DESIGN PRINCIPLES AND PATTERNS

Vinod

http://vinod.co

Background

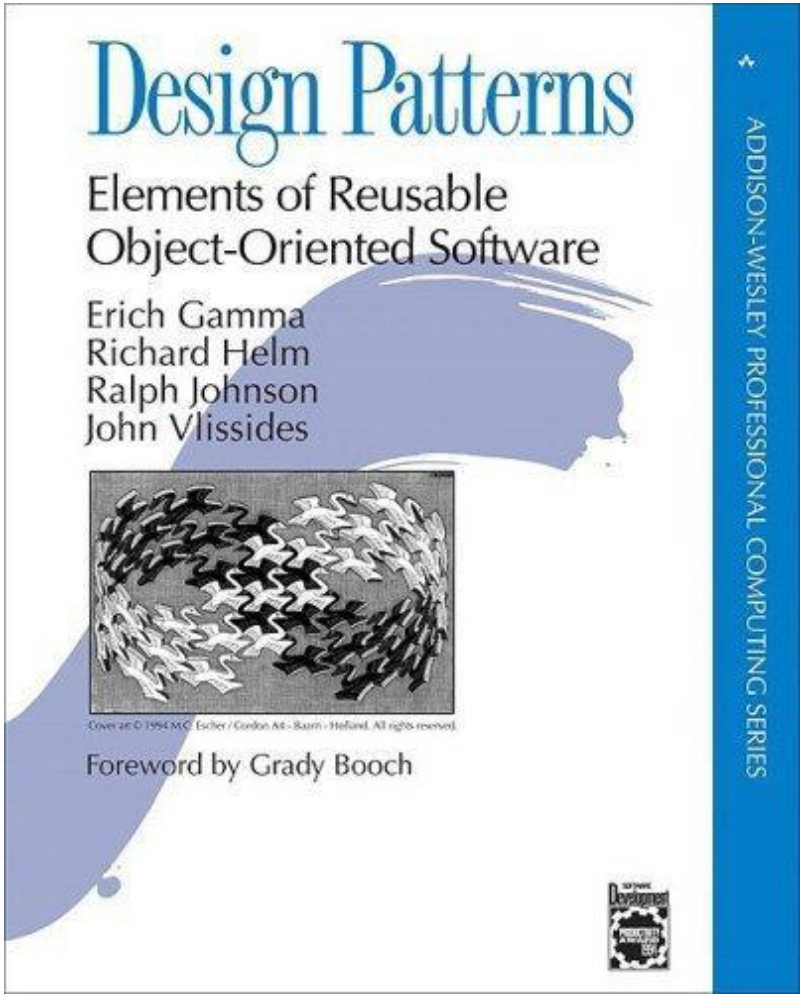
- Patterns originated as an architectural concept by Christopher Alexander (1977/79)
- Reasoning that users know more about the buildings they need than any architect could, he produced and validated a "pattern language" designed to empower anyone to design and build at any scale.

Background

- In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year.

Background

- Design patterns gained popularity in computer science after the book "Design Patterns: Elements of Reusable Object-Oriented Software" was published in 1994 by GOF - "Gangs Of Four"



Design patterns

- Design patterns represent solutions to problems that arise when developing software within a particular context.
  - E.g., problem/solution pairs within a given context
- Describes recurring design structures
- Describes the context of usage

Design patterns

- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
- Especially good for describing how and why to resolve nonfunctional issues
- Patterns facilitate reuse of successful software architectures and designs.

Origins of Design Patterns

"Each pattern describes a problem which 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 in the same way twice"

Christopher Alexander, A Pattern Language, 1977

Context: City Planning and Building architectures

Elements of Design Patterns

- Design patterns have four essential elements:
  - Pattern name
  - Problem
  - Solution
  - Consequences

Pattern Name

- A handle used to describe:
  - a design problem
  - its solutions
  - its consequences
- Increases design vocabulary
- Makes it possible to design at a higher level of abstraction
- Enhances communication
- “The Hardest part of programming is coming up with good variable [function, and type] names.”

Problem

- Describes when to apply the pattern
- Explains the problem and its context
- May describe specific design problems and/or object structures
- May contain a list of preconditions that must be met before it makes sense to apply the pattern

Solution

- Describes the elements that make up the
  - design
  - relationships
  - responsibilities
  - collaborations
- Does not describe specific concrete implementation
- Abstract description of design problems and how the pattern solves it

Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - evaluating design alternatives
  - understanding costs
  - understanding benefits of applying the pattern
- Includes the impacts of a pattern on a system's:
  - flexibility
  - extensibility
  - portability

Design Patterns are NOT

- Designs that can be encoded in classes and reused as is (i.e., linked lists, hash tables)
- Complex domain-specific designs (for an entire application or subsystem)

- They are:
  - "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Where They are Used

- Object-Oriented programming languages [and paradigm] are more open to implementing design patterns

- Procedural languages: need to define
  - Inheritance
  - Polymorphism
  - Encapsulation

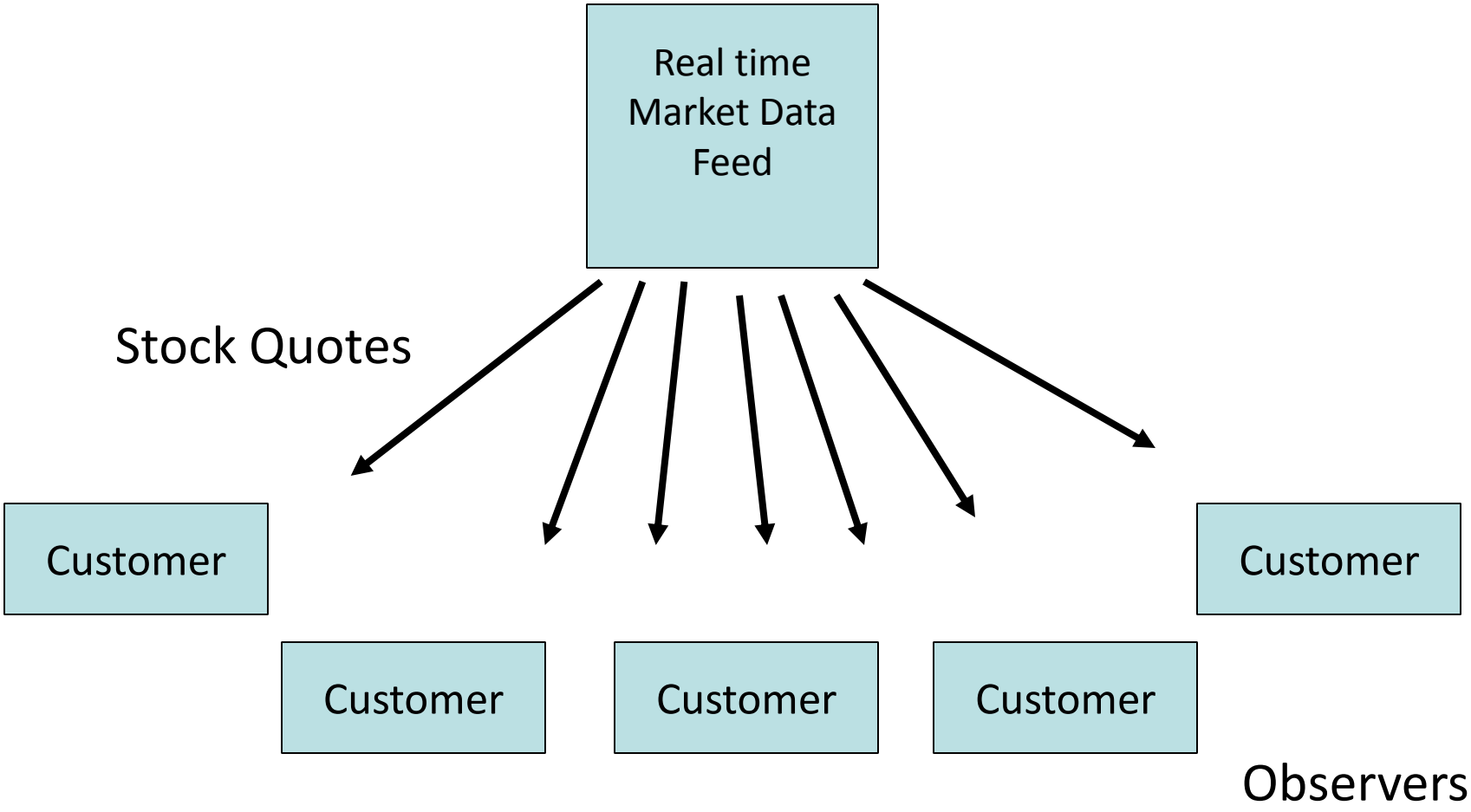
Describing Design Patterns

- Graphical notation is generally not sufficient
- In order to reuse design decisions the alternatives and trade-offs that led to the decisions are critical knowledge
- Concrete examples are also important
- The history of the why, when, and how set the stage for the context of usage

Design Patterns

- Describe a recurring design structure
  - Defines a common vocabulary
  - Abstracts from concrete designs
  - Identifies classes, collaborations, and responsibilities
  - Describes applicability, trade-offs, and consequences

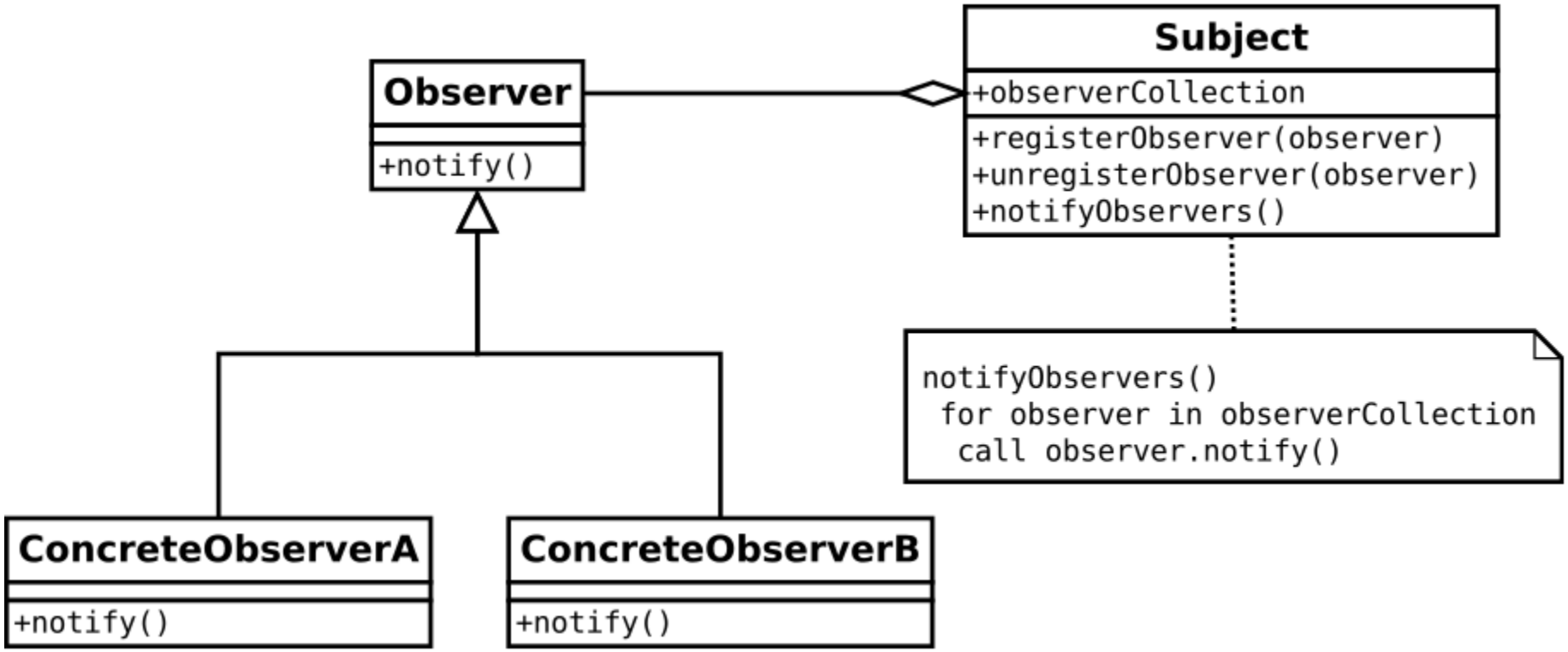
Example: Stock Quote Service



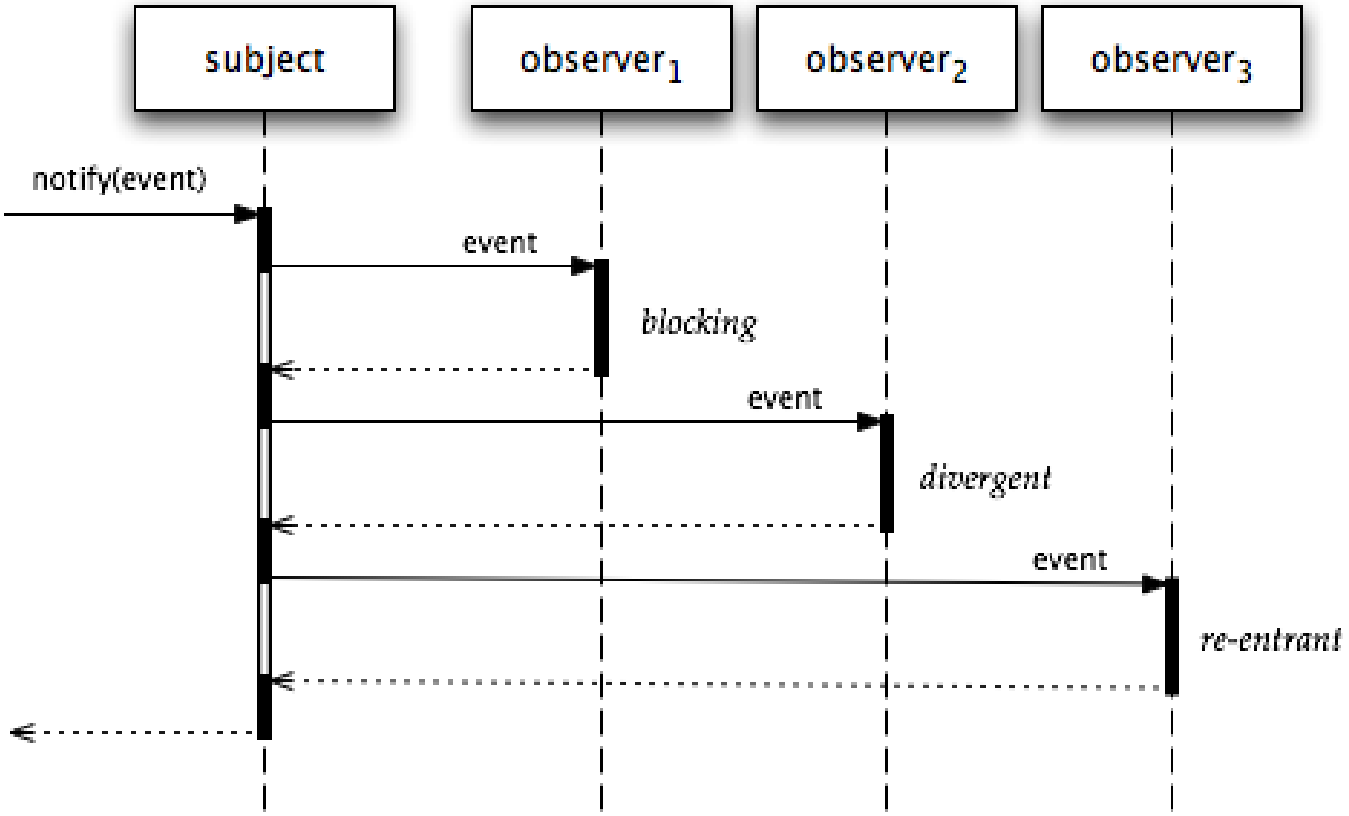
Observer Pattern

- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Key forces:
  - There may be many observers
  - Each observer may react differently to the same notification
  - The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject

Structure of Observer Pattern



Collaborations in Observer Pattern



Design Pattern Descriptions

- Name and Classification: Essence of pattern
- Intent:
  - What it does, its rationale, its context
- AKA:
  - Other well-known names
- Motivation:
  - Scenario illustrates a design problem
- Applicability:
  - Situations where pattern can be applied
- Structure:
  - Class and interaction diagrams
- Participants:
  - Objects/classes and their responsibilities
- Collaborations:
  - How participants collaborate
- Consequences:
  - Trade-offs and results
- Implementation:
  - Pitfalls, hints, techniques, etc.
- Sample Code
- Known Uses:
  - Examples of pattern in real systems
- Related Patterns:
  - Closely related patterns

GoF Design Pattern Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Proxy • Flyweight	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Types of Patterns

- Creational patterns:
  - Deal with initializing and configuring classes and objects
- Structural patterns:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- Behavioral patterns:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Creational Patterns

- Abstract Factory:
  - Factory for building related objects
- Builder:
  - Factory for building complex objects incrementally
- Factory Method:
  - Method in a derived class creates associates
- Prototype:
  - Factory for cloning new instances from a prototype
- Singleton:
  - Factory for a singular (sole) instance

Structural Patterns

- Adapter:
  - Translator adapts a server interface for a client
- Bridge:
  - Abstraction for binding one of many implementations
- Composite:
  - Structure for building recursive aggregations
- Decorator:
  - Decorator extends an object transparently
- Facade:
  - Simplifies the interface for a subsystem
- Flyweight:
  - Many fine-grained objects shared efficiently.
- Proxy:
  - One object approximates another

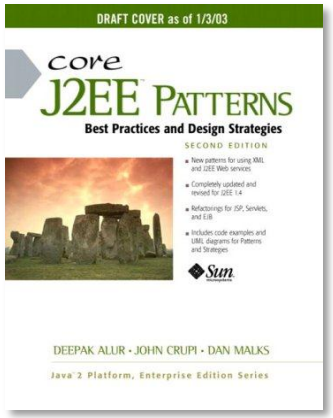
Behavioral Patterns

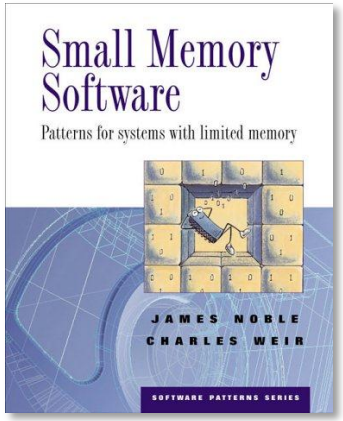
- Chain of Responsibility:
  - Request delegated to the responsible service provider
- Command:
  - Request is first-class object
- Iterator:
  - Aggregate elements are accessed sequentially
- Interpreter:
  - Language interpreter for a small grammar
- Mediator:
  - Coordinates interactions between its associates
- Memento:
  - Snapshot captures and restores object states privately

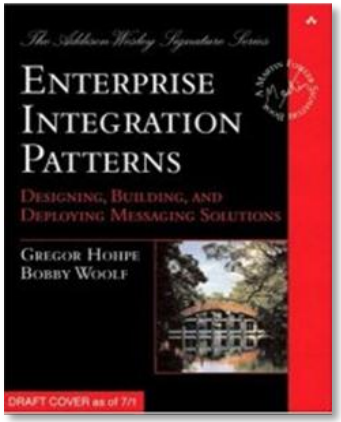
Behavioral Patterns (cont.)

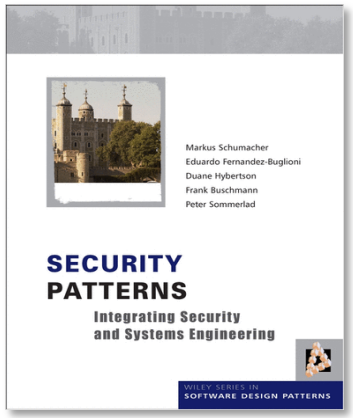
- Observer:
  - Dependents update automatically when subject changes
- State:
  - Object whose behavior depends on its state
- Strategy:
  - Abstraction for selecting one of many algorithms
- Template Method:
  - Algorithm with some steps supplied by a derived class
- Visitor:
  - Operations applied to elements of a heterogeneous object structure

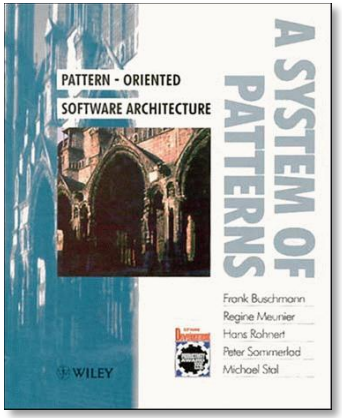
Life Beyond GoF Patterns

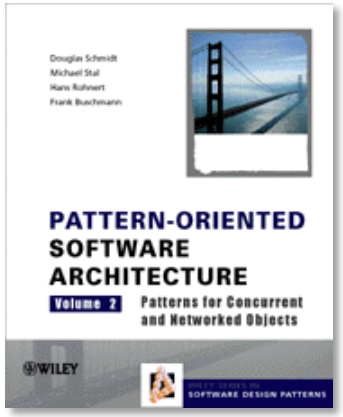


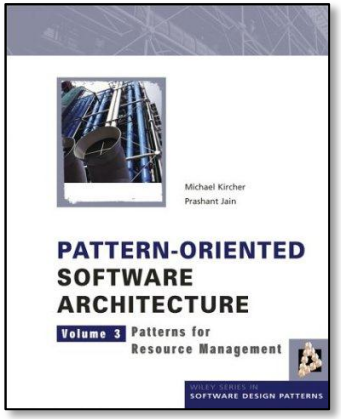


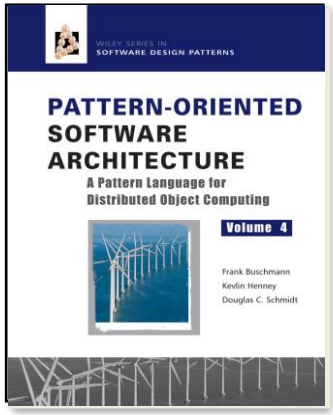


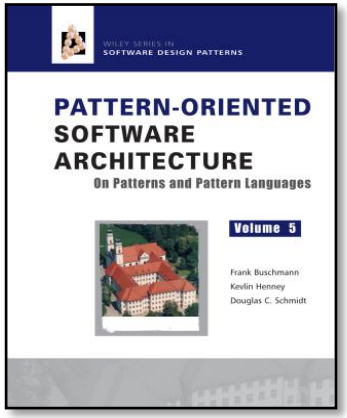












Categorization Terms

- Scope is the domain over which a pattern applies
  - Class Scope: relationships between base classes and their subclasses (static semantics)
  - Object Scope: relationships between peer objects
- Some patterns apply to both scopes.

Class::Creational

- Abstracts how objects are instantiated
- Hides specifics of the creation process
- May want to delay specifying a class name explicitly when instantiating an object
- Just want a specific protocol

Example

- Use Factory Method to instantiate members in base classes with objects created by subclasses
- DaoManager (abstract) class: create application-specific CustomerDao objects conforming to a particular type of implementation
- Application instantiates these CustomerDao objects by calling the factory method getCustomerDao
- Optionally, method is overridden in classes derived from DaoManager
- Subclass JdbcDaoManager overrides getCustomerDao to return a ustomerDao object

Class:: Structural

- Use inheritance to compose protocols or code
- Example:
  - Adapter Pattern: makes one interface (Adaptee's) conform to another
  - Gives a uniform abstraction of different interfaces
  - Class Adapter inherits privately from an Adaptee class
  - Adapter then expresses its interface in terms of the Adaptee's.

Class:: Behavioral

- Captures how classes cooperate with their subclasses to satisfy semantics.

- Example:
  - Template Method: defines algorithms step by step.
  - Each step can invoke an abstract method (that must be defined by the subclass) or a base method.
  - Subclass must implement specific behavior to provide required services

Object Scope

- Object Patterns all apply various forms of non-recursive object composition.
- Object Composition: most powerful form of reuse
- Reuse of a collection of objects is better achieved through variations of their composition, rather than through sub classing.

Object:: Creational

- Abstracts how sets of objects are created

- Example:
  - Abstract Factory: create “product” objects through generic interface
    - Subclasses may manufacture specialized versions or compositions of objects as allowed by this generic interface
  - User Interface Toolkit: 2 types of scroll bars (Motif and Open Look)
    - Don’t want to hard-code specific one; an environment variable decides
  - Class Kit:
    - Encapsulates scroll bar creation (and other UI entities);
    - An abstract factory that abstracts the specific type of scroll bar to instantiate
    - Subclasses of Kit refine operations in the protocol to return specialized types of scroll bars.
    - Subclasses MotifKit and OpenLookKit each have scroll bar operation.

Object:: Structural

- Describe ways to assemble objects to realize new functionality
  - Added flexibility inherent in object composition due to ability to change composition at run-time
  - not possible with static class composition
- Example:
  - Proxy: acts as convenient surrogate or placeholder for another object.
    - Remote Proxy: local representative for object in a different address space
    - Virtual Proxy: represent large object that should be loaded on demand
    - Protected Proxy: protect access to the original object

Object:: Behavioral

- Describes how a group of peer objects cooperate to perform a task that can be carried out by itself.

- Example:
  - Strategy Pattern: objectifies an algorithm
  - Text Composition Object: support different line breaking algorithms
    - Don’t want to hard-code all algs into text composition class/subclasses
  - Objectify different algs and provide them as Compositor subclasses (contains criteria for line breaking strategies)
  - Interface for Compositors defined by Abstract Compositor Class
    - Derived classes provide different layout strategies (simple line breaks, left/right justification, etc.)
  - Instances of Compositor subclasses couple with text composition at run-time to provide text layout
  - Whenever text composition has to find line breaks, forwards the responsibility to its current Compositor object.

When to Use Patterns

- Solutions to problems that recur with variations
  - No need for reuse if problem only arises in one context
- Solutions that require several steps:
  - Not all problems need all steps
  - Patterns can be overkill if solution is a simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
  - Patterns leave out too much to be useful to someone who really wants to understand
  - They can be a temporary bridge

What Makes it a Pattern?

- A Pattern must:
  - Solve a problem and be useful
  - Have a context and can describe where the solution can be used
  - Recur in relevant situations
  - Provide sufficient understanding to tailor the solution
  - Have a name and be referenced consistently

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary
- Patterns help ease the transition to OO technology

Drawbacks to Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a software development process is a human-intensive activity.

Suggestions for Effective Use

- Do not recast everything as a pattern
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- Institutionalize rewards for developing patterns
- Directly involve pattern authors with application developers and domain experts
- Clearly document when patterns apply and do not apply
- Manage expectations carefully.

MVC

Model-View-Controller,

An architectural Pattern

What is MVC?

Architectural design pattern which works to separate data and UI for a more cohesive and modularized system

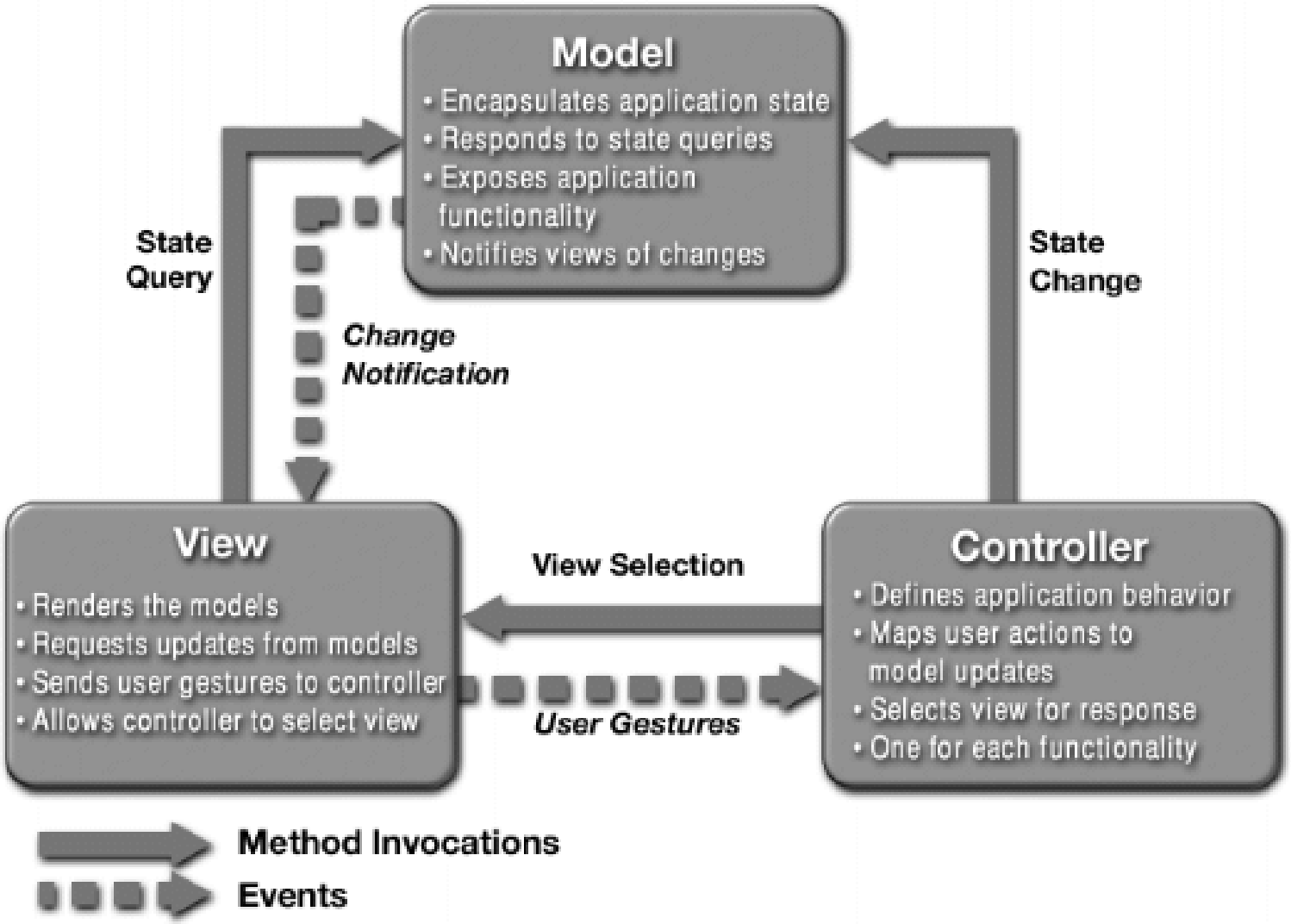
How did it come about?

- Presented by Trygve Reenskaug in 1979
- First used in the Smalltalk-80 framework
  - Used in making Apple interfaces (Lisa and Macintosh)

How does it work?

- User inputs a command
- Controller handles input and updates model or changes the view
- View, which relies on model to show data to user, updates if necessary
- Rinse and Repeat

What is MVC?


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

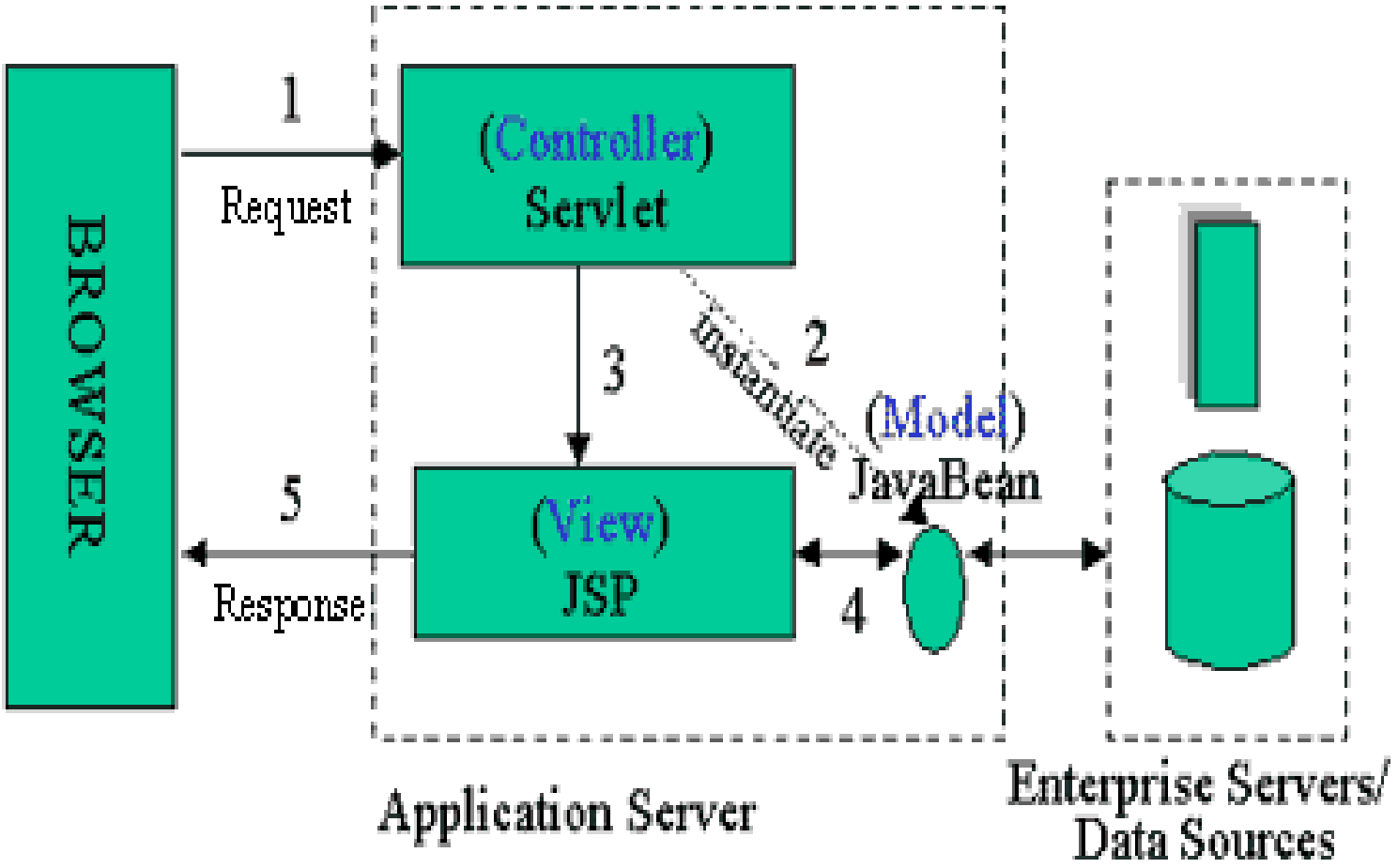
Details of MVC Design Pattern

- Name (essence of the pattern)
  - Model View Controller MVC
- Context (where does this problem occur)
  - MVC is an architectural pattern that is used when developing interactive application such as a shopping cart on the Internet.
- Problem (definition of the reoccurring difficulty)
  - User interfaces change often, especially on the internet where look-and-feel is a competitive issue. Also, the same information is presented in different ways. The core business logic and data is stable.

MVC continued

- Solution (how do you solve the problem)
  - Use the software engineering principle of “separation of concerns” to divide the application into three areas:
    - Model encapsulates the core data and functionality
    - View encapsulates the presentation of the data there can be many views of the common data
    - Controller accepts input from the user and makes request from the model for the data to produce a new view.

MVC Structure for JEE



MVC Architecture

- The Model represents the structure of the data in the application, as well as application-specific operations on those data.

- A View (of which there may be many) presents data in some form to a user, in the context of some application function.

- A Controller translates user actions (mouse motions, keystrokes, words spoken, etc.) and user input into application function calls on the model, and selects the appropriate View based on user preferences and Model state.

Advantages of MVC

- Separating Model from View (that is, separating data representation from presentation)
  - easy to add multiple data presentations for the same data,
  - facilitates adding new types of data presentation as technology develops.
  - Model and View components can vary independently enhancing maintainability, extensibility, and testability.

Advantages of MVC design Pattern

- Separating Controller from View (application behavior from presentation)
  - permits run-time selection of appropriate Views based on workflow, user preferences, or Model state.
- Separating Controller from Model (application behavior from data representation)
  - allows configurable mapping of user actions on the Controller to application functions on the Model.

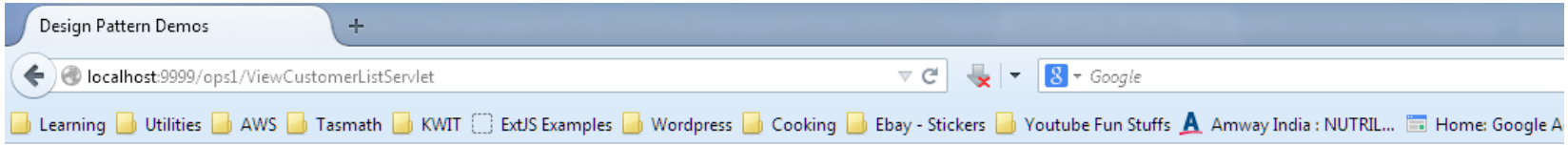
Consequences or Benefits

- We make changes without bringing down the server.
- We leave the core code alone
- We can have multiple versions of the same data displayed
- We can test our changes in the actual environment.
- We have achieved “separation of concerns”

Lab / Practical

Getting started

- Import the project “ops1” into your workspace
- Run the project
- Clicking the link in the homepage should show you this screen



Design Pattern Demos

Customer list

Customer id	Company name	Contact name	Job title	Address	Phone
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57 Berlin 12209 Germany	030-0074321
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq. London WA1 1DP UK	(171) 555-7788
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	0621-08460

Some points

- The application reads the data from the file “customers.csv” and prints the data in a HTML table
  - The hyperlink in the homepage refers to a servlet “ViewCustomerListServlet”

```
<ul>
    <li>
        <a href="ViewCustomerListServlet">View customers</a>
    </li>
</ul>
```

Some points

- The servlet uses the File API to read the content

```
String path = getServletContext().getRealPath("/WEB-INF/classes/customers.csv");

FileReader file = new FileReader(path);
BufferedReader in = new BufferedReader(file);
```

Some points

- The servlet then uses a StringBuffer to generate the required UI
- Contains
  - Static HTML tags and attributes
  - Dynamic data from the file “customers.csv”

```
StringBuffer sb = new StringBuffer(10000);
sb.append("<html>");
sb.append("<head><title>Design Pattern Demos
sb.append("<style type='text/css'>table tr t
sb.append("</head>");
sb.append("<body>");
sb.append("<h1>Design Pattern Demos</h1>");
sb.append("<h2>Customer list</h2>");
sb.append("<table border='1'>");
sb.append("<tr>");
sb.append("<th>Customer id</th>");
sb.append("<th>Company name</th>");
sb.append("<th>Contact name</th>");
sb.append("<th>Job title</th>");
sb.append("<th>Address</th>");
sb.append("<th>Phone</th>");
sb.append("</tr>");
```

```
String row;
in.readLine(); // skip the header
while((row=in.readLine())!=null){
    String fields[] = row.split(",");
    sb.append("<tr>");
    for(int i=0; i<=3; i++){
        sb.append("<td>"+fields[i]+"</td>");
    }
    sb.append("<td>");
    for(int i=4; i<=8; i++){
        sb.append(fields[i]+"<br />");
    }
    sb.append("</td>");
    sb.append("<td>"+fields[9]+"</td>");
    sb.append("</tr>");
}
```

Some points

- The servlet finally renders the response to the user

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.print(sb.toString());
out.close();
```

Problem

- If this approach is followed by your application, the servlet tends to address these issues:
  - Working with the web container objects such as HttpServletRequest, HttpSession, etc (not applicable in the current example)
  - Executing the business logic/ data-access logic
    - Since the servlet runs only in the web container, how can we reuse the business or data-access logic in standalone applications?
    - If two or more servlets require the same business or data-access logic, how do we share the same?
    - If the data-access logic need to be replaced using a different technology such as JDBC or Spring-Data or Hibernate etc, where do we start modifying the application?

Problem

- If this approach is followed by your application, the servlet tends to address these issues (cont.):
  - Rendering the user interface
    - Suppose we would like to have a different servlet for searching customers based on some inputs (like city/country), how can we reuse the UI part of the servlet?
    - Small changes to the UI such as adding new CSS or JavaScript would need lot of tailoring, as embedding such code in a Java String is quite difficult.

Solution

- Separate the concerns
  - Model,
  - View and
  - Controller

Solution

- Model:
  - Data, Data-access logic, Business logic
  - Mostly reusable across different kinds of applications such as Web, Standalone, Enterprise – as these do not depend on any of those APIs
  - How?
    - Organize your classes into Entity, DAO and Service

Solution

- Model:
  - Entity classes represent pure data
    - Private fields, public constructor/s, public properties (getters/setters)
    - Optionally hashcode, equals, toString
    - No dependency on any APIs such as JDBC or Hibernate
    - Some part of your application (typically DAO layer) will create objects of entities or process the entities created by the UI
    - Not aware of the persistence mechanism (JDBC/ORM/File)

Solution

- Model:
  - DAO classes represent how to store the data in secondary storage and retrieve them
    - JDBC
    - ORM (JPA, Hibernate, Mybatis, JDO, and so many of them)
    - File API (Reader/Writer, Serialization)
    - XML serialization
    - JSON serialization

Solution

- Model:
  - Service classes represent business logic
    - For example, in an Order Processing System, a customer placing an order is not just saving an object into the secondary storage. It may involve the following steps:
      - Check the inventory if enough quantity of the requested product exists
      - Check for any dues from the customer
      - Add product/s to the line-item
      - Persist this information
      - Reduce the quantity in the stock
      - Check if it has to be re-ordered to the supplier
    - Each of these steps may involve one or more DAO activities.
    - Collectively all of them should be treated as a single operation
    - Some of the steps followed may also be required in other business operation
    - Service layer does not know about the persistence mechanism. They just use some DAOs.

Solution

- View:
  - If the data is made available, the view can render the user interface
  - Usually contains template code (static HTML/CSS/JS in a web application), which then bind the data in appropriate UI locations (such as DIV or TABLE etc).
    - For example, you can think of a HTML table displaying the customer list and an AWT/Swing applet displaying the same customer list in a Grid
  - There are many different view technologies available to Java
    - JSP, Facelets (XHTML), Velocity, Freemarker,
    - XML, JSON (in RESTful web services)

Solution

- Controller:
  - Since the model does not have a clue of how the data is rendered to the user, and view does not have an idea about how the data arrived into the memory, we need a mechanism to link these two components.
  - Controllers typically have the following workflow:
    - Check for any inputs from the user
    - Use a suitable service (which make use of DAO) to get the data required for the output
    - Store the data in a location that is accessible to the view
    - Invoke the view

Solution

- Controller:
  - In a web application, the best candidate for this role is a Servlet
    - Can check for user inputs using the Servlet API (HttpServletRequest, HttpSession, Cookie etc)
    - Can use any normal Java code such as instantiating or finding a Service object and invoking the necessary methods to get the data
    - Can store the data in any of these “scopes” – request (HttpServletRequest), session (HttpSession) or application (ServletContext)
      - which are accessible to any of the view technologies
    - Can forward to any of the web components (or URLs, both inside the container and outside of the domain)

MVC

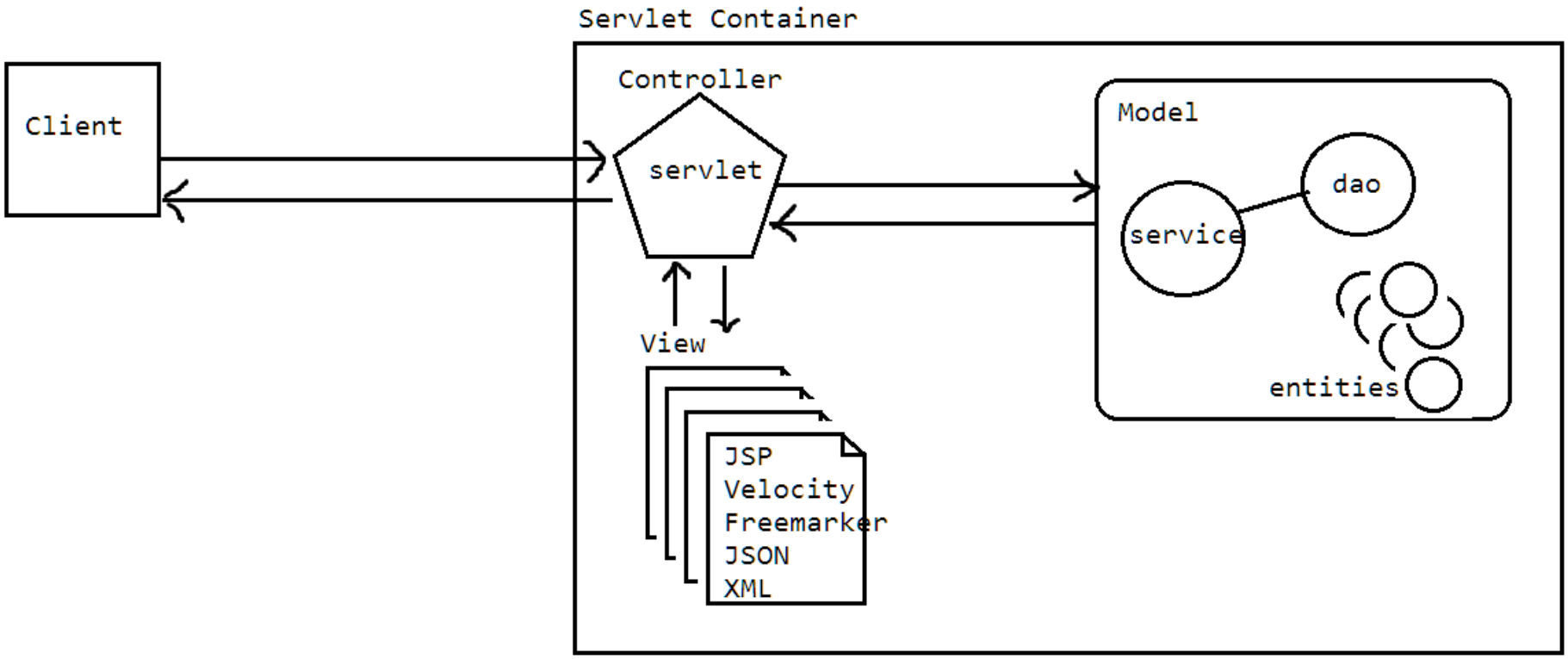
Model–view–controller is an architectural pattern for implementing user interfaces.

It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

MVC

- First described by Krasner and Pope in 1988
  - Was used to build user interfaces in Smalltalk-80

MVC


Servlet Container
Client
Controller
servlet
Model
dao
service
entities
View
JSP
Velocity
Freemarker
JSON
XML

How do we solve the problem in our app?

- Add Java files in the resources section to the appropriate packages

- Customer.java → entity package
- CustomerDaoFileImpl.java → dao package
- DaoException.java → dao package
- CustomerService.java → service package
- ServiceException.java → service package

How do we solve the problem in our app?

- Add

- displayCustomers.jsp → /WEB-INF/views folder

How do we solve the problem in our app?

- The ViewCustomerListServlet should now be modified to
  - Instantiate the service object
  - Get the list of customers using the service object
  - Store the list of customers in request scope
  - Forward to “/WEB-INF/views/displayCustomers.jsp”

Servlet - modified

```
String path = getServletContext().getRealPath("/WEB-INF/classes/customers.csv");

CustomerService service = new CustomerService();
service.setPath(path);

try {
        request.setAttribute("customers", service.getAllCustomers());
} catch (ServiceException e) {
        e.printStackTrace();
}

String forwardTo = "/WEB-INF/views/displayCustomers.jsp";
request.getRequestDispatcher(forwardTo).forward(request, response);
```

How can we reuse the view?

- Let’s take a use case where the user inputs a country to see the customers from that country
  - We can write another servlet to handle the user request, but once the data is available, can be passed to the same view.

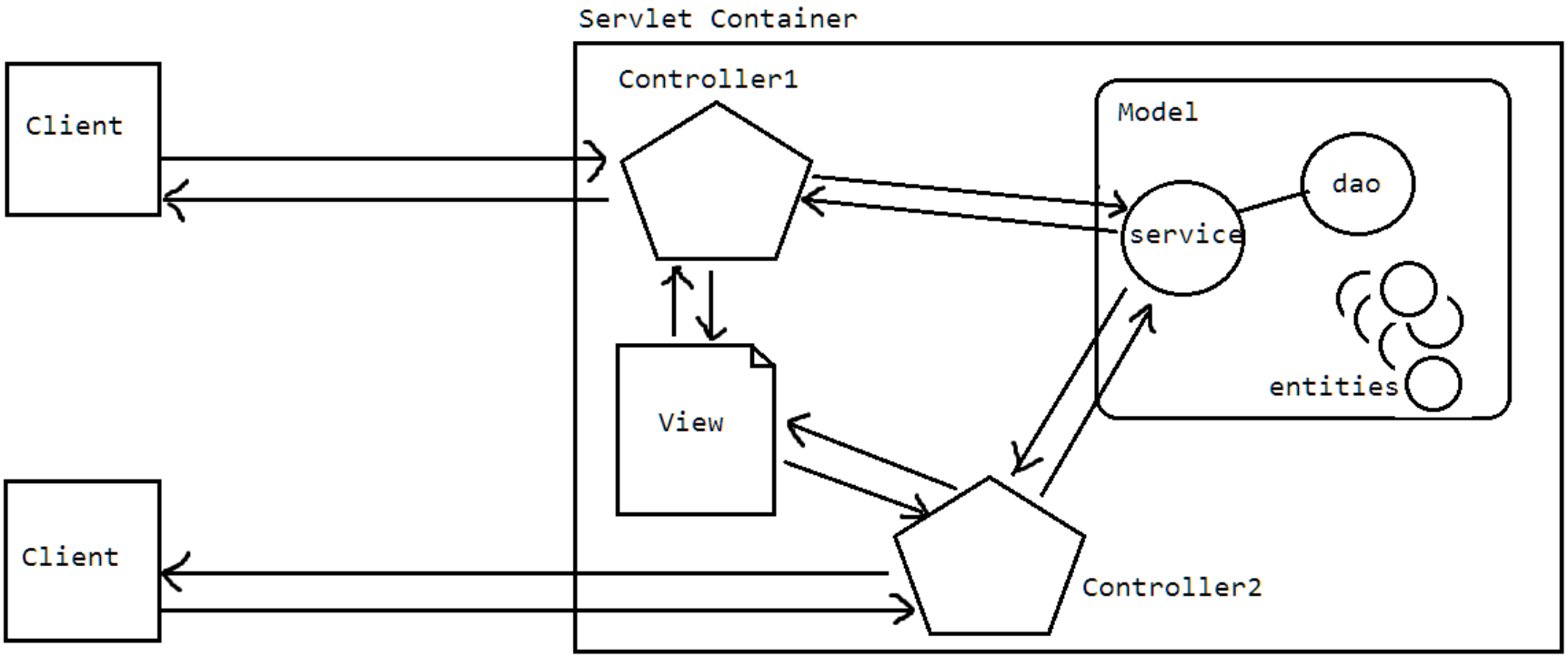
How do we do this in our app?

- Add these files from the “Resources” folder to appropriate folders.

- customersByCountry.jsp → “WebContent” folder
- ViewCustomersFromCountryServlet.java → “web” package

- Add a hyperlink in the “index.jsp” to “customersByCountry.jsp”
- Add the servlet mapping in web.xml for the servlet “ViewCustomersFromCountryServlet”

What did we achieve?



Some points

- The CustomerService class uses the following approach to get an instance of CustomerDaoFileImpl

```
public class CustomerService {

    private CustomerDaoFileImpl dao;

    public CustomerService() {
        dao = new CustomerDaoFileImpl();
    }

    public void setPath(String path) {
        dao.setPath(path);
    }
```

Some points

- This approach makes the two classes (service and dao) tightly copuled with each other.
  - What if the data need to be persisted into a DBMS instead of a file?
    - CustomerDaoJdbcImpl
    - CustomerDaoHibernateImpl
    - CustomerDaoJpaImpl

How do we address this?

- Separate the object creation (for dao) from the service layer.

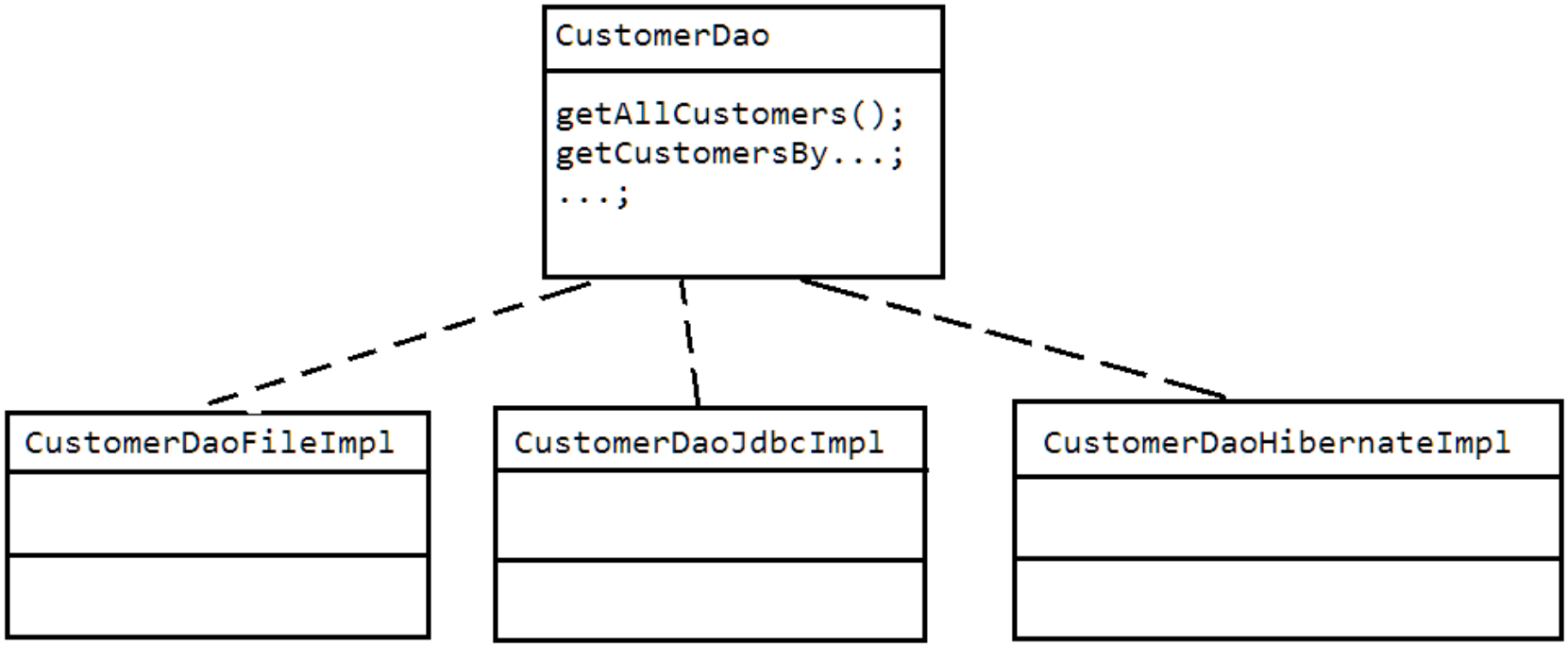
- Instead of

```
CustomerDaoFileImpl dao =
        new CustomerDaoFileImpl();
```

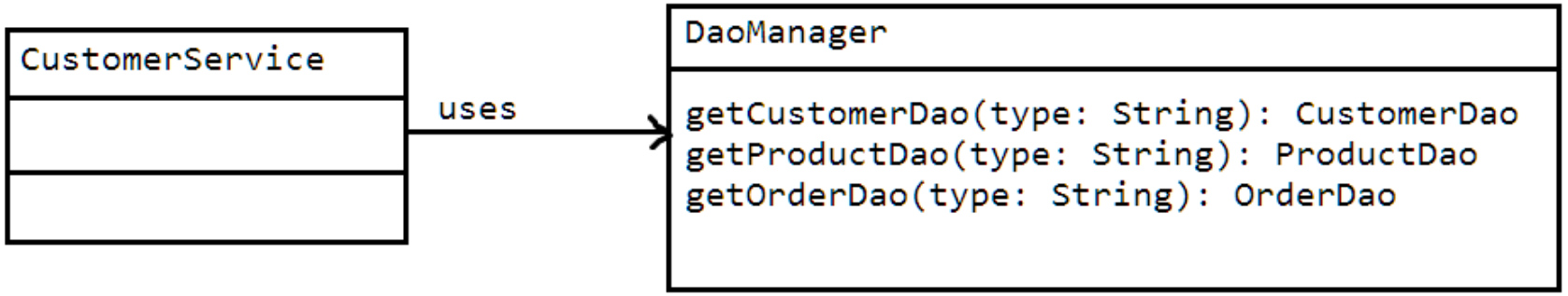
- Do this:

```
CustomerDao dao =
        DaoManager.getCustomerDao("jdbc")
```

How do we address this?



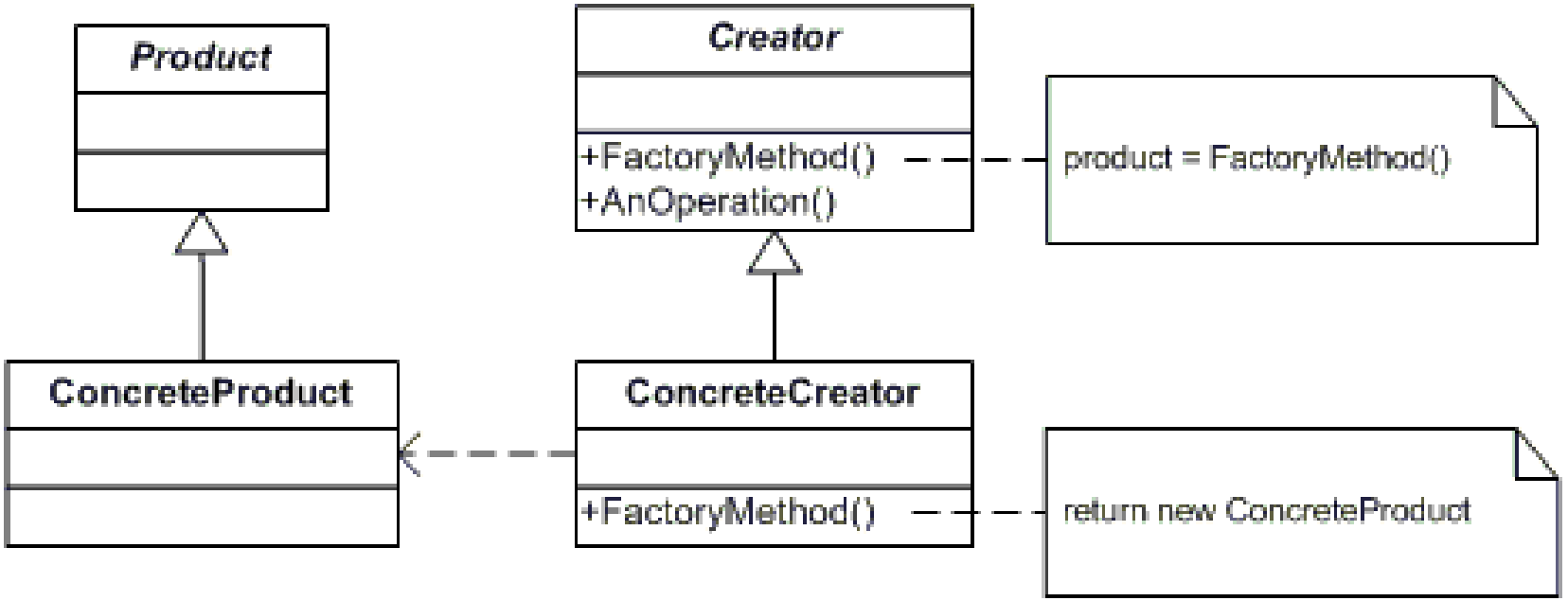
How do we address this?



Factory Method Pattern

- A.K.A
  - Virtual Constructor
- Use factory method pattern when
  - A class can’t anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
- Participants
  - Product (CustomerDao)
  - ConcreteProduct (CustomerDaoJdbcImpl)
  - Creator (DaoManager)
  - ConcreteCreator (DaoManager)
    - Can be a subclass of Creator to override the way ConcreteProduct be instatiated.

Factory Method Pattern



How do we solve the problem in our app?

- Copy the CustomerDao implementation classes, to the dao package
  - CustomerDaoFileImpl.java
  - CustomerDaoJdbcImpl.java
- Create a DaoManager to create and return an instance of CustomerDao****Impl

```
public class DaoManager {
        public static CustomerDao getCustomerDao(String type)
        throws DaoException{
                // based on "type", return an appropriate
                // CustomerDao****Impl instance
        }
}
```

How do we solve the problem in our app?

- In the CustomerService constructor, get an instance of the CustomerDao with the help of DaoManager's factory method

```
dao = DaoManager.getCustomerDao("file");
```

  - Possible values: jdbc, file

Where else can we apply this in our app?

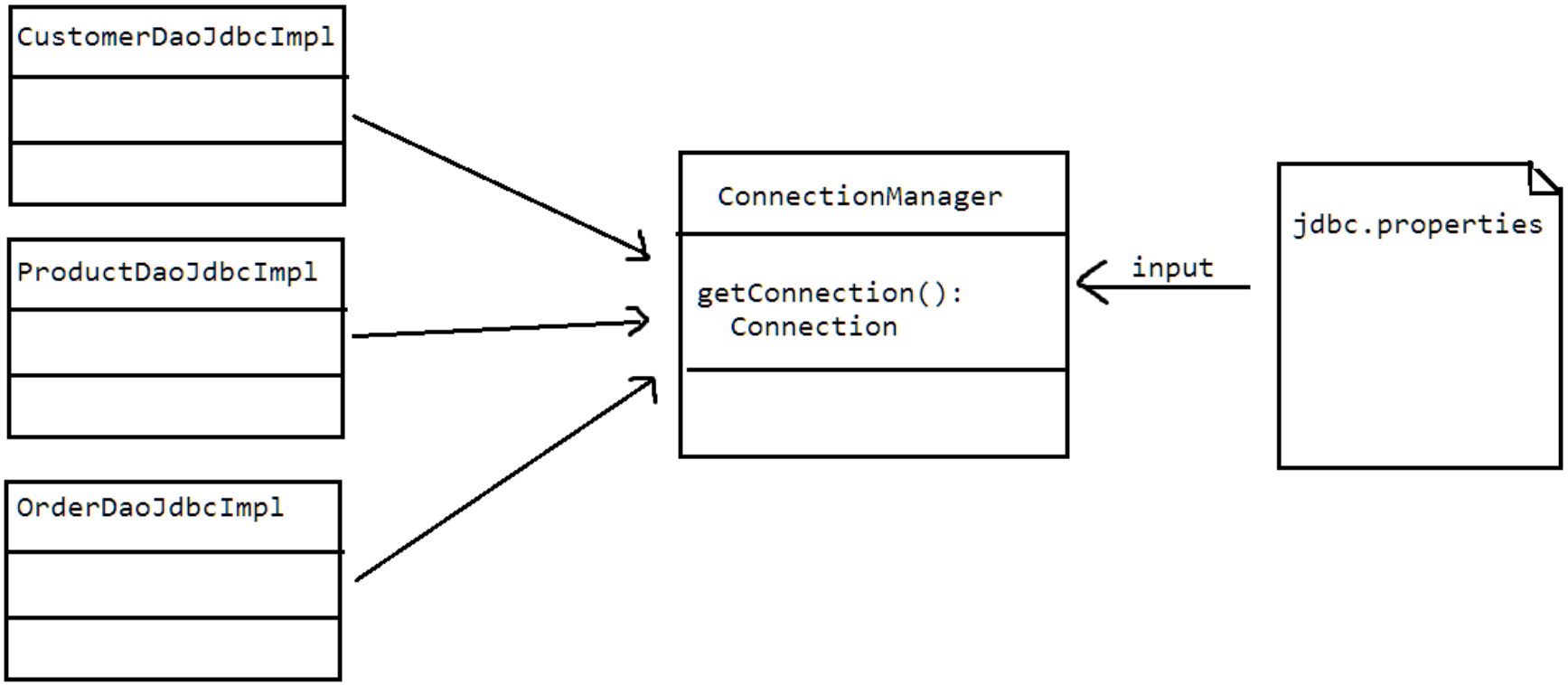
- The CustomerDaoJdbcImpl needs an instance of java.sql.Connection to persist data
- Currently, we have a private method to do the task:

```
private Connection getConnection() throws Exception {
        String driver, url, user, password;
        driver = "org.hsqldb.jdbcDriver";
        url = "jdbc:hsqldb:hsql://localhost/ops";
        user = "sa";
        password = "";
        Class.forName(driver);
        return DriverManager.getConnection(url, user, password);
}
```

Where else can we apply this in our app?

- If we have multiple DAO clasess, we may have to do this in each of those:
  - ProductDaoJdbcImpl
  - OrderDaoJdbcImpl
- Also, if the JDBC parameters change, we have to update in all of these implementations

Can we use factory-method pattern here?



Can we use factory-method pattern here?

- Create a class
  - ConnectionManager
- Add a static method
  - getConnection()
  - Returns java.sql.Connection
  - Reads the jdbc parameters from jdbc.properties file

Factory method patterin in Java API

- java.sql.DriverManager#getConnection
- java.util.Calendar#getInstance()
- java.util.ResourceBundle#getBundle()
- java.text.NumberFormat#getInstance()
- java.nio.charset.Charset#forName()
- java.net.URLStreamHandlerFactory#createURLStreamHandler(String)

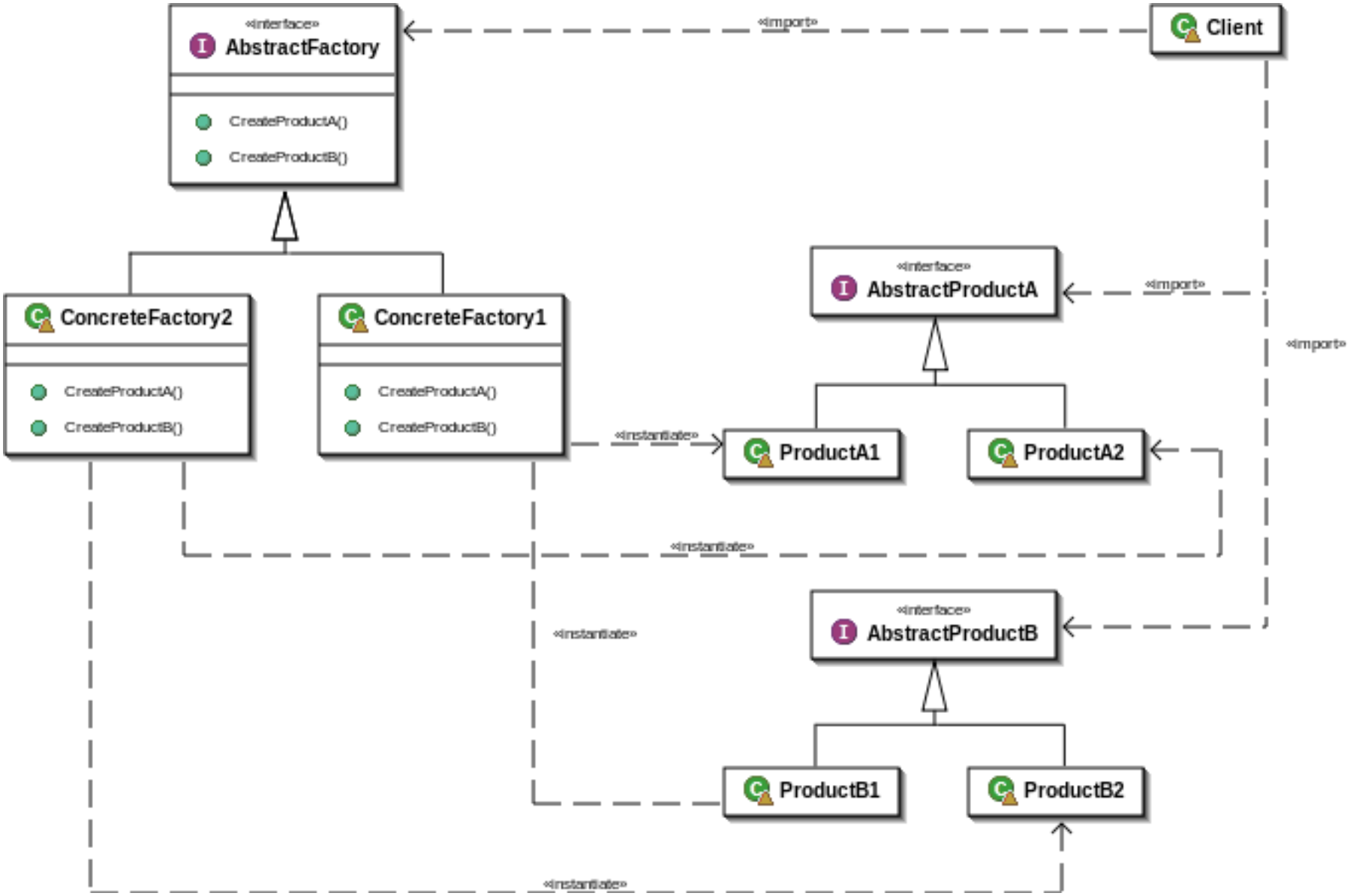
Some points

- Suppose we have multiple DAO interfaces and for each, two types of implementaions (file, jdbc)
  - Many of the service methods would have refered to many of the DAOs
    - Everytime we need a DAO, we have to specify what kind of DAO we need
    - Makes it hard to switch the implementations across all service classes

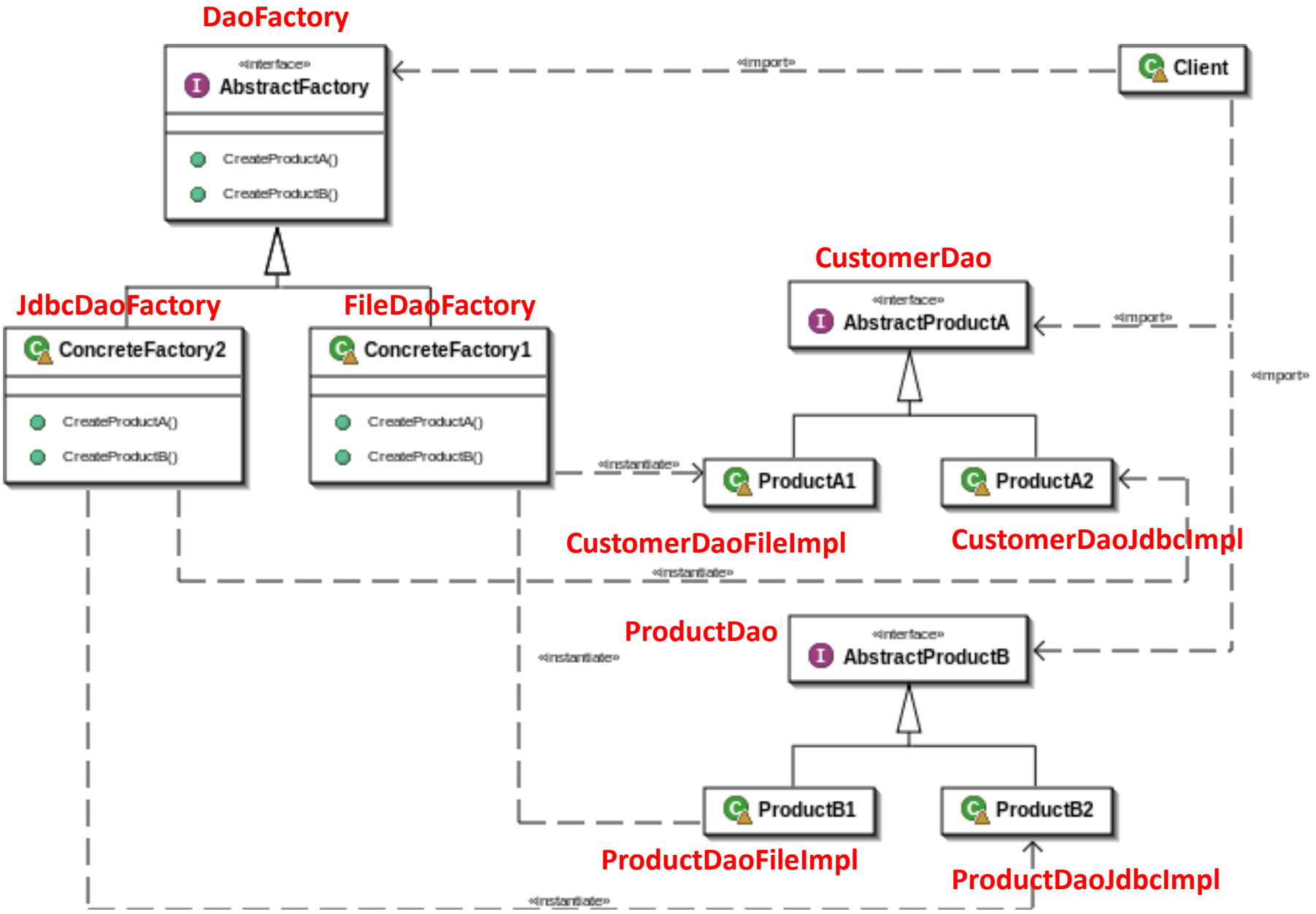
Some points

- Is there a way to toggle a switch at one location, and all DAO instances will be switched?
  - Abstract Factory pattern

Big picture


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«instantiate»
«interface»
AbstractProductA
«import»
«import»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»

Mapped to our app



Abstract Factory patterin in Java API

- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()
- org.hibernate.SessionFactory
- Spring’s BeanFactory and ApplicationContext (implementations)

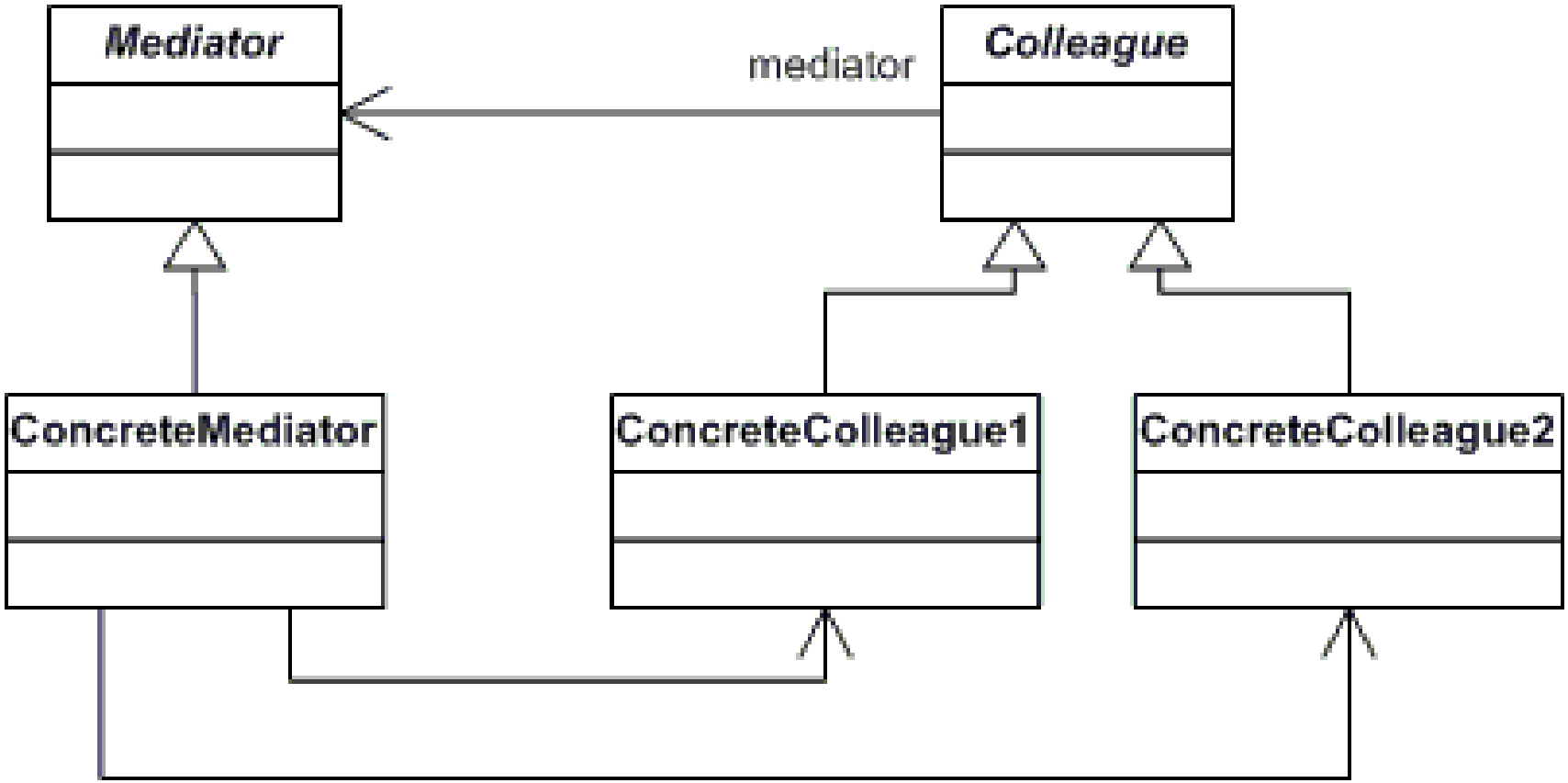
Mediator Pattern

Intent:

Define an object that encapsulate how a set of objects interact.

Mediator promtes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Structure


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

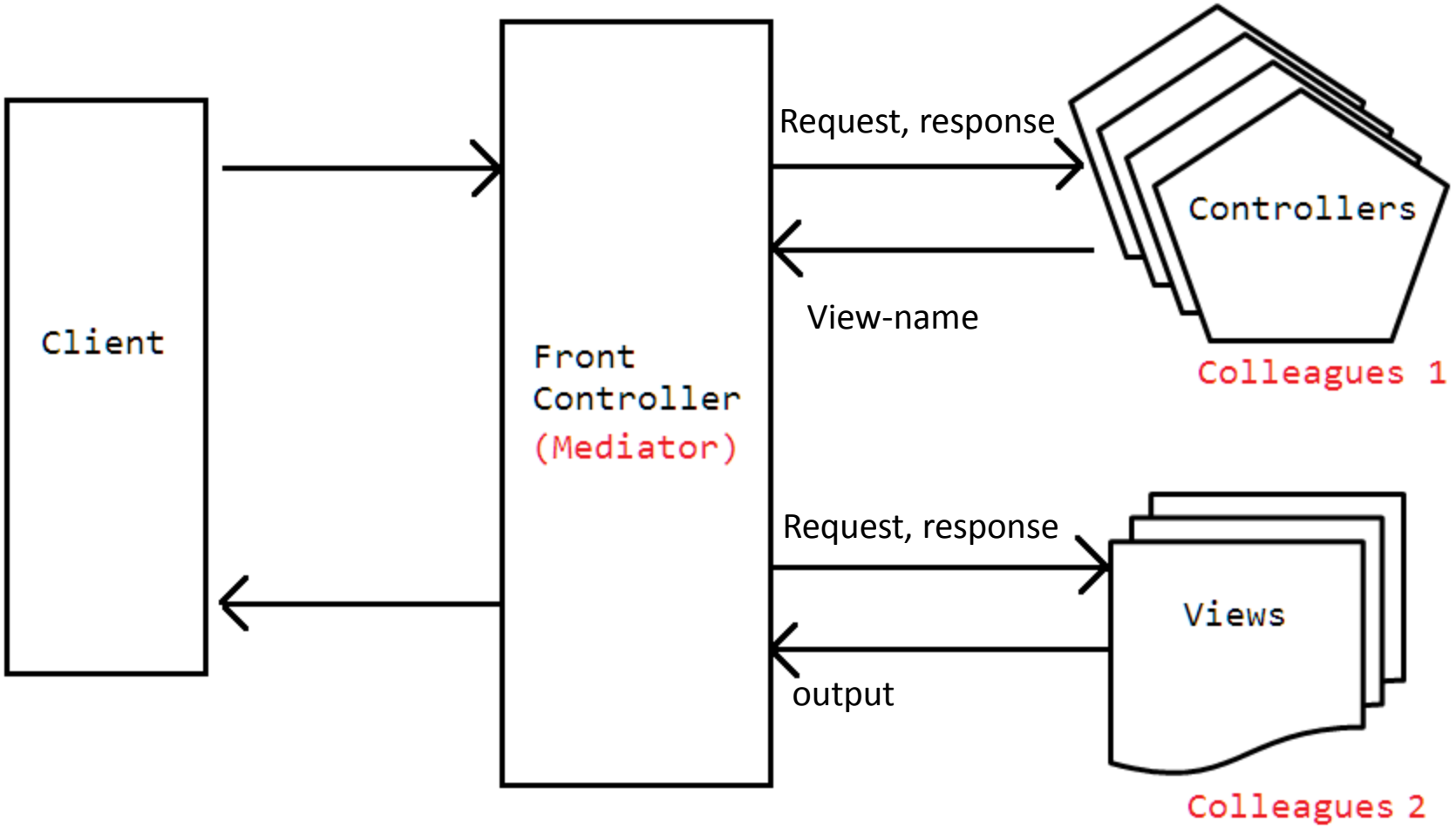
Participants

- Mediator
  - Defines an interface for communicating with colleague objects
- ConcreteMediator
  - Implements cooperative behavior by coordinating colleague objects
  - Knows and maintains its colleagues
- Colleagues
  - Each colleague knows it’s mediator object
  - Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

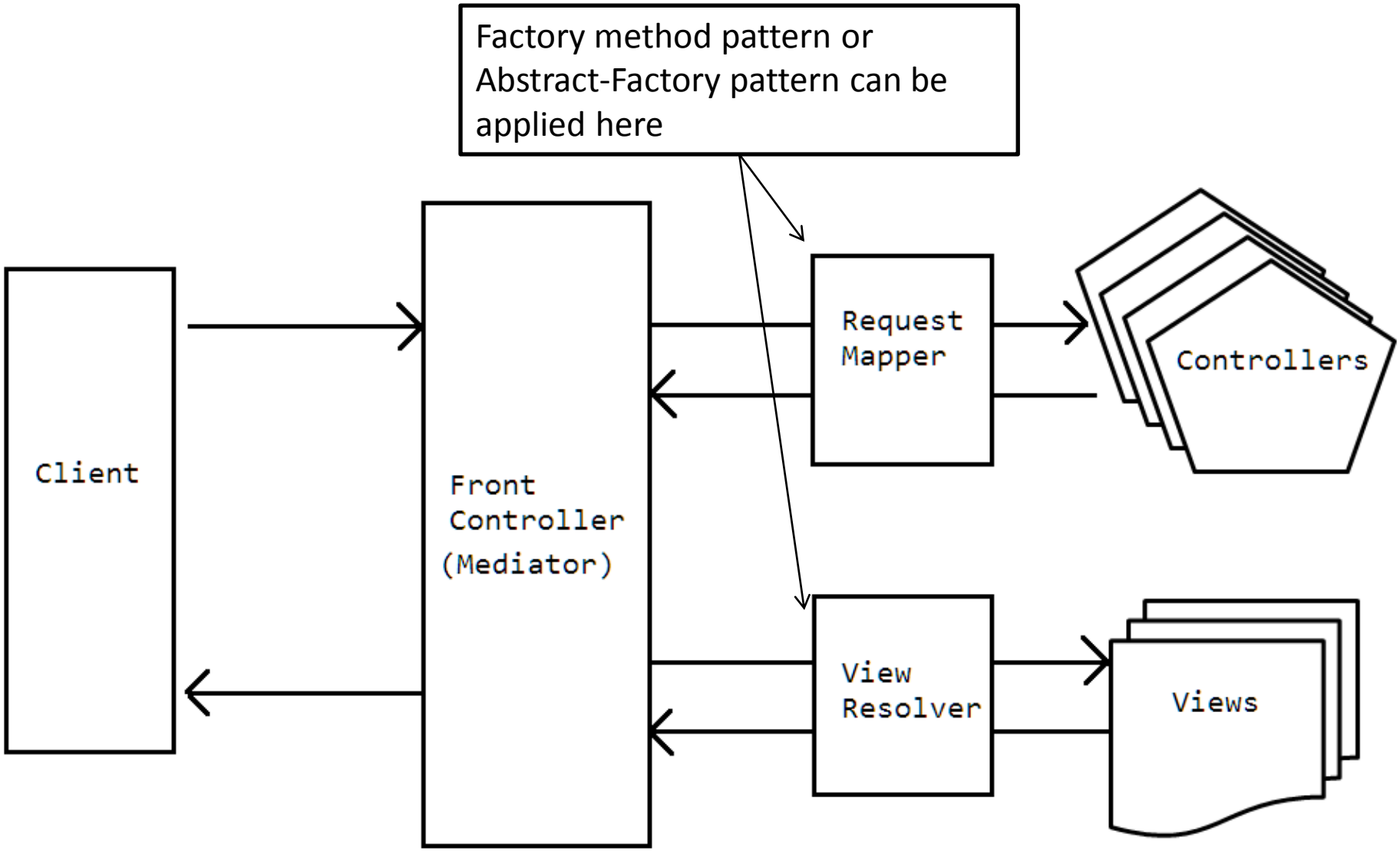
Omitting the abstract Mediator class

There is no need to define an abstract Mediator class, when colleagues work with only one mediator.

Use of mediator in our app



Helper classes for mediator



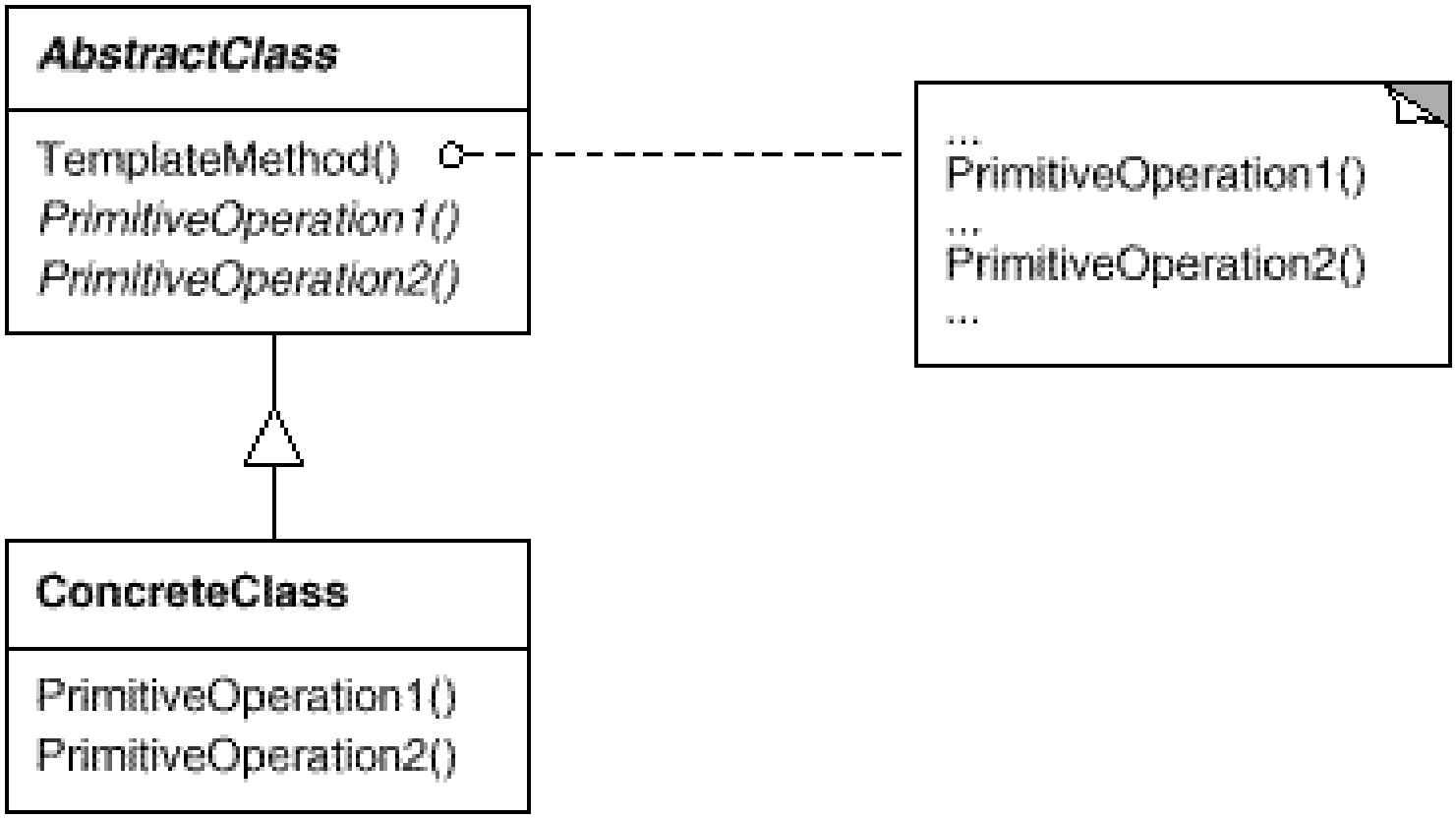
Mediator pattern in Java API

- java.util.Timer (all scheduleXXX() methods)
- java.util.concurrent.Executor#execute()
- java.util.concurrent.ExecutorService
  - (the invokeXXX() and submit() methods)
- java.util.concurrent.ScheduledExecutorService
  - (all scheduleXXX() methods)
- java.lang.reflect.Method#invoke()

Template pattern

- Intent:
  - Define the skeleton of an algorithm in an operation, deffering some steps to subclasses
  - Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

Template pattern - structure



Template pattern - Participants

- AbstractClass
  - Defines abstract primitive operations that concrete subclasses deine to implement steps of an algorithm
  - Implements a template method defining the skeleton of an algorithm
- ConcreteClass
  - Implements the primitive operations to carry out subclass specific steps of the algorithm

Template pattern in Java API

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.

- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

Strategy pattern

- Intent:
  - Define a family of algorithms, enapsulate each of them and make them interchangable.
  - Strategy lets the algorithm vary independently from clients that use it.

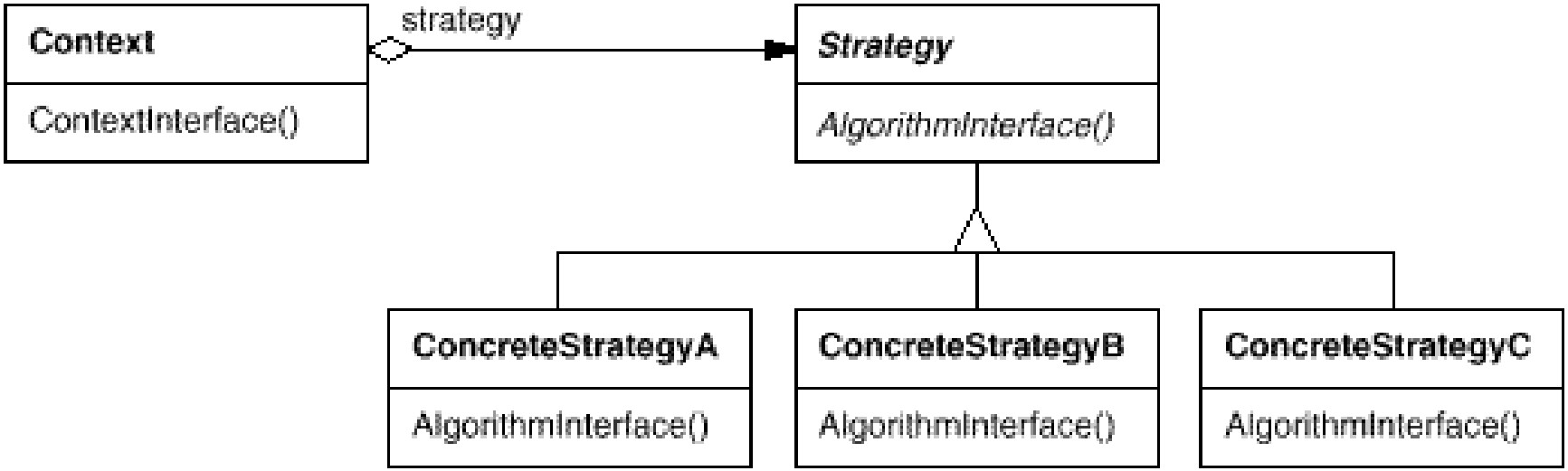
Strategy pattern

- AKA:
  - Policy

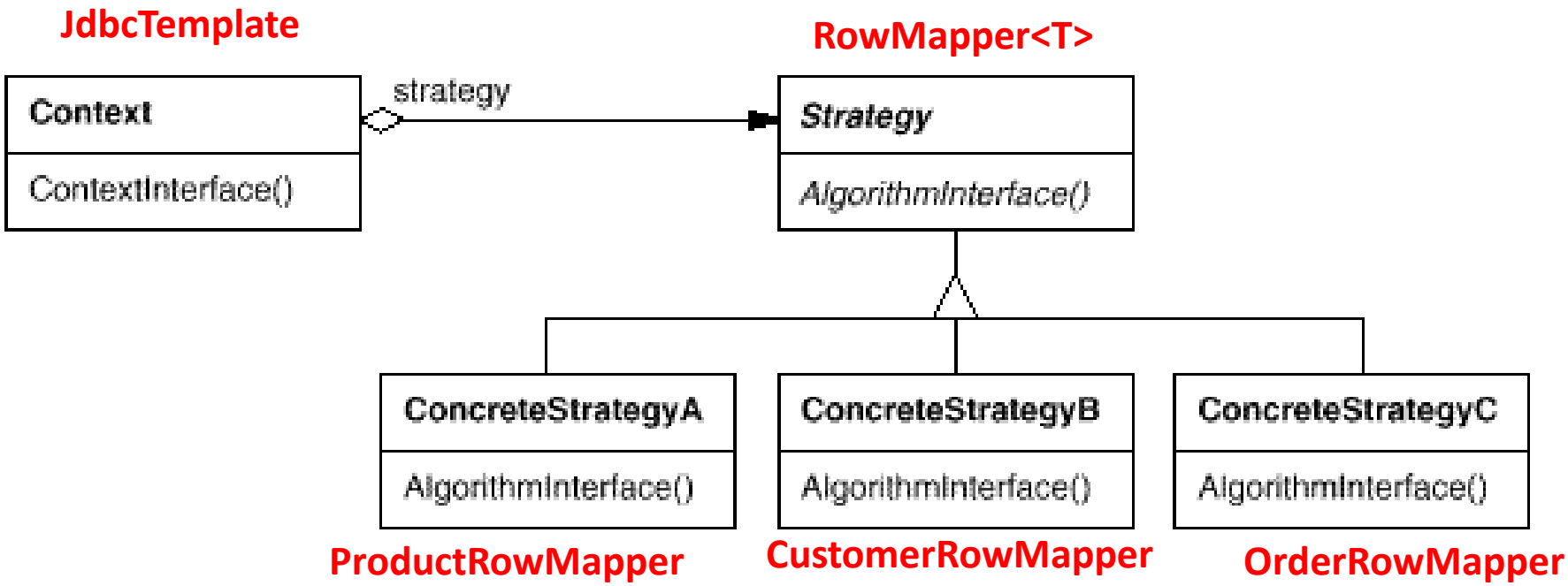
Strategy pattern

- Applicability:
  - Many related classes differ only in their behavior.
  - Different varients of an algorithm.
  - An algorithm uses data that the client shouldn’t know about.

Strategy pattern



Strategy pattern in our app



Participants

- Strategy (Compositor)
  - Declares an interface common to all supported algorithms.
  - Context uses this interface to call the algorithm defined by a ConcreteStrategy
  - Eg, In our app, dao.RowMapper interface defines the contract method for converting a record in a ResultSet into an entity object. This logic vary from entity to entity.
- ConcreteStrategy
  - Implements the algorithm using the Strategy interface
  - Eg, ProductRowMapper, CustomerRowMapper, etc.
- Context (Composition)
  - Is configured with ConcreteStrategy object
  - Maintains a reference to the Strategy object
  - Eg, dao.JdbcTemplate class defines functions like get, find that need a Strategy to convert a row into an entity.

Strategy pattern in Java API

- java.util.Comparator#compare(), executed by among others Collections#sort().

- javax.servlet.http.HttpServlet, the service() and all doXXX() methods take HttpServletRequest and HttpServletResponse and the implementor has to process them (and not to get hold of them as instance variables!).

Chain of responsibility pattern

Intent:

Avoid coupling the sender of a request to its receiver by giving more than one object a change to handle the request.

Chain the receiving objects and pass the request along the chain until an object handles it.

Some examples

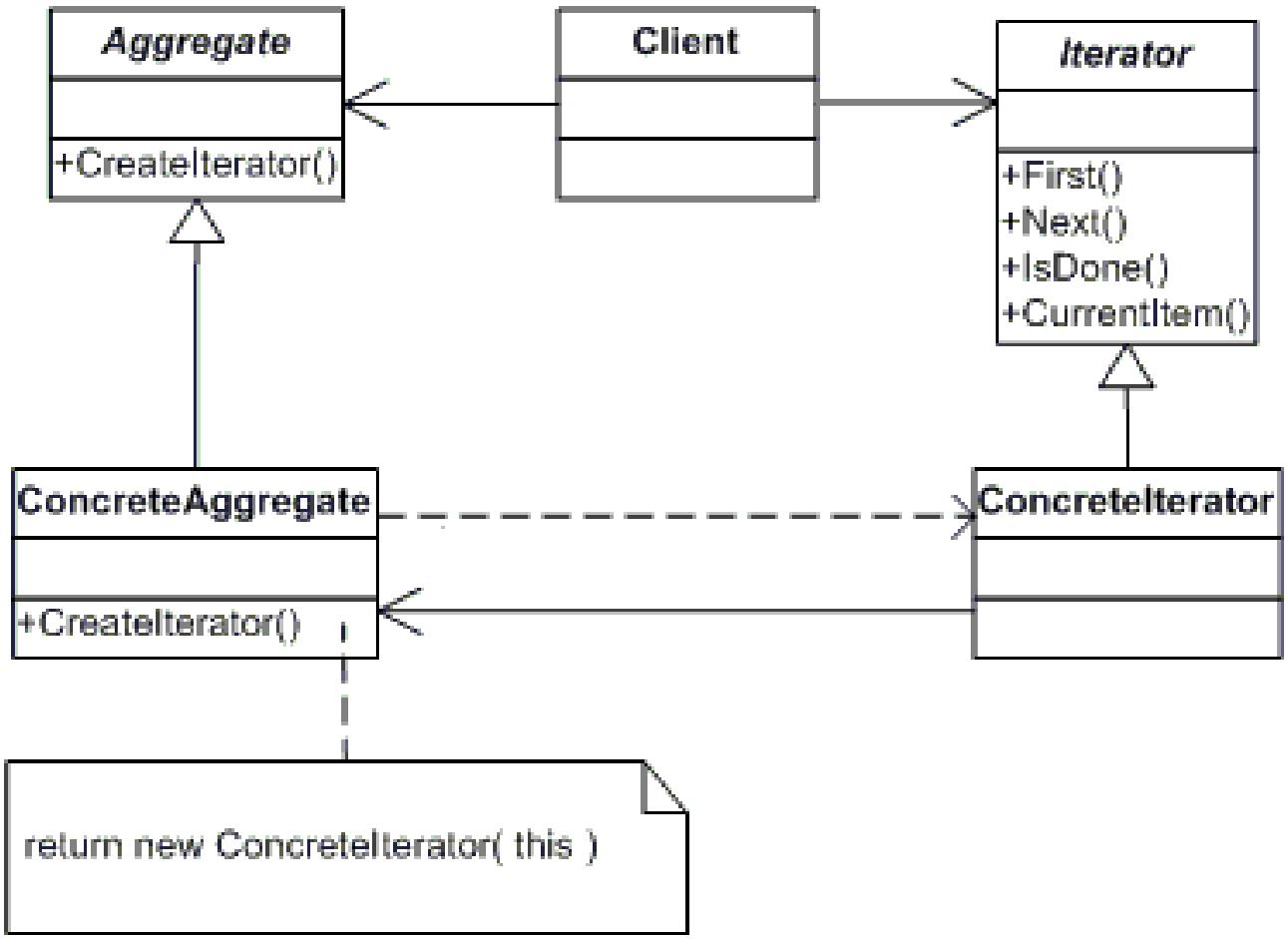
- When an object is constructed, a series of constructors are called until the constructor for java.lang.Object is called.
  - SubClass() → Parent() → GrandParent() → Object()
- PATH environment variable in operatig system
- Exception funnelling across multiple layers of an application
  - SqlException → TemplateException → DaoException → ServiceException → ControllerException
- Servlet filters
- Logger
- Struts 2 interceptors

Iterator pattern

Intent:

Provide an object which traverses some aggregate structure, abstracting away assumptions about the implementation of that structure.

Iterator pattern - structure



Iterator pattern

The simplest iterator has a "next element" method, which returns elements in some sequential order until there are no more.

More sophisticated iterators might allow several directions and types of movement through a complex structure.

Iterator pattern

- Typically an iterator has three tasks that might or might not be implemented in separate methods:
  - Testing whether elements are available
  - Advancing to the next n.th position
  - Accessing the value at the current position

Iterator pattern in Java API

- All implementations of java.util.Iterator (thus among others also java.util.Scanner!).
- All implementations of java.util.Enumeration

Command pattern

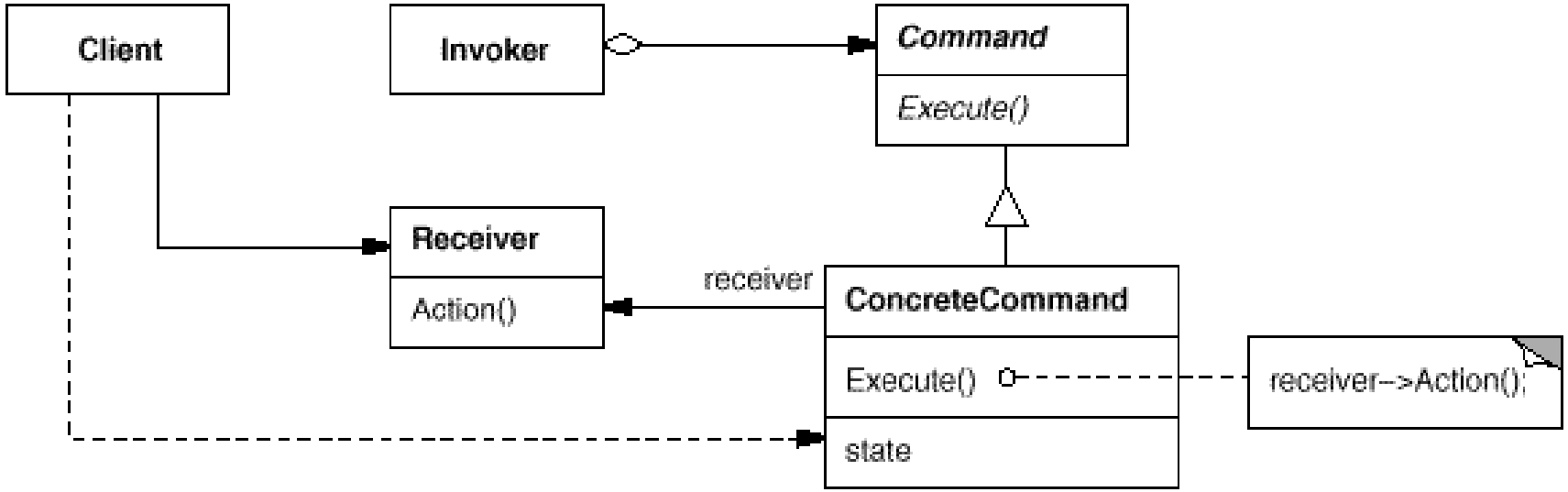
Intent:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queues or log requests.

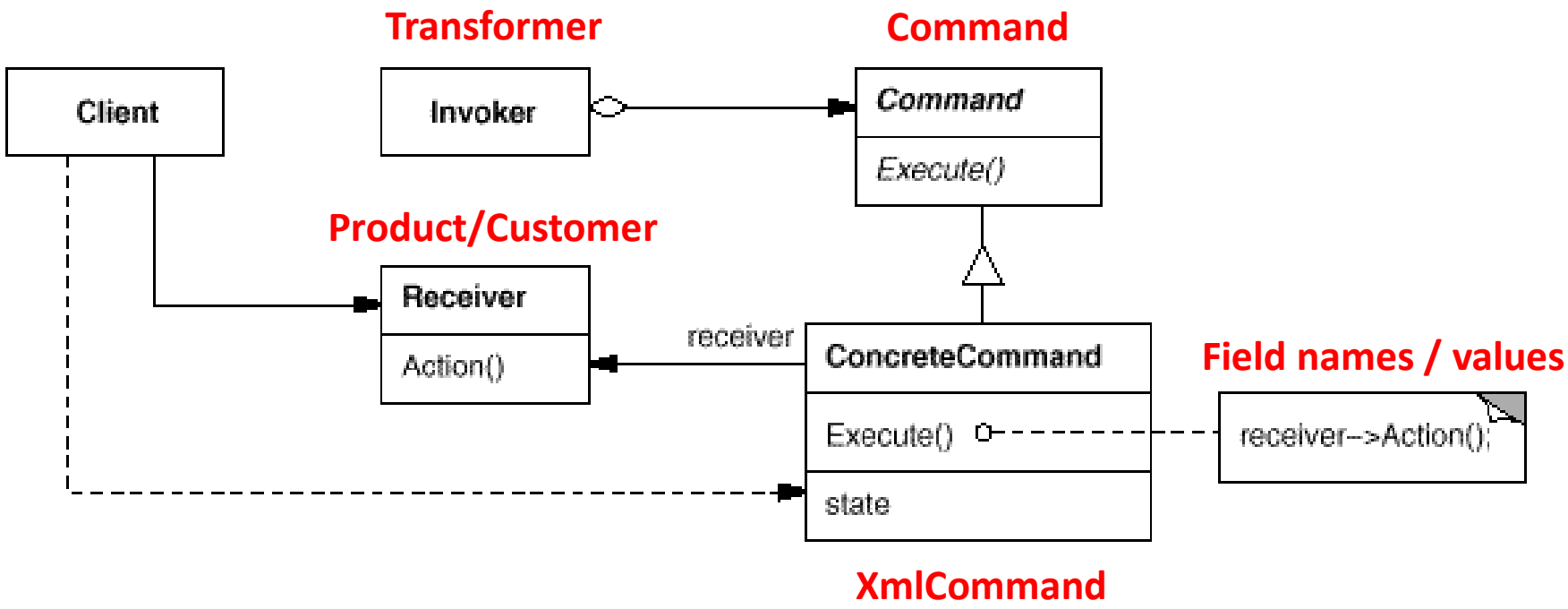
AKA:

Action, Transaction

Command pattern - structure



Command pattern - structure



Command pattern

- Participants:
  - Command
    - Declares an interface for executing operations
  - ConcreteCommand
    - Defines a binding between a receiver object and an action
    - Implements an “execute” method by invoking corresponding operation/s on the receiver
  - Client
    - Creates ConcreteCommand object and sets its receiver
  - Invoker
    - Asks the command to carry out the request
  - Receiver
    - Knows how to perform the operations associated with carrying out a request.

Command pattern

- You can have a series commands added to a collection (script) and use it as a macro
  - An application (GUI) may provide an option to record commonly used user inputs and store it as a macro for future/repeted use.

Command pattern in Java API

- All implementations of java.lang.Runnable
- All implementations of javax.swing.Action

Memento

Intent:

Without violating encapsulation, capture and externalize an object's internal state so that object can be restored to this state later.

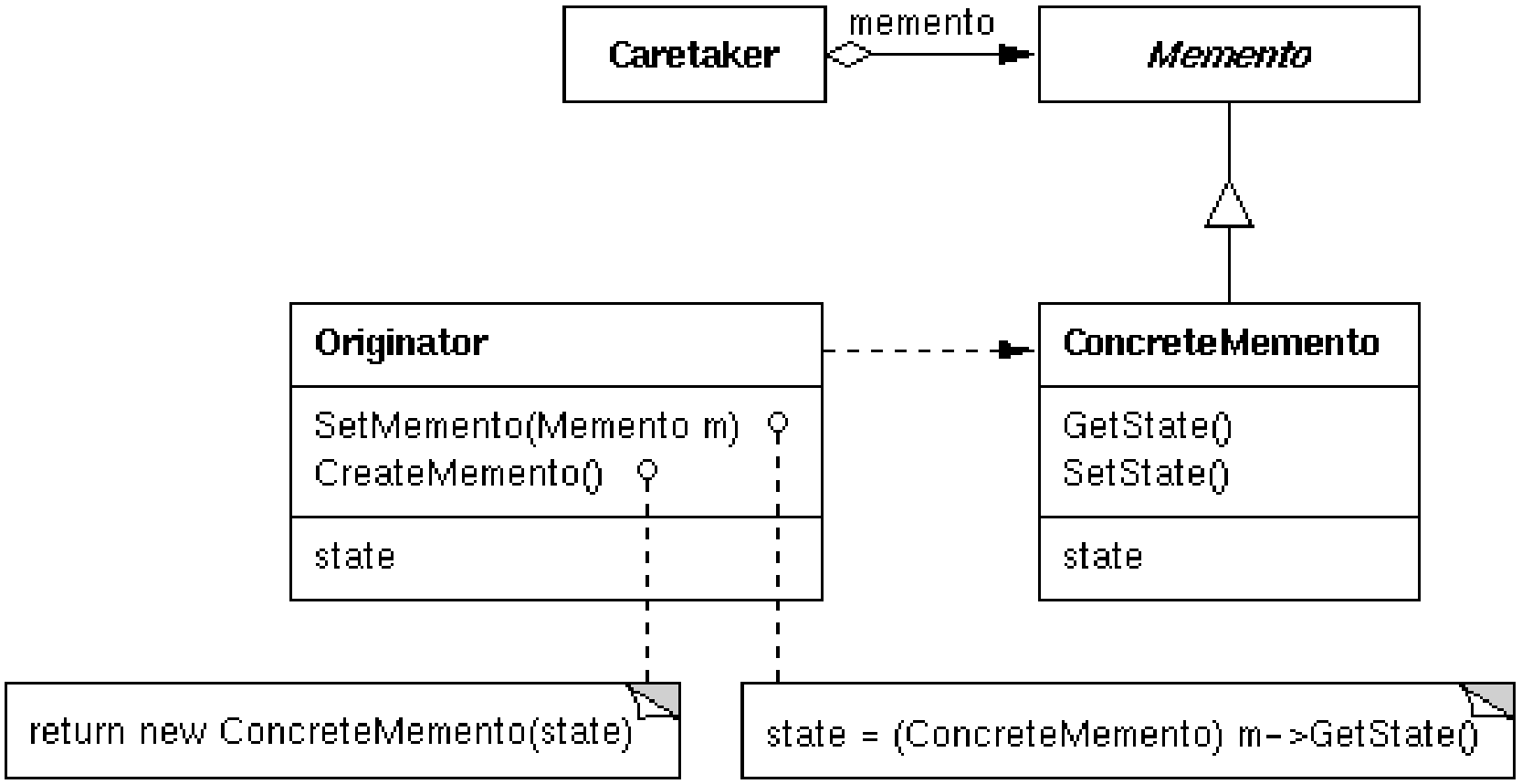
AKA:

Token

Memento

- Applicability:
  - Use the Memento pattern when
    - A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
    - A direct interface to obtaining the state would expose implmentation details and break the object's encapsulation.

Memento - structure



Memento

- Participants:
  - Memento
    - Stores internal state of the originator object
    - The memento may store as much as or as little of the originator's internal state as necessary at its originator's descrition
  - Originator
    - Creates a memento containing a snapshot of its current internal state
    - Uses the memento to restore it's internal state
  - Caretaker
    - Responsible for the memento's safekeeping
    - Never operates on or examines the contents of a memento

Memento in Java API

- java.util.Date (the setter methods do that, Date is internally represented by a long value)
- All implementations of java.io.Serializable
- All implementations of javax.faces.component.StateHolder

Proxy pattern

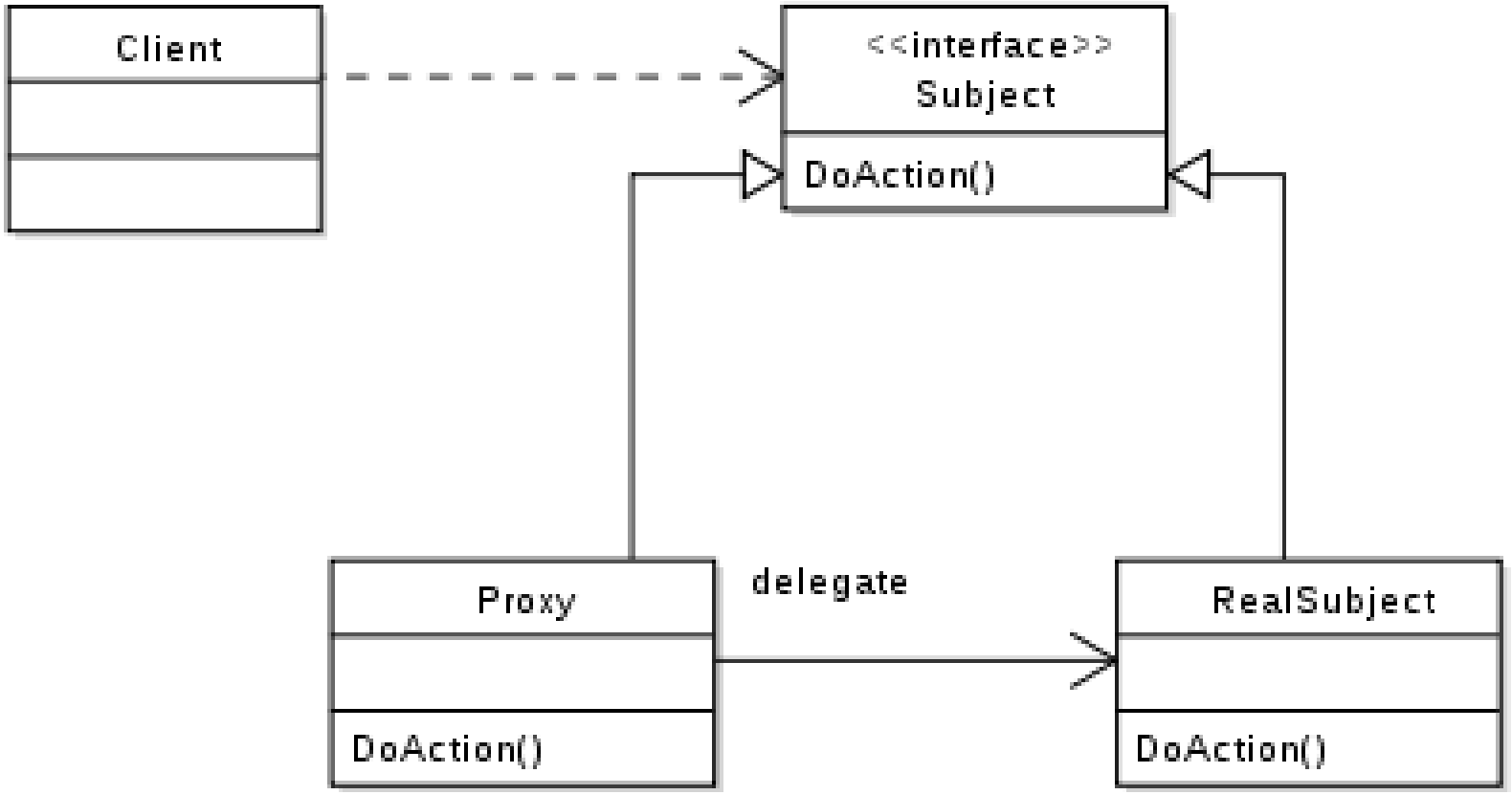
Intent:

Provide a surrogate or placeholder for another object to control access to it.

AKA:

Surrogate

Proxy pattern - structure



Proxy pattern

- Simply speaking, a Proxy object is one through which we control access to the actual object on which the functionality lies.

- Depending on the context in which the Proxy object is used, the Pattern is broadly divided into the following 3 types:
  - Virtual Proxy,
  - Remote Proxy and
  - Access Proxy

Proxy pattern

- Virtual Proxy:
  - Used for Lazy instantiation of objects or for Lazy processing.
  - Suppose you need to support 'resource-hungry' objects that involve high amount of I/O or one those that involved a database transaction.
  - One need not instantiate these objects until they are really required.
  - The real object would get created only when the client actully requests for some of its functionality.
  - This is LazyInstantiationPattern.

Proxy pattern

- Remote Proxy:
  - Used to hide the communication mechanisms between remote objects.
  - In RMI, for example, we have the stubs which act as Remote Proxies for the Skeleton.
  - This is RemoteProxy.

Proxy pattern

- Access Proxy:
  - Used to provide control over a sensitive master object.
  - This proxy object could check for the client's access permission before allowing methods to be executed on the actual object.
  - This is ProtectionProxy.

Proxy pattern in Java API

- java.lang.reflect.Proxy
- java.rmi.*, the whole API actually.

Composite pattern

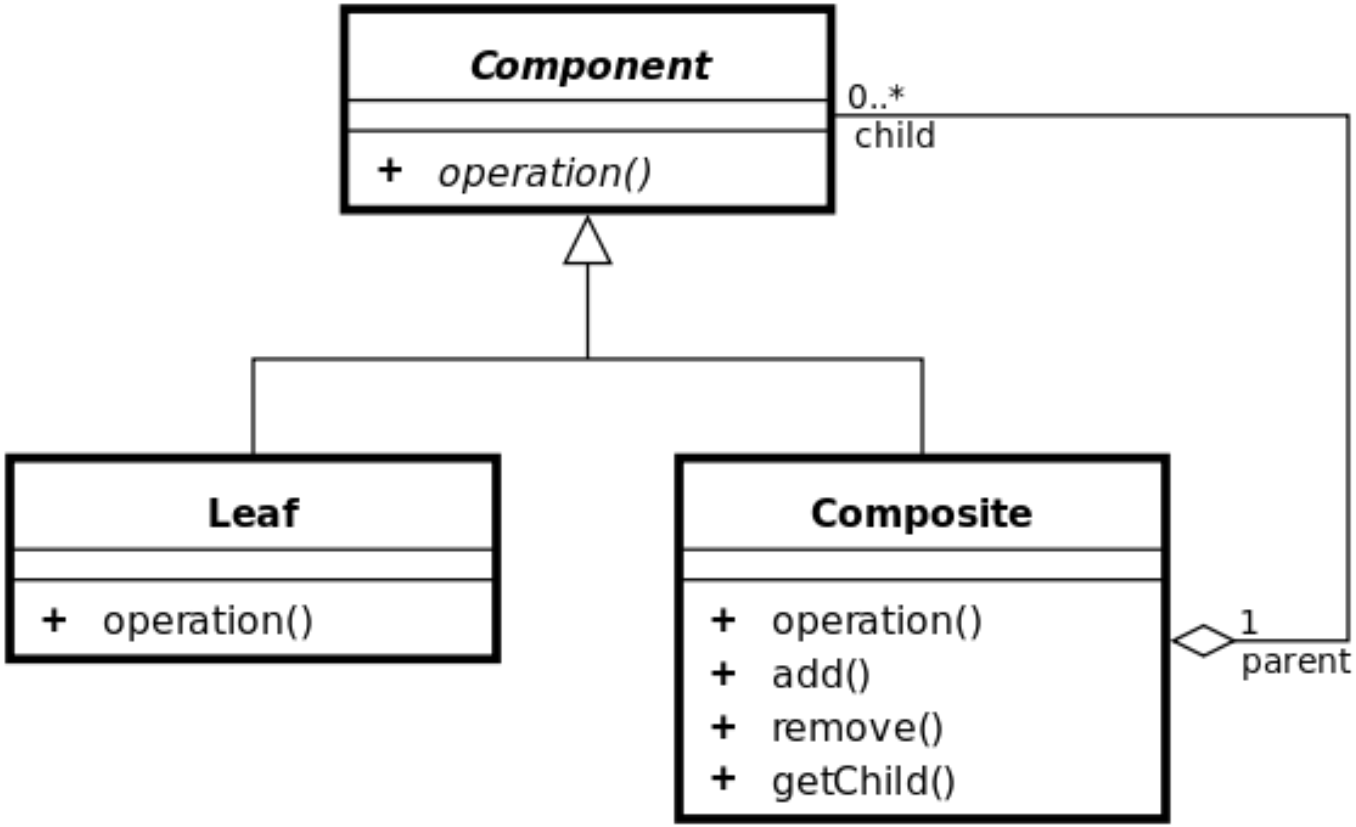
Intent:

Compose objects into tree structures that represent whole-part hierarchies.

Composite lets clients treat individual objects and compositions of objects uniformly.

A leaf has the same interface as a node.

Composite pattern - structure



Composite pattern

- Applicability:
  - Use composite pattern when
    - You want to represent part-whole hierarchies of objects

    - Customer has one or more Order objects
    - Each Order has one or more LineItem objects
    - LineItem is composed of a Product

    - If Customer, Order and LineItem implement “Printable” interface,
      - customer1.print() should print customer data + orders + line-items
      - order1.print() should print order data + line-items

    - You want clients to ignore the difference between compositions of objects and individual objects

Composite pattern

- Participants:
  - Component (Printable)
    - Declares the interface for objects in the composition
    - Delcares an interface for accessing and managing its child components
  - Leaf (LineItem, Product)
    - Represents leaf objects in the composition; has no children.
    - Defines behavior for primitive objects in composition
  - Composite (Customer, Order)
    - Defines behavior for primitive objects in the composition
    - Stores child components
    - Implements child related operations in the component interface.

Composite pattern in Java API

- java.awt.Container#add(Component) (practically all over Swing thus)
- javax.faces.component.UIComponent#getChildren() (practically all over JSF UI thus)

Flyweight pattern

Intent:

Use sharing to support large numbers of fine-grained objects efficiently

Flyweight pattern

- The FlyweightPattern describes how to support a large number of fine grained objects efficiently, by sharing commonalities in state.
  - For example, when designing a word processor application, you might create an object for each character typed.
  - Each Character object might contain information such as the font face, size and weight of each character.
  - The problem here is that a lengthy document might contain tens of thousands of characters, and objects - quite a memory killer!

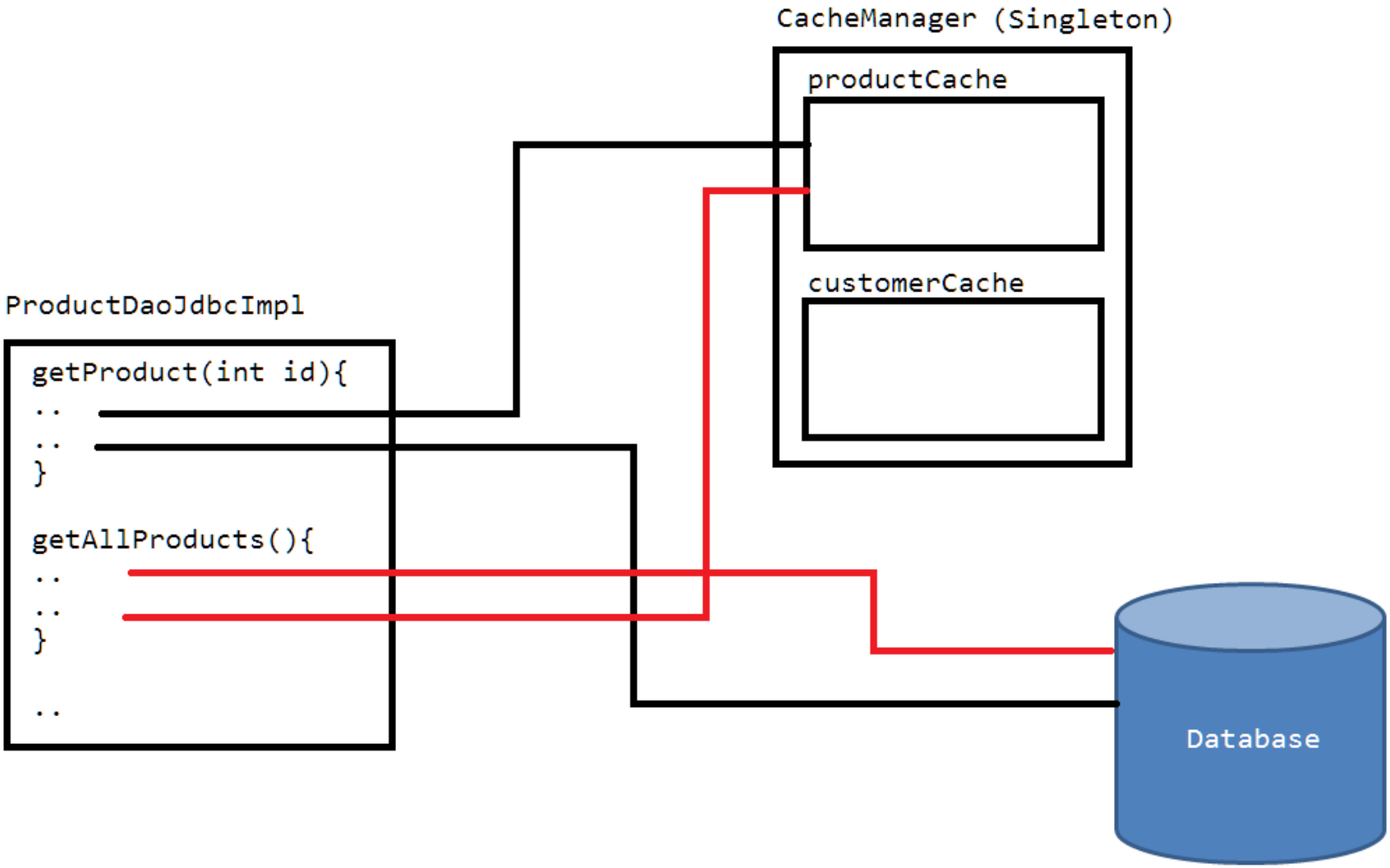
Flyweight pattern

- The Flyweight pattern addresses the problem by creating a new object to store such information, which is shared by all characters with the same formatting.
- So, if I had a ten-thousand word document, with 800 characters in Bold Times-New-Roman, these 800 characters would contain a reference to a flyweight object that stores their common formatting information.
- The key here is that you only store the information once, so memory consumption is greatly reduced.

Flyweight in Java API

- java.lang.String (a cache called “intern”)
- java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short and Long)

Flyweight in our case study?



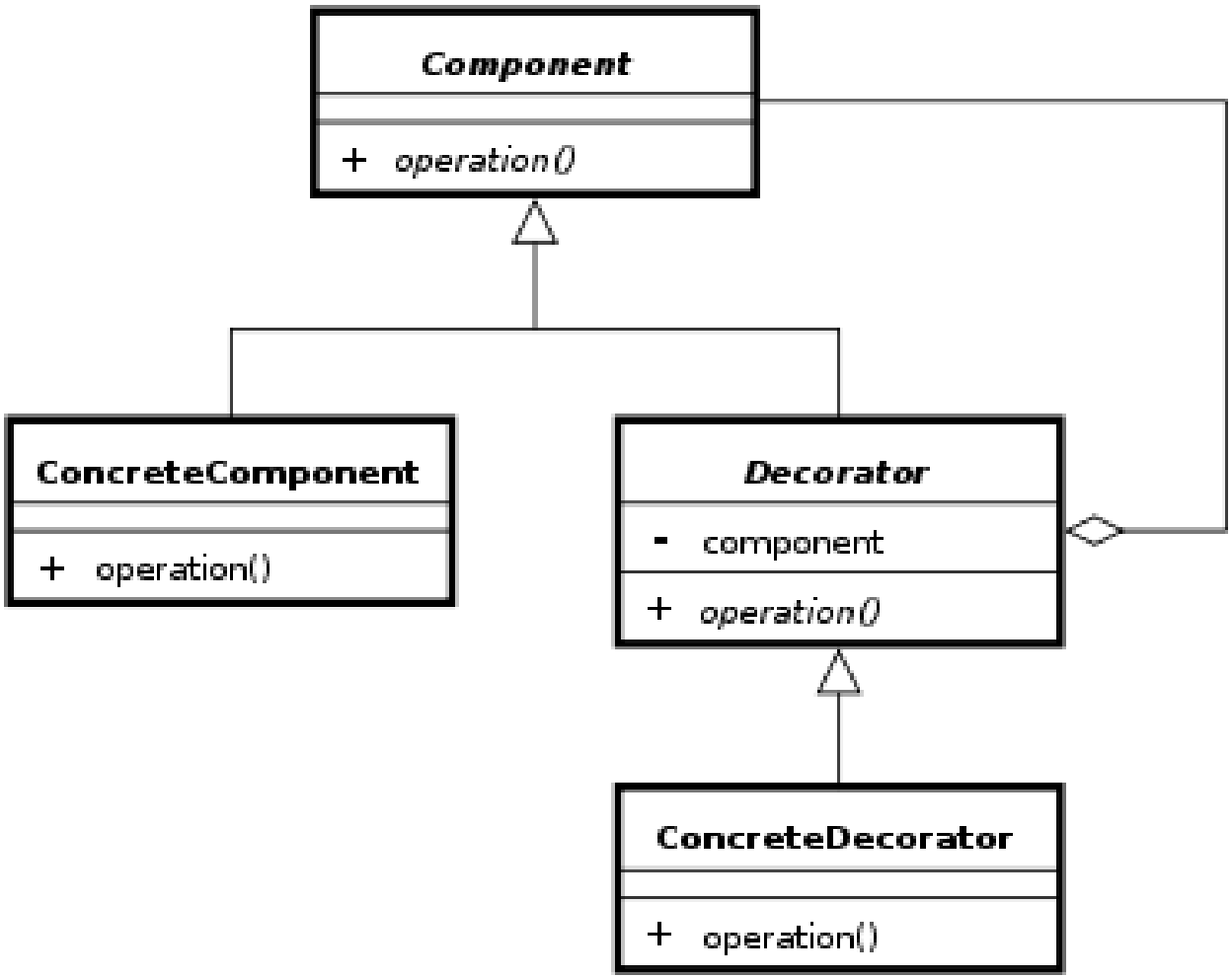
Decorator pattern

Intent:

Attach additional responsibilities to an object dynamically.

Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator pattern - structure



Decorator pattern

- This pattern can also be used as a way to refactor a complex class into smaller pieces.

- Even if you don't need to attach responsibilities dynamically it can be clearer to have each responsibility in a different class.

Decorator patterin in Java API

- All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.
- java.util.Collections, the checkedXXX(), synchronizedXXX() and unmodifiableXXX() methods.
- javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

Adapter pattern

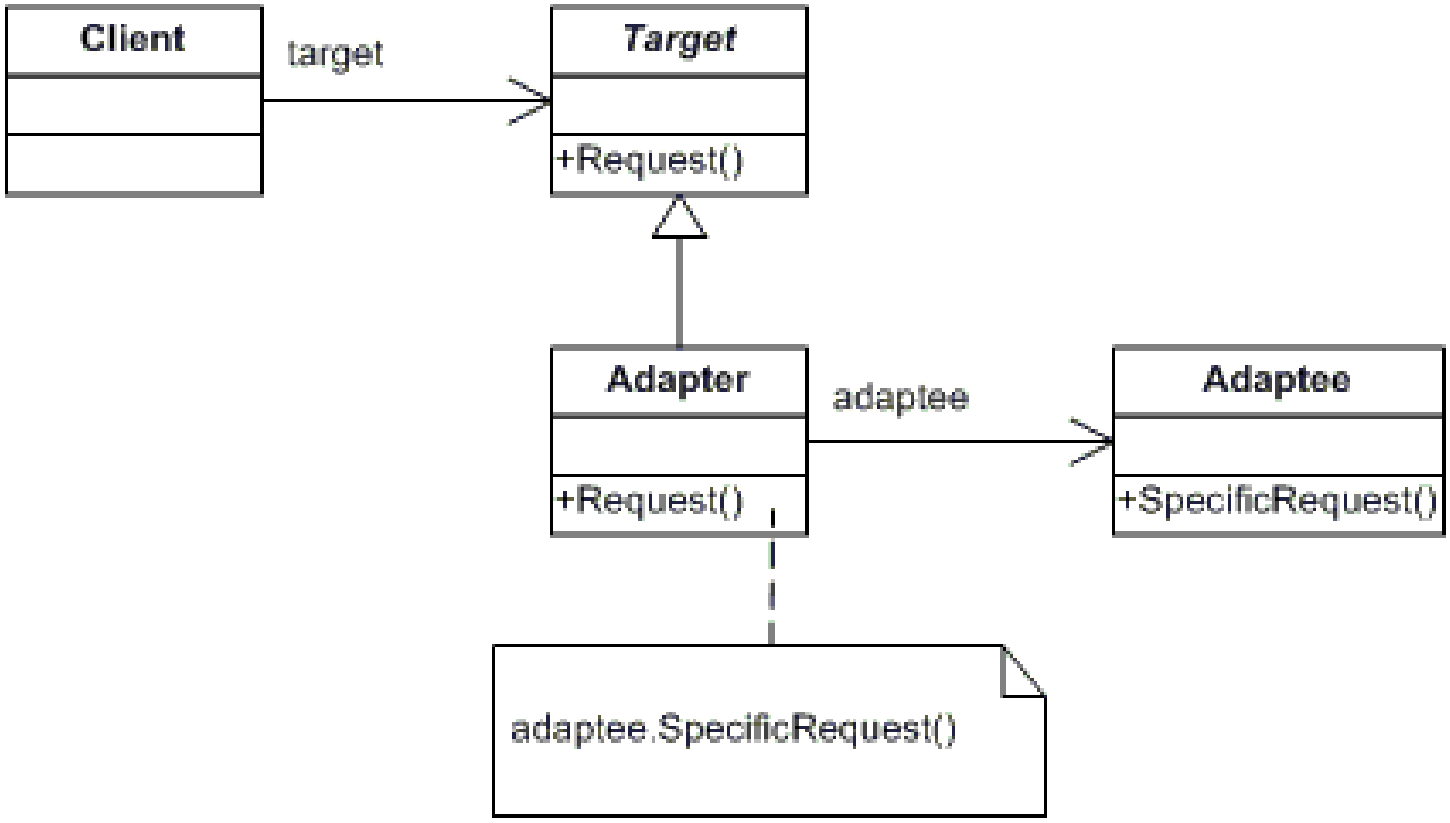
Intent:

Convert the interface of some class b into an interface a that some client class c understands.

AKA:

Wrapper

Adapter pattern - structure



Adapter pattern

- The AdapterPattern lets classes with incompatible interfaces work together.

- This is sometimes called a wrapper because an adapter class wraps the implementation of another class in the desired interface.

- This pattern makes heavy use of delegation where the delegator is the adapter (or wrapper) and the delegate is the class being adapted.

Adapter pattern

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Observer pattern

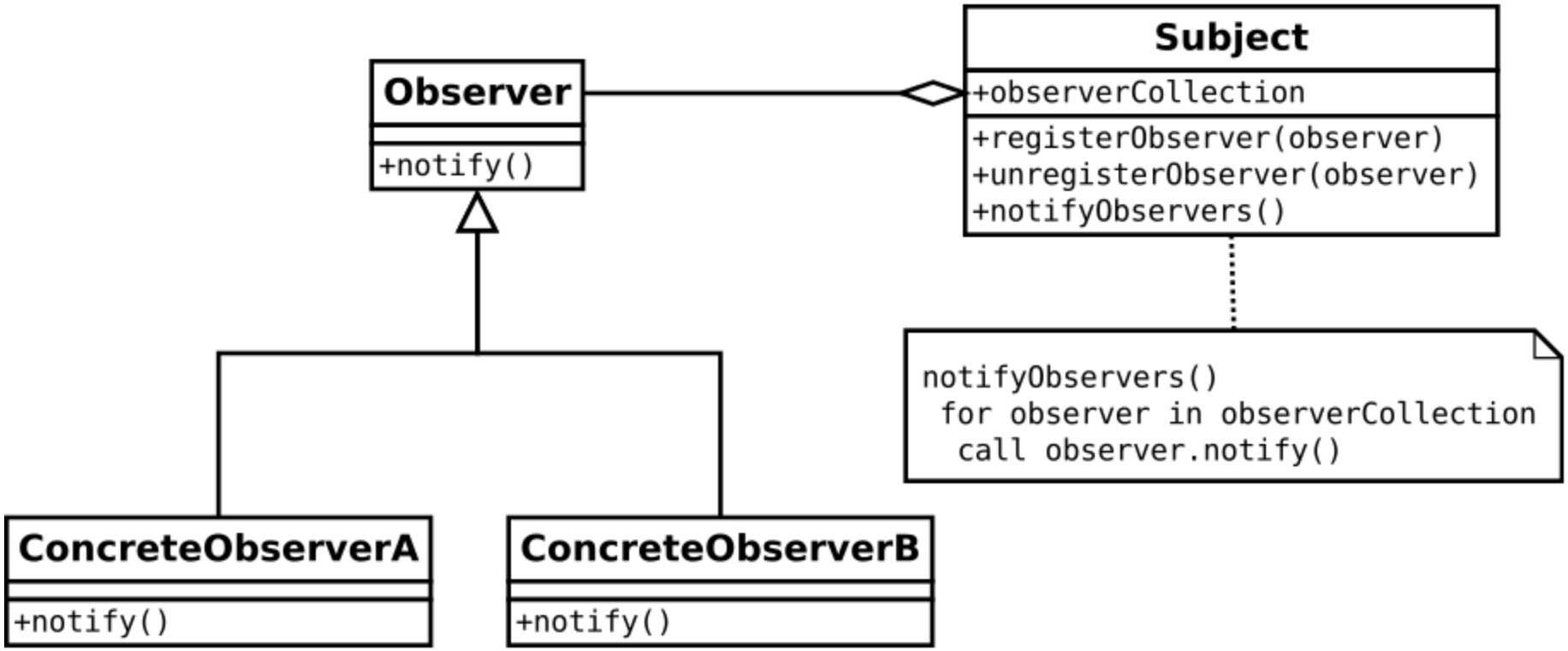
Intent:

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

AKA:

Subject Observer, Publish Subscribe, Callback.

Observer pattern - structure



Observer pattern in our case study

- We can have our DAO implementation class implement the Subject, thus allowing for new observer registrations.

DESIGN PRINCIPLES AND PATTERNS

Vinod

http://vinod.co