejbCreate(String resID,
19        Date start, Date end, CustomerLocal cust,
20        RoomLocal room) throws CreateException {
21      setReservationID(resID);
22      setStartDate(start);
23      setEndDate(end);
24      setCustomer(cust);
25      setRoom(room);
26      return null;
```

Composite Entity Pattern Example

```
27    }
28    public void ejbPostCreate(String resID,
29        Date start, Date end, CustomerLocal cust,
30        RoomLocal room) throws CreateException {
31      setCustomer(cust);
32      setRoom(room);
33    }
34    public void setEntityContext
35        (EntityContext ctx){
36      context = ctx;
37    }
38  public void unsetEntityContext() {
39      context = null;
40    }
41    public void ejbRemove() { }
42    public void ejbActivate() { }
43    public void ejbPassivate() { }
44    public void ejbStore() { }
45    public void ejbLoad() { }
```

Composite Entity Pattern Example

```
46  public ReservationTO getReservationTO() {
47      return new ReservationTO();
48    }
49  }
```

Composite Entity Pattern Example

CMR relationship in deployment descriptor:

```
1   . . .
2   <ejb-relation>
3     <ejb-relationship-role>
4       <ejb-relationship-role-name>
5         ReservationBean
6       </ejb-relationshiprole-name>
7       <multiplicity>One</multiplicity>
8       <relationship-role-source>
9         <ejb-name>ReservationBean</ejb-name>
10      </relationship-role-source>
11      <cmr-field>
12        <cmr-field-name>customer</cmr-field-name>
13      </cmr-field>
14    </ejb-relationship-role>
15    <ejb-relationship-role>
16      <ejb-relationship-role-name>
17        CustomerBean
```

Composite Entity Pattern Example

```
18  </ejb-relationship-rolename>
19      <multiplicity>One</multiplicity>
20      <relationship-role-source>
21        <ejb-name>CustomerBean</ejb-name>
22      </relationship-role-source>
23    </ejb-relationship-role>
24  </ejb-relation>
25  . . .
```

Applying the Composite Entity Pattern: Consequences

Advantages:

- Improves maintainability by simplifying entity bean interactions
- Decreases network overhead by using less remote entity beans
- Decreases network overhead by providing a façade and composite transfer objects
- Shields clients from changes in the database

Applying the Composite Entity Pattern: Consequences

Disadvantage:

Even with local interfaces, CMP, and CMR, entity beans might not always perform as well as POJOs

Applying the Value List Handler Pattern: Problem Forces

- Client applications need an efficient query facility
- Clients need server-side caching to support result sets that are too large for the client
- Large search results are usually primarily read-only
- EJB component `finder` methods are resource-intensive and ill-suited for browsing large result sets
- Clients might want to scroll backwards, as well as forwards, through the search results

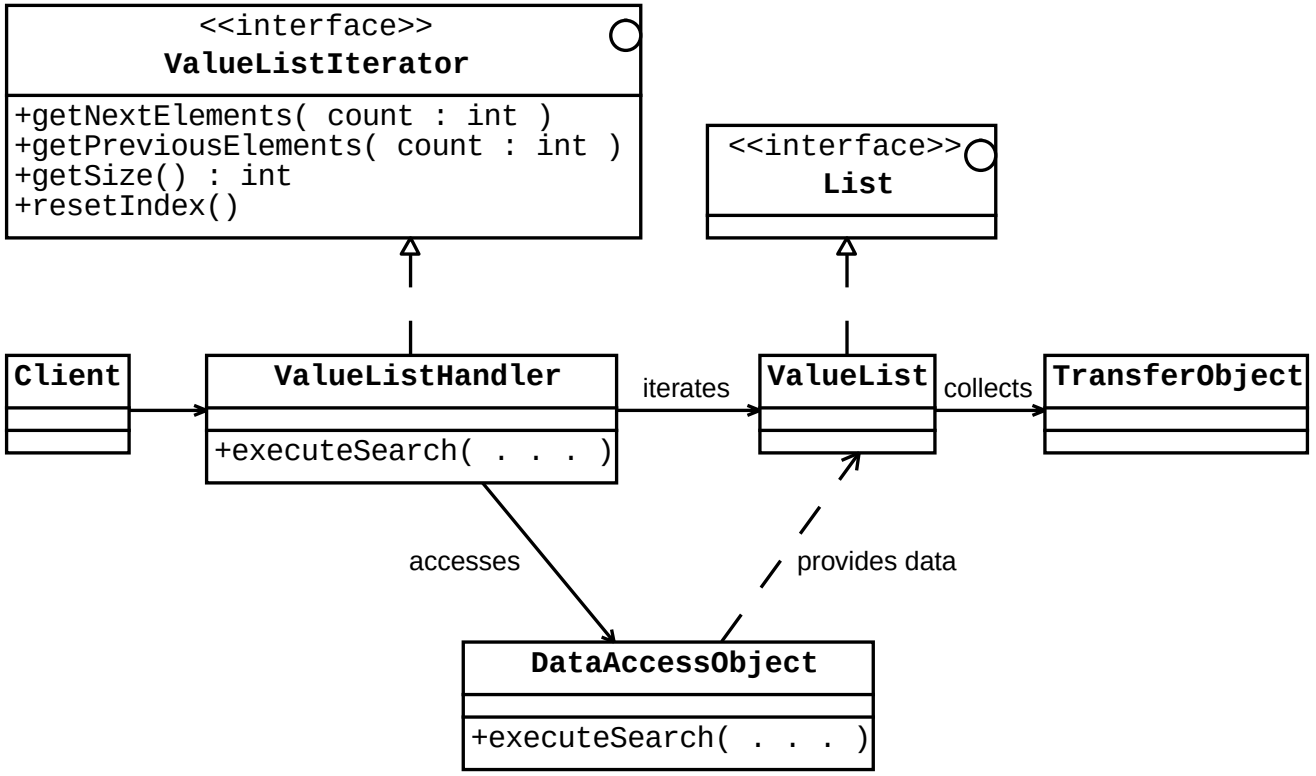
Applying the Value List Handler Pattern: Solution

- The `ValueListHandler` class executes a search query, holds the results, and allows the client to retrieve the results they need
- When the client invokes the `executeSearch` method on `ValueListHandler`, it passes that request to the `DataAccessObject`
- The `DataAccessObject` executes the query and creates the `ValueList` object of `TransferObject` instances

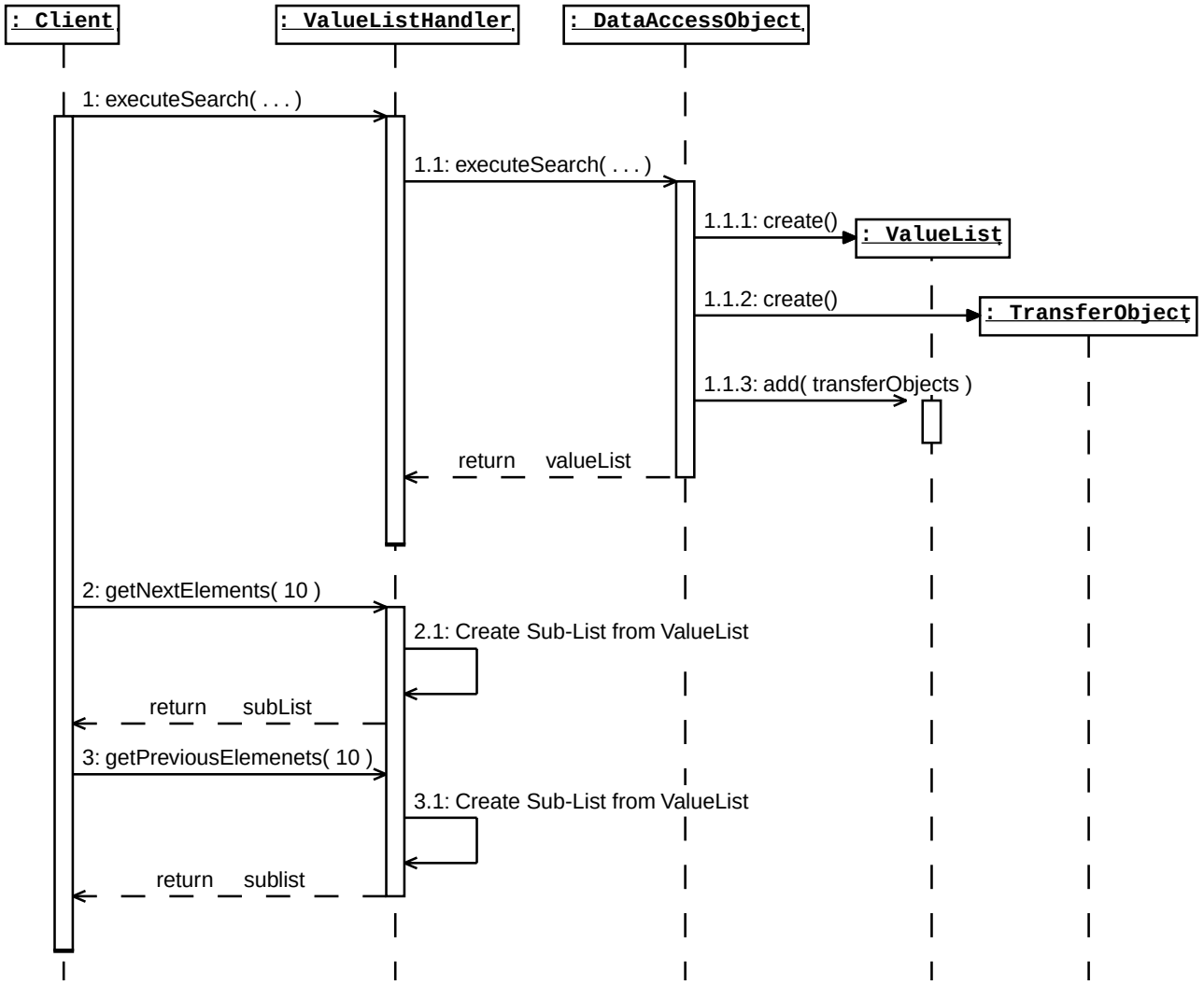
Applying the Value List Handler Pattern: Solution

- The `ValueListHandler` keeps the `ValueList` reference
- The client can invoke the `getNextElements` and `getPreviousElements` methods to get results

Value List Handler Pattern Structure



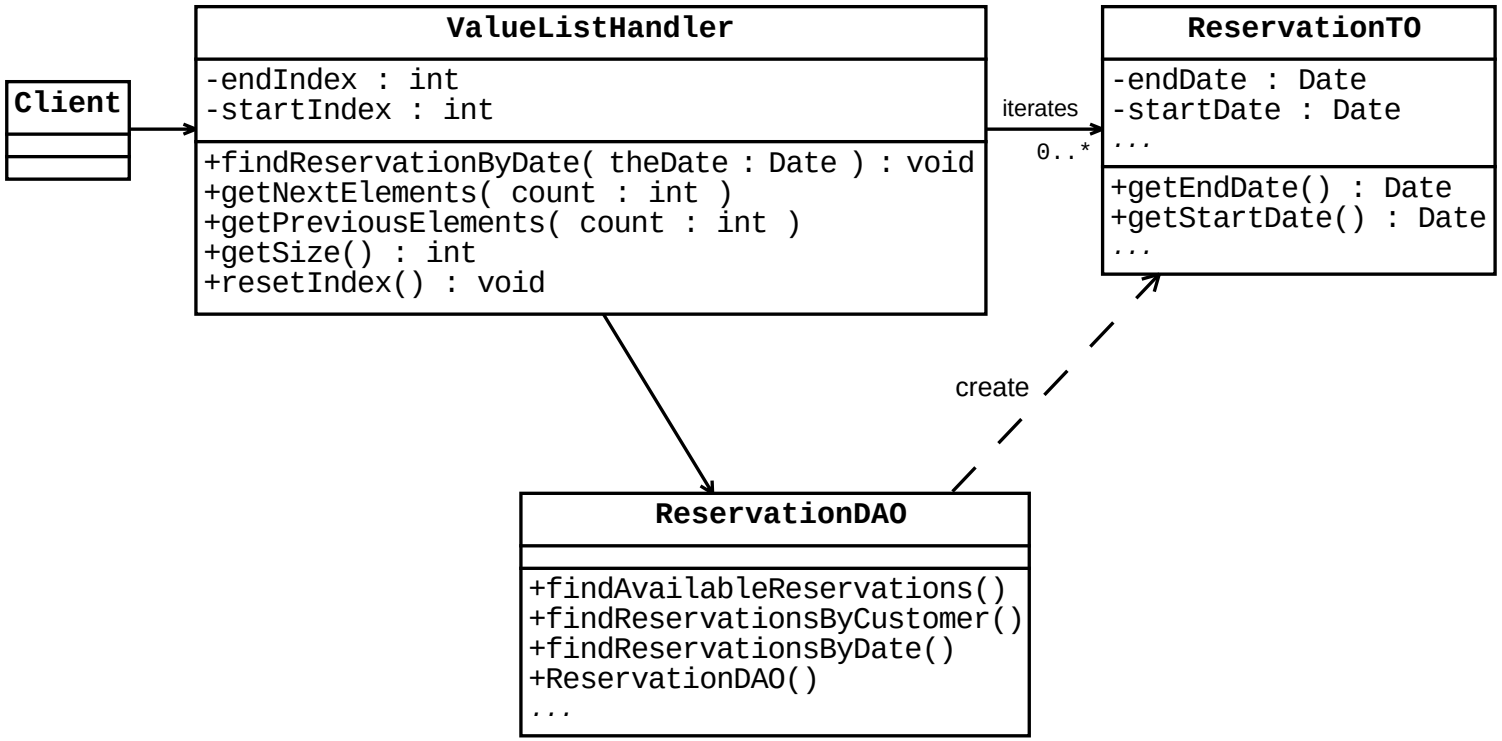
Value List Handler Pattern Sequence



Applying the Value List Handler Pattern: Strategies

- **Stateful Session Bean Handler** – A value list handler is usually a stateful session bean or fronted by a stateful session bean
- **POJO Handler** – If an application does not use EJB components, the value list handler can be implemented as a plain Java object that is directly accessed by a business delegate
- **Value List from DAO** – A DAO can return results in a variety of formats including a JDBC ResultSet, a JDBC Rowset, or a collection

Value List Handler Pattern Example



Value List Handler Pattern Example

Value list handler code:

```
1   public class ValueListHandler {
2     private List allResults;
3     private int startIndex, endIndex;
4
5     public void findReservationByDate(Date theDate)
6       throws Exception {
7       ReservationDAO dao = new ReservationDAO();
8       allResults =
9         dao.findReservationByDate(theDate);
10    }
11    public ReservationTO[] getNextElements
12        (int howMany) {
13      if (howMany > allResults.size()-endIndex) {
14        howMany = allResults.size() - endIndex;
15      }
16      ReservationTO[] subset =
17        new ReservationTO[howMany];
```

Value List Handler Pattern Example

```
18      int index;
19      for (index=0; index<howMany; index++) {
20        subset[index] = (ReservationTO)
21          allResults.get(startIndex+index);
22      }
23      startIndex = endIndex;
24      endIndex = startIndex+index;
25      return subset;
26    }
```

Value List Handler Pattern Example

```
27  public ReservationTO[] getPreviousElements(int howMany) {
28      if (howMany > startIndex) {
29        howMany = startIndex;
30      }
31      ReservationTO[] subset = new ReservationTO[howMany];
32
33      startIndex -= howMany;
34      int index;
35      for (index=0; index<howMany; index++) {
36        subset[index] = (ReservationTO)
37          allResults.get(startIndex+index);
38      }
39      endIndex = startIndex+index;
40      return subset;
41    }
42    public int getSize() {
43      return allResults.size();
44    }
45    public void resetIndex() {
```

Value List Handler Pattern Example

```
46      startIndex = endIndex = 0;
47      }
48  }
```

Applying the Value List Handler Pattern: Consequences

Advantages:

- Avoids EJB component `finder` method overhead
- Supports server caching of result sets
- Enables client control over dispatching results
- Reduces network overhead

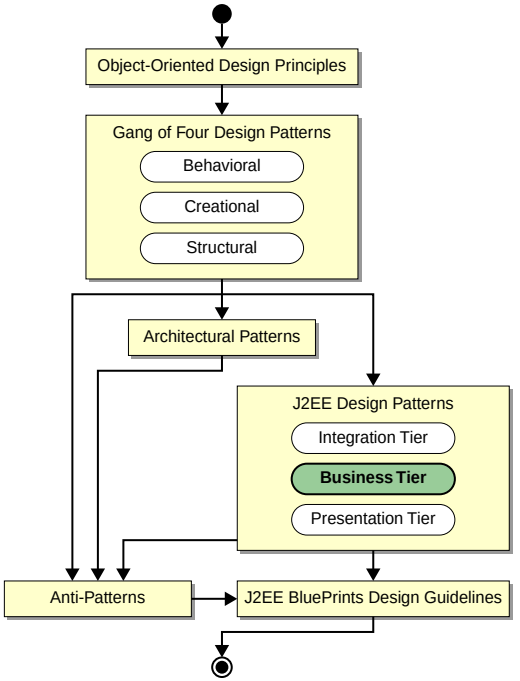
Applying the Value List Handler Pattern: Consequences

Disadvantages:

- Performance can be impeded when caching some query patterns
- This pattern is less effective when results are updated frequently

Summary

Intra-Business Tier patterns help expedite client access to business objects. In summary, these patterns contribute as follows:



- **Application Service** – Provides a central place to put business logic between the service facades and the business objects.
- **Business Object** – Describes how to organize and separate business data from business logic and workflow.
- **Transfer Object Assembler** – Builds complex business objects from multiple transfer objects.

Summary

- **Composite Entity** – Organizes related persistence objects into a composite structure.
- **Value List Handler** – Manages large result sets on behalf of a client.

Module 10

Using Presentation Tier Patterns

Objectives

- Describe basic characteristics of the Presentation Tier J2EE Patterns
- Describe the Model 2 Architecture and the Apache Struts Framework
- Apply the Intercepting Filter pattern
- Apply the Front Controller pattern
- Apply the Application Controller pattern
- Apply the Context Object pattern

Introducing the Presentation Tier Patterns

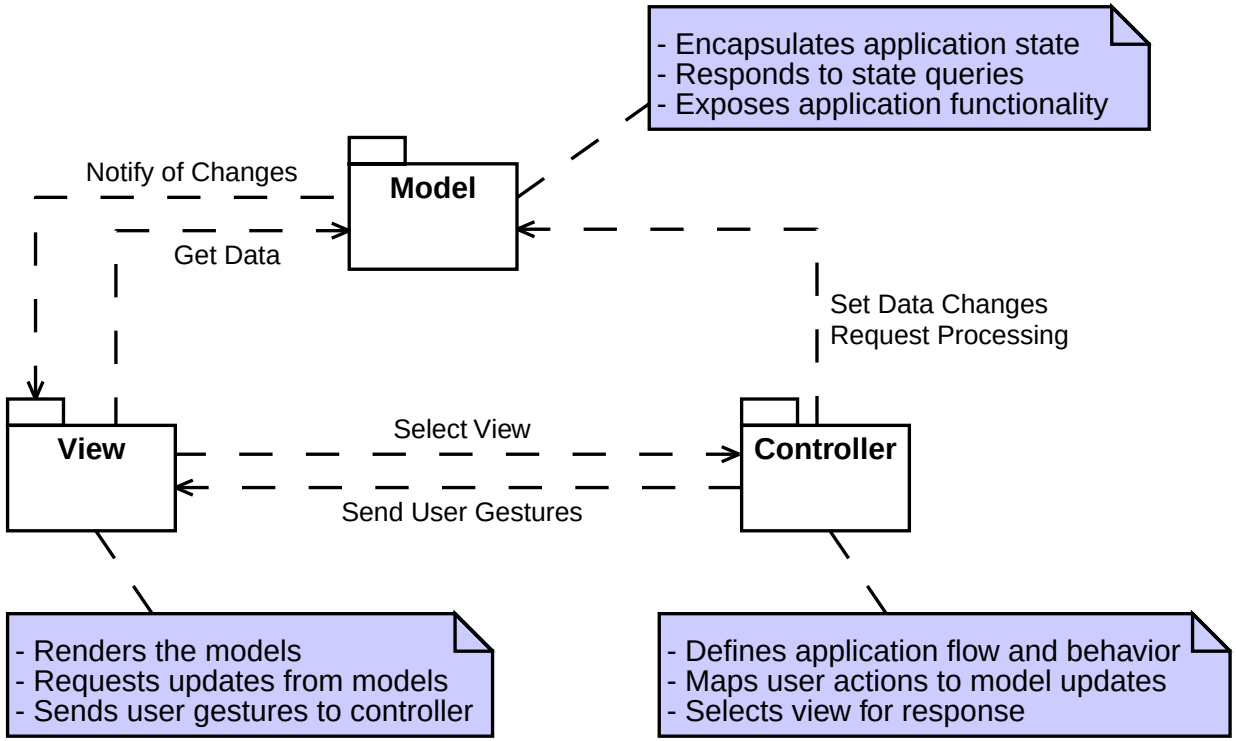
Presentation Tier patterns:

- Provide guidance for designing individual presentation tier components by role
- Provide mechanisms to reduce copy-and-paste reuse
- Enforce role separation between logic and presentation components

Presentation Tier Patterns

Pattern	Primary Function
Intercepting Filter	Provides a simple mechanism for pre-processing and post-processing of HTTP requests and responses.
Front Controller	Provides a single entry point to the presentation tier. It can handle security, validation, and flow control.
Application Controller	Separates the action invocation management and view dispatching management from the front controller component.
Context Object	Passes data that is in context-specific objects without passing those objects out of their context.

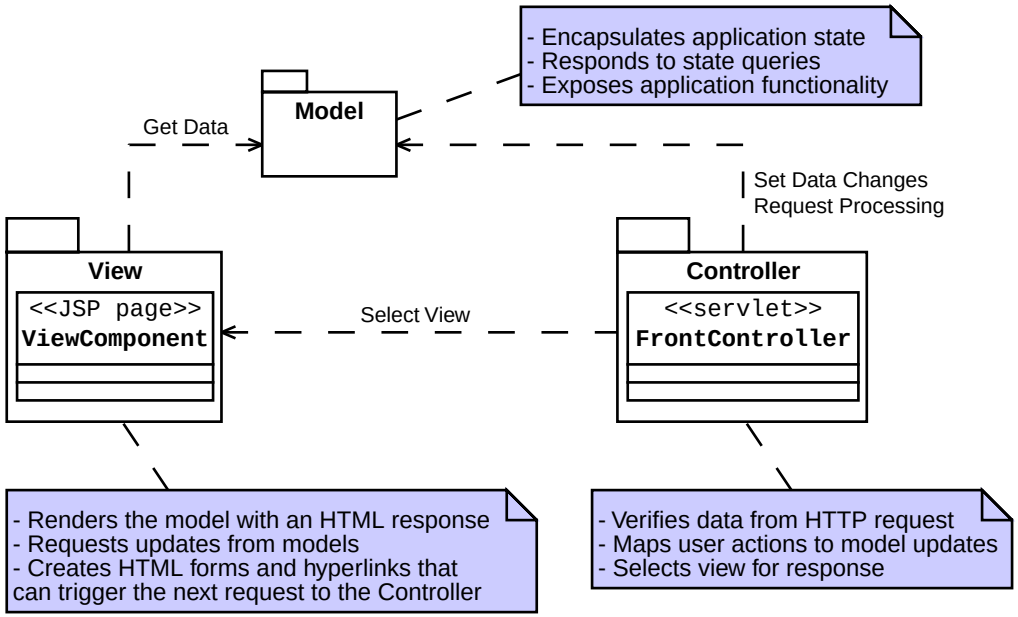
Traditional Model View Controller Architecture



The traditional MVC architecture requires modification to be useful for Web applications

Model 2 Architecture

- Model 2 architecture describes how to apply the MVC architecture to a Java technology web application
- The Presentation Tier patterns fit into this architecture



Apache Struts Framework

- Many development teams create Model 2 architecture applications without a pre-written framework
- You might want to use a Model 2 architecture framework
- Apache Jakarta Struts is one of the most popular Model 2 Architecture frameworks
- Struts provides a servlet, custom tags, JavaBeans components, and other helpers to facilitate the implementation of a Model 2 Architecture application
- The second edition of the Presentation Tier patterns discuss how they are implemented in Struts

Applying the Intercepting Filter Pattern: Problem Forces

Some requests might require pre-processing and some responses may require post-processing.

Pre-Processing Tasks	**Post-Processing Tasks**
User authentication	Response customization for a certain type of client
Session validate	Response encryption
Denial of service attack detection	Response compression
Request decryption	Language translation
Request decompression	Debugging
Request auditing	

Applying the Intercepting Filter Pattern: Problem Forces

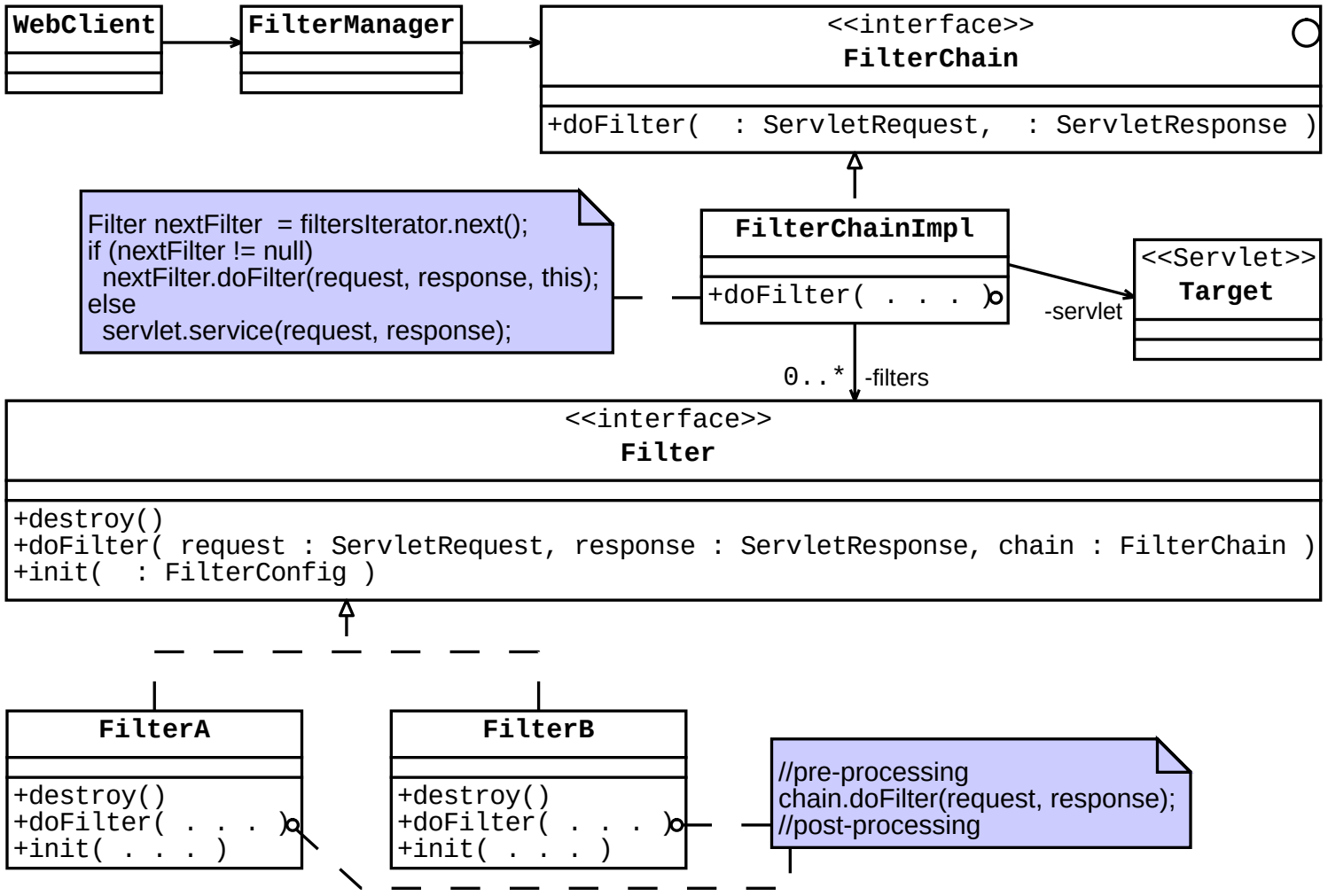
Placing all of the pre-processing and post-processing tasks in one component can result in:

- Components with poor cohesion
- Tasks that are difficult to add and remove
- Tasks that are reusable only through undesirable copy-and-paste techniques

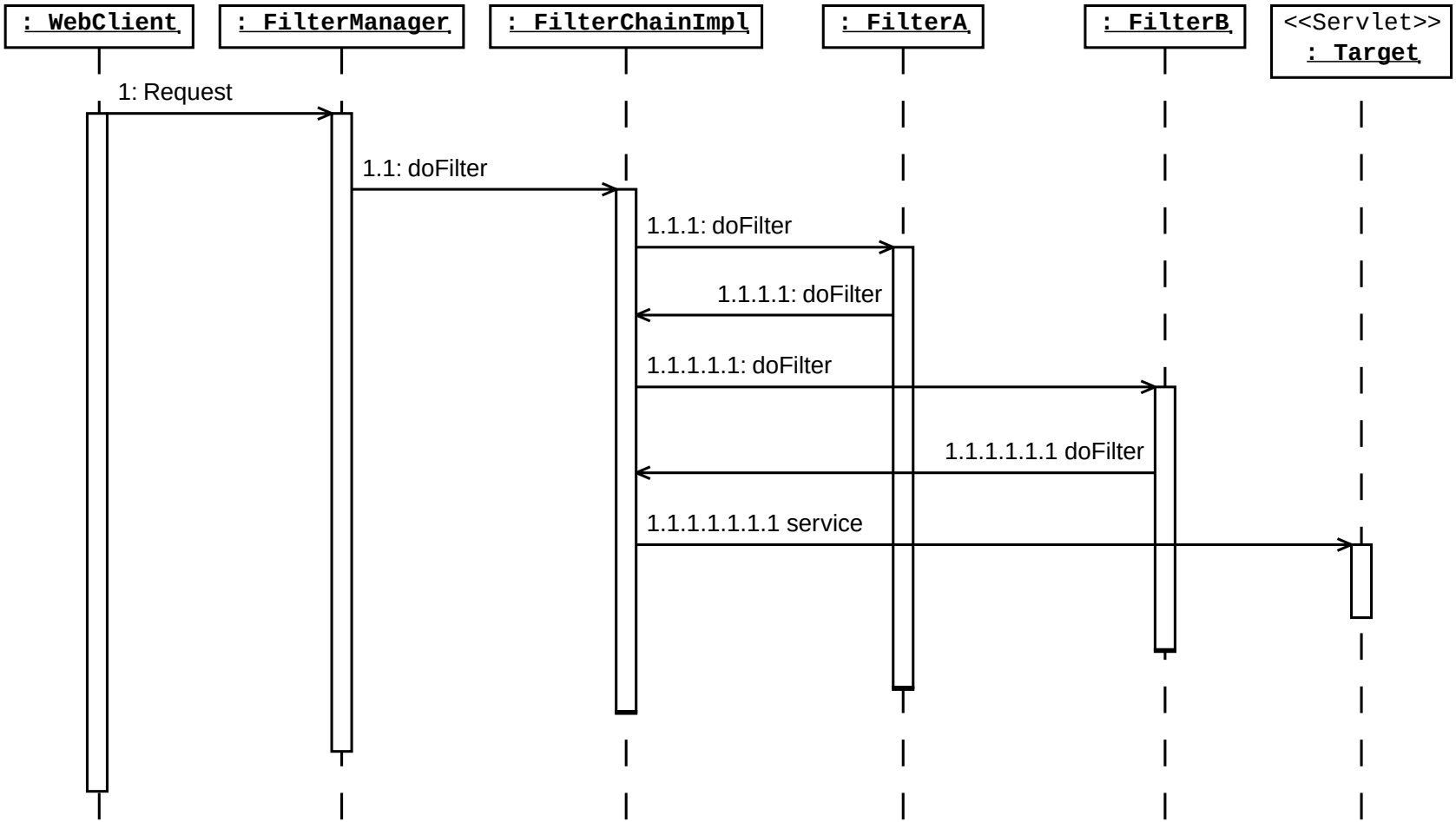
Applying the Intercepting Filter Pattern: Solution

- Incoming requests go through each declared filter
- Outgoing responses go back through each declared filter
- The pattern can be implemented using different designs, but the result is comparable to the GoF Decorator pattern
- Filters can be added or removed declaratively
- Java Servlet 2.3 or later containers provide built-in support for filters
- Pre-2.3 Java Servlet containers must implement a custom filter strategy

Intercepting Filter Pattern Structure



Intercepting Filter Pattern Sequence



Applying the Intercepting Filter Pattern: Strategies

- **Custom Filter** – With pre-2.3 Java Servlet containers, you must define the filter pattern programmatically
- **Standard Filter** – In Java Servlet 2.3 containers, filters are added declaratively through the deployment descriptor
- **Base Filter** – Base filter class serves as a superclass for all other filter classes
- **Template Filter** – The base filter can be used with the GoF Template Method pattern
- **JAX-RPC Filter** – The filter is used in the client or integration tier in front of a web service

Applying the Intercepting Filter Pattern: Example

The timing filter:

```
1   package sl500;
2   import java.io.*;
3   import javax.servlet.*;
4   public class TimerFilter implements Filter {
5     private FilterConfig filterConfig = null;
6
7     public void init(FilterConfig filterConfig)
8         throws ServletException {
9       this.filterConfig = filterConfig;
10    }
11
12    public void destroy() { }
13
14    public void doFilter(ServletRequest request,
15          ServletResponse response,
16          FilterChain chain)
17          throws IOException, ServletException {
18      long start = System.currentTimeMillis();
19
20      chain.doFilter(request, response);
21
22      long end = System.currentTimeMillis();
23      filterConfig.getServletContext().log
24        ("Request processed in " + (end - start) +
25        " milliseconds");
26    }
27  }
```

Applying the Intercepting Filter Pattern: Example

Part of the `web.xml` file

```
1   <filter>
2     <filter-name>TimerFilter</filter-name>
3     <filter-class>sl500.TimerFilter</filter-class>
4   </filter>
5   <filter-mapping>
6     <filter-name>TimerFilter</filter-name>
7     <url-pattern>/*</url-pattern>
8   </filter-mapping>
```

Applying the Intercepting Filter Pattern: Consequences

Advantages:

- Filters can be applied to all incoming requests without having to duplicate code
- Filters can easily be combined in different combinations
- Filters are easily reusable across applications
- Filters can be added and removed declaratively

Applying the Intercepting Filter Pattern: Consequences

Disadvantage:

Filters can be inefficient if a lot of data is shared between them

Applying the Front Controller Pattern: Problem Forces

The system lacks a centralized request-handling mechanism for activities needing to be completed for each request. This leads to these activities being littered across many components

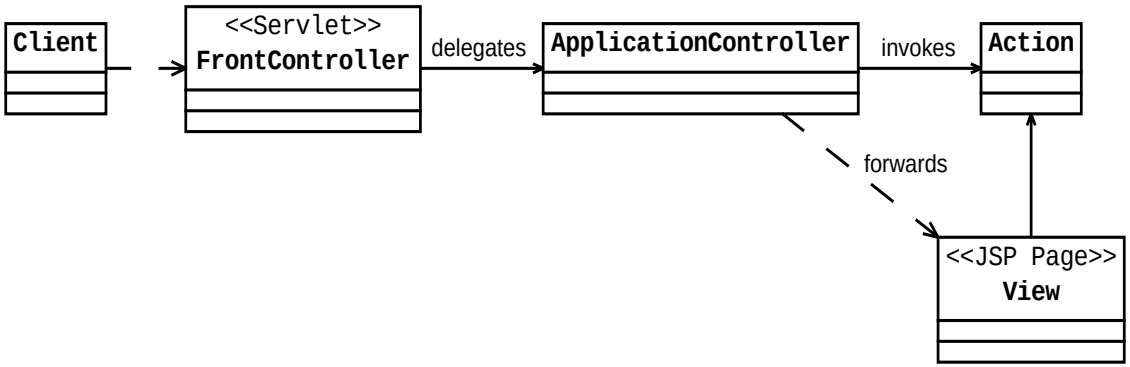
- Common system services, such as security and auditing, should not be duplicated in each view component
- Workflow management and a controller are needed
- Multiple views can be used to respond to similar business requests
- Tracking a user's progress through the site is easier with a centralized point of contact

Applying the Front Controller Pattern: Solution

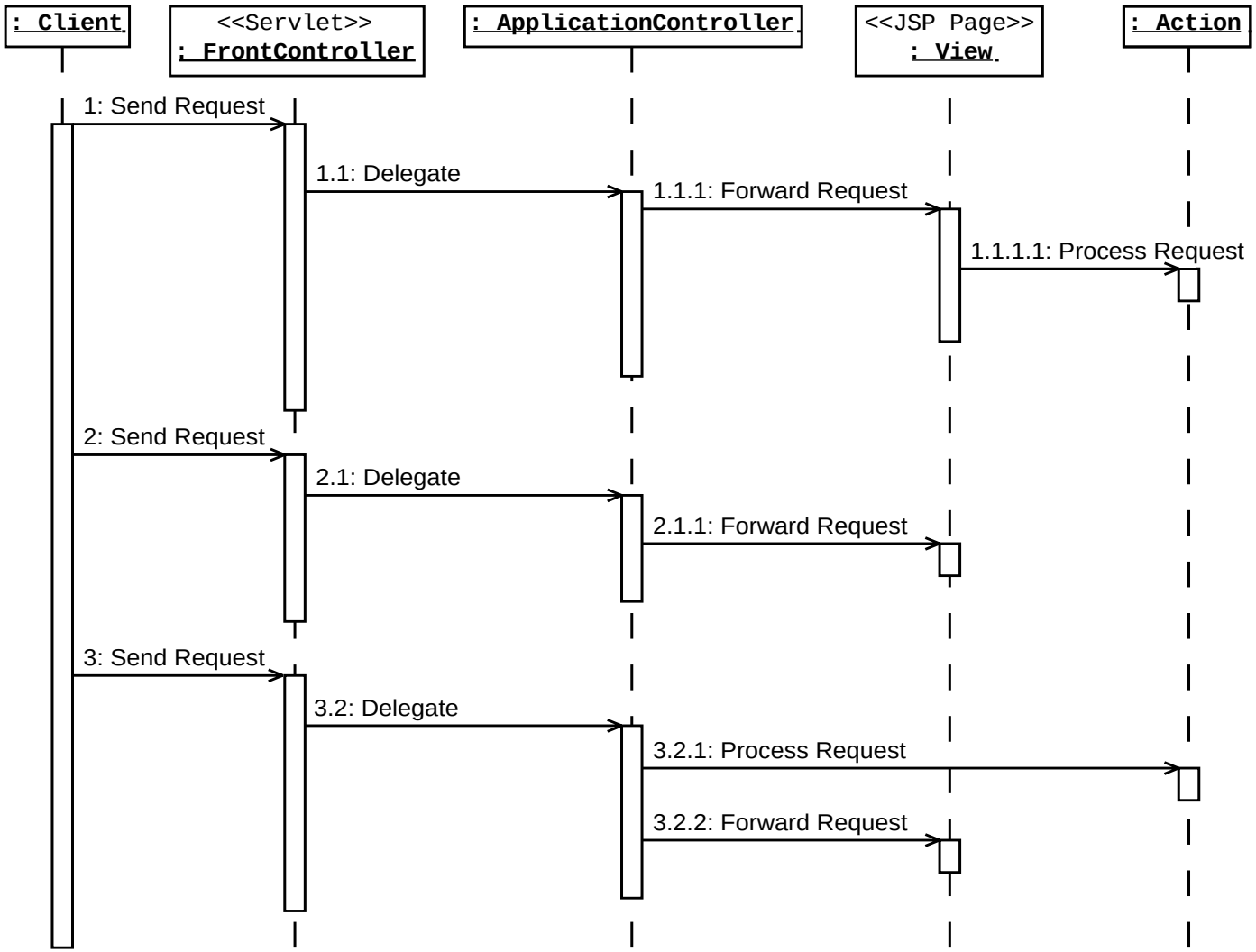
Use a Front Controller as the initial point of contact for handling a request. The controller manages request processing, including:

- Invoking security services
- Delegating business processing
- Managing the choice of an appropriate view
- Handling errors
- Managing the selection of content creation strategies

Front Controller Pattern Structure



Front Controller Pattern Sequence



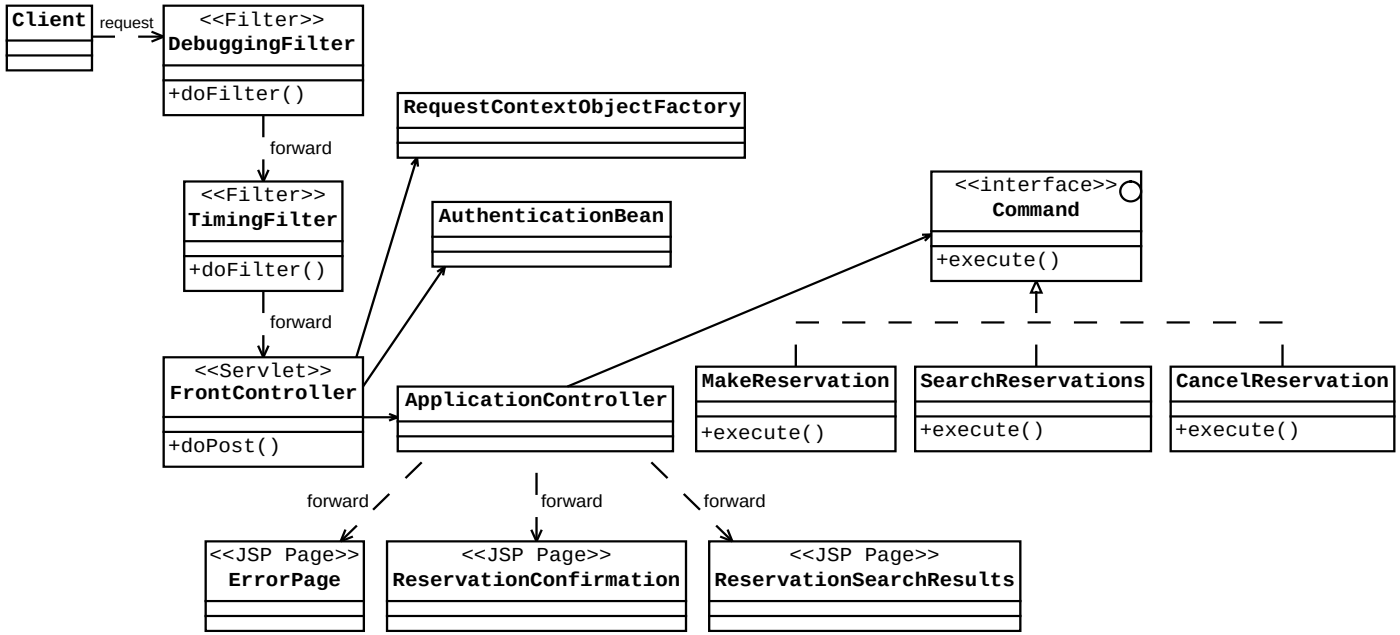
Applying the Front Controller Pattern: Strategies

- **Servlet Front Controller vs. JSP Page Front Controller** – A `FrontController` is usually a servlet
- **Dispatcher in Controller** – If dispatcher functionality is minimal, it can be included in the front controller
- **Filter Controller** – Some front controller functionalities can be placed in a filter

Applying the Front Controller Pattern: Strategies

- **Physical vs. Logical Resource Mapping** – Rather than clients requesting actual physical URLs such as `http://www.server.com/servlet/package.Controller`, use a logical URL mapping as an alias for the servlet
- **Multiplexed Resource Mapping** – Logical resource mapping can include wildcards, such as `*.do` which is mapped to the Struts `ActionServlet` controller

Front Controller Pattern Example



Applying the Front Controller Pattern: Consequences

Advantages:

- Avoids duplication of common request processing code
- Allows central control of the Web application flow
- Allows the tracking of a particular user and logging of related activity
- Manages validation and error handling centrally

Disadvantage:

The front controller can have poor cohesion

Applying the Application Controller Pattern: Problem Forces

The controller has two particularly important responsibilities:

- **Action Management** – The controller decides what action to invoke, which in turn typically invokes the business processing
- **View Management** – The controller decides which view to forward the request to

Applying the Application Controller Pattern: Problem Forces

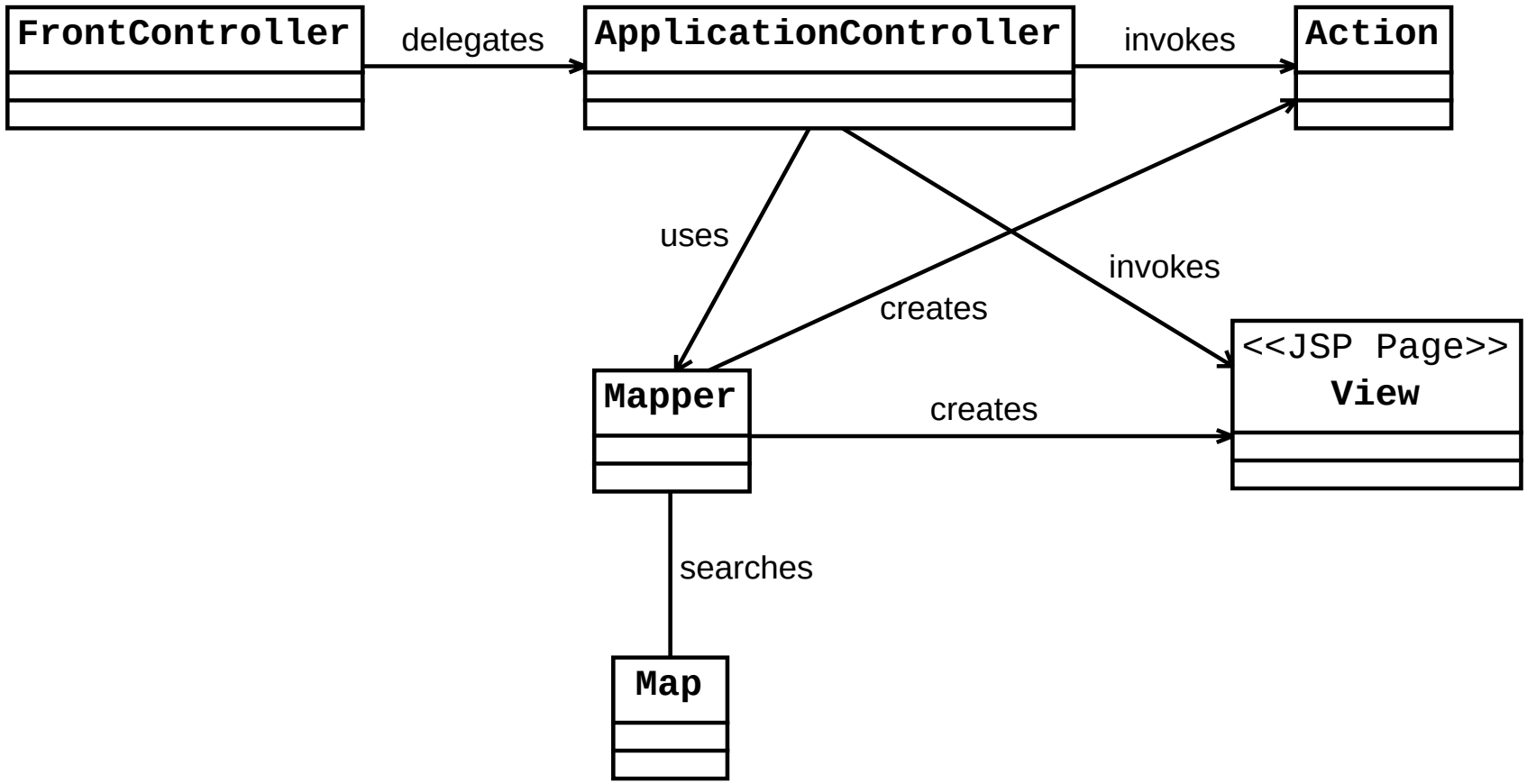
Placing the action management and view management in the front controller component can cause problems:

- The front controller might have poor cohesion
- This code is coupled to the servlet technology, making the code harder to reuse and test

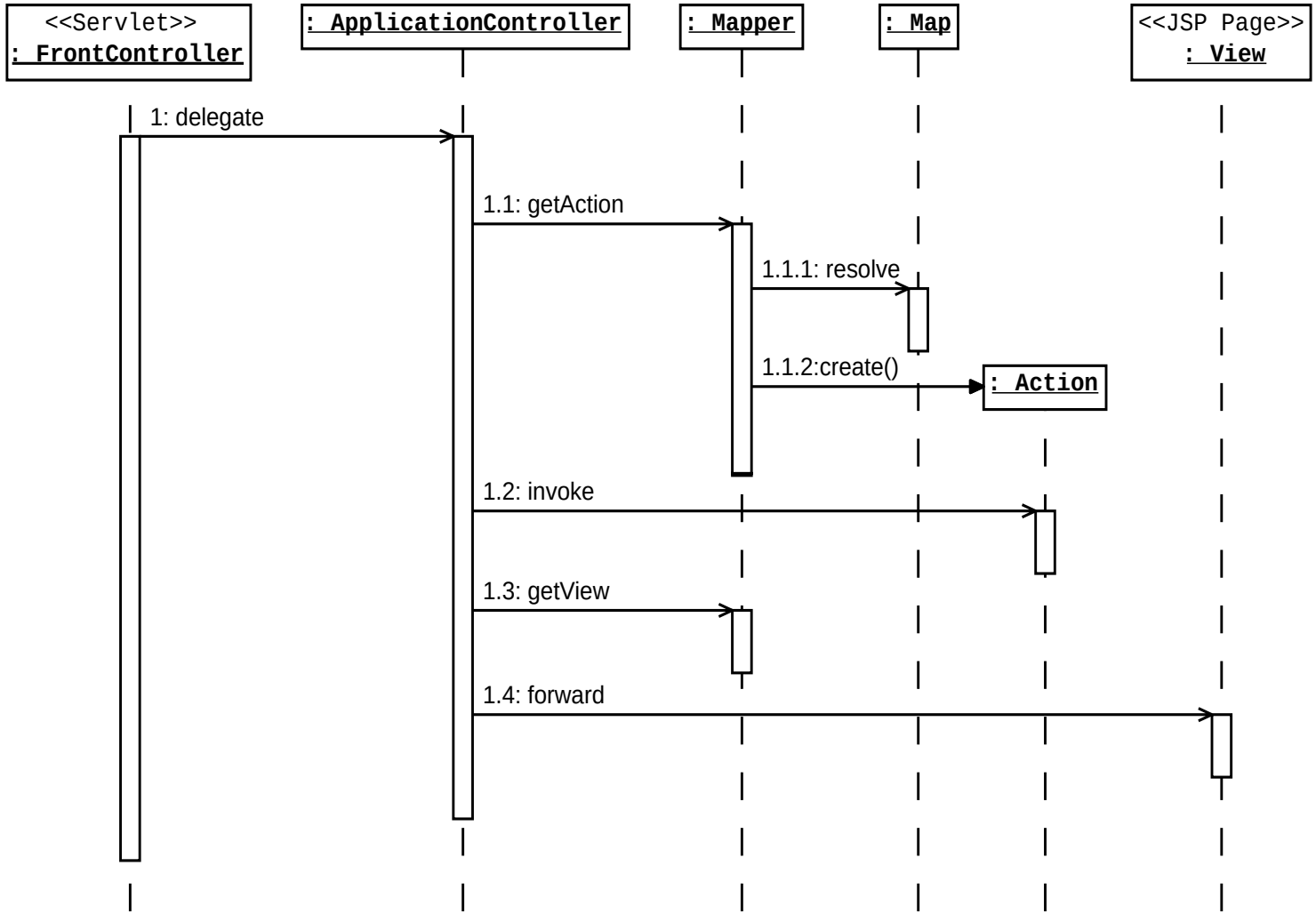
Applying the Application Controller Pattern: Solution

- The action management and view management are placed in the `ApplicationController` class
- The `FrontController` servlet does the initial request processing and invokes the `ApplicationController`
- The `ApplicationController` usually uses a `Mapper` class that determines what action to invoke and what view to forward to

Application Controller Pattern Structure



Application Controller Pattern Sequence



Applying the Application Controller Pattern: Strategies

- **Command Handler** – An application controller that performs action management. Application controllers usually invoke actions using the GoF Command pattern and the mapper component might use an XML configuration file to determine which action command should be invoked based on the request URL.
- **View Handler** – An application controller that performs view management. Most commonly an application controller is both a command handler and a view handler. The result of the action command might be used to determine from the XML configuration file which view the request should be forwarded to.

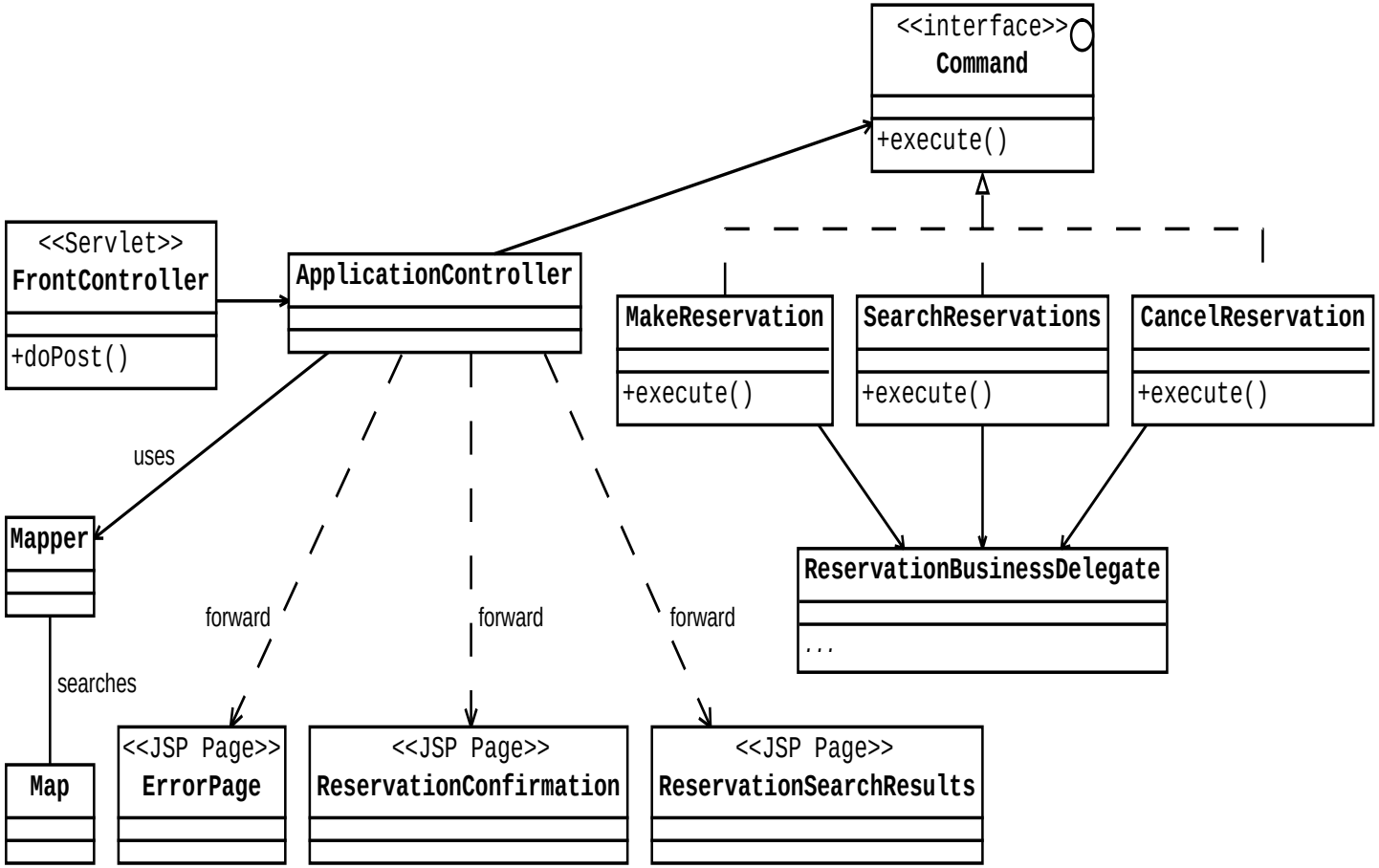
Applying the Application Controller Pattern: Strategies

- **Transform Handler** – XML Stylesheet Language Transformations (XSLT) are used instead of JSP pages to generate the view. The application controller determines an XSLT stylesheet to use and invokes that stylesheet.
- **Navigation and Flow Control** – In some way, the application controller controls the application flow.

Applying the Application Controller Pattern: Strategies

- **Custom Simple Object Access Protocol (SOAP) Message Handling** – Rather than being a request from a web user, the request can be a web services message.
- **JAX-RPC Message Handling** – This strategy is similar to the Custom SOAP Message Handling strategy, but it uses JAX-RPC to automate much of the work.

Application Controller Pattern Example



Application Controller Pattern Example

- Struts implements the Application Controller with the `RequestProcessor` class
- Each action that should be performed when a request is received is implemented as a command (`Action`) object
- For each type of request that requires business processing:
  - Create a subclass of the `Action` class
  - Implement the process and invoke business logic from the `execute` method
  - Create a mapping in the Struts configuration file, from the request URI to that `Action` class

Application Controller Pattern Example

Example Struts Mapping Code:

```
<action path="/reserveRoom" type="hotel.MakeReservation">
    <forward name=”success” path="/ReservationConfirmation.jsp"/>
    <forward name=”failure” path="/ErrorPage.jsp"/>
</action>
```

- The `MakeReservation` object's `execute` method returns an `ActionForward` object specifying "success" or "failure"
- The `RequestProcessor` then forwards the request to the `ReservationConfirmation.jsp` or `ErrorPage.jsp` page

Applying the Application Controller Pattern: Consequences

Advantages:

- Improves cohesion by separating action management and view management from the front controller
- Improves reusability by keeping the application controller protocol generic and thus reusable by different client types

Disadvantage:

Introduces an extra component that might be unnecessary for smaller applications

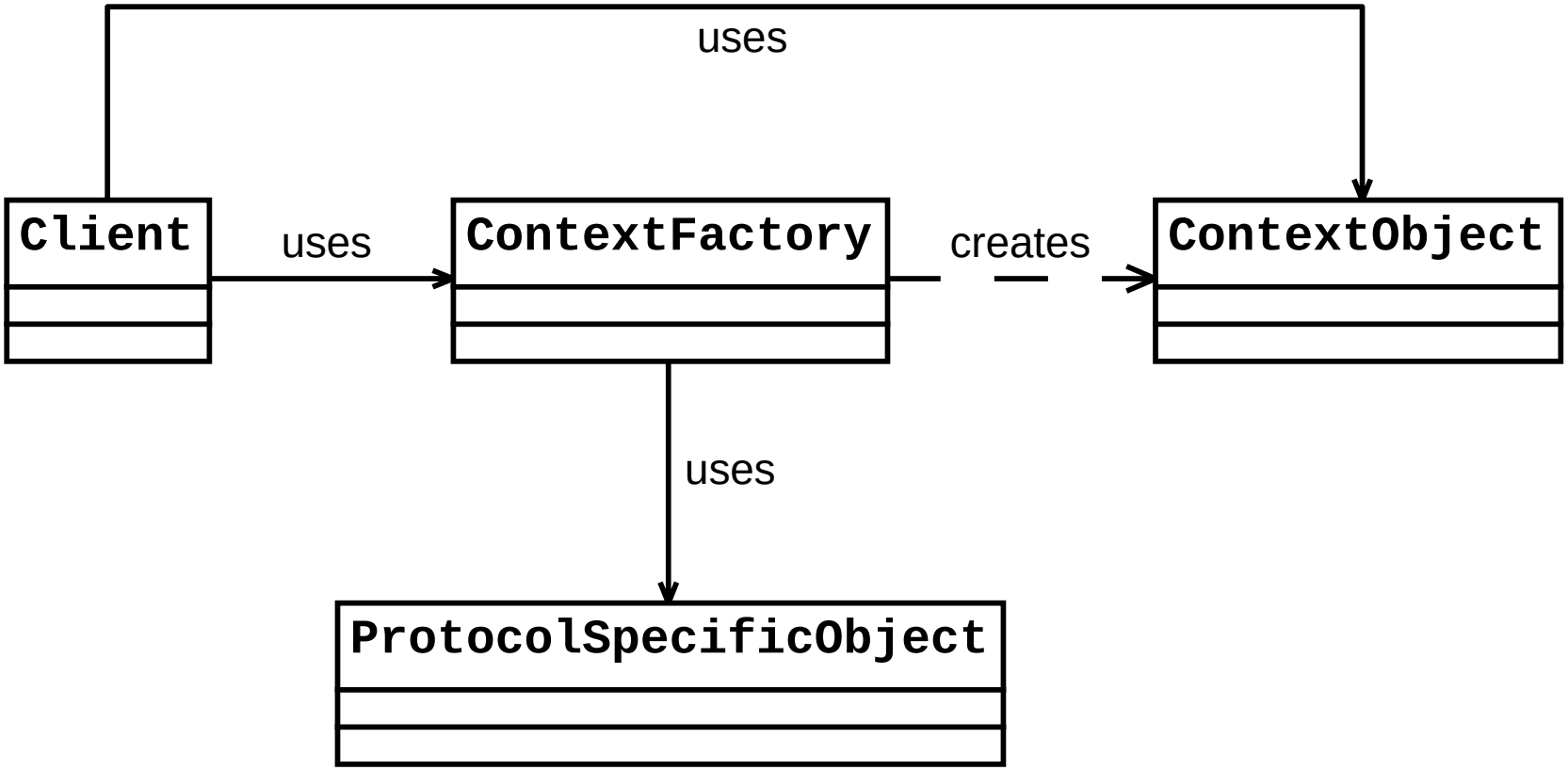
Applying the Context Object Pattern: Problem Forces

- Context specific objects, such as `HttpServletRequest` objects often contain data that is used throughout the request processing, including in other contexts such as the business tier
- If context specific objects are passed into classes in other contexts, the flexibility and reusability of those classes is reduced

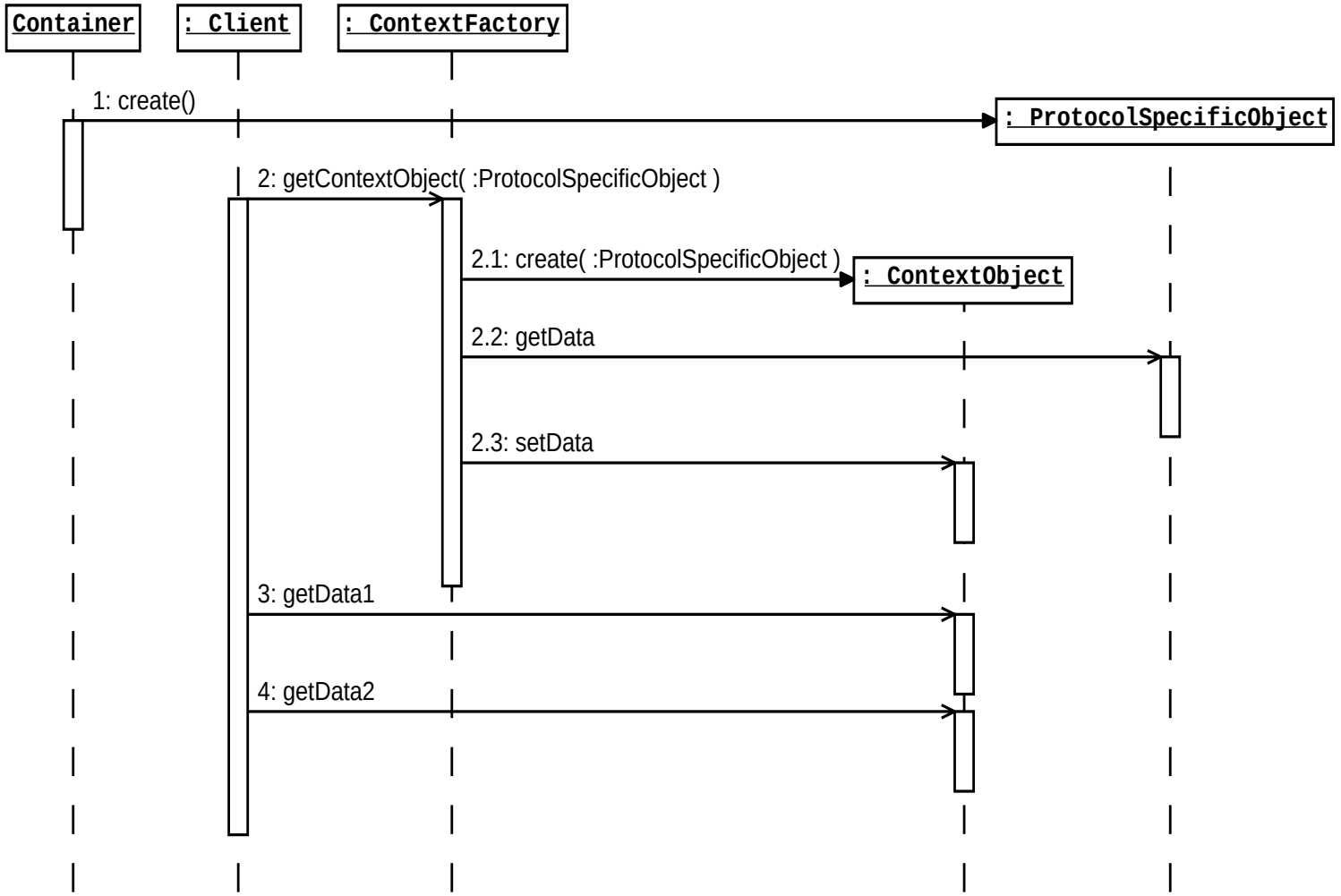
Applying the Context Object Pattern: Solution

- An object that encapsulates the data from the context specific object is created
- The context object can be a set of simple get methods, or it can have one `HashMap` generic getter method
- The context object can keep a reference to the protocol specific class, or it can keep a copy of the data from the protocol specific object
- The context object might also contain data validation and conversion logic
- A context object might seem similar to a transfer object, but they have different intents

Context Object Pattern Structure



Context Object Pattern Sequence



Applying the Context Object Pattern: Strategies

Strategies for `ServletRequest` context objects:

- **Request Context Map Strategy** – Context object holds the immutable Map object returned by the `ServletRequest` object's `getParameterMap` method
- **Request Context POJO Strategy** – A POJO context object has a get method for each piece of data. The constructor takes `ServletRequest` and populates instance variables with the data in `ServletRequest`
- **Request Context Validation Strategy** – For either of the above strategies, provide a `validate` method

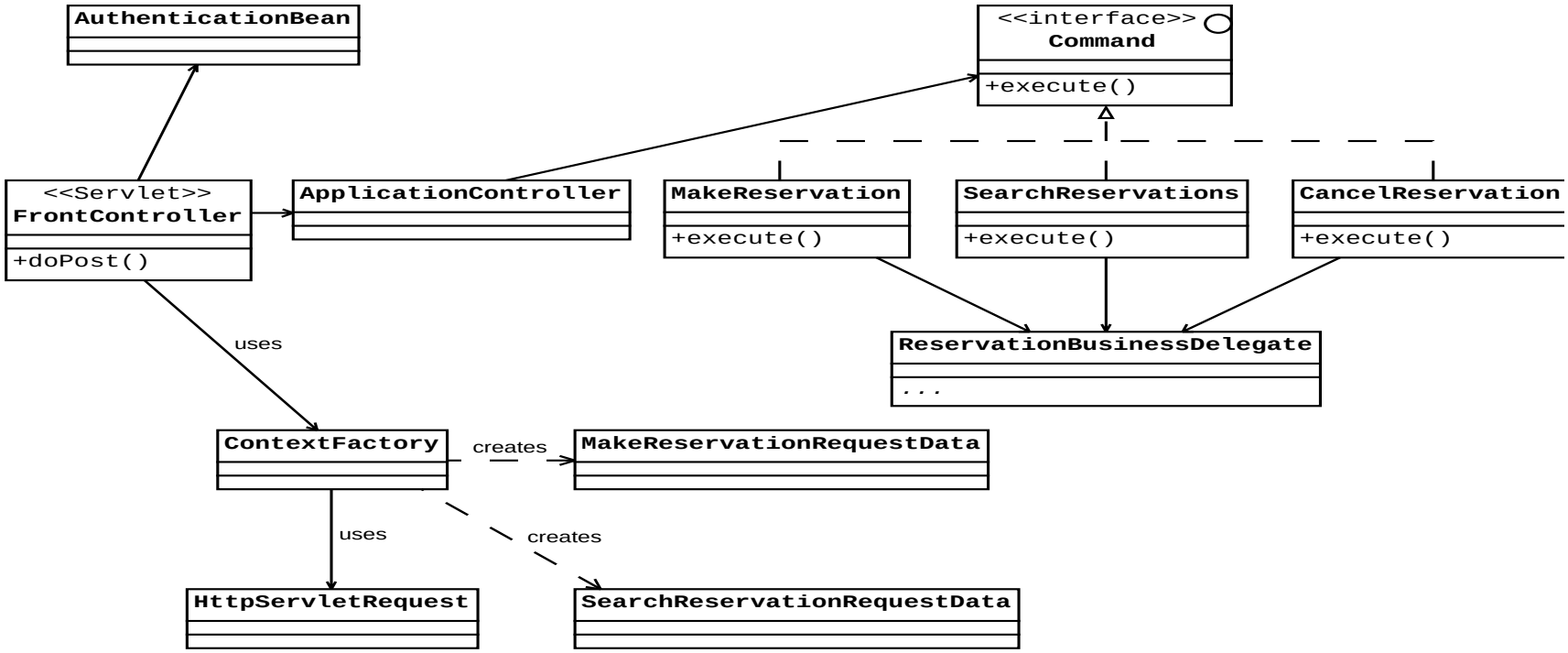
Applying the Context Object Pattern: Strategies

- **Configuration Context Strategy** – Context object stores configuration state, for example `javax.servlet.jsp.PageContext`
- **Security Context Strategy** – Context object stores security state, for example `javax.security.auth.login.LoginContext`

Applying the Context Object Pattern: Strategies

- **Context Object Factory Strategy** – Use a factory to create context objects
- **Context Object Auto-Population Strategy** – Rather than calling the methods to pass the data from the protocol-specific object to the context object one attribute at a time, you can use code that automatically copies all data over

Context Object Pattern Example



Context Object Pattern Example

```
1   import javax.servlet.*;
2
3   public class MakeReservationRequestData extends Request{
4     private String roomType;
5     private int numNights;
6     //...other fields
7
8     public void initialize(ServletRequest request) {
9       setRoomType(request.getParameter("roomType"));
10      int numNights = Integer.parseInt
11          (request.getParameter("numNights"));
12      setNumNights(numNights);
13    }
14    public boolean validate() {
15      //...Validation rules checked here
16      return true;
17    }
```

Context Object Pattern Example

```
18    public void setRoomType(String rt) {
19      roomType = rt;
20    }
21    public void setNumNights(int nn) {
22        numNights = nn;
23    }
24    public String getRoomType() {
25      return roomType;
26    }
27    public int getNumNights() {
28      return numNights;
29    }
30    //...more get/set methods
31  }
```

Applying the Context Object Pattern: Consequences

Advantages:

- Improves the reusability of components
- Improves testability since context objects can be filled with test data
- Reduces dependencies on protocol specific classes that might change in different technology versions

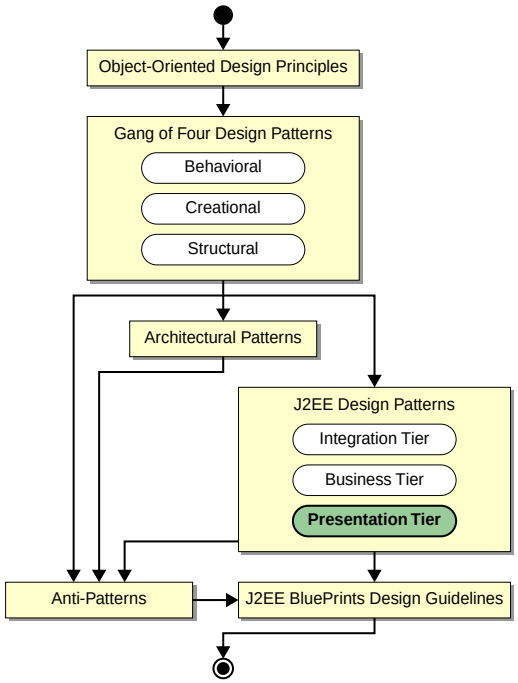
Applying the Context Object Pattern: Consequences

Disadvantage:

- Increases the amount of code that must be developed and maintained
- Slightly decreases performance by making copies of data and adding a layer of indirection

Summary

- **Intercepting Filter** – Provides a simple mechanism for pre-processing and post-processing HTTP requests and responses
- **Front Controller** – Provides a single entry point to the presentation tier; may handle security and validation
- **Application Controller** – Separates the action invocation and view dispatching from the front controller
- **Context Object** – Passes data from context-specific objects to other scopes



Module 11

More Presentation Tier Patterns

Objectives

- Apply the View Helper pattern
- Apply the Composite View pattern
- Apply the Dispatcher View pattern
- Apply the Service to Worker pattern

More Presentation Tier Patterns

Pattern	Primary Function
View Helper	Provides an object to carry out presentation tier logic in order to remove it from view components, like JSP pages.
Composite View	Provides a simple way to create views from smaller view components.
Dispatcher View	Combines the Front Controller and View Helper patterns. The view components request business processes and thus might have to assume some controller responsibility.
Service To Worker	Combines the Front Controller and View Helper patterns. The controller requests business processes.

Applying the View Helper Pattern: Problem Forces

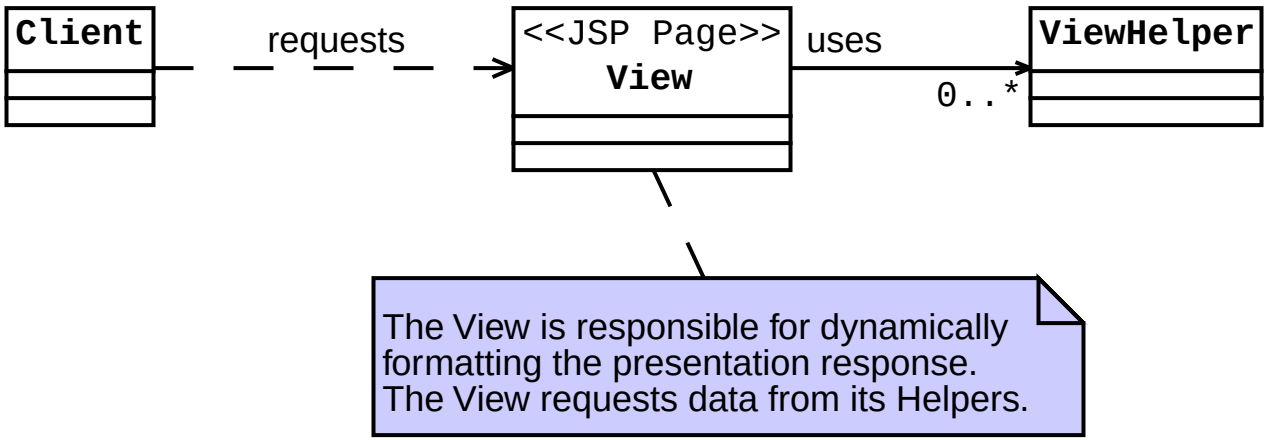
Combining data preparation and formatting logic with view generation logic leads to:

- Poor role separation between Web content developers and programmers
- Brittle view components
- Copy-and-paste reuse

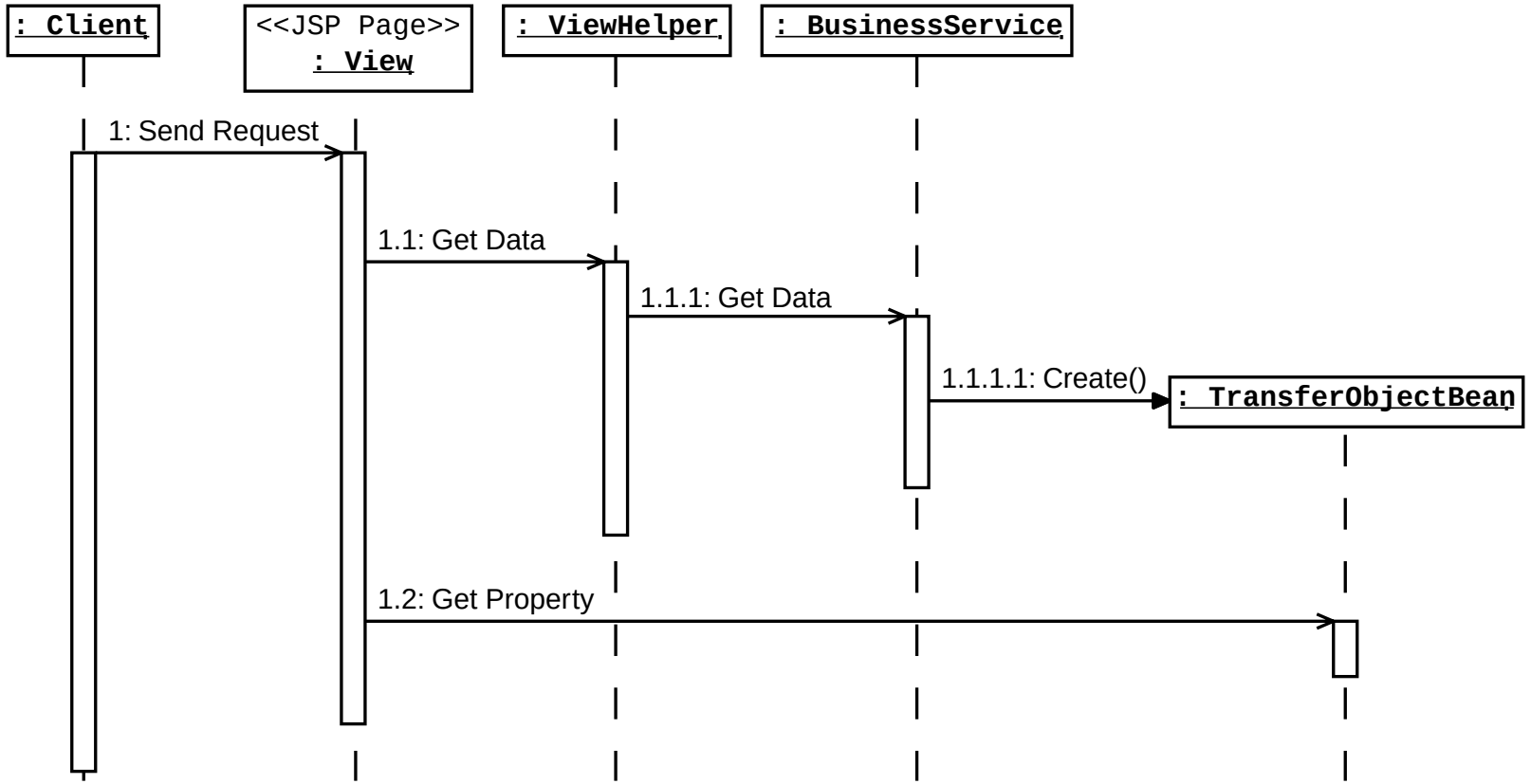
Applying the View Helper Pattern: Solution

- The view focuses on presentation logic
- Business logic and data formatting logic moved to helper classes
- Helper classes can be either JavaBeans components, custom tags or simple tag files

View Helper Pattern Structure



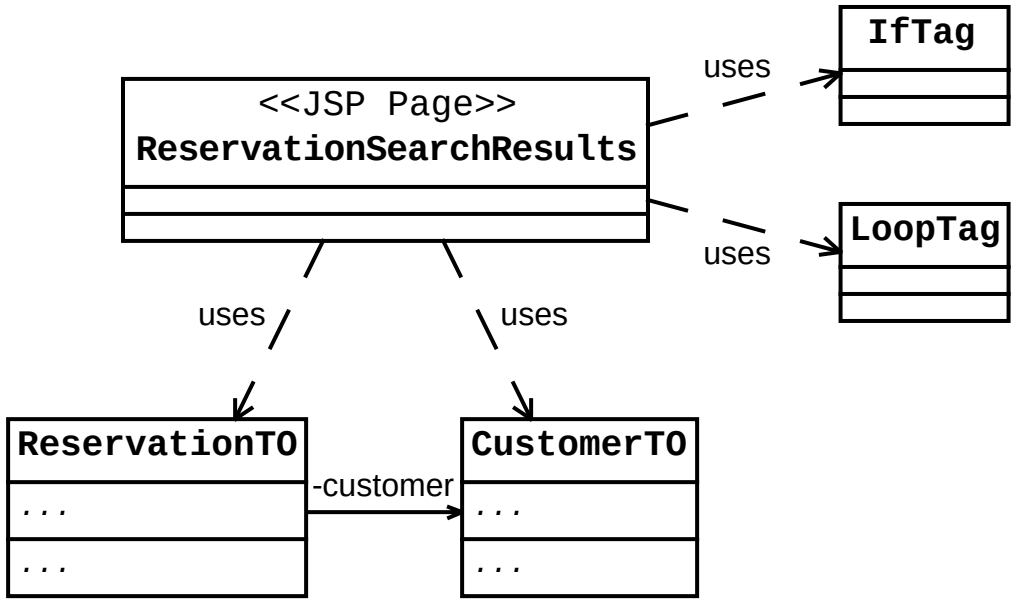
View Helper Pattern Sequence



Applying the View Helper Pattern: Strategies

- **JSP Page View vs. Servlet View** – The `View` component is usually a JSP page
- **JavaBeans Component Helper** – The `ViewHelper` component is a JavaBeans component
- **Custom Tag Helper** – `ViewHelper` is a custom tag
- **Tag File Helper** – `ViewHelper` is a JSP 2.0 tag file
- **Business Delegate as Helper** – `ViewHelper` is a business delegate
- **Transformer Helper** – `ViewHelper` is implemented as an eXtensible Stylesheet Language Transformer (XSLT)

View Helper Pattern Example



Applying the View Helper Pattern: Example

```
1   <%@ attribute name="reservations" type="java.util.Collection" %>
2   <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
3   <table>
4     <c:forEach var="reservation"
5             items="${reservations}" varStatus="status">
6       <tr>
7         <td><c:out value="${status.count}"/></td>
8         <td><c:out value="${reservation.roomType}"/></td>
9         <td><c:out value="${reservation.startDate}"/></td>
10        <td><c:out value="${reservation.endDate}"/></td>
11      </tr>
12    </c:forEach>
13  </table>
```

```
1   <%@ taglib prefix="tag" tagdir="/WEB-INF/tags" %>
2   <tag:buildResTable reservations="${reservations}"/>
```

Applying the View Helper Pattern: Consequences

Advantages:

- Using `ViewHelper` components results in a cleaner separation between the view and the data processing logic
- Separating the data preparation logic from the view reduces dependencies that developers might have on the same resources
- Data preparation and formatting logic is more reusable

Applying the Composite View Pattern: Problem Forces

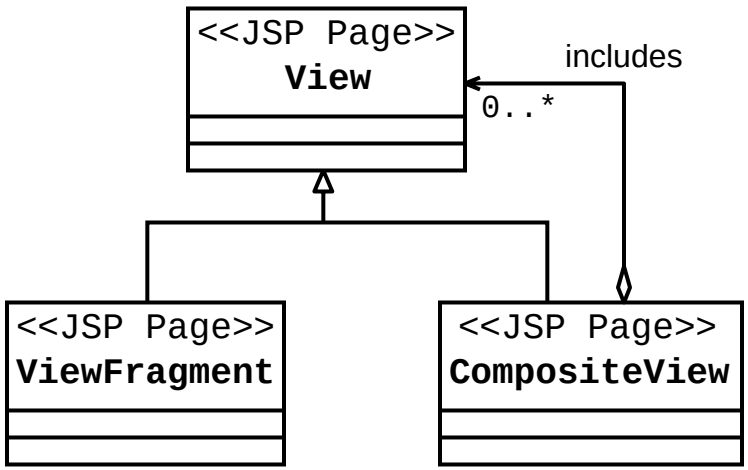
Copying and pasting content on multiple pages causes several problems:

- A content change can require changing many JSP pages
- A consistent look and feel is difficult to maintain
- Many views are needed to create combinations of content

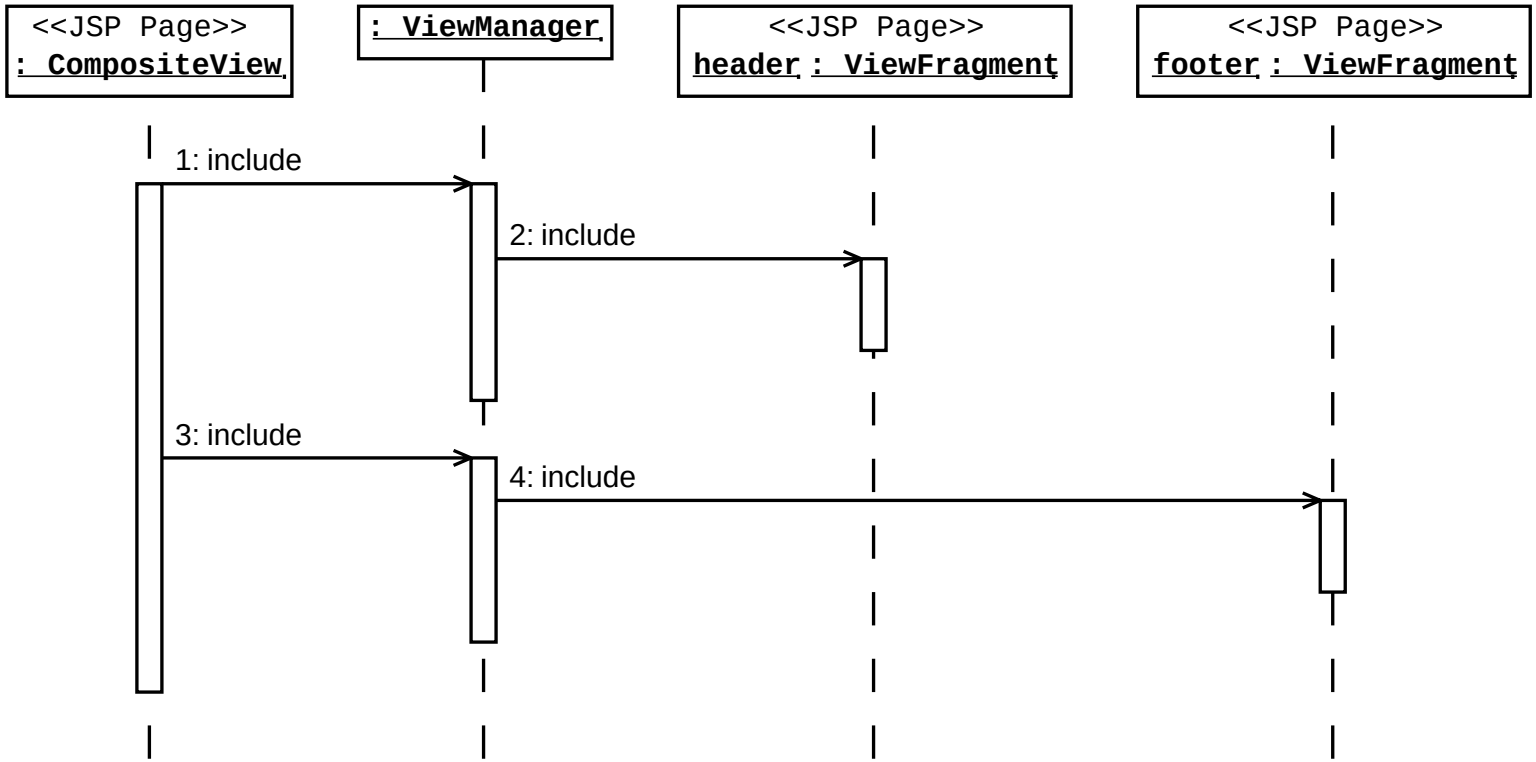
Applying the Composite View Pattern: Solution

- Make a `CompositeView` page that includes `ViewFragment` pages and other `CompositeView` pages
- Use the `CompositeView` page to structure the layout of a page

Composite View Pattern Structure



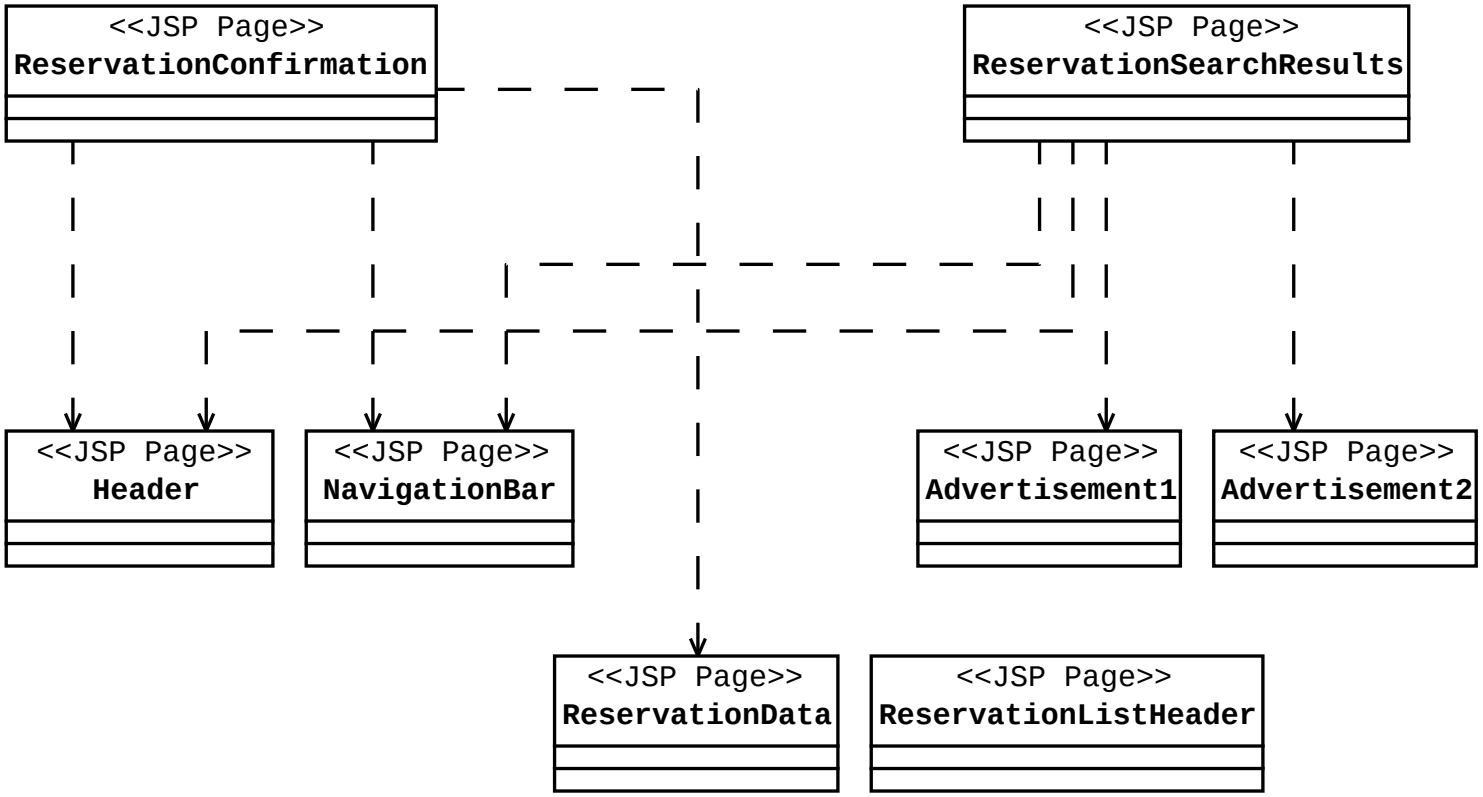
Composite View Pattern Sequence



Applying the Composite View Pattern: Strategies

- **Early-Binding Resource** – Include pieces of the template at translation time (`<%@ include>`)
- **Late-Binding Resource** – Include pieces of the template at runtime (`<jsp:include>`)
- **JavaBeans Component View Management** – Views are included using JavaBeans components
- **Custom Tag View Management** – Views are included using custom tags. Apache Tiles can be used to implement this pattern. Tiles is often used with Struts.
- **Transformer View Management** – Views are managed using an XSL Transformer

Composite View Pattern Example



Composite View Pattern Example

```
1   <TABLE BORDER='0'>
2     <TR>
3       <TD>
4         <%@ include file="../incl/Header.jsp" %>
5       </TD>
6     </TR>
7     <TR>
8       <TD>
9         <jsp:include page="../incl/NavigationBar.jsp"/>
10      </TD>
11      <TD>
12        <jsp:include page="../incl/ReservationData.jsp"/>
13      </TD>
14    </TR>
15  </TABLE>
```

Applying the Composite View Pattern: Consequences

Advantages:

- Fragments can be reused in numerous views
- Manageability is improved because fragments or templates can be changed in one place
- Templates can dynamically include different fragments based on the request or user

Applying the Composite View Pattern: Consequences

Disadvantages:

- Invalid content can be created if templates and fragments are not properly configured
- Runtime inclusion of fragments can decrease performance

Applying the Dispatcher View Pattern

- The Dispatcher View pattern is one way to combine the View Helper, Front Controller, and Application Controller patterns
- This pattern reduces the role of the controller components
- The Service to Worker pattern is an alternative to this pattern, and it might be more appropriate for more complex Web applications

Applying the Dispatcher View Pattern: Problem Forces

- Without controller components, controller activities can be dispersed throughout the system
- Without view helpers, JSP pages might have large quantities of scriptlet code
- If the Front Controller, Application Controller, and the View Helper patterns are used, you need to decide whether the controller or view components should request the business services

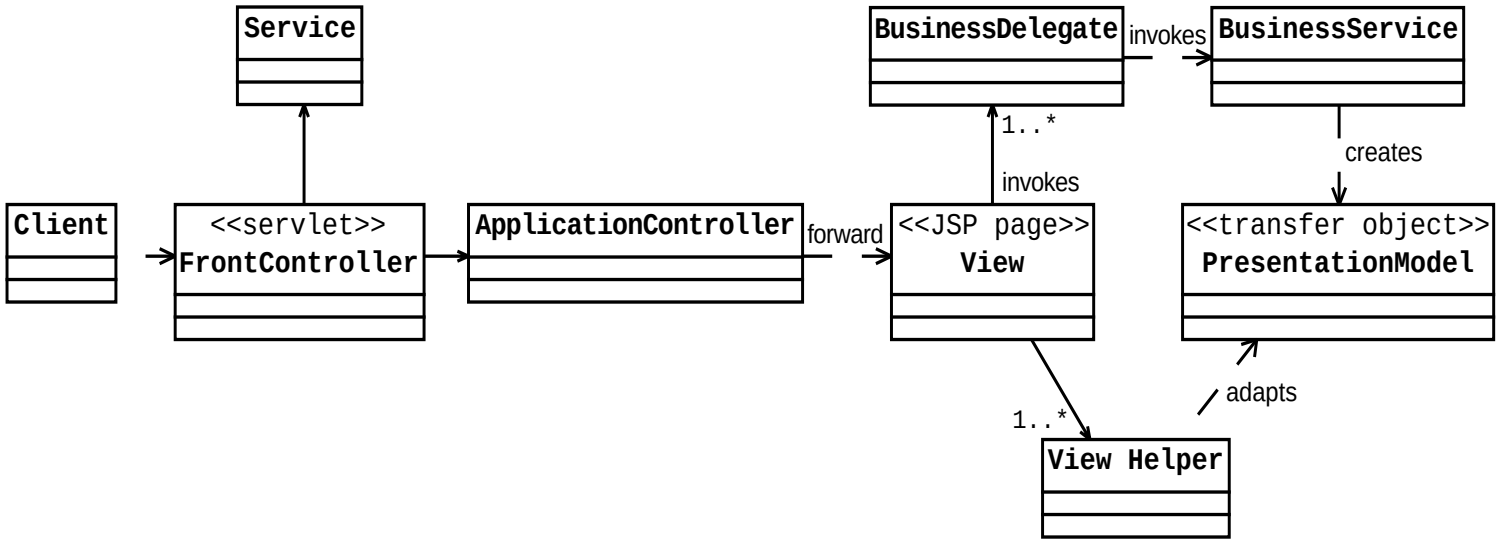
Applying the Dispatcher View Pattern: Solution

The controller component:

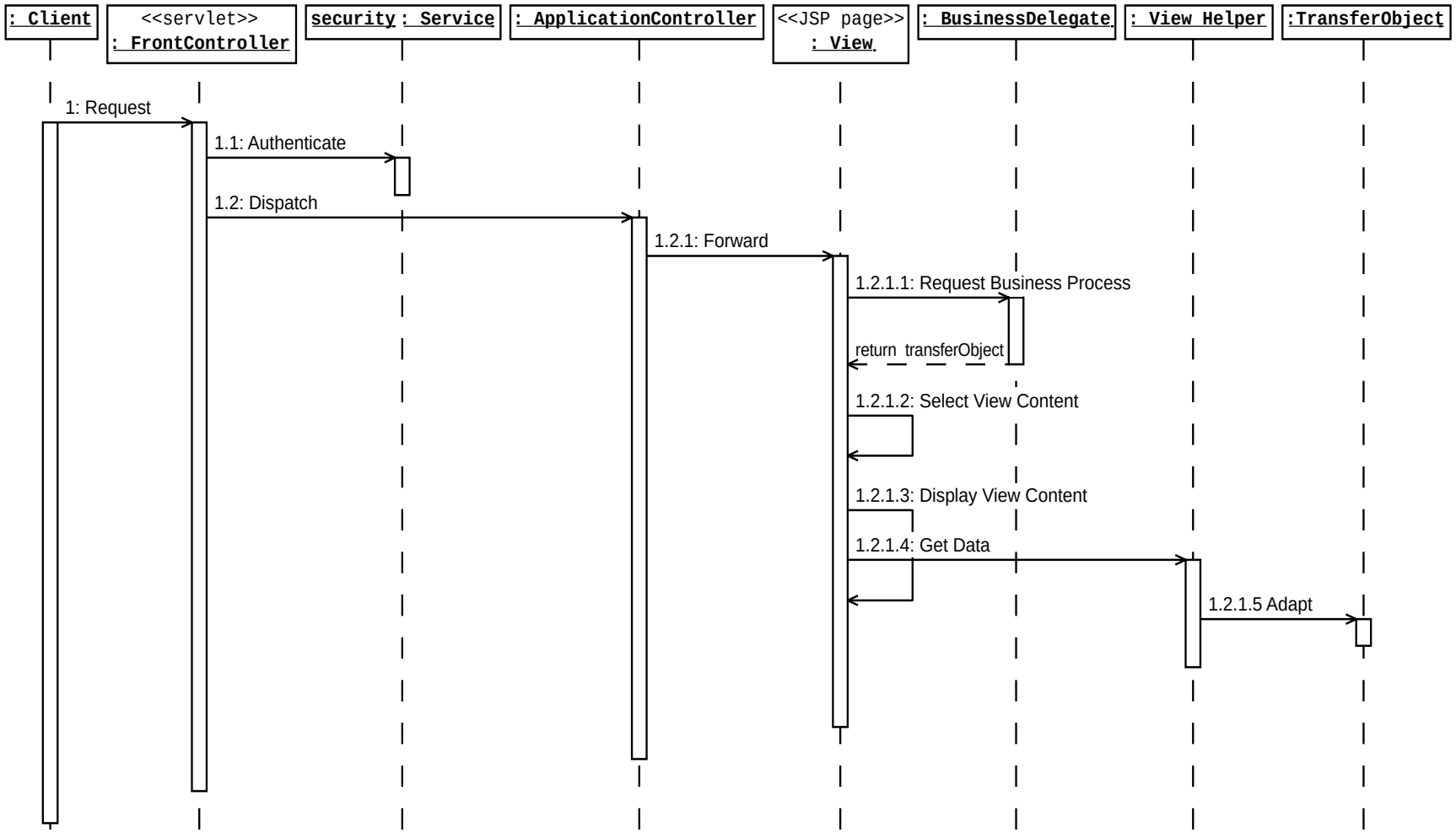
- Invokes common `Service` components
- Does not invoke business services or adapt the display model
- Dispatches using information included in the request state, as opposed to return values from `BusinessService` objects

The `View` component uses the `BusinessDelegate` components, which can be JavaBeans components or custom tags to invoke business services and adapt the display model

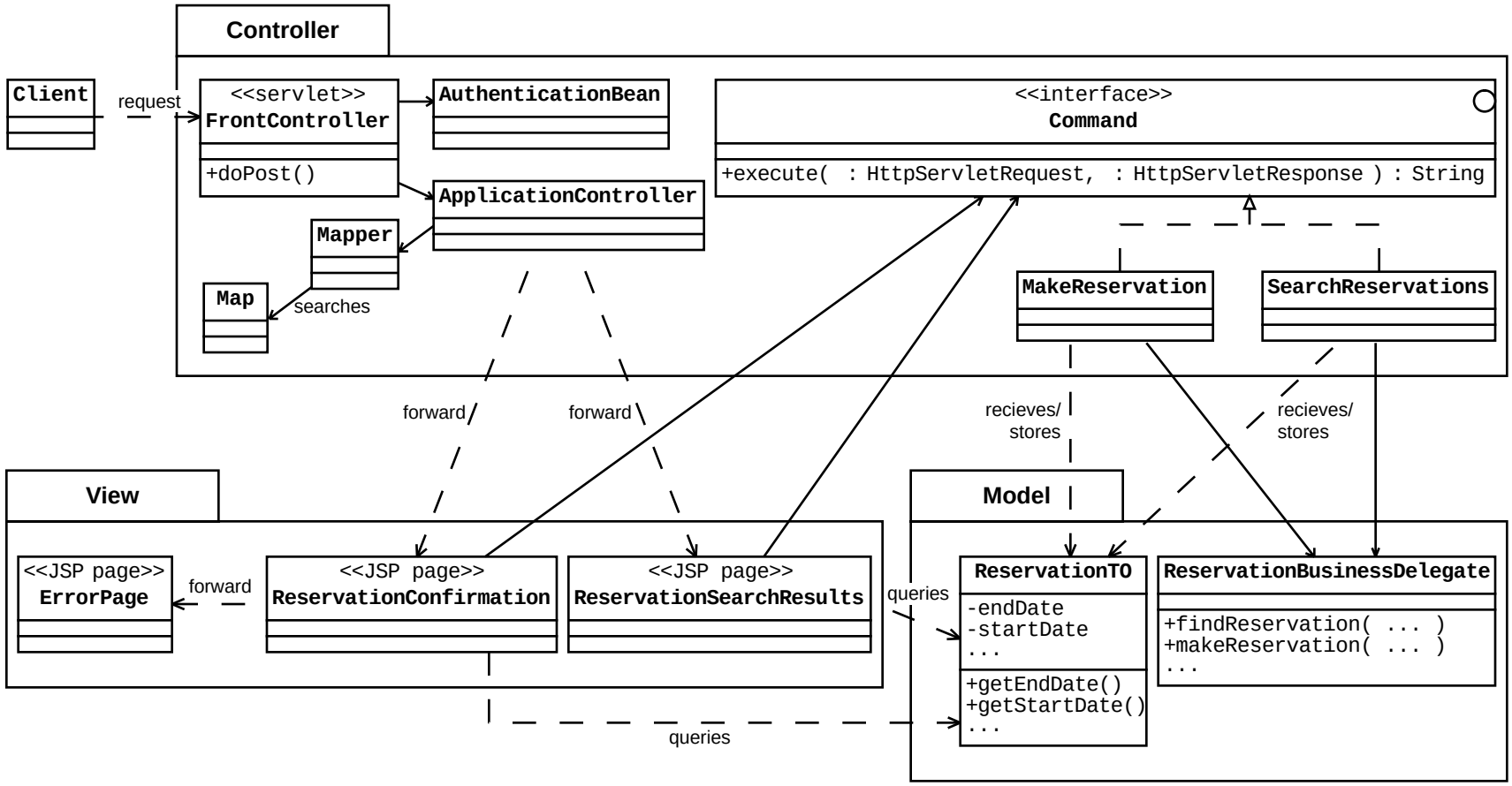
Dispatcher View Pattern Structure



Dispatcher View Pattern Sequence



Dispatcher View Pattern Example



Dispatcher View Pattern Example

The front controller code:

```
1   public class ApplicationController {
2   //. . .
3     public void process(HttpServletRequest request,
4                         HttpServletResponse response)
5             throws IOException, ServletException {
6       String action = request.getParameter("action");
7       String viewName=mapper.getView(action);
8       RequestDispatcher view;
9       view = request.getRequestDispatcher(viewName);
10      view.forward(request, response);
11    }
```

Dispatcher View Pattern Example

The command code:

```
1   public class MakeReservation implements Command {
2     public String execute(HttpServletRequest request,
3                           HttpServletResponse response) {
4       String startDate = request.getParameter("startdate");
5       //. . .get other request parameters
6       //. . .validate request parameters
7       try {
8         ReservationBusinessDelegate rbd =
9           new ReservationBusinessDelegate();
10        ReservationTO resData = rbd.makeReservation(. . .);
11        if (resData == null) {
12          return "failure";
13        } else {
14          request.setAttribute("resData", resData);
15          return "success";
16        } . . .
```

Dispatcher View Pattern Example

The pattern view page:

```
1   <myTag:executeLogic>
2   <jsp:useBean id='result' scope='request' class='java.lang.String'/>
3   <%if (result.equals("failure") {%>
4     <jsp:forward page="errorPage.jsp">
5   <%}%>
6
7   <jsp:useBean id='resData'
8       scope='request' class='ReservationTO'/>
9   Your reservation for
10  <jsp:getProperty name='resData' property='startDate'/>,
11  has been confirmed.
```

Applying the Dispatcher View Pattern: Consequences

Advantage:

Offers most of the advantages of the Front Controller, Application Controller, and View Helper patterns

Applying the Dispatcher View Pattern: Consequences

Disadvantages:

- The `View` components can have a large amount of scriptlet code to request business processing
- Many `View` components can have identical code to request the same business processing
- The `View` components will likely either produce different views or dispatch to other `View` components to react to different business processing results

Applying the Service to Worker Pattern

- The Service to Worker pattern is another way to combine the View Helper, Front Controller, and Application Controller patterns
- This pattern increases the role of the controller components

Applying the Service to Worker Pattern: Problem Forces

- Without controller components, controller activities may be dispersed throughout the system
- Without view helpers, JSP pages may have large quantities of scriptlet code
- If the Front Controller, Application Controller, and the View Helper patterns are used, you need to decide whether the controller or view components should request the business services

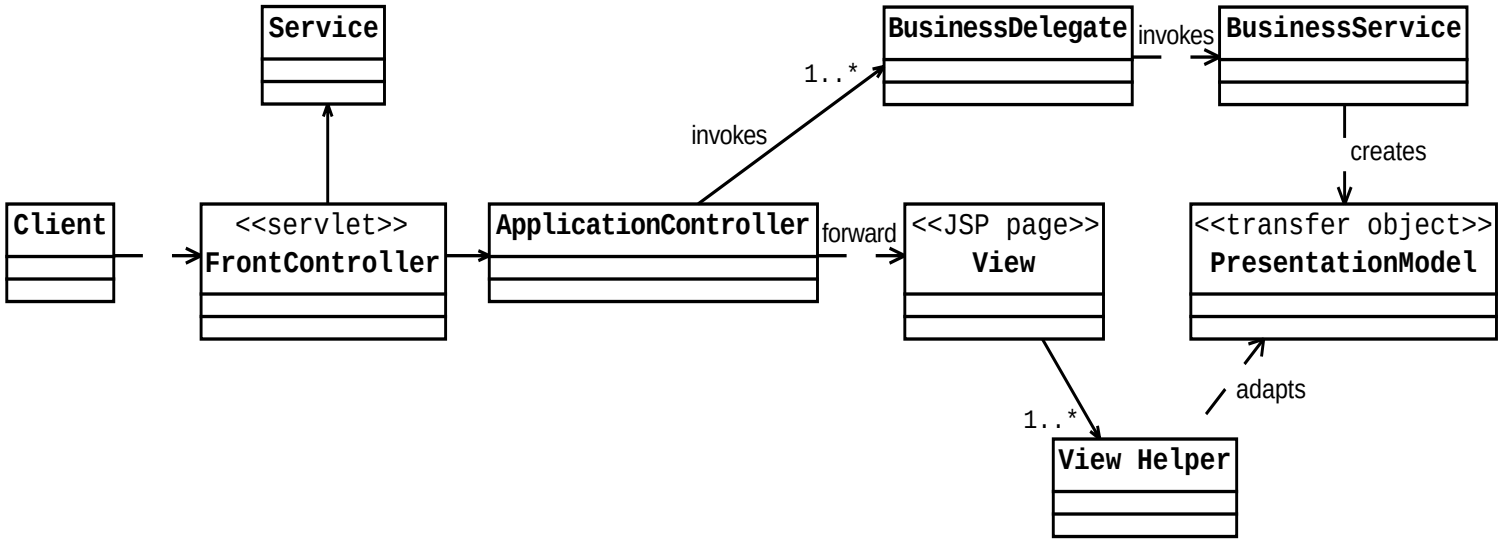
Applying the Service to Worker Pattern: Solution

The controller components:

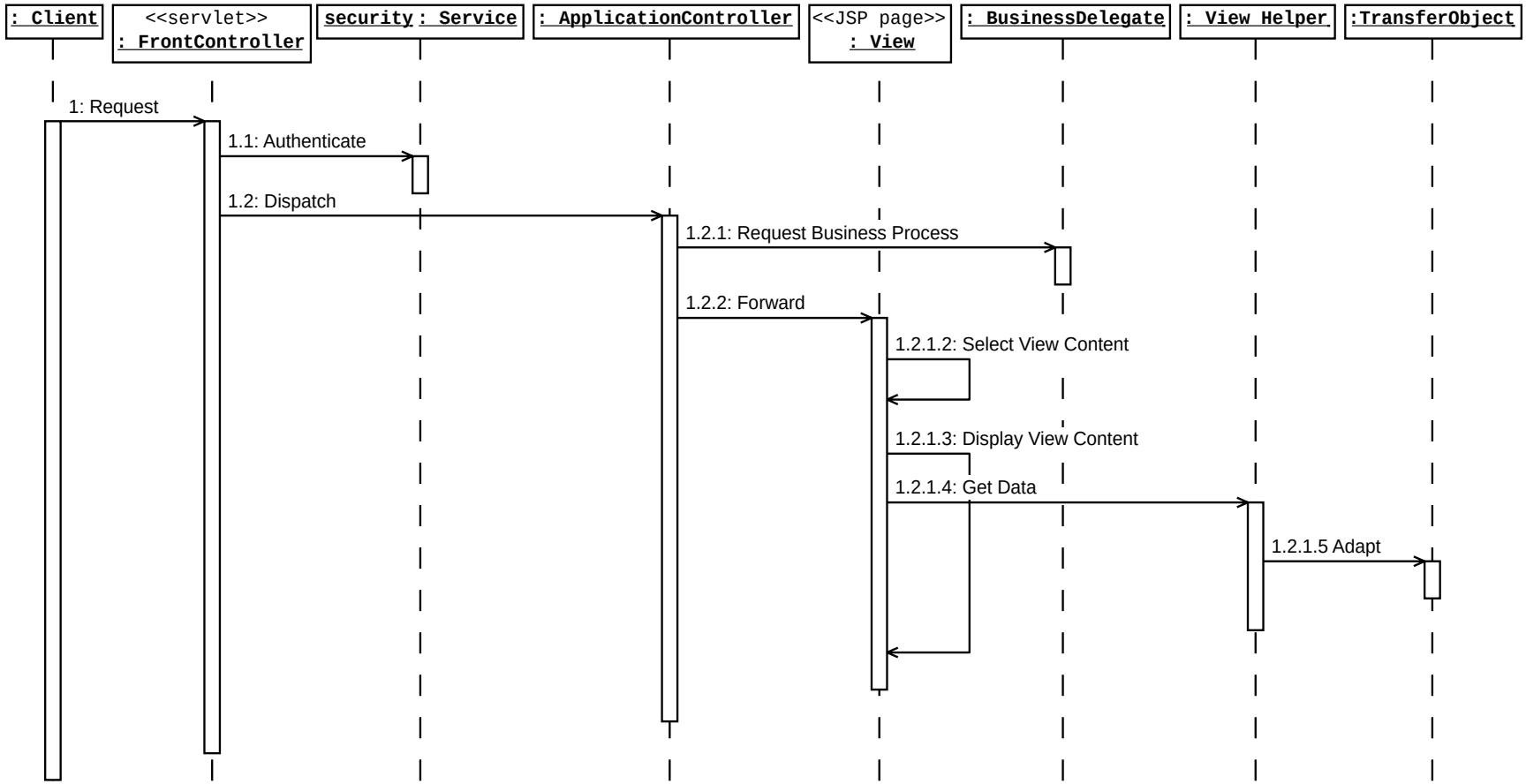
- Invoke common `Service` components
- Invoke the business processing that is appropriate for this request
- Dispatch to a particular `View` component, based on the request state and the results of the business processing

The `View` component is only responsible for querying the `Helper` objects for data produced by the controller's business processing request and creating the display content.

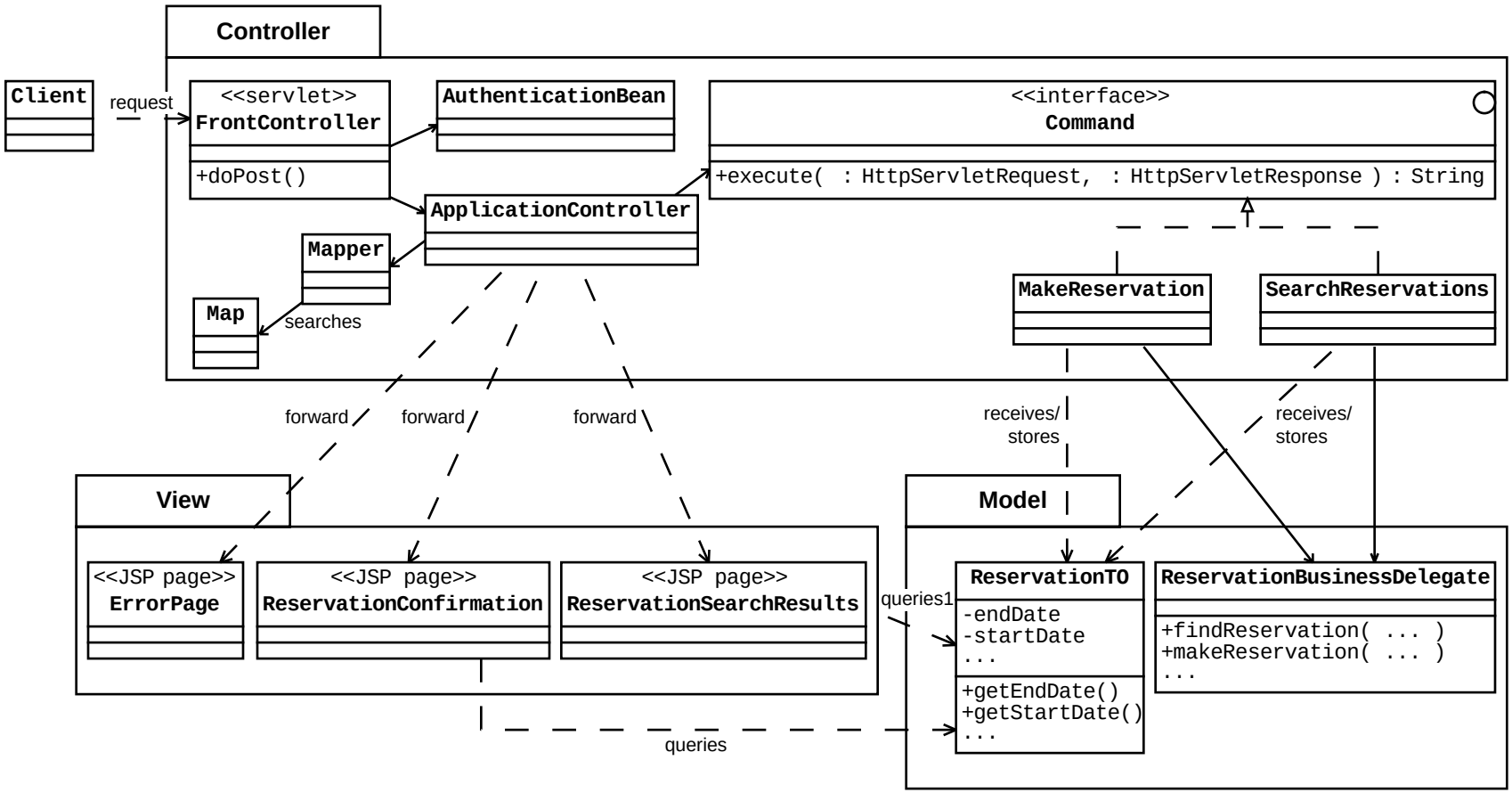
Service to Worker Pattern Structure



Service to Worker Pattern Sequence



Service to Worker Pattern Example



Service to Worker Pattern Example

The controller code:

```
1   public class ApplicationController {
2     //. . .
3     public void process(HttpServletRequest request,
4                         HttpServletResponse response)
5             throws IOException, ServletException {
6       String action = request.getParameter("action");
7       Command cmd = mapper.getCommand(action);
8       String result = cmd.execute(request, response);
9       String viewName = mapper.getView(action, result);
10      RequestDispatcher view;
11      view = request.getRequestDispatcher(viewName);
12      view.forward(request, response);
13    }
```

Service to Worker Pattern Example

The view page:

```
1   <jsp:useBean id='resData'
2       scope='request' class='ReservationTO'/>
3   Your reservation for
4   <jsp:getProperty name='resData' property='startDate'/>,
5   has been confirmed.
```

Applying the Service to Worker Pattern: Consequences

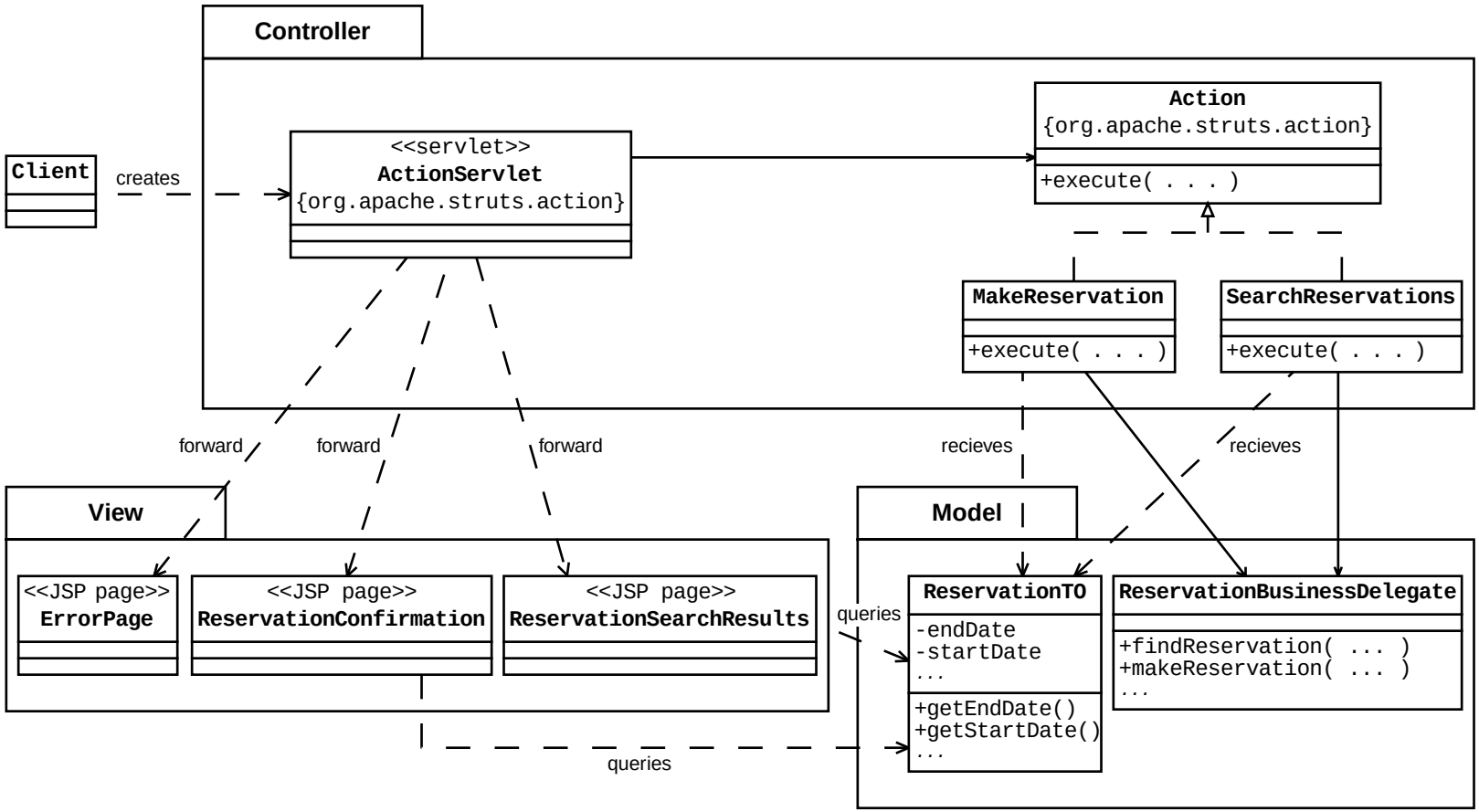
Advantages:

- Offers all the advantages of the Front Controller, Application Controller, and View Helper patterns
- Compared to the Dispatcher View pattern, `View` components are likely be more cohesive, have less scriptlet code, and be easier to adapt and maintain

Disadvantage:

Depending on how it is implemented, the `FrontController` component can have poor cohesion

Service to Worker Pattern In Struts



Summary

- **View Helper** – Provides an object to carry out presentation tier logic to remove it from view components, like JSP pages.
- **Composite View** – Provides a simple way to create views from smaller view components.
- **Dispatcher View Pattern** – Combines the Front Controller and View Helper patterns. The view components request business processes and thus might have to assume some controller responsibility.
- **Service to Worker Pattern** – Combines the Front Controller and View Helper patterns. The controller requests business processes.

Module 12

Exploring AntiPatterns

Objectives

- Define AntiPatterns
- Describe Integration Tier AntiPatterns
- Describe Business Tier AntiPatterns
- Describe Presentation Tier AntiPatterns

Introducing AntiPatterns

- AntiPatterns identify and document well-known code development practices that have a negative impact on a software system
- Studying AntiPatterns helps you avoid them and learn refactored solutions
- In this module, AntiPatterns are grouped into three sets of patterns:
  - Integration Tier AntiPatterns
  - Business Tier AntiPatterns
  - Presentation Tier AntiPatterns

Describing Integration Tier AntiPatterns

- These AntiPatterns involve poor usage of the J2EE platform integration tier technologies
- Work hand in hand with the integration tier J2EE patterns to avoid poor designs and find good designs for the integration tier

Not Pooling Connections AntiPattern

AntiPattern problem:

Creating and destroying database connections every time a query is run is expensive

Symptoms and consequences:

- Performance is poor
- Scalability is poor
- Database driver and location data might be hardcoded in many parts of the system

Not Pooling Connections AntiPattern

Refactored solution:

- Use a connection pool
- Use the `DataSource` interface

Monolithic Consumer AntiPattern

AntiPattern problem:

- Business logic is placed in the `onMessage` method of a message driven bean
- This business logic can only be invoked by sending a JMS message to a messaging server

Symptoms and consequences:

- Clients don't have the option of invoking business logic using a synchronous method invocation
- Testing of the business logic is more difficult

Monolithic Consumer AntiPattern

Refactored solution:

- The message driven bean:
  - Reads and prepares the data in the message
  - Invokes a method on a regular class or a session bean
- Clients that want to synchronously request a service can call the logic directly in the business object
- Service Activator pattern's structure can be used to achieve the refactored solution

Fat Message AntiPattern

AntiPattern problem:

- Asynchronous messages delivered through messaging servers can become too fat
- Fat messages cause:
  - Network overhead increases
  - Strain on messaging server that might have to persist the messages
  - Strain on senders and receivers that have to marshall and unmarshall the data

Fat Message AntiPattern

Symptoms and consequences:

- Network overhead increased
- Strain on the messaging server increases
- Strain on senders and receivers increases

Refactored solution:

- Carefully consider the necessary parameters for an asynchronous service, just as you do for remote method invocations
- Consider the possibility of only passing an ID or a reference to the data

Hot Potato AntiPattern

AntiPattern problem:

- With guaranteed delivery, the messaging server has to re-deliver messages until it receives acknowledgement:
  - With CMT, the acknowledgement is tied to the transaction outcome
  - With BMT, the acknowledgment is not tied to the transaction outcome
- If the message is not processed due to an ongoing condition, the messaging server will continuously resend the message

Hot Potato AntiPattern

Symptoms and consequences:

- Message driven beans or other JMS receivers continually receive the same invalid messages for processing
- The messaging server and the receivers are burdened from repeatedly delivering and receiving messages that can not be processed

Hot Potato AntiPattern

Refactored solution:

- Message that cannot be processed by another bean, should probably be acknowledged, and the problem should be announced, perhaps through email or through another queue of problem messages
- It can be necessary to use BMT so you can rollback the transaction and still acknowledge the message
- Some messaging servers have Dead Message Queues where the message is sent after a certain number of attempts

Describing Business Tier AntiPatterns

- This group of AntiPatterns involves poor usage of the J2EE platform business tier technologies
- These AntiPatterns work hand in hand with the business tier J2EE patterns to avoid poor designs and find good designs for the business tier

Sledgehammer for a Fly AntiPattern

AntiPattern problem:

Like a sledgehammer, EJB components are very powerful tools—too powerful for some problems:

- EJB component services can lead to increased overhead, increased complexity, increased development time, and increased maintenance
- If you need many of the services, then the costs are greatly outweighed by the savings
- If you do not need these services, then you are paying a price for services that you are not using

Sledgehammer for a Fly AntiPattern

Symptoms and consequences:

- Increased development complexity and decreased developer efficiency
- More lines of code and more classes and interfaces than are required for the job
- Slower performance

Sledgehammer for a Fly AntiPattern

Refactored solution:

- Consider whether EJB components are going to benefit your situation
- *Bitter EJB* has a simple logging EJB component example:
  - The interfaces, classes, and deployment descriptors for the session and entity bean include ten files and 300 lines of code
  - For logging functionality, all the EJB technology services such as, scalability, distributed transaction, state management, persistence, and security offer little value

Local and Remote Interfaces Simultaneously AntiPattern

AntiPattern problem:

- The local interface methods must declare that they might throw `RemoteException` objects
- The code can have unintended side effects, depending on whether it was invoked locally or remotely
- Remote interfaces should be coarse grained
- The choice between local and remote should be carefully made and enforced

Local and Remote Interfaces Simultaneously AntiPattern

Symptoms and consequences:

- Poor performance due to inadequate consideration of local versus remote invocation issues
- Unexpected side affects because code is not properly designed for local versus remote invocation

Refactored solution:

- Carefully choose remote boundaries
- Use local or remote interfaces to implement your decisions

Accesses Entities Directly AntiPattern

AntiPattern problem:

If presentation tier components directly access entity beans:

- You can have too many remote invocations to the entity beans
- Presentation tier components are responsible for finding or creating the beans, coordinating the method invocations, and ensuring proper transactional control

Accesses Entities Directly AntiPattern

Symptoms and consequences:

- Performance suffers due to too many remote method invocations
- Transactions barriers might not be handled correctly
- Presentation tier developers must have much more intimate knowledge of the business tier implementation

Refactored solution:

Uses session beans to separate the presentation tier components from the business entity components

Mirage AntiPattern

AntiPattern problem:

- Many developers use BMP due to server's based on EJB 1.*x* architecture (EJB server's) CMP limitations
- CMP offers many benefits over BMP:
  - CMP can provide data caching, lazy loading, and other server optimizations
  - CMP code does not contain vendor specific SQL
  - CMP is easier to develop and maintain
  - CMP 2.0 provides container managed relationships (CMR)

Mirage AntiPattern

Symptoms and consequences:

Using BMP unnecessarily might lead to:

- Needlessly high development and maintenance effort
- Less efficient entity bean performance

Refactored solution:

- Consider using CMP instead of BMP
- There are times when BMP is the only option
- Check into the reputation of your servers CMP implementation, or do your own performance tests

Cacheless Cow AntiPattern

AntiPattern problem:

Considerable server capacity is wasted if you never cache the results of previous requests and reuse them when the same or different client makes the same request

Symptoms and consequences:

The system does not scale as well as it could because it wastes time recalculating responses that have already been acquired

Cacheless Cow AntiPattern

Refactored solution:

- Places where cache might be added:
  - In front of the Web server
  - In front of the presentation tier components
  - In front of the business tier components
  - In the integration tier
- Might use vendor provided hardware or software caching
- Might use home grown caching solutions like caching commands in the Command pattern or caching in the Value List Handler pattern

Cacheless Cow AntiPattern

Refactored solution:

There are many issues to consider when developing a cache:

- Ensure the cached results are sufficient and that there is no need to rerun the process
- Decide what data should be cached and for how long
- Consider when the cached data might become out of date
- Consider how to allow fast concurrent reads while protecting the data when it is being modified

Conversational Baggage AntiPattern

AntiPattern problem:

This pattern is documented in *Bitter EJB* by Tate, Clark, Lee, and Linskey:

- While many tasks in enterprise applications require keeping session state, many other tasks do not
- Session state adds substantial overhead to an application and can significantly decrease scalability
- If you are not focused on this issue, you are likely to create stateful tasks that do not require session state

Conversational Baggage AntiPattern

Symptoms and consequences:

Performance and scalability degrades

Refactored solution:

Attempt to make tasks stateless whenever you reasonably can

Golden Hammers of Session State AntiPattern

AntiPattern problem:

- Four common places to store session state:
  - Client tier, perhaps in cookies
  - Presentation tier in a `HttpSession` class
  - Business tier in stateful session beans
  - Resource tier in a database
- There is no golden hammer for session state management; the situation should dictate the choice

Golden Hammers of Session State AntiPattern

Symptoms and consequences:

Using an inappropriate session management tool can lead to poor performance and scalability

Refactored solution:

When making a session state tool choice, consider:

- The location of the data to be used
- The size and type of the data
- The implications of losing the data
- Types of clients

Describing Presentation Tier AntiPatterns

- This group of AntiPatterns involve poor usage of the J2EE platform presentation tier technologies
- These AntiPatterns work with the Presentation Tier J2EE patterns to avoid poor designs and find goo designs for the presentation tier

Including Common Functionality in Every Servlet AntiPattern

AntiPattern problem:

- Pre-processing and post-processing tasks are implemented or invoked in the service methods of each servlet
- An AntiPattern only with Java Servlet 2.3 and later containers

Symptoms and consequences:

- Tasks must be programmatically added and removed from multiple servlets
- The servlets have poor cohesion

Including Common Functionality in Every Servlet AntiPattern

Refactored solution:

- Pre-processing and post-processing tasks are placed in filters
- These filters can be applied to JSP pages as well as servlets
- The Intercepting Filter pattern is the refactored solution for this AntiPattern

Embedded Navigational Information AntiPattern

AntiPattern problem:

- JSP pages usually include URLs for hyperlinks or buttons and for fragment files that are included in the JSP page
- The JSP pages must be modified if the file names or URLs change

Symptoms and consequences:

- Users receive page not found errors
- The application flow is difficult to modify

Embedded Navigational Information AntiPattern

Refactored solution:

- With Front Controller pattern and logical resource mapping, the URLs are unlikely to frequently change.
- You can use custom tags to include fragments instead of the `jsp:include` action or `<%@include>` directive. An include custom tag can take a generic attribute value such as `banner` and look in a configuration file to see what specific resource that the string maps to

Ad Lib TagLibs AntiPattern

AntiPattern problem:

Model or controller functionality might be moved from JSP pages into custom tags

Symptoms and consequences:

- If business logic is placed in custom tags, it will not be easily used by rich clients
- Custom tags that are doing model or controller functionality are likely to need many attributes
- Custom tags are slightly more complicated to create, debug, and maintain than ordinary Java technology classes

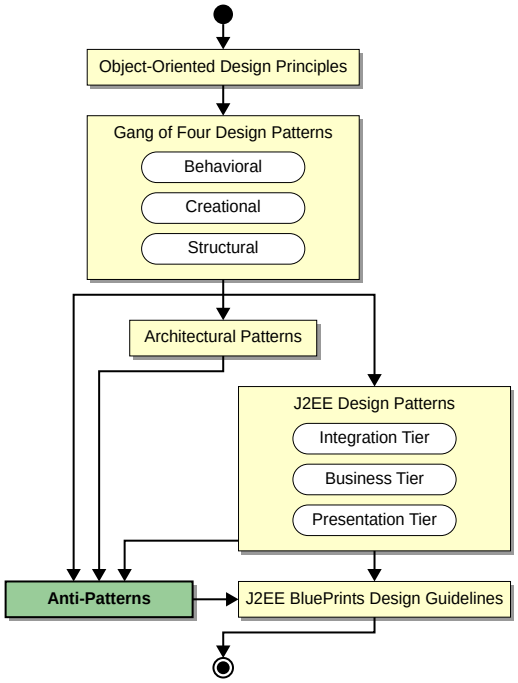
Ad Lib TagLibs AntiPattern

Refactored solution:

> When you are writing custom tags, ask yourself whether this logic should really be invoked by the JSP page

Summary

- AntiPatterns help us focus on the common mistakes made in development and provide refactored solutions to avoid or remedy these problems
- Studying patterns and AntiPatterns provide a natural benefit of seeing development issues from both a positive and a negative perspective



Module 13

Applying J2EE BluePrints Design Guidelines

Objectives

- Describe the J2EE BluePrints design guidelines
- Describe the Java Pet Store demo software
- Describe the J2EE patterns used in the Java Pet Store demo software

Describing the J2EE BluePrints Design Guidelines

- Provides enterprise application developers with the concepts, technologies, and design principles needed to quickly make secure, robust, scalable, and extensible applications
- Includes guidance and practical examples to show you how to make the best use of the architecture and features of the J2EE platform

Object-Oriented Guidelines

Many of the J2EE BluePrints design guidelines are specializations of general object-oriented guidelines. Modules should:

- Be separately compilable so they can be developed independently
- Have few interfaces to communicate with as few other modules as possible
- Use loose coupling to other modules
- Have explicit interfaces so that whenever two modules communicate, it is obvious from public declarations

Client Tier Guidelines

The J2EE Client tier can be implemented as a:

- Browser client
- Java technology-based Application client
- Java technology-based MIDlet client
- Non-Java technology-based client

Web Tier Guidelines

The J2EE BluePrints design guidelines recommends using an existing Web framework, because:

- Most Web frameworks encourage the separation of presentation, control, and business logic
- Most Web frameworks provide centralized control and customization of templating, localization, data validation, access control, and logging
- Time can be saved by not building the framework structure from scratch
- A framework should be stable, thoroughly tested, and have a community support structure

Business Tier Guidelines

- Entity beans may be appropriate for persistent data when:
  - Multiple clients need to access the data
  - The data is not tied to a user session
  - Data must be available in a portable manner
  - The data must be handled properly inside of transactions
- Use stateful session beans to keep most conversational state, instead of in the client or Web tier
- Use stateless session beans for providing services without maintenance of client state

Business Tier Guidelines

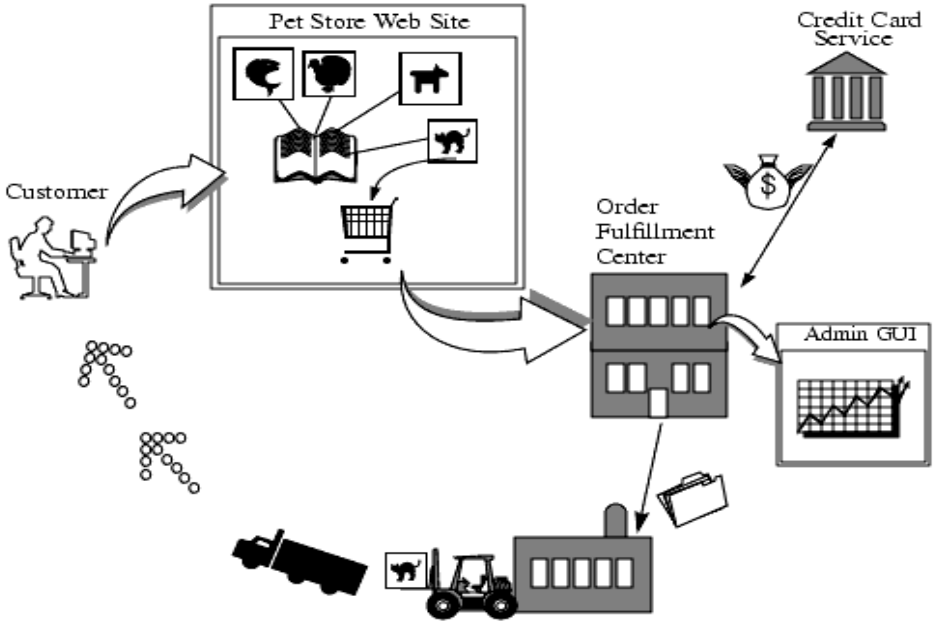
- Use message driven beans for asynchronous services or to loosely couple EJB components with other systems
- Keep EJB technology code agnostic of the type of client
- Keep the business logic in the business tier
- Use session beans as a façade to entity beans
- Avoid using fine grained remote entity objects
- Use Data Access Objects (DAO), if container managed persistence (CMP) entity beans are not appropriate

Integration Tier Guidelines

- Uses access objects to access Enterprise Information Systems (EIS) functions and data
- Use command beans that invoke EIS functions, and data access objects to access EIS data
- Access objects guidelines:
  - Do not make assumptions about the deployment environment
  - Ensure they are usable by different component types
  - Use transaction and security management
  - Follow the programming restrictions of the components that might use them
  - Use connection pooling

Describing the Java Pet Store Demo Software

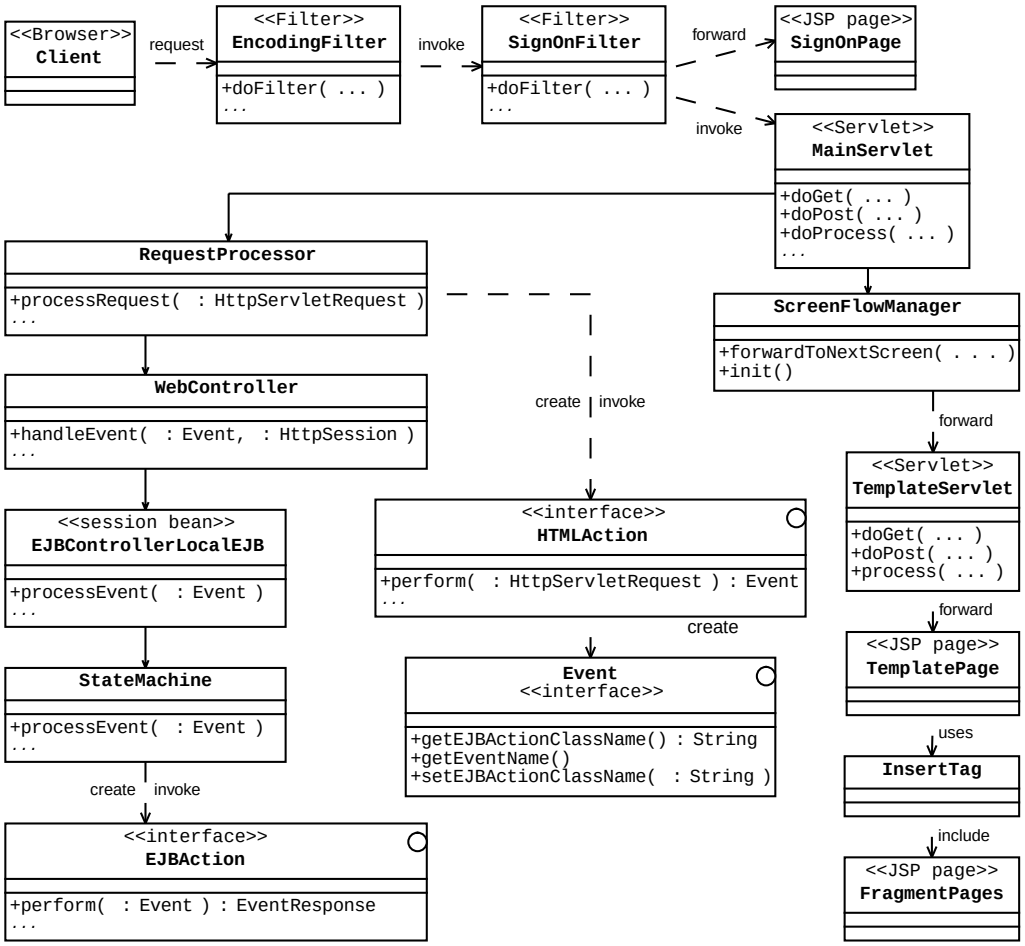
The Java Pet Store demo software illustrates good design and construction while demonstrating the reasoning behind the design choices



Analyzing J2EE Patterns in the Java Pet Store Demo Software Web Application Framework

- The Java Pet Store demo software customer Web site uses the Web Application Framework (WAF)
- This framework is a Model 2 Architecture framework, which is somewhat similar to Struts, but also quite different in many regards

Java Pet Store Demo Software Web Application Framework



J2EE Patterns in the Web Application Framework

The Java Pet Store demo software Web Application framework implements the following J2EE Patterns:

- **Front Controller pattern** – The `MainServlet` class
- **Intercepting Filter pattern** – The `EncodingFilter` and `SignOnFilter` classes
- **View Helper pattern** – The JSP pages use custom tags and JavaBeans components to reduce scriptlet usage
- **Service to Worker pattern** – The view components invoke the `RequestProcesses` class
- **Composite View Pattern** – The views include fragment views

The `SignOnFilter` Class Mapping in the `web.xml` File

```
1   <filter>
2     <filter-name>SignOnFilter</filter-name>
3     ...
4     <filter-class>
5       com.sun.j2ee.blueprints.signon.web.SignOnFilter
6     </filter-class>
7   </filter>
8   <filter-mapping>
9     <filter-name>SignOnFilter</filter-name>
10    <url-pattern>/*</url-pattern>
11  </filter-mapping>
```

The SignOnFilter Class

```
1   package com.sun.j2ee.blueprints.signon.web;
2
3   import javax.servlet.*;
4   ...
5   public class SignOnFilter implements Filter {
6
7     private HashMap protectedResources;
8     private FilterConfig config = null;
9     ...
10    public void init(FilterConfig config)
11        throws ServletException {
12      this.config = config;
13      URL protectedResourcesURL=
14          config.getServletContext().getResource
15          ("/WEB-INF/signon-config.xml");
16      SignOnDAO dao =
17          new SignOnDAO(protectedResourcesURL);
18      ...
19      protectedResources =
20          dao.getProtectedResources();
21    }
22
23    public void doFilter( ServletRequest request,
24      ServletResponse response, FilterChain chain)
25      throws IOException, ServletException {
26      HttpServletRequest hreq =
27        (HttpServletRequest) request;
28      String targetURL = hreq.getRequestURI();
```

The SignOnFilter Class

```
29      ...
30      // check if the user is signed on
31      boolean signedOn = false;
32      if (hreq.getSession().getAttribute
33          (SIGNED_ON_USER)!= null {
34        signedOn = ((Boolean) hreq.getSession().
35
getAttribute(SIGNED_ON_USER)).booleanValue();
36      } else {
37        hreq.getSession().setAttribute
38        (SIGNED_ON_USER, new Boolean(false));
39      }
40
41  // jump to the resource if signed on
42      if (signedOn) {
43        chain.doFilter(request, response);
44        return;
45      }
46
47     // find out if the patterns match the target URL
48      Iterator it =
49          protectedResources.keySet().iterator();
50      while (it.hasNext()) {
51        String protectedName = (String) it.next();
52        ProtectedResource resource =
53          (ProtectedResource)
54          protectedResources.get(protectedName);
55       String urlPattern = resource.getURLPattern();
56
```

The SignOnFilter Class

```
57  // now check against the targetURL
58  if (urlPattern.equals(targetURL)) {
59  // put the orginal url in the session
60    hreq.getSession().setAttribute
61          (ORIGINAL_URL, targetURL);
62          config.getServletContext()
63          .getRequestDispatcher("/" + signOnPage)
64          .forward(request, response);
65        return;
66        }
67      }
68      // No matches if we made it to here
69      chain.doFilter(request, response);
70    }
71    ...
72  }
```

Front Controller Servlet Mapping in the `web.xml` File

```
1   <servlet>
2     <servlet-name>MainServlet</servlet-name>
3     <description>The Front Controller for the Pet Store
4     </description>
5     <servlet-class>
6       com.sun.j2ee.blueprints.waf.controller.web.MainServlet
7     </servlet-class>
8   ...
9   </servlet>
10  ...
11  <servlet-mapping>
12    <servlet-name>MainServlet</servlet-name>
13    <url-pattern>*.do</url-pattern>
14  </servlet-mapping>
```

The `mapping.xml` Class Mapping

```
1   <url-mapping url="order.do" screen="order_complete.screen">
2     <web-action-class>
3       com.sun.j2ee.blueprints.....OrderHTMLAction
4     </web-action-class>
5   </url-mapping>
6
7   <url-mapping url="createcustomer.do" useFlowHandler="true" >
8     <web-action-class>
9       com.sun.j2ee.....CustomerHTMLAction
10    </web-action-class>
11    <flow-handler
12      class="com.sun.j2ee......CreateUserFlowHandler"/>
13  </url-mapping>
```

The MainServlet Class

```
1   package
1       com.sun.j2ee.blueprints.waf.controller.web;
2   ...
3   public class MainServlet extends HttpServlet {
4     ...
5     private HashMap urlMappings;
6     private HashMap eventMappings;
7
8     public void init(ServletConfig config)
9       throws ServletException {
10      ...
11      urlMappings =
12      URLMappingsXmlDAO.loadRequestMappings
13        (requestMappingsURL);
14      context.setAttribute
15        (WebKeys.URL_MAPPINGS, urlMappings);
16      eventMappings = URLMappingsXmlDAO.
17        loadEventMappings(requestMappingsURL);
18      context.setAttribute
19          (WebKeys.EVENT_MAPPINGS, eventMappings);
20      ...
21    }
22    public  void doGet(...) ... {
23      doProcess(request, response);
24    }
25    public  void doPost(...) ... {
26      doProcess(request, response);
27    }
```

The `MainServlet` Class

```
28
29    private void doProcess
30          (HttpServletRequest request,
31          HttpServletResponse response)
32          throws IOException, ServletException {
33      ...
34      getRequestProcessor().
35          processRequest(request);
36      getScreenFlowManager().
37        forwardToNextScreen(request,response);
38      ...
39  }
40    ...
41  }
```

The RequestProcessor Class processRequest Method

```
1   String actionClassString = //appropriate
2       //<web-action-class> for the request
3       //URL that came in
4   HTMLAction action = (HTMLAction)
5       getClass().getClassLoader().
        loadClass(actionClassString).newInstance();
6   ...
7   action.doStart(request);
8   Event ev = action.perform(request);
9   EventResponse eventResponse = null;
10  if (ev != null) {
11    // set the ejb action class name on the event
12    EventMapping eventMapping= getEventMapping(ev);
13    if (eventMapping != null) {
14      ev.setEJBActionClassName
15        (eventMapping.getEJBActionClassName());
16    }
17    ComponentManager sl = (ComponentManager)
18    request.getSession().
19        getAttribute(WebKeys.COMPONENT_MANAGER);
20    WebController wcc = sl.getWebController
21        (request.getSession());
22    eventResponse =
23      wcc.handleEvent(ev, request.getSession());
24  }
25  action.doEnd(request, eventResponse);
```

The DefaultWebController Class

```
1   package com.sun.j2ee.blueprints.waf.controller.web;
2   ...
3   public class DefaultWebController implements WebController {
4     ...
5
6     public synchronized EventResponse handleEvent
7       (Event ev, HttpSession session)
8       throws EventException {
9       DefaultComponentManager cm = (DefaultComponentManager)
10        session.getAttribute(WebKeys.COMPONENT_MANAGER);
11      EJBControllerLocal controllerEJB
12        =cm.getEJBController(session);
13      return controllerEJB.processEvent(ev);
14    }
15    ...
16  }
```

The EJBControllerLocalEJB Class

```
1   package
2       com.sun.j2ee.blueprints.waf.controller.ejb;
3   ...
4   public class EJBControllerLocalEJB
5         implements SessionBean {
6
7     protected StateMachine sm;
8     protected SessionContext sc;
9
10    public void ejbCreate() {
11      sm = new StateMachine(this, sc);
12    }
13
14    public EventResponse processEvent(Event ev)
15        throws EventException {
16      return (sm.processEvent(ev));
17    }
18    public void setSessionContext
19        (SessionContext sc) {
20      this.sc = sc;
21    }
22    public void ejbRemove() {
23      sm = null;
24    }
25    public void ejbActivate() {}
26    public void ejbPassivate() {}
27  }
```

The StateMachine Class

```
1   package
2       com.sun.j2ee.blueprints.waf.controller.ejb;
3   ...
4   public class StateMachine
5         implements java.io.Serializable {
6     ...
7     public EventResponse processEvent(Event ev)
            throws EventException {
8       String actionName =
9           ev.getEJBActionClassName();
10      EventResponse response = null;
11      ...
12      action = (EJBAction)
13          Class.forName(actionName).newInstance();
14      ...
15      action.init(this);
16      action.doStart();
17      response = action.perform(ev);
18      action.doEnd();
19      ...
20      return response;
21    }
22  }
```

The ScreenFlowManager Class

```
1   package
2   com.sun.j2ee.blueprints.waf.controller.web.flow;
3   ...
4   public class ScreenFlowManager
5         implements java.io.Serializable {
6     ...
7     public void forwardToNextScreen
8       (HttpServletRequest request,
9           HttpServletResponse response)
10          throws java.io.IOException,
11          FlowHandlerException,
12          javax.servlet.ServletException {
13      String selectedURL = request.getRequestURI();
14      ...
15      URLMapping urlMapping =
16          getURLMapping(selectedURL);
17      if (!urlMapping.useFlowHandler()) {
18        currentScreen = urlMapping.getScreen();
19      } else {
20        FlowHandler handler = null;
21        String flowHandlerString =
22          urlMapping.getFlowHandler();
23        handler = (FlowHandler) getClass().
24            getClassLoader().
25            loadClass(flowHandlerString).
26            newInstance();
27        handler.doStart(request);
```

The ScreenFlowManager Class

```
28          String flowResult =
29            handler.processFlow(request);
30          handler.doEnd(request);
31          currentScreen =
32            urlMapping.getResultScreen(flowResult);
33          if (currentScreen == null)
34            currentScreen = flowResult;
35          ...
36        }
37        ...
38        context.getRequestDispatcher("/"
39          +currentScreen).forward(request, response);
40      }
41    }
```

The screendefinitions_en_US.xml File

```
1   <screen-definitions>
2     <default-template>/template.jsp
3     </default-template>
4     <screen name="main">
5       <parameter key="title"
6         value="Welcome to the BluePrints Petstore"
7         direct="true"/>
8       <parameter key="banner" value="/banner.jsp" />
9       <parameter key="sidebar"
10                value="/sidebar.jsp" />
11      <parameter key="body" value="/main.jsp" />
12      <parameter key="mylist" value="/mylist.jsp" />
13      <parameter key="advicebanner"
14          value="/advice_banner.jsp" />
15      <parameter key="footer" value="/footer.jsp" />
16     </screen>
17    <screen name="cart">
18      <parameter key="title" value="Cart"
19                direct="true"/>
20      <parameter key="banner" value="/banner.jsp" />
21      <parameter key="sidebar"
22                value="/sidebar.jsp" />
23      <parameter key="body" value="/cart.jsp" />
24      <parameter key="mylist" value="/mylist.jsp" />
25      <parameter key="footer" value="/footer.jsp" />
26      <parameter key="advicebanner"
27                value="/advice_banner.jsp" />
28     </screen>
```

The TemplateServlet Class

```
1   package
2     com.sun.j2ee.blueprints.waf.view.template;
3   ...
4   public class TemplateServlet extends HttpServlet {
5     ...
6     public void process (HttpServletRequest request,
7                HttpServletResponse  response)
8                throws IOException,ServletException {
9       String screenName = null;
10      String selectedUrl = request.getRequestURI();
11      int lastPathSeparator =
12          selectedUrl.lastIndexOf("/") + 1;
13      int lastDot = selectedUrl.lastIndexOf(".");
14      if (lastPathSeparator != -1 && lastDot != -1
15          && lastDot > lastPathSeparator) {
16        screenName = selectedUrl.substring
17            (lastPathSeparator, lastDot);
18      }
19      //code omitted to cache request parameters
        //and request attributes
20      ...
21      String templateName = localeScreens.
22                getTemplate(screenName);
23      if (templateName != null) {
24      ...
25      context.getRequestDispatcher(templateName).
26        forward(request, response);
27  ...
```

The Template.jsp Page

```
1   <%@ taglib uri="/WEB-INF/template.tld"
2              prefix="template" %>
3   <%@ page contentType="text/html;charset=UTF-8" %>
4   <html>
5   <head>
6   ...
7   <body bgcolor="#FFFFFF">
8   <table width="100%" height="100%" border="0"
9         cellpadding="5" cellspacing="0">
10    <tr height="100" valign="top">
11      <td colspan="3">
12        <template:insert parameter="banner" />
13      </td>
14    </tr>
15    <tr valign="top">
16      <td width="20%" valign="top">
17          <template:insert parameter="sidebar" />
18      </td>
19      <td width="60%" valign="top">
20          <template:insert parameter="body" />
21      </td>
22      <td valign="top">
23          <template:insert parameter="mylist" />
24      </td>
25    </tr>
26  ...
```

The InsertTag Custom Tag

```
1   public class InsertTag extends TagSupport {
2
3     private boolean directInclude = false;
4     private String parameter = null;
5     private Parameter parameterRef = null;
6     private Screen screen = null;
7     public void setParameter(String parameter){
8       this.parameter = parameter;
9     }
10    public int doStartTag() throws JspTagException {
11      ...
12      screen = (Screen)pageContext.getRequest().
13        getAttribute(WebKeys.CURRENT_SCREEN);
14      if ((screen != null) && (parameter != null))
15        parameterRef = (Parameter)
16          screen.getParameter(parameter);
17      ...
18      if (parameterRef != null)
19        directInclude = parameterRef.isDirect();
20      return SKIP_BODY;
21    }
```

The InsertTag Custom Tag

```
22    public int doEndTag() throws JspTagException {
23      if (directInclude && parameterRef != null) {
24        pageContext.getOut().print(parameterRef.getValue());
25      } else if (parameterRef != null)  {
26      if (parameterRef.getValue() != null)
27        pageContext.getRequest().getRequestDispatcher
28          (parameterRef.getValue()).include
29          (pageContext.getRequest(),
30           pageContext.getResponse());
31      }
32      return EVAL_PAGE;
33    }
34  }
```

Analyzing the J2EE Patterns in the Java Pet Store Demo Software Administrative Classes

Design Pattern Implementation in the Administrative Client Functionality

- **Business Delegate pattern** – The `AdminRequestBD` class is a business delegate
- **Service Locator pattern** – The `ServiceLocator` class locates various resources such as EJB technology references
- **Session Façade pattern** – The `OPCAdminFacadeEJB` class is a session façade that provides a unified interface to several session and entity beans
- **Transfer Object pattern** – The `OrderTO` class is a transfer object that returns order data back to the client tier

The AdminRequestBD Class

```
1   package com.sun.j2ee.blueprints.admin.web;
2   ...
3   public class AdminRequestBD {
4   private String OPC_ADMIN_NAME =
5         "java:comp/env/ejb/remote/OPCAdminFacade";
6     private OPCAdminFacade opcAdminEJB = null;
7     public AdminRequestBD()
8         throws AdminBDException{
9       try {
10        OPCAdminFacadeHome home =
11          (OPCAdminFacadeHome)
12          ServiceLocator.getInstance().
13          getRemoteHome(OPC_ADMIN_NAME,
14          OPCAdminFacadeHome.class);
15        opcAdminEJB = home.create();
16      } catch (CreateException ce) {
17        ce.printStackTrace();
18       throw new AdminBDException(ce.getMessage());
19      }
20  ...
21    }
22    public OrdersTO getOrdersByStatus(String status)
23          throws AdminBDException {
24      try {
25        return
26            opcAdminEJB.getOrdersByStatus(status);
27      } catch (RemoteException re) {
28     re.printStackTrace();
29     throw new AdminBDException(re.getMessage());}}}
```

The ServiceLocator Class

```
1   package
2       com.sun.j2ee.blueprints.servicelocator.web;
3   ...
4   public class ServiceLocator {
5     private InitialContext ic;
6     private Map cache; //holds references to EJBHomes
7     private static ServiceLocator me;
8
9     static {
10        me = new ServiceLocator();
11        ...
12    }
13    private ServiceLocator()
14        throws ServiceLocatorException  {
15      try {
16        ic = new InitialContext();
17        cache = Collections.synchronizedMap
18                    (new HashMap());
19      } catch (NamingException ne) {
20            throw new ServiceLocatorException(ne);
21      } ...
22    }
23    static public ServiceLocator getInstance() {
24      return me;
25    }
26    public EJBLocalHome getLocalHome
27          (String jndiHomeName)
28          throws ServiceLocatorException {
```

The ServiceLocator Class

```
29      EJBLocalHome home = null;
30       try {
31         if (cache.containsKey(jndiHomeName)) {
32           home = (EJBLocalHome)
33               cache.get(jndiHomeName);
34         } else {
35             home = (EJBLocalHome)
36                 ic.lookup(jndiHomeName);
37             cache.put(jndiHomeName, home);
38         }
39       } catch (NamingException ne) {
40           throw new ServiceLocatorException(ne);
41       } catch (Exception e) {
42           throw new ServiceLocatorException(e);
43       }
44       return home;
45     }
46     public EJBHome getRemoteHome(...) ... {...}
47     public QueueConnectionFactory
                getQueueConnectionFactory(...) ...{...}
48     public Queue getQueue(...) ... {...}
49     public TopicConnectionFactory
                getTopicConnectionFactory(...) ...{...}
50     public  Topic getTopic(...) ... {...}
51     public DataSource getDataSource(...) ... {...}
52     public URL getUrl(...) ... {...}
53     public boolean getBoolean(...) ... {...}
54     public String getString(...) ... {...}
55   }
```

The MutableOrderTO Class

```
1   package com.sun.j2ee.blueprints.opc.admin.ejb;
2   ...
3   public interface OrdersTO extends Serializable {
4     public Iterator iterator();
5     public int size();
6     public boolean contains(Object o);
7     public boolean containsAll(Collection c);
8     public boolean equals(Object o);
9     public int hashCode();
10    public boolean isEmpty();
11    public Object[] toArray();
12    public Object[] toArray(Object[] a);
13
14    static class MutableOrdersTO
15         extends ArrayList implements OrdersTO {
16    }
17  }
```

The OPCAdminFacadeEJB Class

```
1   package com.sun.j2ee.blueprints.opc.admin.ejb;
2   ...
3   public class OPCAdminFacadeEJB
4       implements SessionBean {
5   private String PURCHASE_ORDER_EJB =
6       "java:comp/env/ejb/local/PurchaseOrder";
7     private String PROCMGR_ORDER_EJB =
8       "java:comp/env/ejb/local/ProcessManager";
9     private SessionContext sc;
10    private PurchaseOrderLocalHome poLocalHome;
11   private ProcessManagerLocal processManagerLocal;
12
13    public void ejbCreate() throws CreateException {
14      try {
15        ServiceLocator serviceLocator =
16            new ServiceLocator();
17        poLocalHome = (PurchaseOrderLocalHome)
18            serviceLocator.getLocalHome
19            (PURCHASE_ORDER_EJB);
20        ProcessManagerLocalHome
21            processManagerLocalHome =
22            (ProcessManagerLocalHome)
23            serviceLocator.getLocalHome
24            (PROCMGR_ORDER_EJB);
25        processManagerLocal =
26            processManagerLocalHome.create();
27      } catch(ServiceLocatorException se) {
28         throw new EJBException(se);
```

The OPCAdminFacadeEJB Class

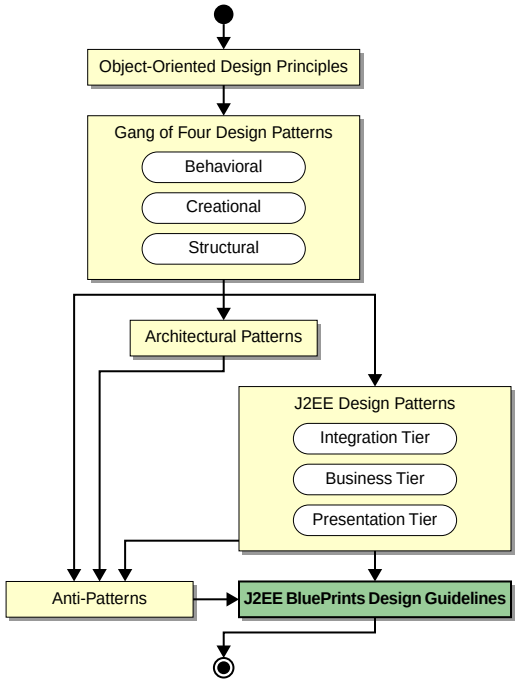
```
29  } catch (CreateException ce) {
30          throw new EJBException(ce);
31      }
32    }
33  ...
34  public OrdersTO getOrdersByStatus(String status)
35           throws OPCAdminFacadeException {
36      OrdersTO.MutableOrdersTO retVal =
37      new OrdersTO.MutableOrdersTO();
38      PurchaseOrderLocal po;
39     ProcessManagerLocal mgr = processManagerLocal;
40      try {
41       PurchaseOrderLocalHome pohome = poLocalHome;
42        Collection orders =
43            mgr.getOrdersByStatus(status);
44        Iterator it = orders.iterator();
45        while((it!= null) && (it.hasNext())) {
46          ManagerLocal mgrEjb =
47            (ManagerLocal) it.next();
48          po = pohome.
49            findByPrimaryKey(mgrEjb.getOrderId());
50          Date gotDate = new Date(po.getPoDate());
51          String podate = (gotDate.getMonth()+1) +
52              "/" + gotDate.getDate() + "/" +
53              (gotDate.getYear()+1900);
54          retVal.add(new OrderDetails(po.getPoId(),
55          po.getPoUserId(), podate,
56          po.getPoValue(), status));
```

The OPCAdminFacadeEJB Class

```
57      }
58  } catch(FinderException fe) {
59        System.err.println
60          ("finder Ex while getOrdByStat :"
61          + fe.getMessage());
62        throw new OPCAdminFacadeException("Unable to find "+
63        "PurchaseOrders of given status : " +
64        fe.getMessage());
65      }
66      return(retVal);
67    }
68    ...
69  }
```

Summary

- The J2EE BluePrints design guidelines provide best practices for designing enterprise systems using J2EE platform technology
- The Java Pet Store demo software is a sample Web application that illustrates these best practices
- By studying the Java Pet Store demo software , you can see an example of how the J2EE patterns can be applied to create a large enterprise application



Appendix A

Quick Reference for UML

There are no overheads associated with this appendix.

Appendix B

J2EE Technology Review

Objectives

- Describe the major technologies of the J2EE platform
- Describe Java Remote Method Invocation (Java RMI)
- Describe Java IDL API
- Describe the JAX-RPC API
- Describe the Java Naming and Directory Interface API
- Describe the Java Servlet API
- Describe JSP technology
- Describe the Java Message Service API
- Describe EJB technology
- Describe JDBC technology
- Describe the J2EE Connector Architecture

Reviewing the J2EE Platform

The J2EE platform is a development platform and framework for building component based enterprise Java technology applications

J2EE includes:

- Specification
- The reference implementation server
- J2EE BluePrints design guidelines
- The Java Pet Store Demo application

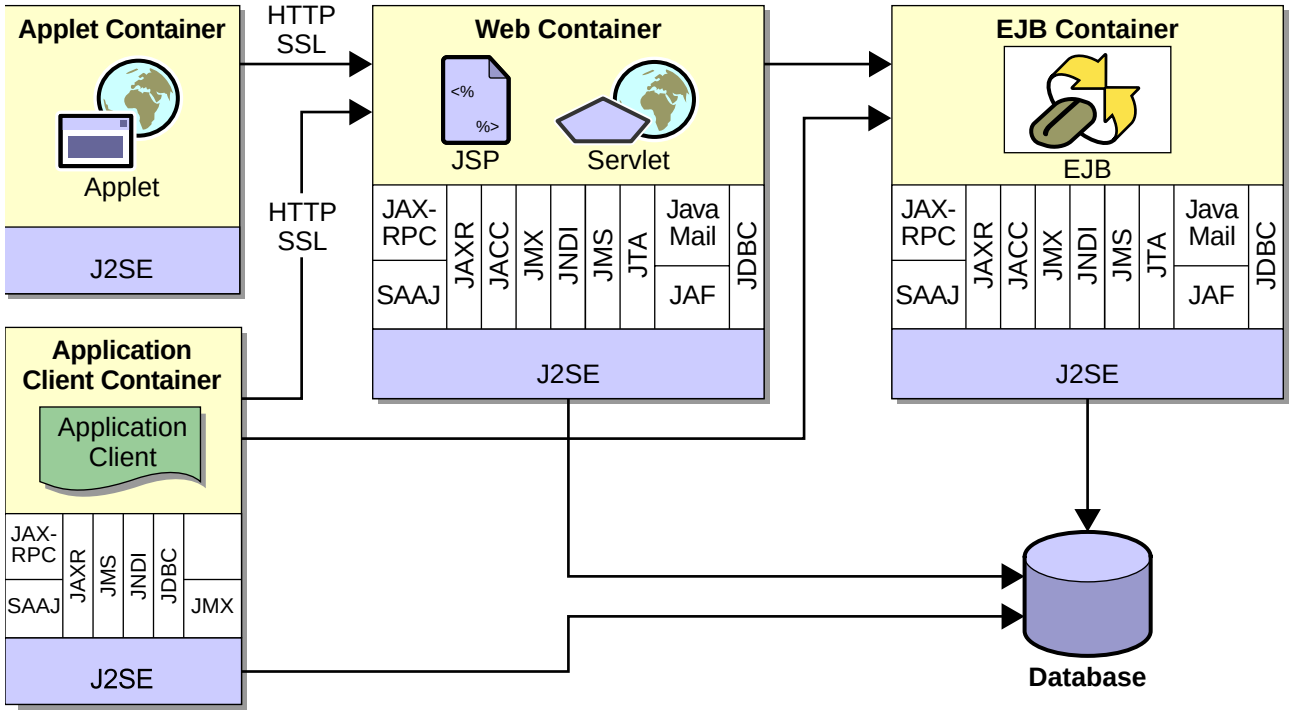
Reviewing the J2EE Platform

- Java Remote Method Invocation (RMI)
- Java IDL API
- Java Naming and Directory Interface (JNDI) API
- Java Servlet API
- JavaServer Pages (JSP pages)
- Enterprise JavaBeans (EJB) components
- Java API for XML Parsing (JAXP) software
- Java API for XML-based RPC (JAX-RPC)
- SOAP with Attachments API for Java (SAAJ)
- Java API for XML Registries (JAXR) software

Reviewing the J2EE Platform

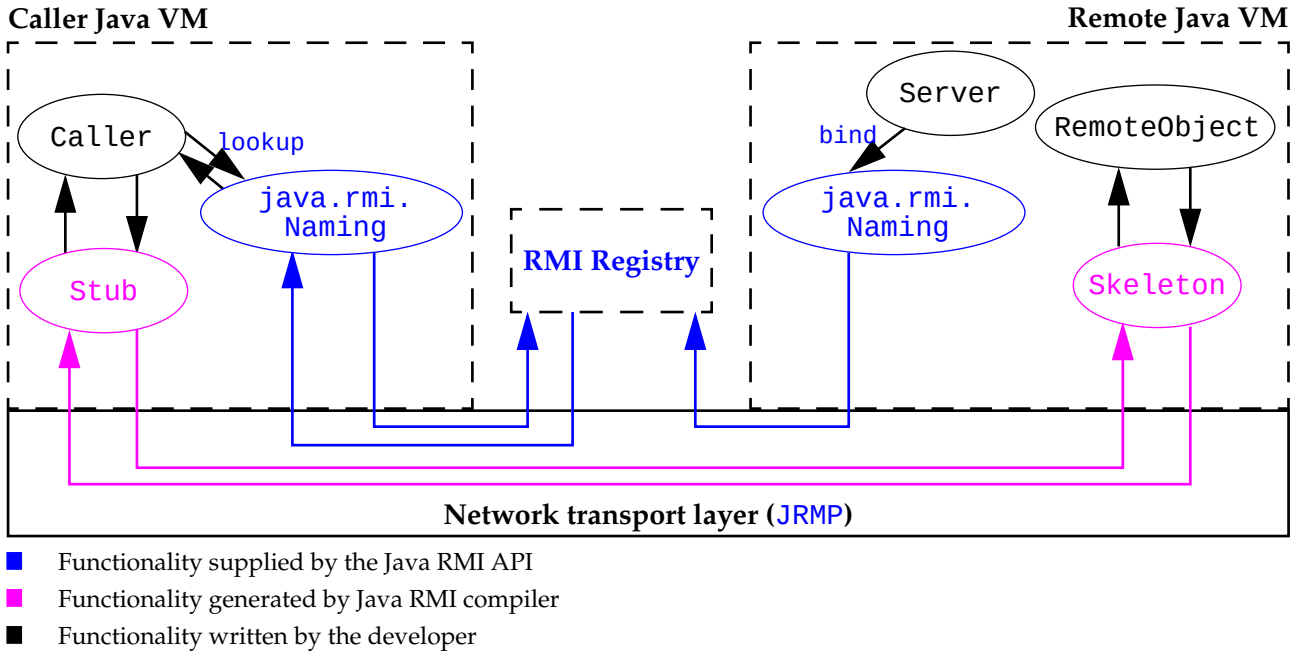
- JDBC API
- Java Message Service API (JMS)
- JavaMail™ API
- Java Transaction API (JTA)
- Java Authentication and Authorization Service (JAAS)
- J2EE connector architecture
- J2EE management model

Multi-Tier J2EE Architecture Basics



Understanding Java Remote Method Invocation

- Light-weight distributed communication
- Uses stubs and skeletons
- The `java.rmi` package contains the API classes



Understanding Java Remote Method Invocation

These are the basic steps to create and run a Java RMI application:

1. Define a remote interface:

```
1   import java.rmi.*;
2   public interface Borrowable extends java.rmi.Remote {
3     public boolean checkOut(String patron) throws RemoteException,
4         PatronException;
5     public boolean checkIn() throws RemoteException;
6     public boolean catalog() throws RemoteException;
7   }
```

Understanding Java Remote Method Invocation

2. Define a remote class:

```
1   import java.rmi.*;
2   import java.rmi.server.*;
3
4   public class Book extends UnicastRemoteObject
5       implements Borrowable {
6     String title;
7     String author;
8     String publisher;
9     String callNo;
10
11    public Book(String t, String a, String p,
12                String c) throws RemoteException {
13      title = t;
14      author = a;
15      publisher = p;
16      callNo = c;
```

Understanding Java Remote Method Invocation

```
17  }
18
19    public boolean checkOut(String patron) throws PatronException {
20       boolean result = false;
21       if (checkOut == true)
22        result = true;
23       return result;
24    }
25    public boolean checkIn() {
26       boolean result = false;
27      if (checkIn == true)
28        result = true;
29      return result;
30    }
31    public boolean catalog() {
32      boolean result = false;
33      if (catalogued == true)
34        result = true;
```

Understanding Java Remote Method Invocation

```
35      return result;
36    }
37  }
```

Understanding Java Remote Method Invocation

3. Define a server class:

```
1   public class BookLauncher {
2     public boolean createBook(String t, String a, String p, String c){
3       Book bookRef = new Book(t, a, p, c);
4       Naming.rebind("//libServer/Book", bookRef);
5       // To put it in the RMI Registry, an object reference and a
6       //logical name for use by callers is needed
7     }
8     public static void main(String args[]) {
9       (new BookLauncher()).createBook(
10                      args[0], args[1], args[2], args[3]);
11    }
12  }
```

Understanding Java Remote Method Invocation

4. Define a caller (client):

```
1   import java.rmi.*;
2   import library.*;
3
4   public class BookUser {
5
6     Borrowable b;
7     public boolean createBook(String t, String a, String p, String c){
8       b = (Borrowable) Naming.lookup(“//libServer/Book”);
9       // Need to know the name of the server
10      // Need to know the logical name of the remote object
11    }
12
13    public boolean checkOut(String patron) {
14  b.checkOut(patron);
15    }
16    // Other methods as necessary
17  }
```

Understanding Java Remote Method Invocation

5. Compile all the code:

```
javac *.java
```

6. Use the `rmic` command to generate the stubs and skeletons which serve as caller and sever-side proxies for the remote object:

```
rmic Book.class
```

7. Start the Java RMI registry:

```
rmiregistry
Run the server program.
java Booklauncher
```

8. Run the caller program:

```
java BookUser
```

Understanding the Java IDL API

- The Java IDL API is the Java technology implementation of CORBA
- The Object Management Group maintains CORBA
- You can write CORBA applications using Java technology

Characteristic	**CORBA**	**Java RMI**
Programming languages	Java, C, C++, COBOL, Smalltalk, LISP, PL/I, and more	Specific to the Java programming language
Naming service	COS Naming Service	Java RMI Registry
Complexity	Quite complex	Much simpler than CORBA
Required software	Must obtain an ORB	Core API in the Java 2 platform, Standard Edition (J2SE™ platform)

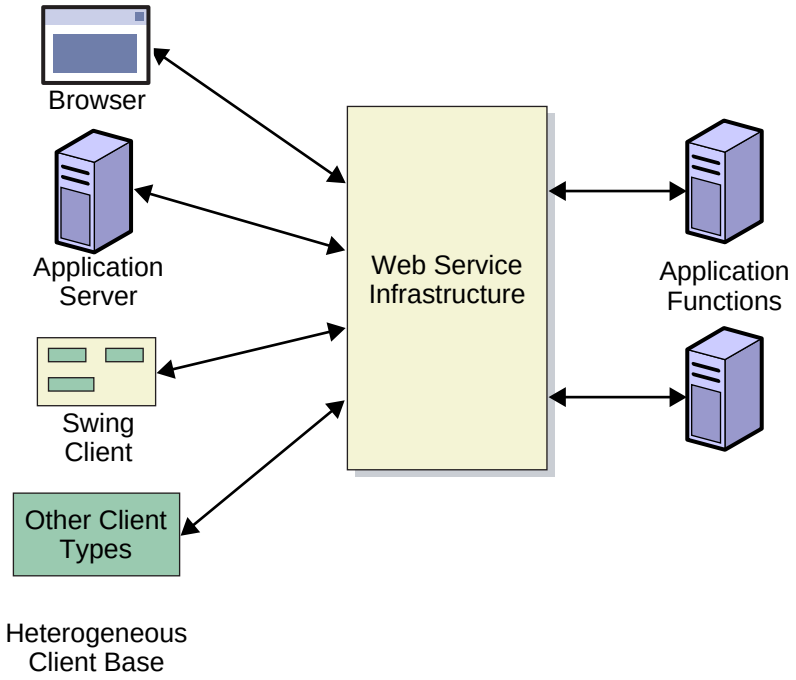
Understanding the Java IDL API

Characteristic	CORBA	Java RMI
Additional services	Other services often available, such as event processing and transaction management	Only a naming service is provided
Network protocol	Uses Internet Inter-ORB Protocol (IIOP)	By default, uses the Java Remote Method Protocol (JRMP)
Interface definition	Definition of attributes and behaviors in the language neutral Interface Definition Language	Work is done only using Java technology

Understanding the JAX-RPC API

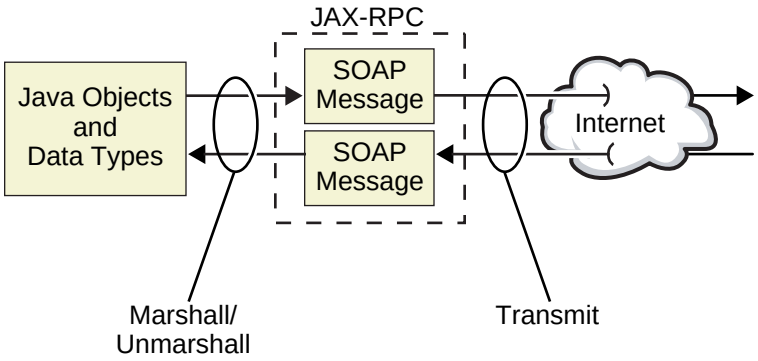
- Web services enable clients and services to communicate regardless of the object model, programming language, or runtime environment used on either side of the communication link
- XML based messages
- HTTP is the most commonly used web service transport protocol
- Simple Object Access Protocol (SOAP) is the most common format for the messages

Understanding the JAX-RPC API



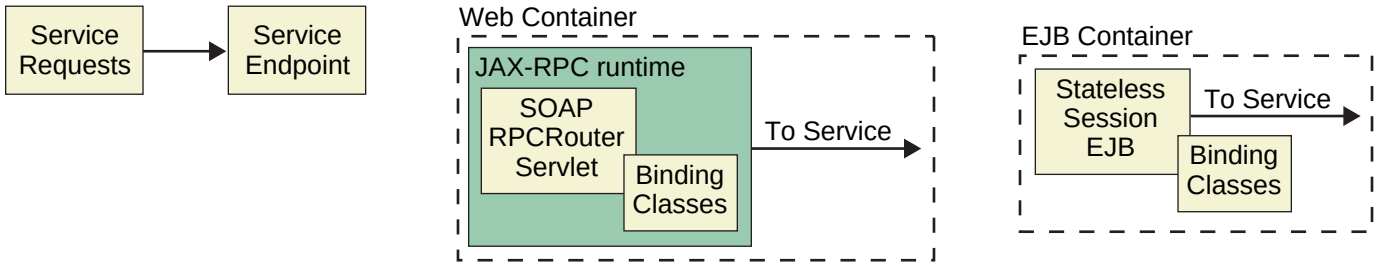
Understanding the JAX-RPC API

- The Java API for XML-based RPC (JAX-RPC) is an extensible API for performing XML-based remote procedure calls
- A JAX-RPC implementation must support HTTP 1.1 as the transport mechanism for SOAP 1.1 messages
- The JAX-RPC specification describes JAX-RPC services according to the WSDL 1.1 specification



Understanding the JAX-RPC API

- JAX-RPC uses a service endpoint model where service requests are directed to the service endpoint
- The service endpoint unmarshalls the service message and invokes the specified service using the method and parameter information contained within the message.
- JAX-RPC supports two endpoint model implementation types, as well as servlets, and stateless session beans



Understanding the JAX-RPC API

- A service interface defines the functionality provided by a web service
- A JAX-RPC interface must extend `java.rmi.Remote`
- All of the interface methods must throw a `java.rmi.RemoteException` and use data types supported by JAX-RPC
- The interface must not declare any constants

```
1   import java.rmi.Remote;
2   import java.rmi.RemoteException;
3
4   public interface InterfaceName extends Remote {
5     // business method
6     public ReturnType methodName(paramList)
7       throws RemoteException;
8   }
```

Understanding the JAX-RPC API

A JAX-RPC service implementation class is a Java class that implements the methods declared on the service interface

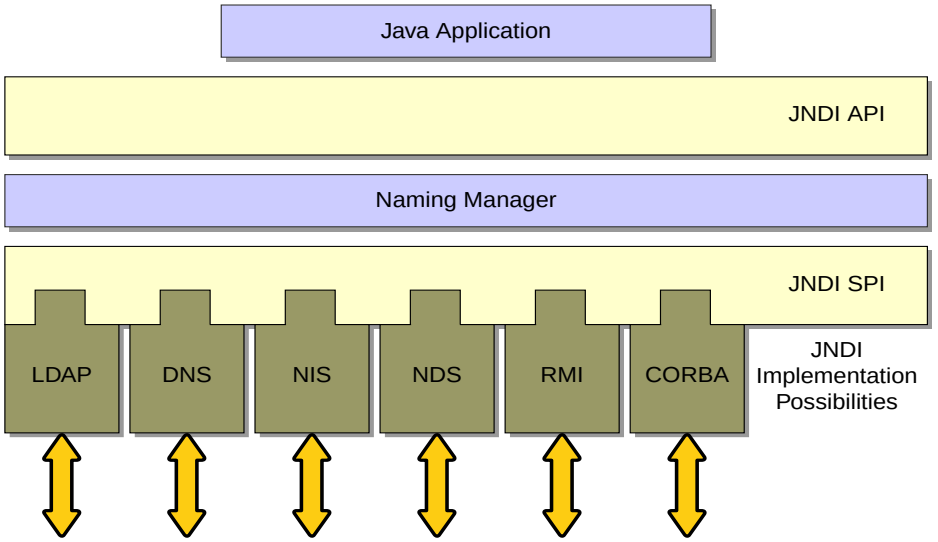
```
1   public class ClassName implements InterfaceName {
2     // constructor
3     public ClassName() {
4       // do nothing
5     }
6   // the actual service
7     public ReturnType methodName(paramList) {
8     method body;
9     }
10  }
```

Understanding the JAX-RPC API

- Having created a service interface and implementation class, a service developer generates the necessary server-side web service infrastructure files:
  - The server tie files
  - The service properties file required for deployment
  - Serializer and deserializer files
  - A WSDL file that describes the service
- You can do these tasks using the tools provided with the Java Web Services Developer Pack (Java WSDP) and automate them using one or more Ant build scripts
- Alternatively, you can rely on the functionality in an IDE to perform this work for you

Understanding the Java Naming and Directory Interface™ (JNDI) API

- Allows access to multiple naming services
- Can change naming services without changing code
- The naming service is like a phone book. It looks up names and gets back objects



JNDI API Class

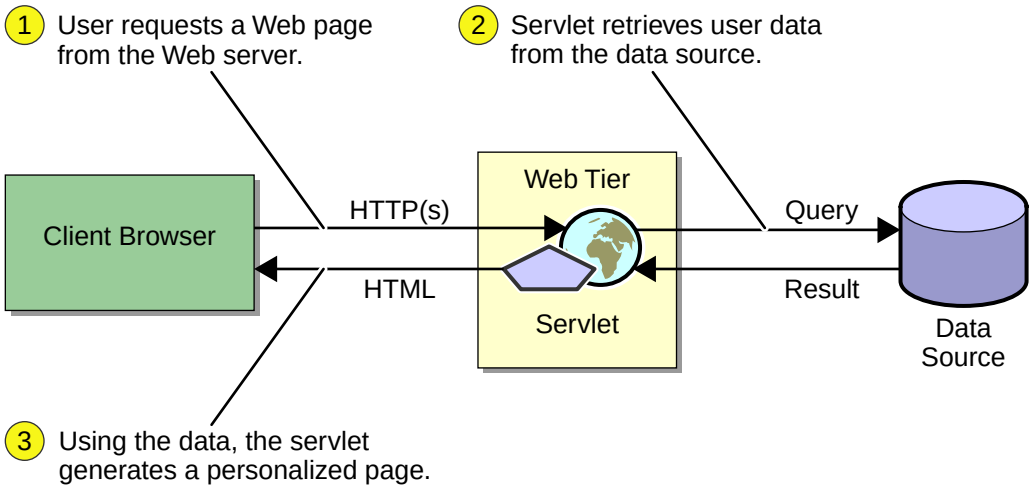
```
1   public class CatalogManager {
2     public CatalogQueryBeanHome findCatalogQueryBean(String jndiname){
3       try {
4         InitialContext ctx = new InitialContext();
5         Object ref = ctx.lookup(jndiname);
6         CatalogQUeryBeanHome home =
7         (CatalogQueryBeanHome)javax.rmi.PortableRemoteObject.narrow
8             ("CatalogQueryBean", CatalogQUeryBeanHome.class);
9       }catch(Exception e) {
10        e.printStackTrace();
11      }
12    }
13  }
```

Understanding the Java Naming and Directory Interface™ API

- The `javax.naming` package contains the API classes
- Create an `InitialContext` object:
  - Use a `java.util.Properties` object to specify the URL and driver classes
- Call the `lookup` method
- Use the `javax.rmi.PortableRemoteObject.narrow` method to narrow the results
- Cast the results

Understanding Servlets

- Server managed Web components
- Servlets create dynamic Web content
- Replace CGI and similar technologies
- The `javax.servlet` package contains the API classes



Servlet Class

```
1   import java.io.*;
2   import java.util.*;
3   import javax.servlet.*;
4   import javax.servlet.http.*;
5   import libraryApp.*;
6
7   public class CatalogServlet extends HttpServlet {
8     private CatalogQueryBeanHome cqbh;
9     public void init() throws ServletException {
10      try {
11        InitialContext ctx = new InitialContext();
12        Object ref = ctx.lookup("CatalogQueryBean");
13        CatalogQueryBeanHome cqbh = (CatalogQueryBeanHome)
14        javax.rmi.PortableRemoteObject.narrow
15            ("CatalogQueryBean", CatalogQUeryBeanHome.class);
16      }catch(Exception e) {
17        e.printStackTrace();
18        }
19    }
```

Servlet Class

```
20
21    public void destroy() { }
22    public void doGet (HttpServletRequest request,
23                       HttpServletResponse response)
24                       throws ServletException, IOException {
25      HttpSession session = request.getSession();
26      try {
27        String category = request.getParameter("SearchType");
28        String searchValue = request.getParameter("SearchValue");
29        CatalogQueryBean cqb = cqbh.create();
30        Collection titles = cqb.query(category, searchValue);
31        cqb.remove();
32      }catch(CreateException crex) {
33        System.out.println
34          ("Error creating CatalogQueryBean " + crex.getMessage());
35      }
36      // Handle other exception types
37      // set content-type header before accessing the Writer
```

Servlet Class

```
38      response.setContentType("text/html");
39      response.setBufferSize(8192);
40      PrintWriter out = response.getWriter();
41      // then write the data of the response
42      out.println
43      ("<html><head><title>Catalog Query Results</title></head>");
44      out.println("<body>");
45      Iterator iter = titles.getItems();
46      while (iter.hasNext()) {
47         String title = (Book)iter.next();
48        out.println(title);
49      }
50      out.println("</body>");
51      out.println("</html>");
52      out.close();
53    }
54  }
```

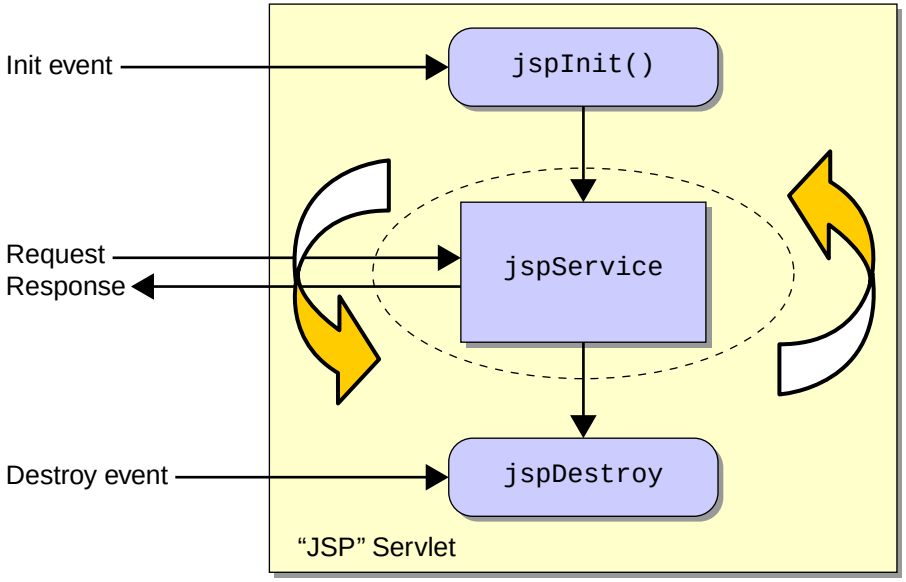
Understanding Servlets

- Subclass of `HttpServlet`
- `doGet` – Called when a client sends a GET request:
  - `HttpServletRequest` object; use the `getParameter` method to get parameters for the Web page
  - `HttpServletResponse` object:
    - Use the `setContentType` method to set mime type
    - Use a `PrintWriter` object to send HTML back to the client

Understanding JavaServer Pages™ Technology

- Server managed Web components
- Allows for separation of developer roles
- Improves application maintainability
- Helps prevent cut-and-paste reuse
- Automatic recompilation
- Contains HTML with special tags
- Get translated and compiled into Servlet classes

JSP Page Flow



HTML Form Processed by a JSP Page

```
1   <HTML>
2   <HEAD>
3   <TITLE>KDL University Library System</TITLE>
4   </HEAD>
5   <BODY>
6   <FORM action="/Catalog.jsp" method="get">
7   Search Type:
8   <INPUT type=”text” name=”type”><BR>
9   Search Text:
10  <INPUT type=”text” name=”value”><BR>
11  <INPUT type="submit" value="Submit">
12  </FORM>
13  </BODY>
14  </HTML>
```

JSP Page

```
1   <%@ include file="Header.jsp" %>
2   <jsp:useBean id="Catalog" class="libraryApp.CatalogBean" />
3   <% String searchType = request.getParameter("type"); %>
4   <jsp:setProperty name="Catalog"
5         property="type" value="<%= searchType %>" />
6   <% String searchValue = request.getParameter("value"); %>
7   <jsp:setProperty name="Catalog" property=
8         "value" value="<%= searchValue %>" />
9   <% try { %>
10    <P>Account:
11    <jsp:getProperty name="Catalog" property="searchResult" />
12    </P>
13  <% } catch (Exception e) { %>
14    <P>Error retrieving item from library ctalog:
15    <jsp:getProperty name="Catalog" property="value" />
16    </P>
17  <% } %>
18  <%@ include file="Footer.jsp" %>
```

Understanding JavaServer Pages Technology

- Comments – Text ignored by JSP page processor

```
<!-- This is a comment -->
```

- Directives – Compile-time instructions

```
<%@ page language = "java" import = "java.util.*" %>
```

- Declarations – Declare Java technology variables

```
<%! int x = 5 %>
```

- Expressions – Produce output

```
<%= clock.getYear(); %>
```

Understanding JavaServer Pages Technology

- Scriptlets

```
<% int x = 5; out.println(x); %>
```

- JavaBeans components

```
<jsp:useBean id="name" />
```

- Custom tags

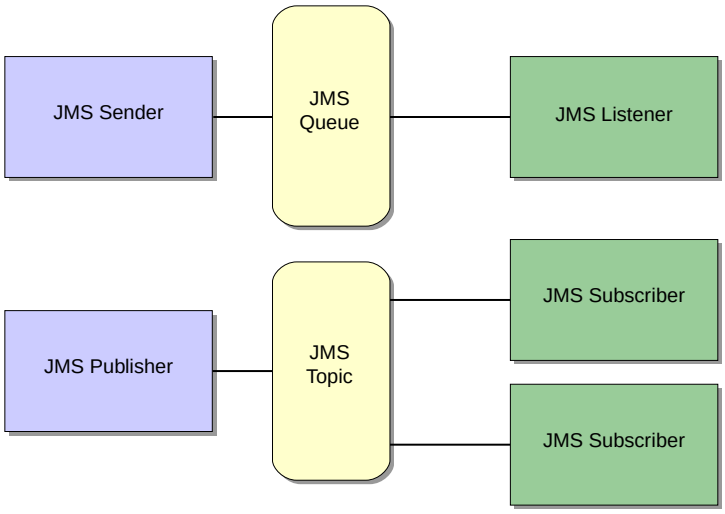
```
<%@ taglib uri="/tlds/library.tld" prefix="library" %>
<P>New books in our collection</P>
<library:showNewBooks month="January" >
```

- Java Standard Tag Library (JSTL) and Expression Language (EL)

```
<c:forEach var=”book” items=”${bookList.books}”>
<li> <c:out value=”book.name”/> </li>
</c:forEach>
```

Understanding the Java Message Service (JMS) API

- Allows access to multiple messaging services without rewriting code
- The `javax.jms` package contains the API classes
- Provides a way to do asynchronous messaging



Understanding the JMS API

1. Create a sender:

```
1   import javax.jms.*;
2   import javax.naming.*;
3   public class PurchaseRequestSender {
4     public static void main(String[] args) {
5       Context  context = null;
6       QueueConnectionFactory qcFactory = null;
7       QueueConnection queueConnection = null;
8       QueueSession queueSession = null;
9       Queue queue = null;
10      QueueSender queueSender = null;
11      TextMessage message = null;
12      try {
13        context = new InitialContext();
14      } catch (NamingException ne) {
15        System.out.println("Could not create “ +
16          “JNDI context: " + ne.toString());
17        System.exit(1);
18      }
19      try {
20        qcFactory = (QueueConnectionFactory)
21            context.lookup("java:comp/env/jms/
22            PurchasingQueueConnectionFactory");
23        queue = (queue) context.lookup
24          ("java:comp/env/jms/PurchaseQueue");
25      } catch (NamingException e) {
26        System.out.println
27          ("JNDI lookup failed: " +e.toString());
28        System.exit(1);
```

Understanding the JMS API

```
29      }
30      try {
31        queueConnection =
32          qcFactory.createQueueConnection();
33        queueSession =
34        queueConnection.createQueueSession(false,
35          Session.AUTO_ACKNOWLEDGE);
36        queueSender =
37          queueSession.createSender(queue);
38        message = queueSession.createTextMessage();
39        message.setText
40          ("// Purchase request data goes here");
41        queueSender.send(message);
42      } catch (JMSException jmse) {
43        System.out.println("Exception occurred: " +
44          jmse.toString());
45      } finally {
46        if (queueConnection != null) {
47          try {
```

Understanding the JMS API

```
48            queueConnection.close();
49          } catch (JMSException e) {
50            System.out.println
51              ("JMS error: " +e.getMessage());
52            e.printStackTrace();
53          }
54        }
55      }
56    }
57  }
```

Understanding the JMS API

This JMS sender code does the following things:

a. Looks up a queue connection factory.
b. Looks up a queue.
c. Creates a connection using the factory.
d. Creates a session using the connection.
e. Creates a sender using the session and passing in the queue.
f. Creates a message using the sender.
g. Sends the message.
h. Closes the connection.

Understanding the JMS API

2. Create a message receiver:

```
1   import javax.jms.*;
2   import javax.naming.*;
3   public class PurchaseOrderReceiver {
4     public static void main(String[] args) {
5       Context context = null;
6       QueueConnectionFactory qcFactory = null;
7       QueueConnection queueConnection = null;
8       QueueSession queueSession = null;
9       Queue queue = null;
10      QueueReceiver  queueReceiver = null;
11      TextMessage message = null;
12      try {
13        context = new InitialContext();
14      } catch (NamingException e) {
15        System.out.println("Could not create
16          InitiialContext: " + e.toString());
17        System.exit(1);
```

Understanding the JMS API

```
18      }
19      try {
20        qcFactory = (QueueConnectionFactory)
21        context.lookup("java:comp/env/jms
22          PurchasingQueueConnectionFactory");
23        queue = (Queue)context.lookup
24          ("java:comp/env/jms/PurchaseQueue");
25      } catch (NamingException e) {
26        System.out.println
27            ("JNDI lookup failed: " + e.toString());
28        System.exit(1);
```

Understanding the JMS API

```
29      }
30      try {
31        queueConnection =
32            qcFactory.createQueueConnection();
33        queueSession =
34              queueConnection.createQueueSession(
35              false,Session.AUTO_ACKNOWLEDGE);
36        queueReceiver = queueSession
37              .createReceiver(queue);
38        queueConnection.start();
39        while (true) {
40          Message m = queueReceiver.receive(1);
41          if (m != null) {
42              if (m instanceof TextMessage) {
43                message = (TextMessage) m;
44                System.out.println("Reading “ +
45                “message: " + message.getText());
46              } else {
47                break;
```

Understanding the JMS API

```
48              }
49            }
50          }
51        } catch (JMSException e) {
52          System.out.println("Exception occurred: "
53            + e.toString());
54        } finally {
55          queueConnection.close();
56          ...
```

Understanding the JMS API

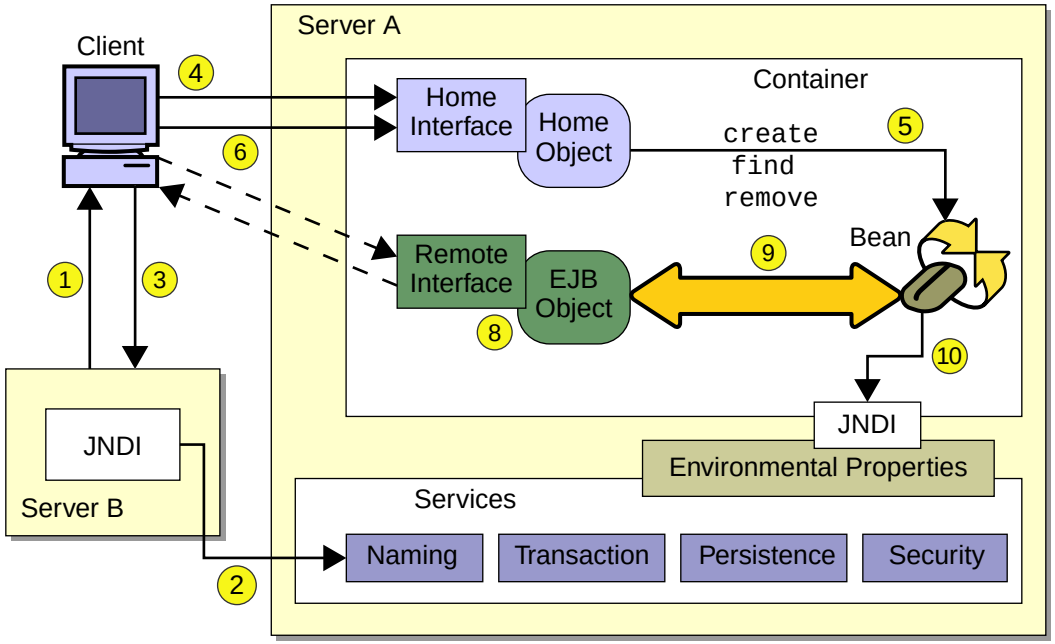
This JMS receiver code does the following things:

1. Looks up a queue connection factory.
2. Looks up a queue.
3. Creates a connection using the factory.
4. Creates a session using the connection.
5. Creates a receiver using the session and passing in the queue.
6. Starts the connection.
7. Receives the message.
8. Closes the connection.

Understanding Enterprise JavaBeans™ Technology Components

- Container managed business components
- Components for business logic
- Container provides these services:
  - Security
  - Transactions
  - Concurrency
  - Resource management
  - Persistence

Overall Enterprise JavaBeans Architecture



Overall Enterprise JavaBeans Architecture: Parts of an EJB Component

Home Object:

- Is a factory that creates, locates, and destroys beans
- Provided by the container
- Implements the `Home` interface that extends the `javax.ejb.EJBHome` interface

Overall Enterprise JavaBeans Architecture: Parts of an EJB Component

EJB Object:

- A proxy that forwards business calls to the bean
- Provided by the container
- Implements the component interface that extends the `javax.ejb.EJBObject` interface

Overall Enterprise JavaBeans Architecture: Parts of an EJB Component

Deployment Descriptor:

- Is an XML document
- Contains structural information about the EJB component
- Is used for deployment by the container

Overall Enterprise JavaBeans Architecture: Remote and Local Interfaces

- Developer can choose the Home and Component interfaces to be remote or local
- Remote interfaces allow access to EJB components across the network
- Local interfaces are more efficient for J2EE platform components on the same physical host

Three Types of EJB Components

	Session Bean	**Entity Bean**	**Message-Driven Bean**
Primary Purpose	Modeling synchronous business logic	Modeling entities, especially tables in an RDBMS or objects in an ODBMS	Modeling asynchronous business logic
Interface	`SessionBean`	`EntityBean`	`MessageDrivenBean`
Client State	Can be conversational (stateful or stateless)	Models persistent state	Stateless. Not accessible by the client except through the JMS API

Session Beans

	Stateful Session Bean	**Stateless Session Bean**
Conversational State	Stateful	Stateless
Instance owned by client?	Generally yes	Generally no
`create` method parameters	Can take parameters for initialization	Cannot accept parameters for initialization
Use	Modeling conversational business logic, such as needed for a shopping cart	Modeling one-shot business logic, such as querying a catalog or validating a credit card number. Sometimes more efficient than a stateful session bean

Session Bean Home Interface

```
1   import javax.ebj.*;
2   import java.rmi.*;
3
4   public interface ShoppingCartHome extends javax.ejb.EJBHome {
5     public ShoppingCart create() throws
6             CreateException, RemoteException;
7     public ShoppingCart create(String custID) throws
8             CreateException, RemoteException;
9   }
```

Session Bean Component Interface

```
1   import java.rmi.*;
2   import javax.ejb.*;
3   public interface ShoppingCart extends javax.ejb.EJBObject {
4     public Collection getItemsInCart() throws RemoteException;
5     public boolean addToCart(String itemID, int qty)
6           throws RemoteException;
7     public boolean removeFromCart(String itemID, int qty)
8           throws RemoteException;
9     public checkOut(String custID)
10          throws RemoteException, CustomerException;
11  }
```

Session Bean Class

```
1   import javax.ejb.*;
2   import java.util.*;
3
4   public class ShoppingCartBean implements javax.ejb.SessionBean {
5     private SessionContext context;
6     private String customerID;
7     private LinkedList items;
8     // Initialize the bean instance with no parameters
9     public void ejbCreate() {}
10    // Initialize the bean instance with one parameter
11    public void ejbCreate(String custID) {
12      customerID = custID;
13    }
14    // Save object which provides environmental access
15    public void setSessionContext(SessionContext sc) {
16      context = sc;
17    }
```

Session Bean Class

```
18    // Destroy bean instance
19    public void ejbRemove() { }
20    // The next two methods are container callbacks
21    // for resource management
22    public void ejbPassivate() { }
23    public void ejbActivate() { }
24    // Remote interface methods
25    public Collection getItemsInCart() {
26      return items;
27    }
28    public boolean addToCart(String itemID, int qty) {
29      // Add item logic
30    }
31    public boolean removeFromCart(String itemID, int qty) {
32      // Remove item logic
33    }
34    public checkOut(String custID) {
35      // Perform order check out operations
36    }}
```

Session Bean Class

Must contain:

- A codeless, no argument constructor
- Home related methods; `ejbCreate` method for every `create` method in the `Home` interface
- Business methods; every business method in the component interface must be in the bean class
- `javax.ejb.SessionBean` interface methods

Session Bean Class

Session Bean Methods:

- `ejbCreate` – Initialization code
- `ejbRemove` – Clean up code before the container destroys the instance
- `ejbPassivate` – Clean up code before the container writes a stateful session bean to secondary storage
- `ejbActivate` – Regain resources after the container rebuilds a stateful session bean from secondary storage

Entity Beans

- Represent a persistent business entity
- Persisted in a data store
- Must have container-managed transactions (CMT)
- Are pooled
- Can be bean-managed persistence (BMP) or container-managed persistence (CMP):
  - BMP – Developer writes data access code
  - CMP – Container writes data access code

BMP Entity Bean Home Interface

```
1   import javax.ejb.*;
2   import java.rmi.*;
3   import java.util.*;
4   public interface BookHome extends javax.ejb.EJBHome {
5     public Book create(String anAuthor, String aTitle,
6           String aCallNo, String aSubject)
7           throws CreateException, RemoteException;
8      public String findByPrimaryKey(String theCallNo)
9           throws FinderException, RemoteException;
10  }
```

BMP Entity Bean Component Interface

```
1   import javax.ejb.*;
2   import java.rmi.*;
3   import java.util.*;
4   public interface Book extends EJBObject {
5     public boolean checkOut(String cardID) throws RemoteException;
6     public String getAuthor() throws RemoteException;
7     public String getTitle() throws RemoteException;
8     public String getStatus() throws RemoteException;
9   }
```

BMP Entity Bean Class

```
1   import javax.ejb.*;
2   import java.util.*;
3   public class BookBean
4         implements javax.ejb.EntityBean {
5     private String author;
6     private String title;
7     private String callNo;
8     private String subject;
9     private String status;
10    private EntityContext context;
11
12    public String ejbCreate(String anAuthor,
13            String aTitle, String aCallNo,
14            String aSubject) {
15      author = anAuthor;
16      title = aTitle;
17      callNo = aCallNo;
18      subject = aSubject;
19      // Do an insert into the database
```

BMP Entity Bean Class

```
20      //  Return the primary key
21      return aCallNo;
22    }
23    public String ejbPostCreate(String anAuthor,
24            String aTitle, String aCallNo,
25            String aSubject) {
26      // There must be an ejbPostCreate() method for
27      // every ejbCreate() method which uses the
28      // same parameters, even if it does nothing.
29    }
```

BMP Entity Bean Class

```
30
31    public void setEntityContext(EntityContext ctx)
32    {
33      context = ctx;
34    }
35    public void unsetEntityContext() { }
36    public void ejbRemove() {
37      // Delete the row from the database
38    }
39    public void ejbActivate() {
40      callNo = (String) context.getPrimaryKey();
41    }
42    public void ejbPassivate() {
43      author = null;
44      title = null;
45      callNo = null;
46      subject = null;
47      status = null;
48    }
```

BMP Entity Bean Class

```
49    public void ejbLoad() {
50      callNo = (String) context.getPrimaryKey();
51      // Do a database select and initialize the data
52      // members of the bean with the data
53    }
54
55    public void ejbStore() {
56      // Do a database update based on the content
57    // of the data members
58    }
```

BMP Entity Bean Class

```
59    public String findByPrimaryKey
60        (String theCallNo) throws FinderException {
61      boolean found = true;
62      // Do query using theCallNo.
63      // if unsuccessful, set found to false.
64      if(found == false)
65        throw new FinderException
66          (theCallNo + "not found in database");
67      else
68        return theCallNo;
69    }
70    public boolean checkOut(String cardID) {
71      boolean result = false;
72      // Process check out request
73      return result;
74    }
75    public String getAuthor() {
76      return author;
77    }
```

BMP Entity Bean Class

```
78    public String getTitle() {
79      return title;
80    }
81    public String getStatus() {
82      return status;
83    }
84  }
```

Entity Beans

A BMP bean must contain:

- A codeless no argument constructor.
- The persistent fields.
- Home related methods – `ejbCreate` and `ejbPostCreate` method for every `create` method in the Home interface. `ejbFindXXX` for every `findXXX` in the Home interface.
- Business methods – Every business method in the component interface must be in the bean class.
- One or more `javax.ejb.EntityBean` interface methods.

Entity Beans

Entity Bean Methods:

- `ejbCreate` – Inserts a row
- `ejbRemove` – Deletes a row
- `ejbPassivate` – Clean up code before container writes a stateful session bean to secondary storage
- `ejbActivate` – Regain resources after container rebuilds a stateful session bean from secondary storage

Entity Beans

Entity Bean Methods:

- `setEntityContext` – Constructor code
- `unsetEntityContext` – Cleanup code before container destroys instance
- `ejbLoad` – Puts data from database into bean
- `ejbStore` – Puts data from bean into database

CMP Entity Bean Home Interface

```
1   import javax.ejb.*;
2   import java.rmi.*;
3
4   public interface PatronHome extends EJBHome {
5     public Patron create(String name, String addr, String cardID)
6         throws RemoteException, CreateException;
7     public String findByPrimaryKey(String cardID)
8         throws RemoteException, FinderException;
9   }
```

CMP Entity Bean Component Interface

```
1   import javax.ejb.*;
2   import java.util.*;
3   import java.rmi.*;
4
5   public interface Patron extends EJBObject {
6     public String getName() throws RemoteException;
7     public Collection getHoldings() throws RemoteException;
8     public float getFines() throws RemoteException;
9   }
```

CMP Entity Bean Class

```
1   import javax.ejb.*;
2   import java.util.*;
3   import java.rmi.*;
4
5   public abstract class PatronBean implements
EntityBean {
6     private EntityContext context;
7     public String ejbCreate(String name,
8           String addr, String cardID) {
9       setName(name);
10      setAddress(addr);
11      setLibCardNo(cardID);
12      return getLibCardNo();
13    }
14    public void ejbPostCreate(String name,
15          String addr, String cardID) {
16    }
17    public void setEntityContext(EntityContext ctx)
{
18      context = ctx;
19    }
```

CMP Entity Bean Class

```
20    public void unsetEntityContext() { }
21    public void ejbRemove() { }
22    public void ejbActivate() {}
23    public void ejbPassivate() {}
24    public void ejbLoad() {}
25    public void ejbStore() {}
26    public abstract String getName();
27    public abstract void setName(String name);
28    public abstract String getLibCardNo();
29    public abstract void setLibCardNo(String lcn);
30    public abstract String getAddress();
31    public abstract void setAddress(String addr);
32  }
```

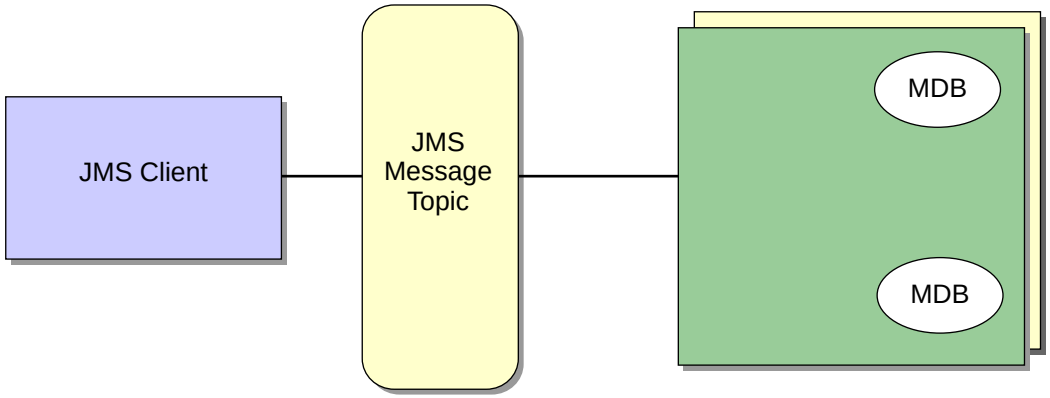
Entity Beans

A CMP bean must contain:

- A codeless no argument constructor
- Abstract getters and setters in place of persistent fields
- Home related methods – `ejbCreate` and `ejbPostCreate` method for every `create` method in the `Home` interface.
- Business methods – Every business method in the component interface must be in the bean class
- `javax.ejb.EntityBean` interface methods

Message-Driven Beans

- Are asynchronously invoked through the JMS API
- Have no home or component interface
- Are similar to stateless session beans:
  - Hold no conversational state
  - Pooled
  - Handle requests from multiple clients



Message-Driven Bean Class

```
1   import java.io.Serializable;
2   import java.rmi.*;
3   import javax.ejb.*;
4   import javax.naming.*;
5   import javax.jms.*;
6   public class PurchaseOrderMakerBean
7      implements MessageDrivenBean, MessageListener {
8     private MessageDrivenContext mdContext;
9     public void setMessageDrivenContext(
10          MessageDrivenContext mdc) {
11      mdContext = mdc;
12    }
13    public void ejbCreate() { }
14    public void onMessage(Message inMessage) {
15      try {
16        if (inMessage instanceof TextMessage) {
17          TextMessage msg =(TextMessage) inMessage;
18          // Extract data from msg and create order
19        } else {
```

Message-Driven Bean Class

```
20          System.out.println("Wrong message type “
21            + inMessage.getClass().getName());
22        }
23      } catch (Exception e) {
24        e.printStackTrace();
25        mdContext.setRollbackOnly();
26      }
27    }
28    public void ejbRemove() { }
29  }
```

Message-Driven Beans

Must contain:

- A codeless no argument constructor
- `javax.ejb.MessageDrivenBean` interface methods:
  - `ejbCreate` – Initialization code
  - `setMessageDrivenContext` – Optionally save a reference to the context
- A `javax.jms.MessageListener` interface method:

  `onMessage` – Code for how to process a message

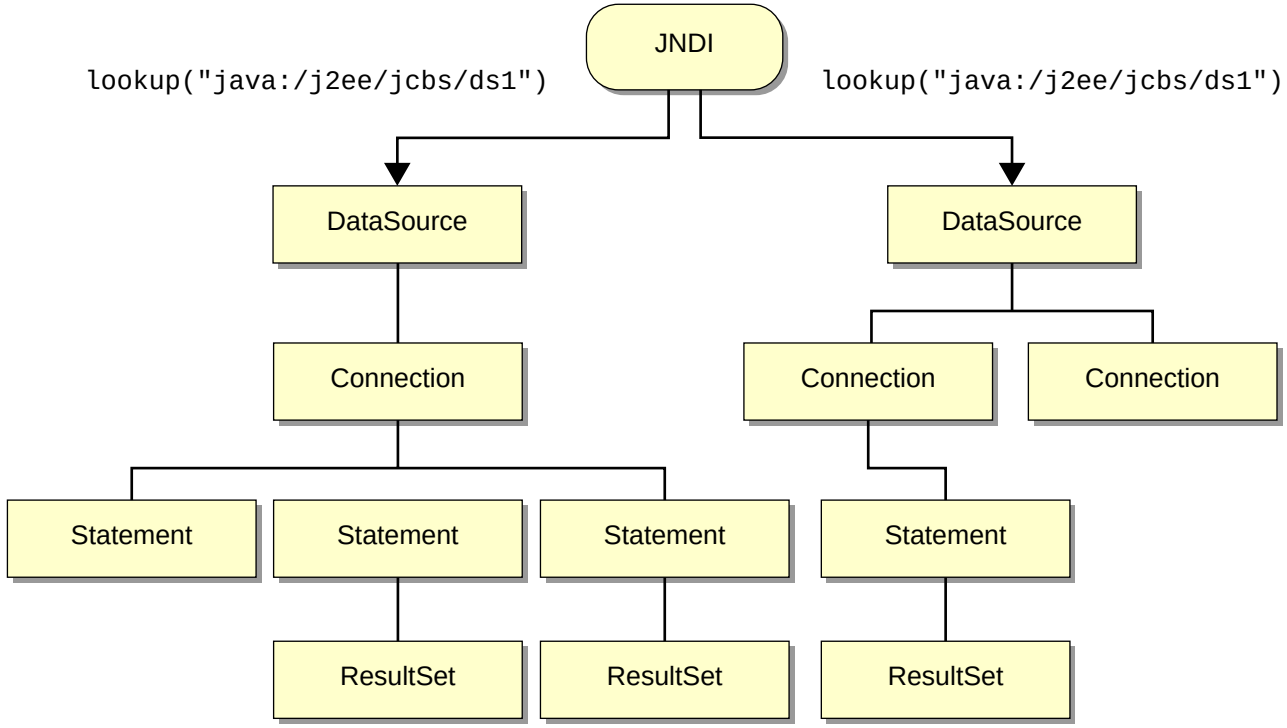
There are no interfaces, and the client is a standard JMS client

Understanding JDBC™ Technology

- Allows access to multiple databases without rewriting code
- Steps for database access:

1. Look up the `DataSource` object using JNDI.
2. Get a connection.
3. Create a statement.
4. Execute a query.
5. Process the results.
6. Close the connection.

JDBC Technology Class Interactions on the J2EE Platform



JDBC Technology Using the DataSource Class

```
1   import java.sql.*;
2   import javax.sql.*;
3   import java.util.*;
4   import javax.naming.*;
5
6   public class CatalogSearcher {
7     DataSource ds;
8     Connection conn;
9     PreparedStatement stmt;
10    ResultSet rs;
11    InitialContext context;
12    public static void main(String[] args) {
13      CatalogSearcher cs = new CatalogSearcher();
14      cs.search("Lewis, C. S.");
15    }
16    public CatalogSearcher() {
17      try {
18        context = new InitialContext();
```

JDBC Technology Using the DataSource Class

```
19          ds = (DataSource)context.lookup
20                ("java:comp/env/jdbc/catalog");
21        } catch(NamingException ne) {
22          System.out.println
23            ("JNDI failure:  " + ne.getMessage());
24          System.exit(1);
25        }
26      }
```

JDBC Technology Using the DataSource Class

```
27    public Collection search(String authorName) {
28      ArrayList al = new ArrayList();
29      try {
30        conn = ds.getConnection();
31        stmt = conn.prepareStatement(
32          "select * from Catalog where author = ?");
33        stmt.setString(1, authorName);
34        ResultSet rs = stmt.executeQuery();
35        int x = 0;
36        while(rs.next() == true) {
37          String title = rs.getString("Title");
38          String pub = rs.getString("Publisher");
39          String date = rs.getString("Date");
40          String callNo = rs.getString("CallNo");
41          Material m = new Material
42              (title, pub, date, callNo);
43          al.add(x, m);  // Add item from
44        }
```

JDBC Technology Using the DataSource Class

```
45        stmt.close();
46        conn.close();
47      } catch(SQLException sqle) {
48        System.out.println
49            ("SQL Error:  " + sqle.getMessage());
50        sqle.printStackTrace();
51      }
52      return al;
53    }
54  }
```

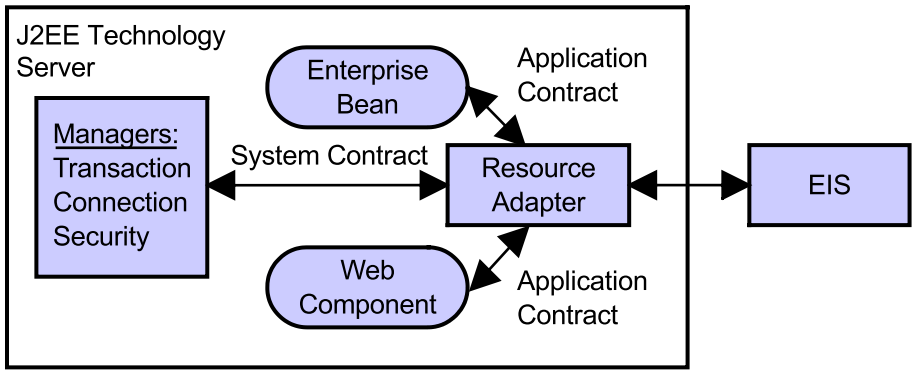
Understanding J2EE Connector Architecture

- J2EE applications might need to access a wide variety of enterprise information systems (EIS)
- EIS integration components might need the ability to integrate with the services of the J2EE server
- The vendor supplies a resource archive (RAR) package, which conforms to the J2EE Connector Architecture (J2EE CA) specification.
- Benefits:
  - Fewer restrictions on the code
  - Full integration with the J2EE server
  - Can support CCI

Understanding J2EE Connector Architecture

Resource adapters can integrate with these key services of the application server:

- Transaction management
- Resource pooling
- Security



Understanding J2EE Connector Architecture

- CCI is a set of interfaces that resemble the JDBC API interfaces. The J2EE components use this API to communicate with the EIS.
- CCI support is not compulsory:
  - Resource adapter vendors are encouraged to implement CCI.
  - Vendors can provide an alternative application contract.
  - Some resource adapters do not require CCI.

The use of CCI does not completely eliminate some API calls that are specific to the legacy system.

Summary

- J2EE platform is a framework for building enterprise applications using Java technology
- The major technologies supported by J2EE include:
  - Java RMI
  - Java IDL
  - JNDI API
  - Java Servlet API
  - JavaServer Pages
  - JMS API
  - EJB technology
  - JDBC technology