



# Code.Hub

The first Hub for Developers

Design patterns



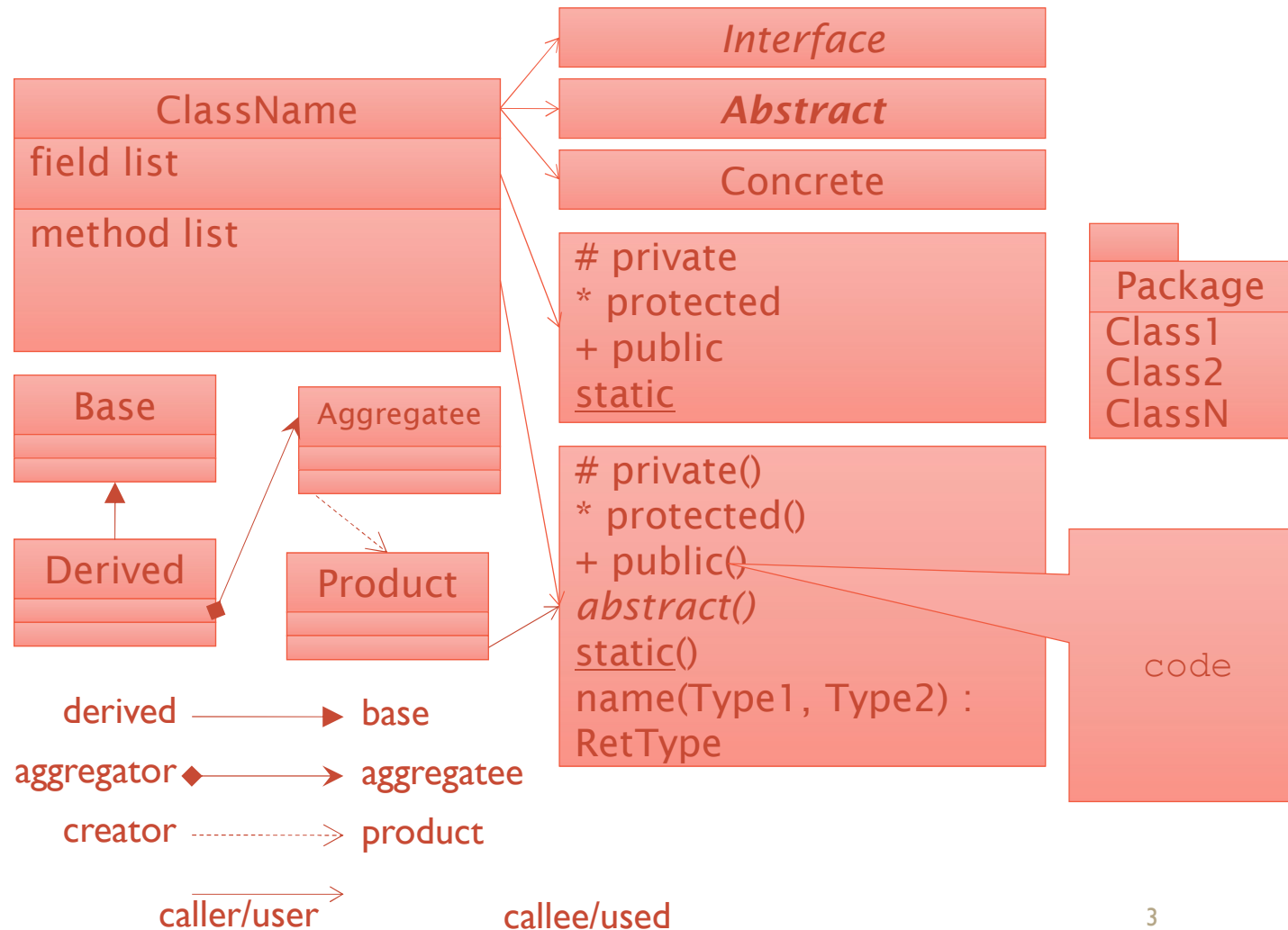
# Design Patterns

based on book of Gang of Four (GoF)

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Elements of Reusable Object-Oriented Software

# UML class diagram recall



# What is a Design Pattern?

## A design pattern

is a **general reusable** solution to a commonly occurring problem in software design.

is **not** a **finished** design that can be transformed directly into code.

is a **description** or **template** for how to solve a problem that can be used in many different situations.

shows **relationships** and **interactions** between classes or objects, without specifying the final application classes or objects that are involved.

# OK, but what is it?

**Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.

**Intent:** A description of the goal behind the pattern and the reason for using it.

**Also Known As:** Other names for the pattern.

**Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.

**Applicability:** Situations in which this pattern is usable; the context for the pattern.

**Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.

**Participants:** A listing of the classes and objects used in the pattern and their roles in the design.

**Collaboration:** A description of how classes and objects used in the pattern interact with each other.

**Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.




**Implementation:** A description of an implementation of the pattern; the solution part of the pattern.

**Sample Code:** An illustration of how the pattern can be used in a programming language.


**Known Uses:** Examples of real usages of the pattern.

**Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

# Classification of DP

 Creational	 Structural	 Behavioral
<p>Abstract Factory</p> <p>Factory method</p> <p>Builder</p> <p>Lazy instantiation</p> <p>Object pool</p> <p>Prototype</p> <p>Singleton</p> <p>Multiton</p> <p>Resource acquisition is initialization</p>	<p>Adapter</p> <p>Bridge</p> <p>Composite</p> <p>Decorator</p> <p>Façade</p> <p>Flyweight</p> <p>Proxy</p>	<p>Null Object</p> <p>Command</p> <p>Interpreter</p> <p>Iterator</p> <p>Mediator</p> <p>Memento</p> <p>Observer</p> <p>State</p> <p>Chain of responsibility</p> <p>Strategy</p> <p>Specification</p> <p>Template method</p> <p>Visitor</p>

# Classification / 2



Concurrency	J2EE	Architectural
Active Object	Business Delegate	Layers
Balking	Composite Entity	Presentation-abstraction-control
Double checked locking	Composite View	Three-tier
Guarded suspension	Data Access Object	Pipeline
Monitor object	Fast Lane Reader	Implicit invocation
Reactor	Front Controller	Blackboard system
Read/write lock	Intercepting Filter	Peer-to-peer
Scheduler	Model-View-Controller	
Event-Based Asynchronous	Service Locator	Service-oriented architecture
Thread pool	Session Facade	Naked objects
Thread-specific storage	Transfer Object	
	Value List Handler	
	View Helper	

# Contents

- Simple patterns

- Singleton
- Template Method
- Factory Method
- Adapter
- Proxy
- Iterator



- Complex patterns

- Abstract Factory
- Strategy
- Mediator
- Façade



- J2EE patterns

- Data Access Objects
- Model-View-Controller
- Session Façade
- Front Controller







# **SIMPLE PATTERNS**

# Singleton



- Ensure a class has only one instance, and provide a global point of access to it.

```
public class Singleton {  
    private Singleton() {}  
    private static Singleton _instance = null;  
    public static Singleton getInstance () {  
        if (_instance == null) _instance = new Singleton();  
        return _instance;  
    }  
    ...  
}
```

## Lazy instantiation

Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

return \_instance;

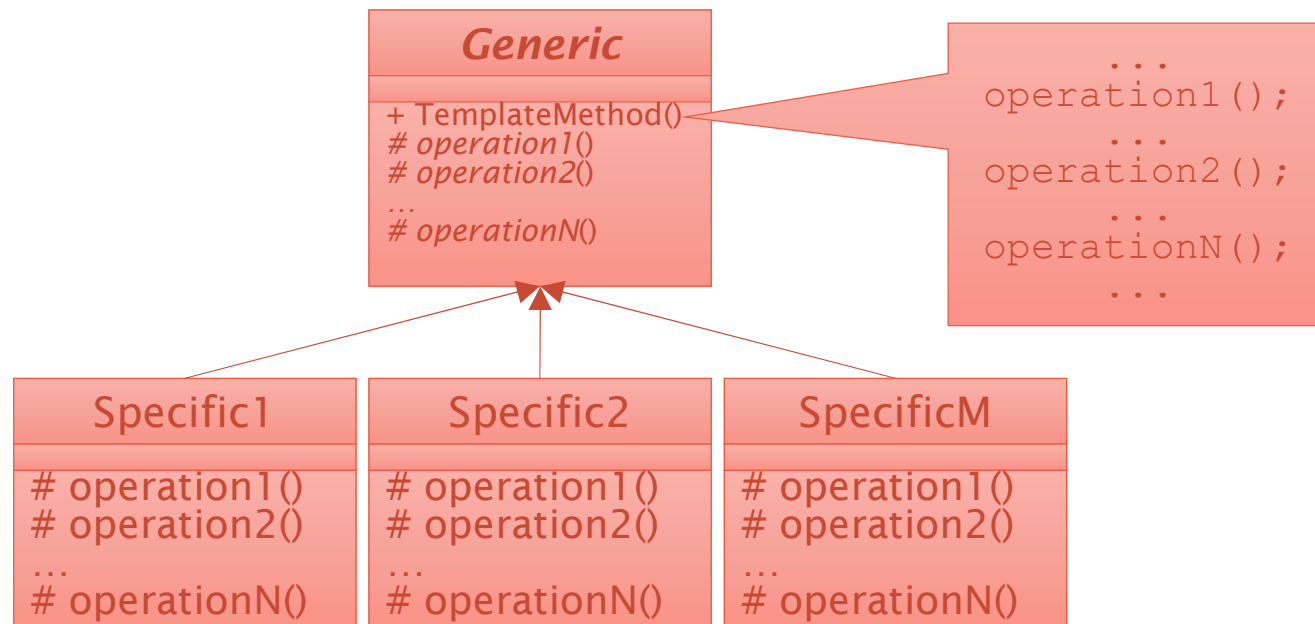
Singleton
# _instance : Singleton
# Singleton() + <u>getInstance()</u> : Singleton

10

# Template Method



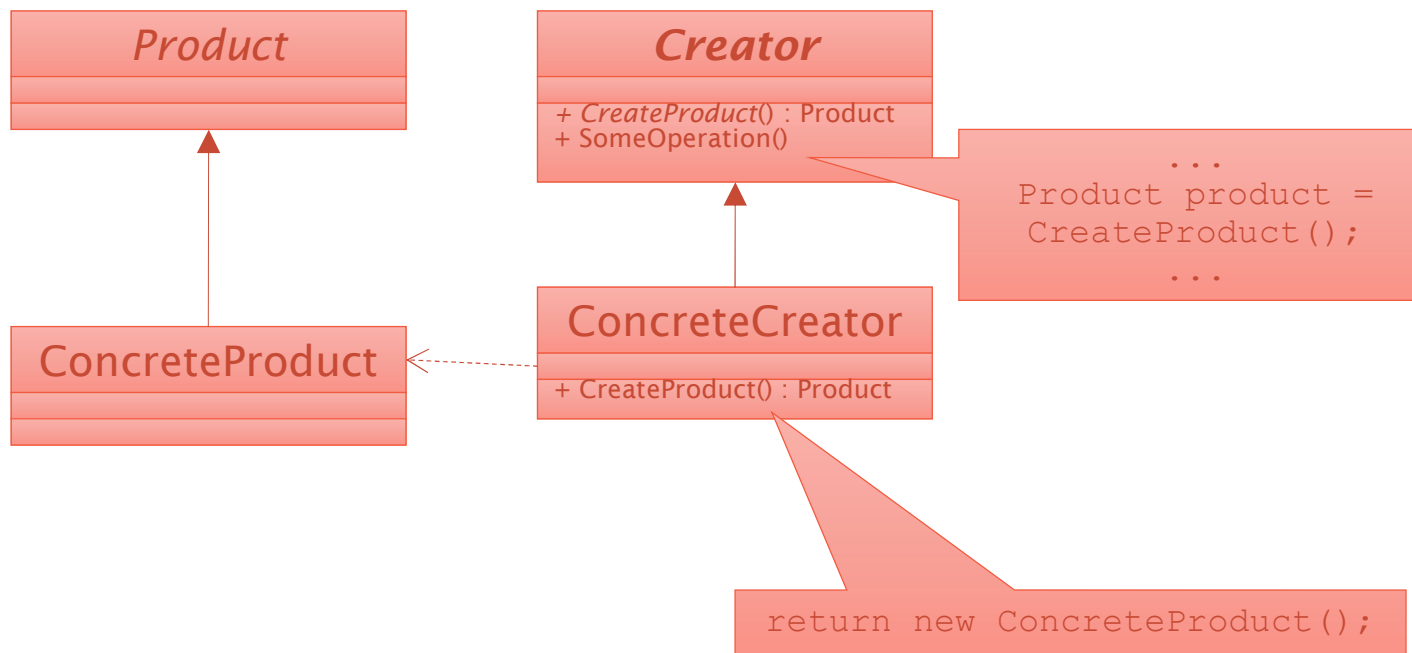
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



# Factory Method



- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



# Example code for factory method

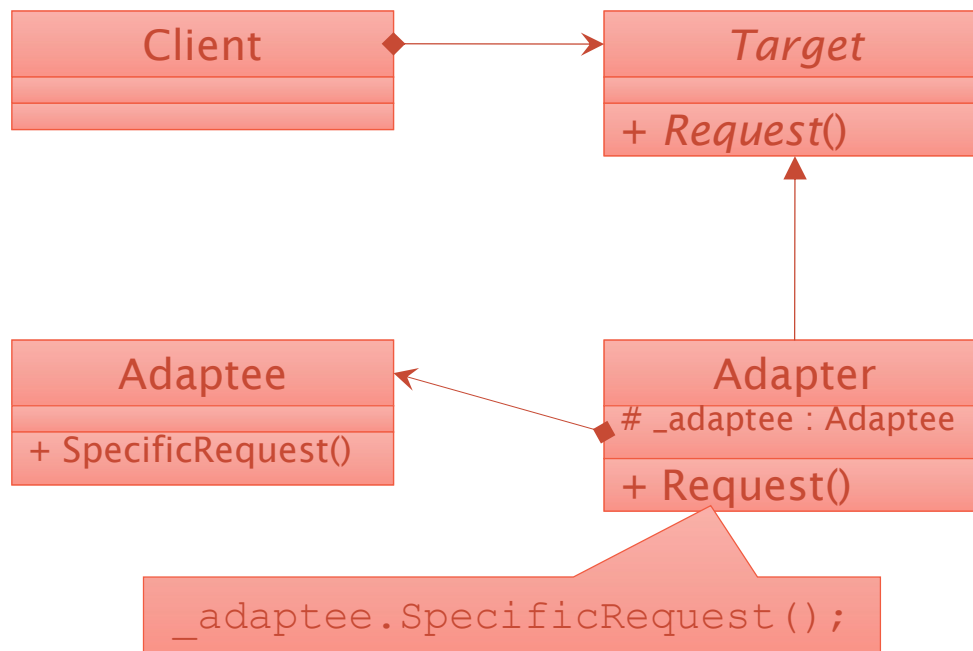
```
public class Product{}
public class SnackProduct extends Product{}
public class DairyProduct extends Product{}
public class FruitProduct extends Product{}

public class Factory{
    public Product createProduct(String productType) throws Exception
    {
        switch(productType)
        {
            case "SnackProduct":
                return new SnackProduct();
            case "DairyProduct":
                return new DairyProduct();
            case "FruitProduct":
                return new FruitProduct();
            default:
                throw new Exception("No such product");
        }
    }
    public static void main(String args[]){
        Factory factory = new Factory();
        Product product = factory.createProduct("SnackProduct");
    }
}
```

# Adapter

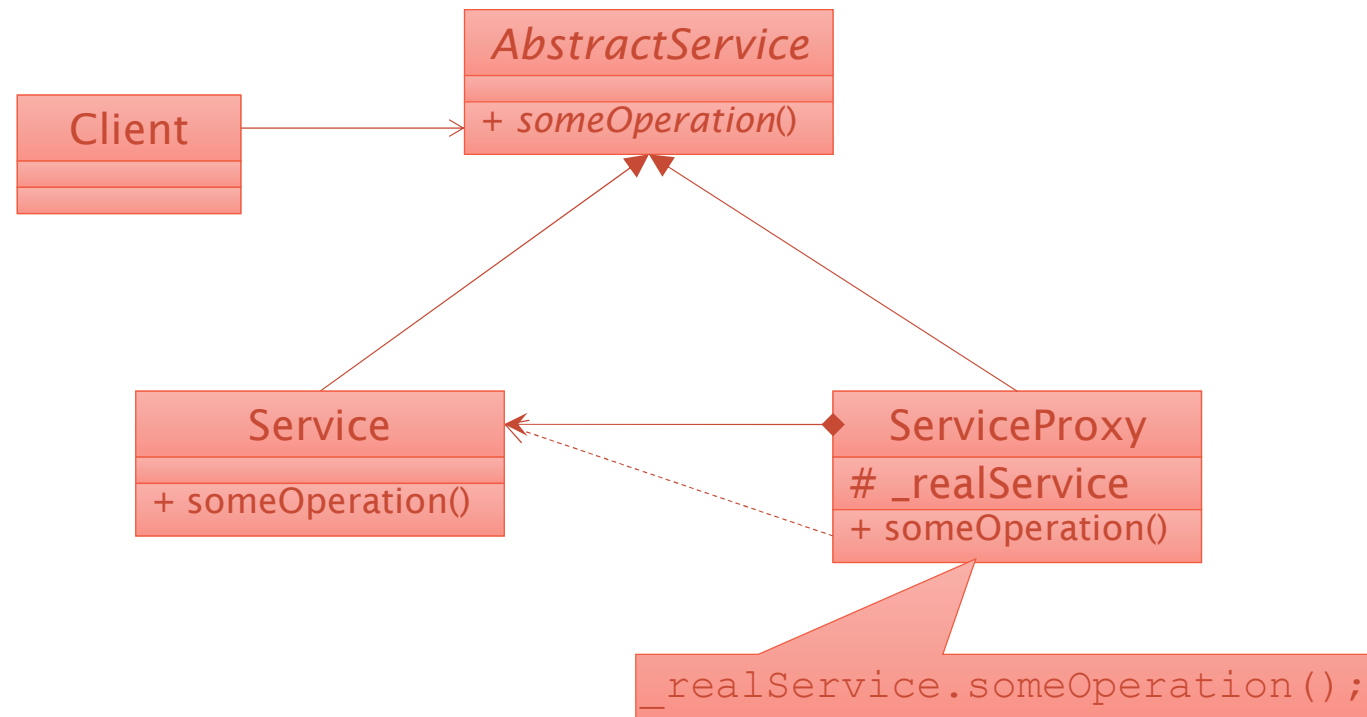


- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



# Proxy

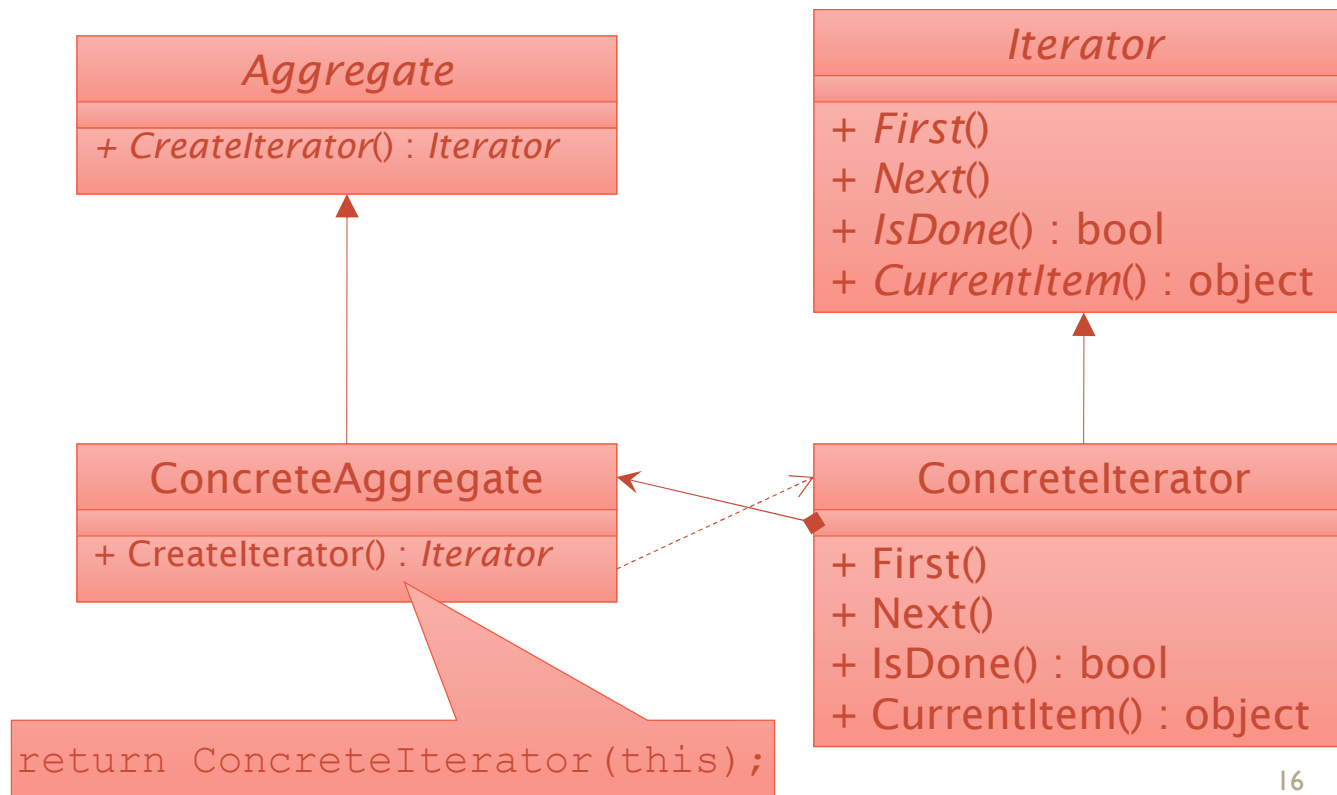
- Provide a surrogate or placeholder for another object to control access to it.



# Iterator



- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.





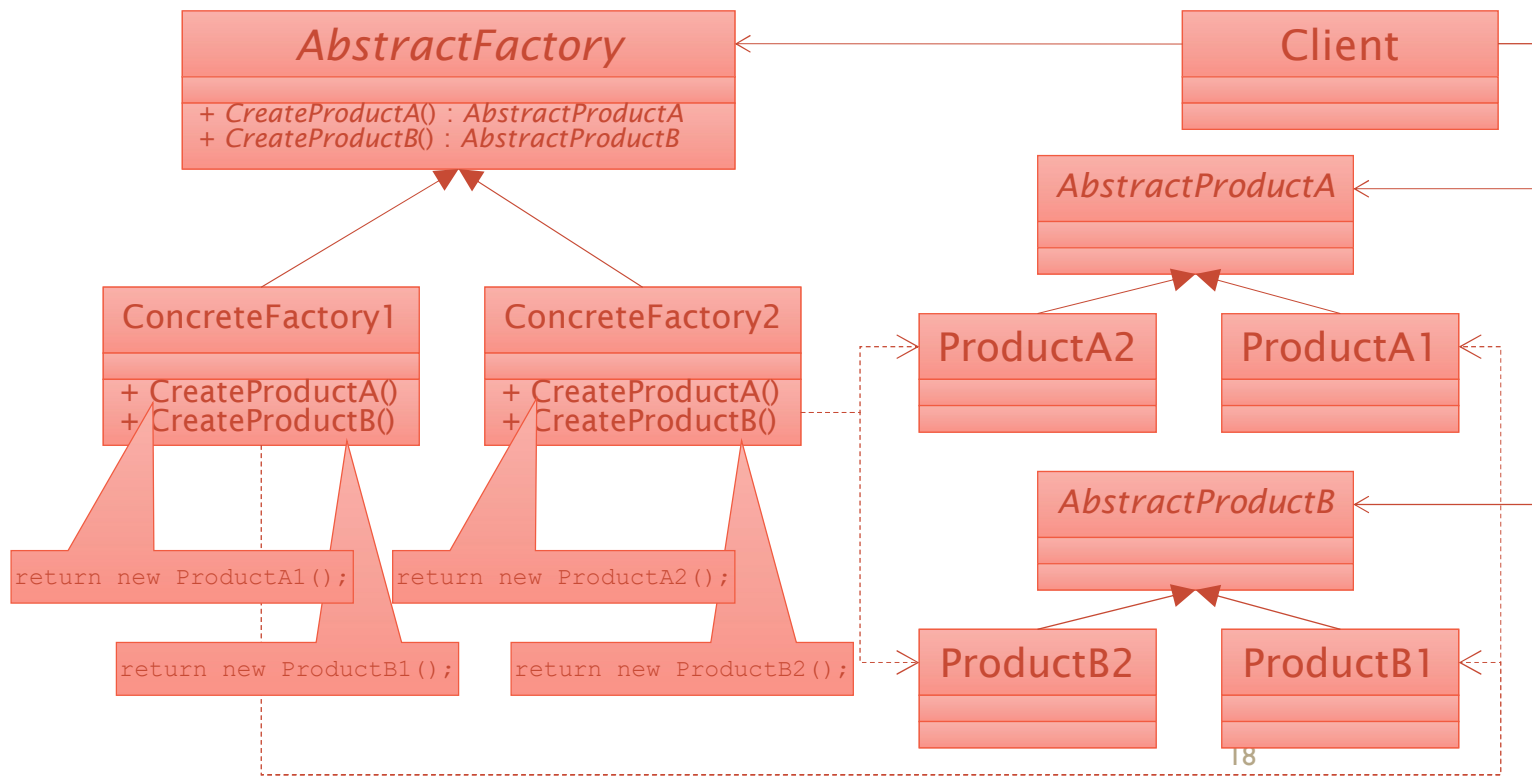


# COMPLEX EXAMPLES

# Abstract Factory



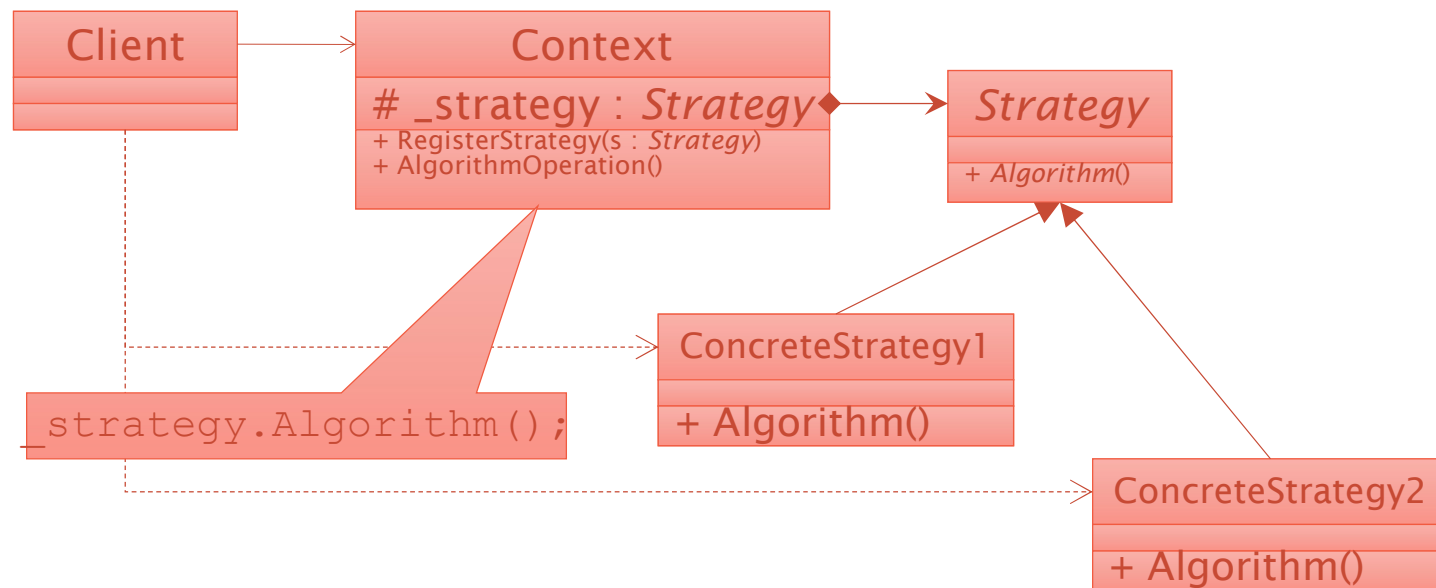
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



# Strategy



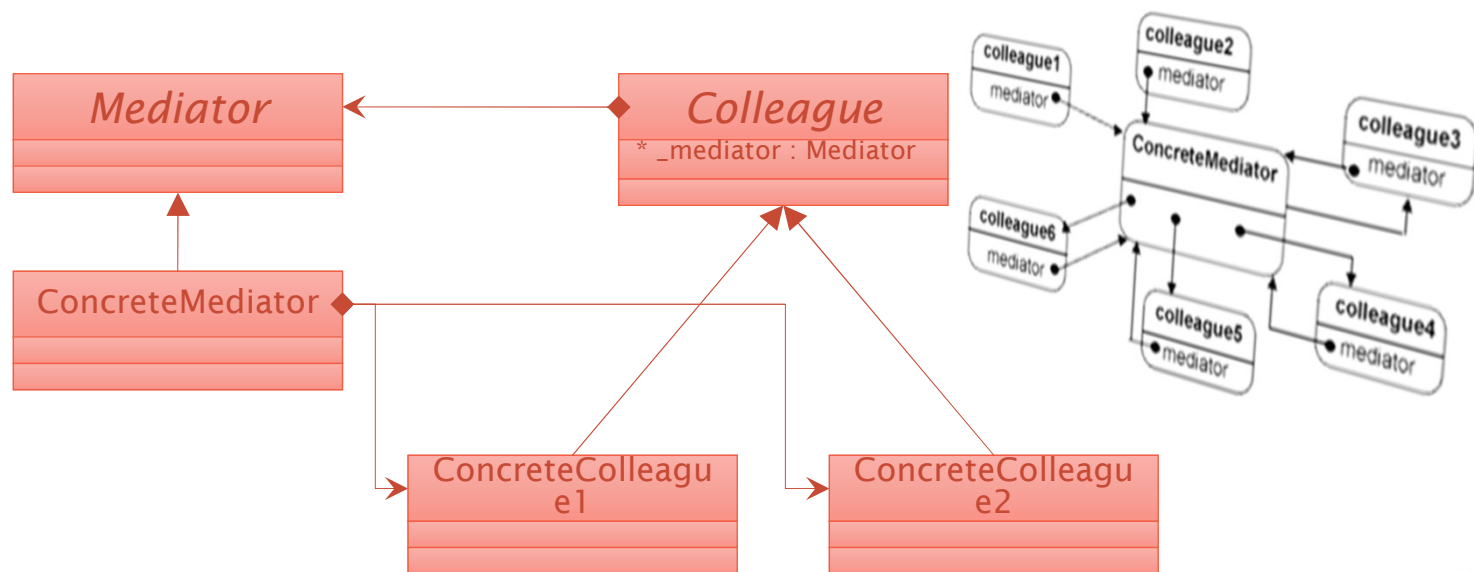
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



# Mediator



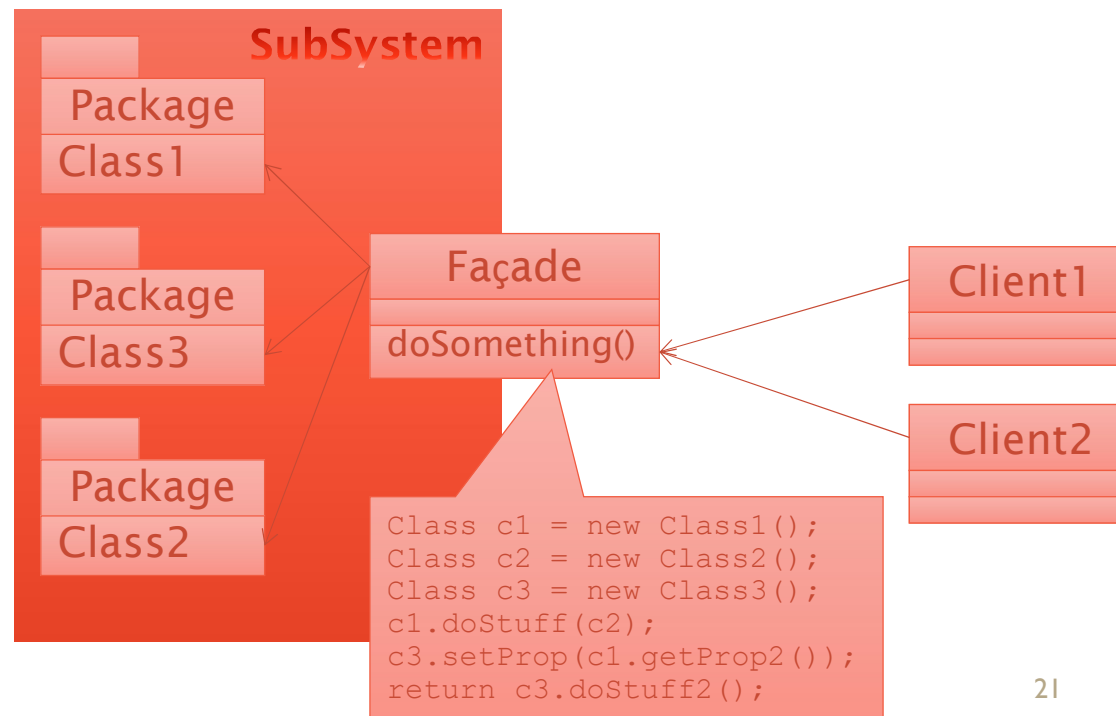
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



# Façade



- Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.





<http://java.sun.com/blueprints/patterns/catalog.html>

# J2EE PATTERN EXAMPLES

# Data Access Objects (DAO)

Code that depends on specific features of data resources ties together business logic with data access logic. This makes it difficult to replace or modify an application's data resources.

This pattern

- separates a data resource's client interface from its data access mechanisms;
- adapts a specific data resource's access API to a generic client interface;
- allows data access mechanisms to change independently of the code that uses the data.



# Data Access Objects (DAO) / 2

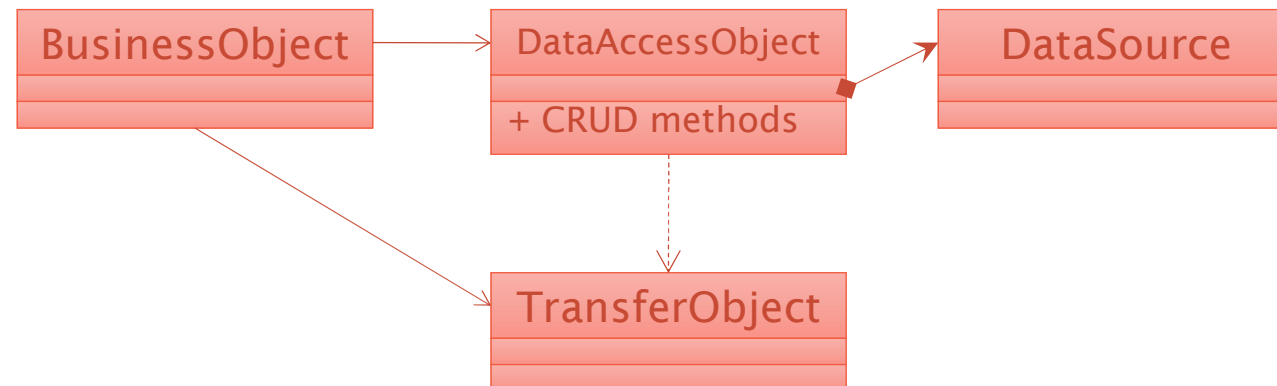


**BusinessObject:** represents the data client.

**DataAccessObject:** abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source.

**DataSource:** represents a data source implementation.

**TransferObject:** the data carrier, DAO may use it to return data to the client.



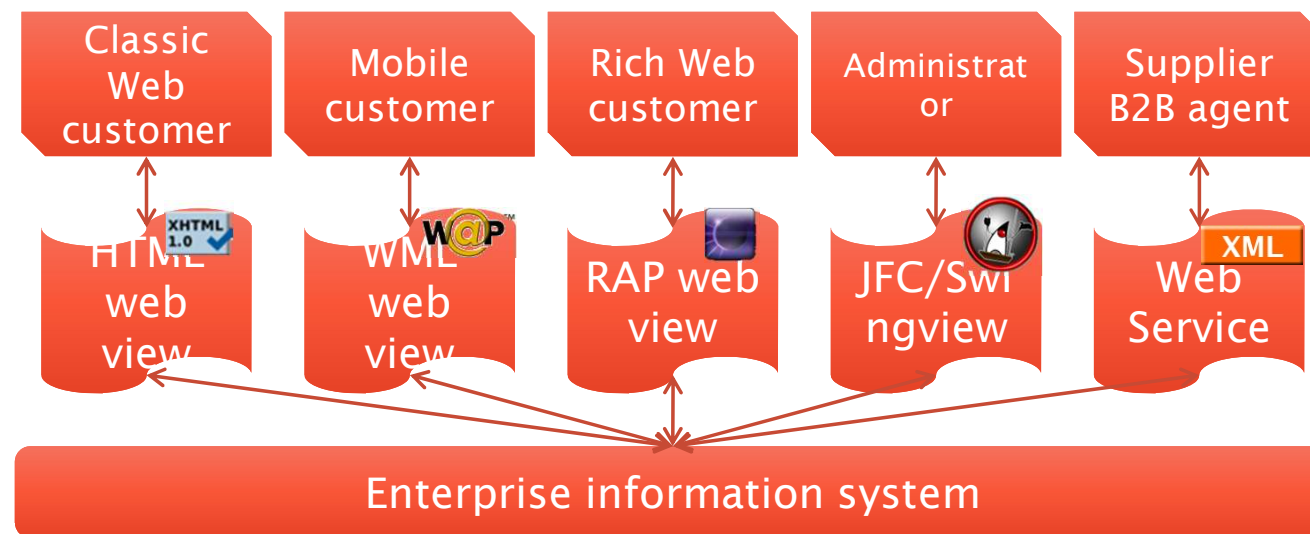




# Model-View-Controller (MVC)

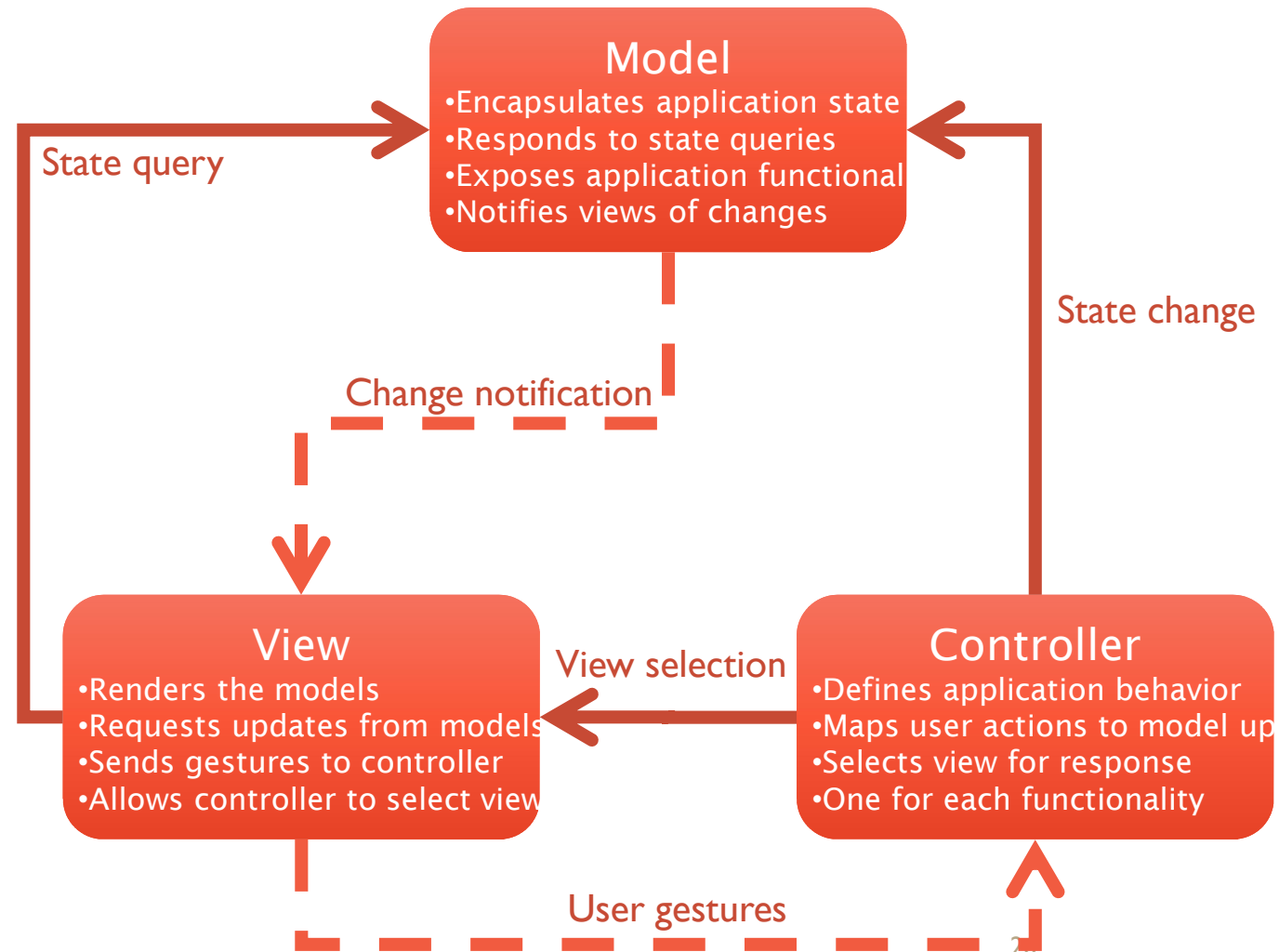


- Application presents content to users in numerous ways containing various data. The engineering team responsible for designing, implementing, and maintaining the application is composed of individuals with different skill sets.





# Model-View-Controller / 2



# Front Controller

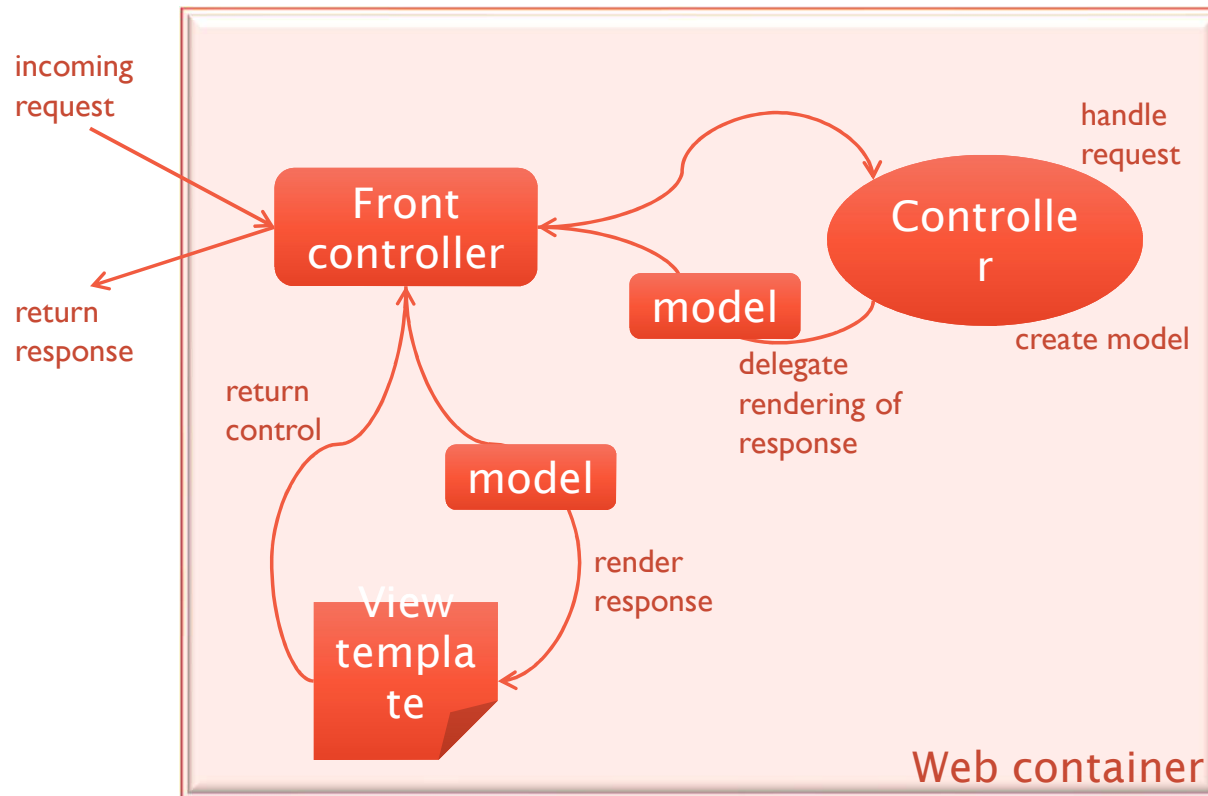


The presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests. Such control mechanisms may be managed in either a centralized or decentralized manner.

## Problems:

- Each view is required to provide its own system services, often resulting in duplicate code.
- View navigation is left to the views. This may result in commingled view content and view navigation.
- Distributed control is more difficult to maintain: changes will need to be made in numerous places.

# Front Controller / 2



# Session Façade



Enterprise beans encapsulate business logic and business data and expose their interfaces, and thus the complexity of the distributed services, to the client tier.

- Tight coupling, which leads to direct dependence between clients and business objects;
- Too many method invocations between client and server, leading to network performance problems;
- Lack of a uniform client access strategy, exposing business objects to misuse.

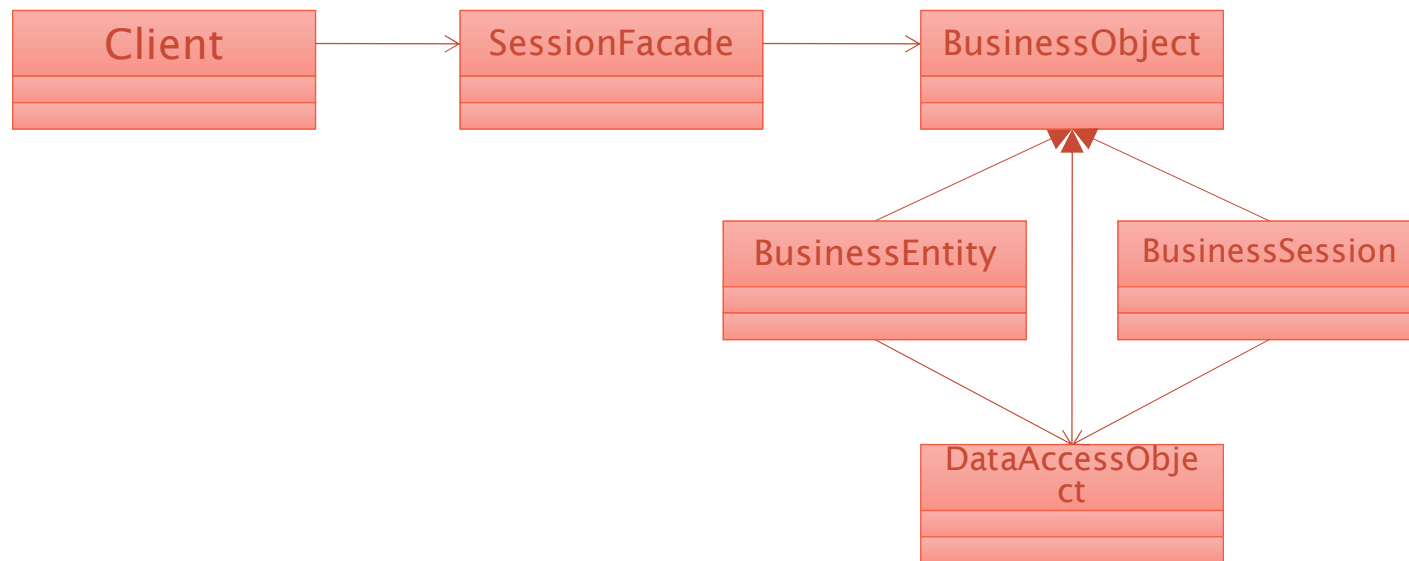
# Session Façade / 2



**Client:** the object which needs access to the business service. This client can be another session bean (Session Facade) in the same business tier or a business delegate in another tier.

**SessionFacade:** a session bean which manages the relationships between numerous BusinessObjects and provides a higher level abstraction to the client.

**BusinessObject:** a role object that facilitates applying different strategies, such as session beans, entity beans and a DAO. A BusinessObject provides data and/or some services.

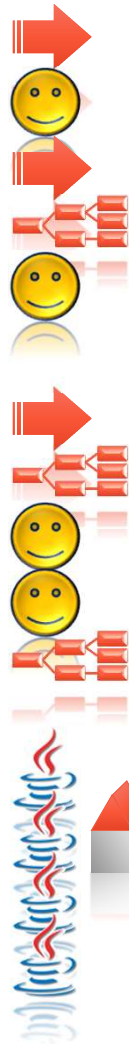




# OUTRODUCTION

# Revision

- **Simple patterns**
  - Singleton
  - Template Method
  - Factory Method
  - Adapter
  - Proxy
  - Iterator
- **Complex patterns**
  - Abstract Factory
  - Strategy
  - Mediator
  - Façade
- **J2EE patterns**
  - Data Access Objects
  - Model-View-Controller
  - Session Façade
  - Front Controller





# Sources

- [https://sourcemaking.com/design\\_patterns](https://sourcemaking.com/design_patterns)
- <https://www.oracle.com/java/technologies/design-patterns-catalog.html>
- [https://www.tutorialspoint.com/design\\_pattern/design\\_pattern\\_overview.htm](https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm)
- <https://www.geeksforgeeks.org/software-design-patterns/>

