Design Patterns

Pre-requisites

- Knowledge of Object Oriented Concepts
- Experience in any OO programming language

A quick recap of OO with Questions

- What is,
  - Abstraction
  - Data Hiding
  - Polymorphism
  - Dynamic Binding ##

Objectives

- After completing this session, you will be able to get an introduction to
  - What design patterns are
  - Gang Of Four Patterns

Architecture Vs. Design

- What is Architecture?
  - The architecture of a system is its 'skeleton'.
  - It's the highest level of abstraction of a system. What kind of data storage is present, how do modules interact with each other, what recovery systems are in place.
  - Just like design patterns, there are Architectural patterns: MVC, 3-tier layered design, etc. ##

Architecture Vs. Design

- What is Software design?
  - It is about designing the individual modules / components.
  - What are the responsibilities, functions, of module x? Of class Y? What can it do, and what not? What design patterns can be used?

- So in short, Software architecture is more about the design of the entire system, while software design emphasizes on module / component / class level ##

ProWord

- Some one has already solved your problem!!

- Oh!!
- Who & how?

What is a design pattern?

- "Reusable solutions to recurring problems that we encounter during software development."

Design Patterns are NOT

- They are not data structures that can be included in classes and reused as is (i.e. linked lists, hash tables, etc)
- They are not complex domain-specific design solutions for the entire application

- Instead, they are:
  - Proven solutions to a general design problem in a particular context which can be customized to suit our needs.

When were these created

- 1994 – The Gang Of Four (GoF - Gamma, Helm, Johnson, and Vlissides) publish the first book on Design Patterns.

Why design patterns?

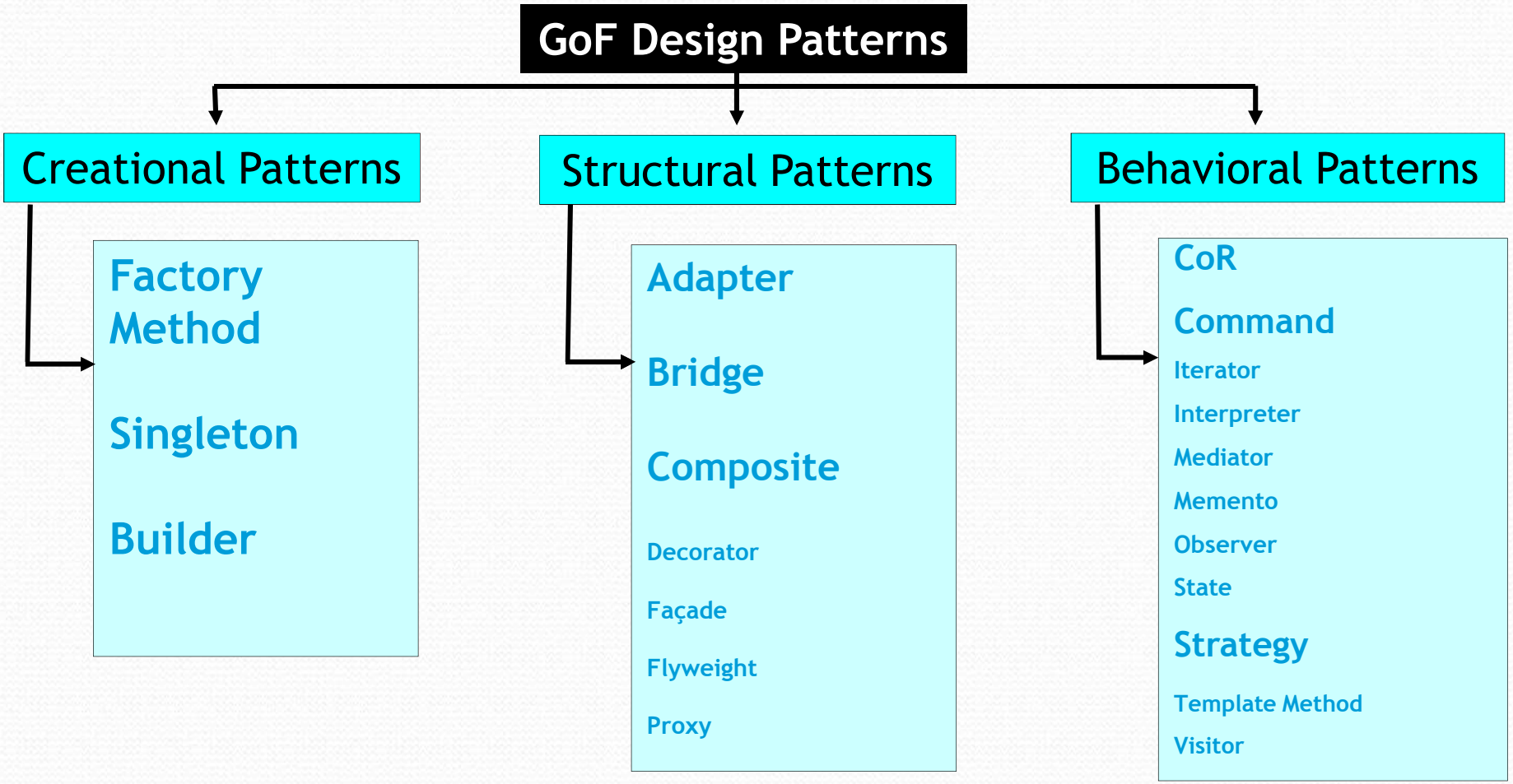
- Patterns enable programmers to "...recognize a problem and immediately determine the solution without having to stop and analyze the problem first."

- What would be the advantages?
  - Reusable solutions
  - Saving time and cost
  - Better quality proven solutions
  - Effective Communication ##

How is Design Pattern Described?

- It has four essential elements
  - Pattern Name
  - Problem Description – Problem and it's Context
  - Solution
  - Consequences – Benefits and limitations ##

Classification of Design Patterns



Creational Design Patterns

Introducing Creational Patterns

- Creational patterns deal with object creation mechanisms
  - They provide guidance on how to create objects when their creation requires decisions

Different Creational Design Patterns

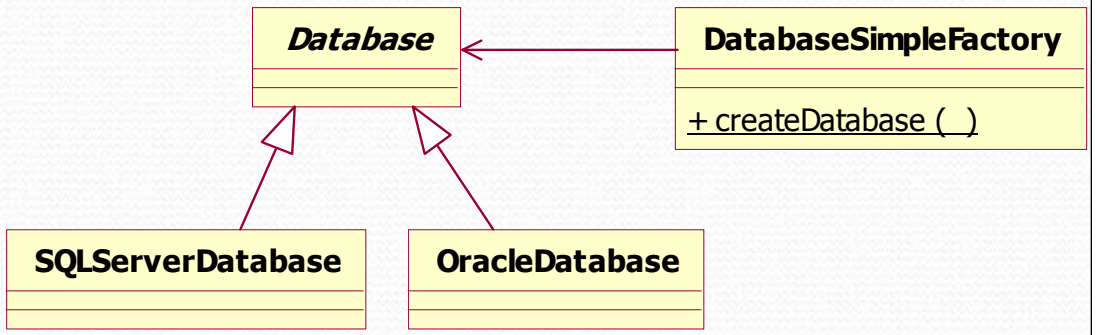
- Creational Design Patterns are further classified as:
  - Factory method
  - Builder
  - Singleton

Factory Pattern

- Returns an instance of one of several possible classes, depending on the data provided to it
  - Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data

Factory - Structure

- **Product (Database):** Defines abstract base class
- **Concrete Product (SQLServerDatabase and OracleDatabase):** Subclasses extending the base class
- **Creator (DatabaseSimpleFactory):** This is the simple factory that decides which type of class to return based on the parameter value of its create operation



Factory - Code

```
//Product
public abstract class
   Database {
}

//Concrete Product
public class OracleDatabase
   extends Database {
}
public class
   SQLServerDatabase
   extends Database {
}
```

```
//Creator
public class
   DatabaseSimpleFactory {
public static Database
   createDatabase(String spec)
   {
/* Depending on value of spec,
   appropriate database type is
   returned */
}
}
```

Factory Method

- Pros:
  - **Loose Coupling:** Factory Method eliminates the need to bind application classes into client code.
  - **Object Extension:** It enables classes to provide an extended version of an object.
  - **Appropriate Instantiation:** Right object is created from a set of related classes.

Usage of Builder Pattern

- **Builder Pattern** builds complex objects from simple ones step-by-step.
- It separates the construction of a complex object from its representation so that the same construction process can create different representations.

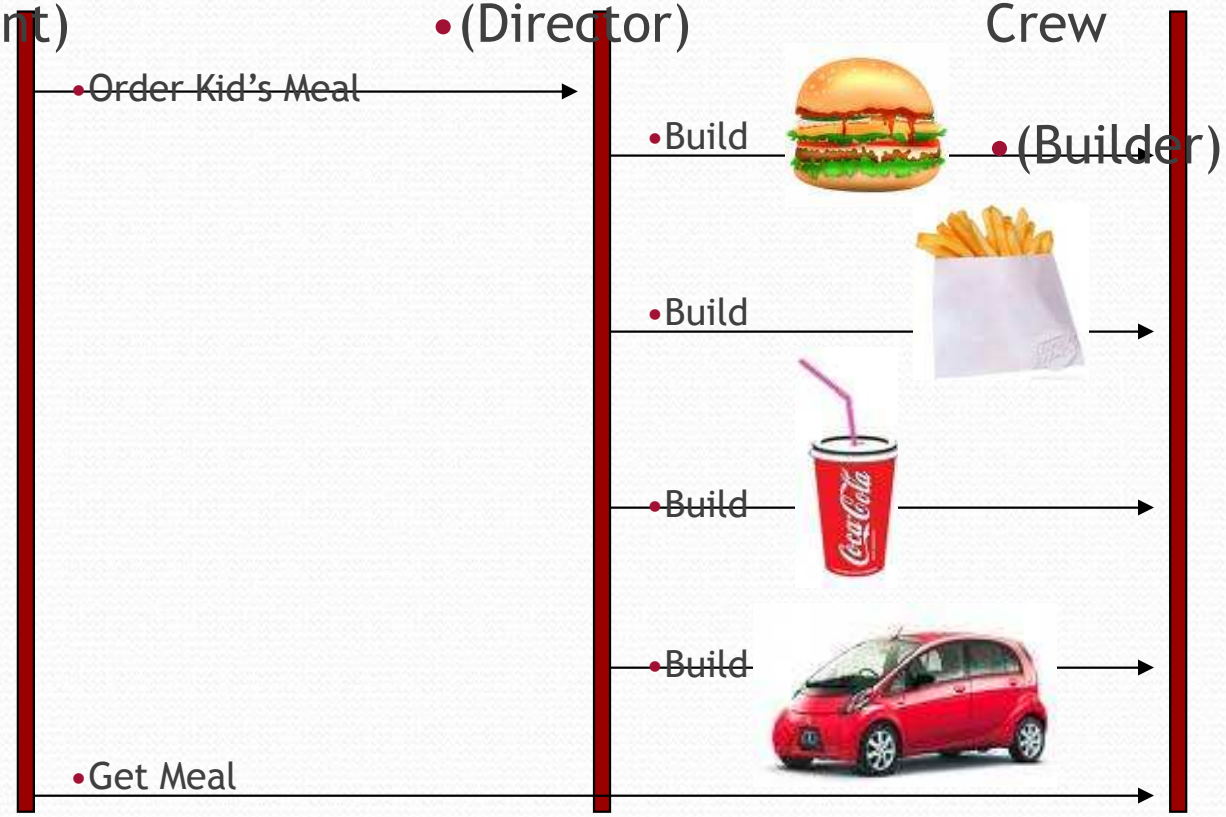
Example of Builder Pattern

- Customer
- Cashier
- Restaurant

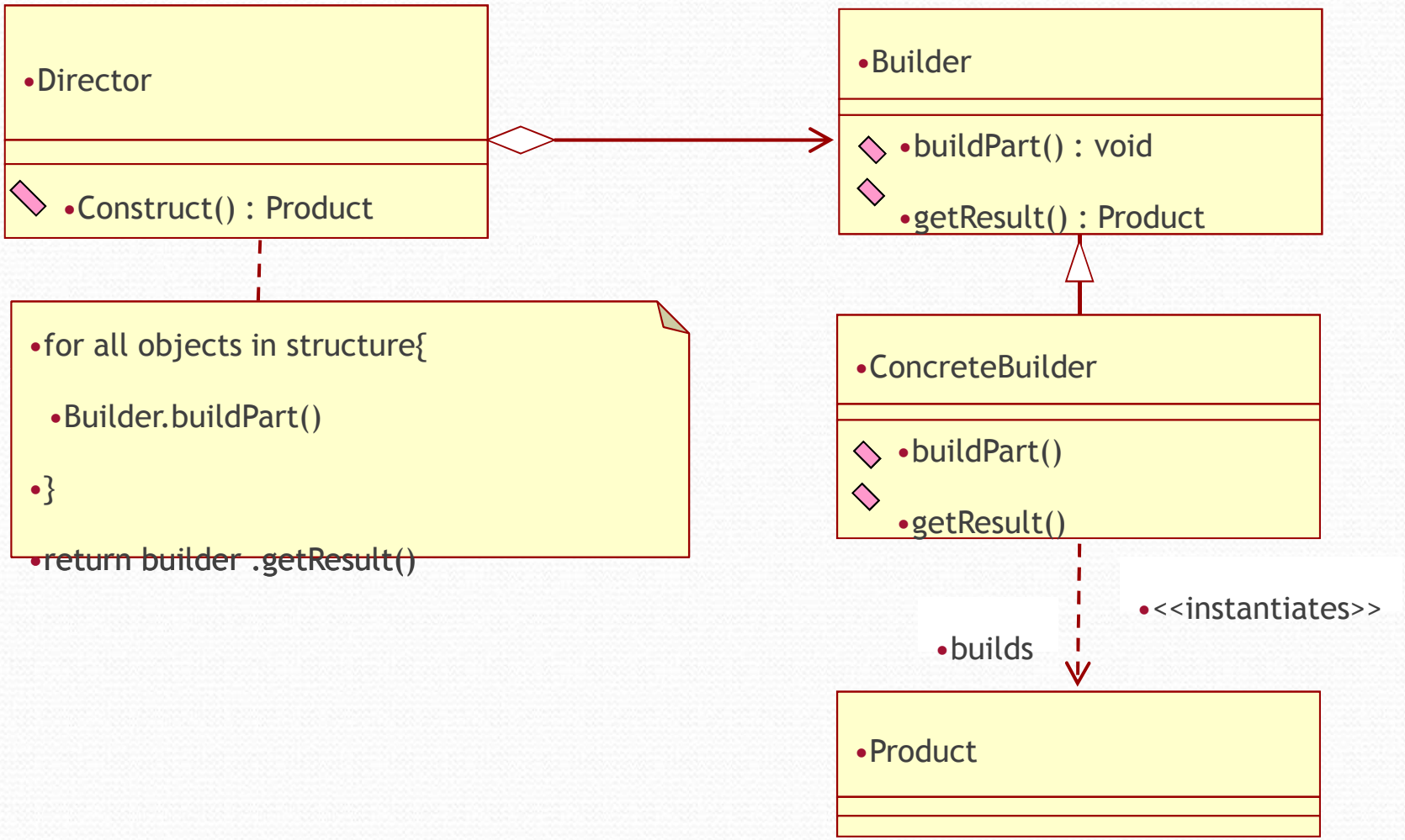
- (Client)
- (Director)

Crew

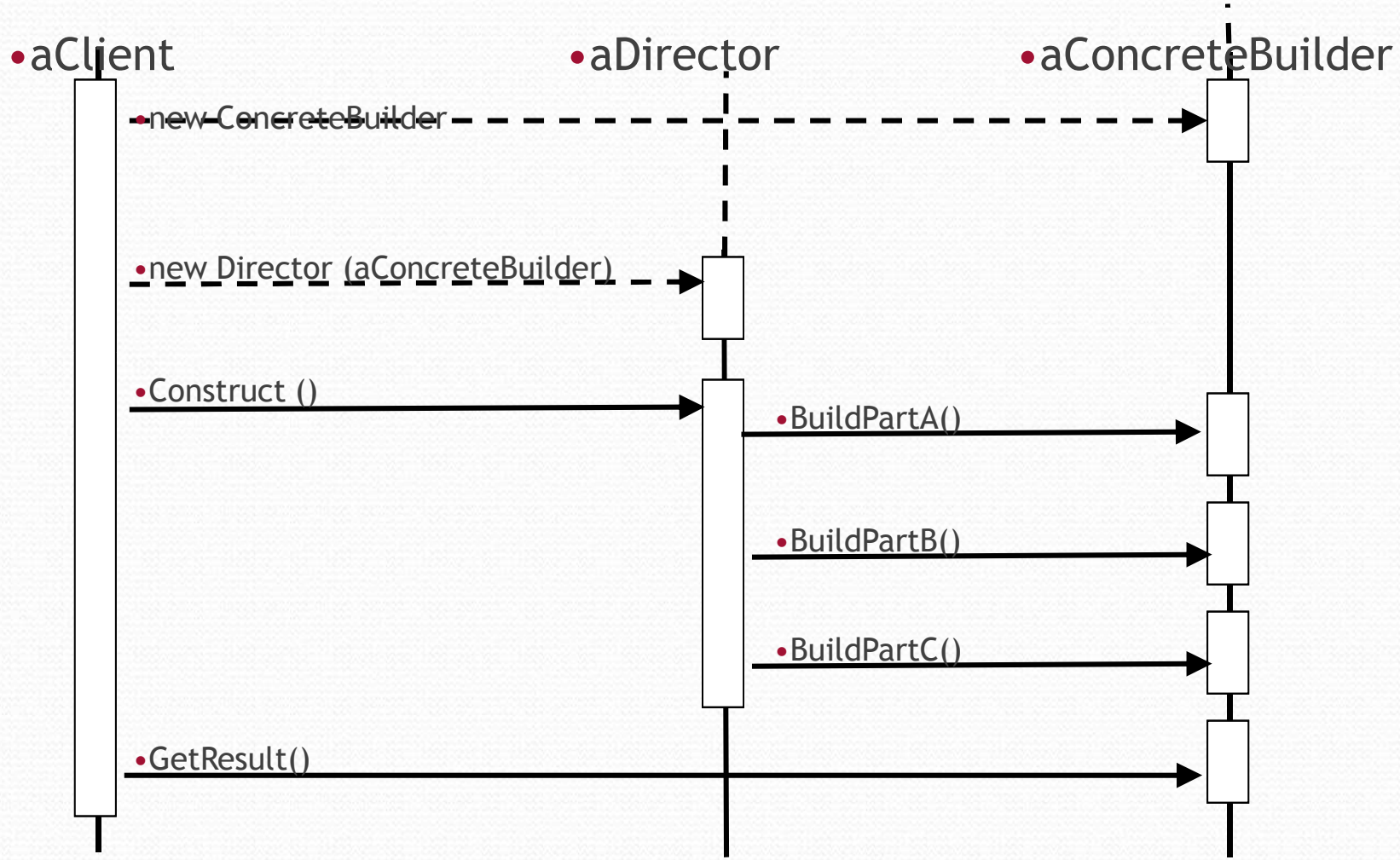
- (Builder)



Structure of Builder Pattern



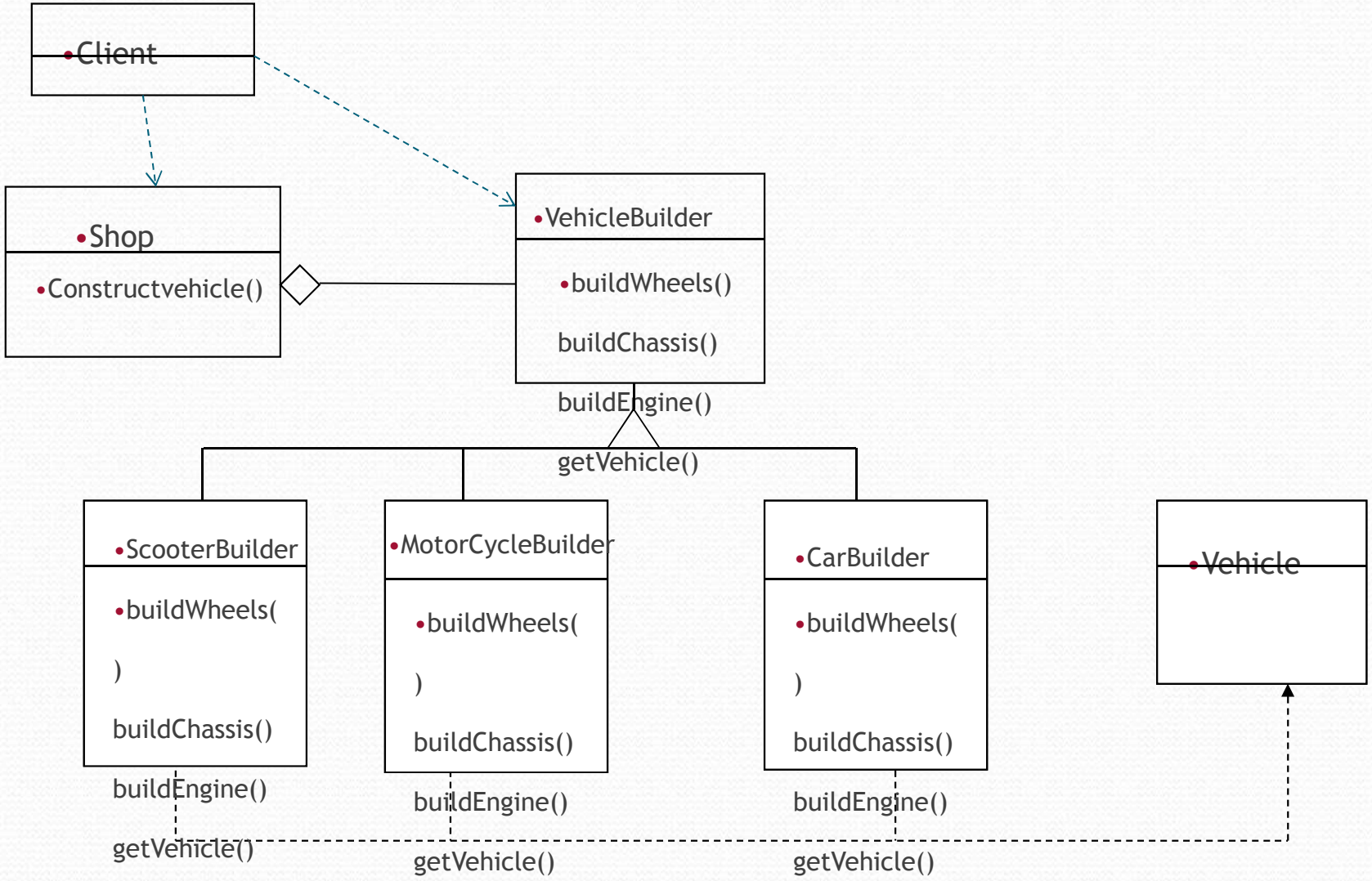
Simplified Structure of Builder Pattern



Example of Builder Pattern

- Customer wants to purchase a vehicle from a vehicle shop. Shop assembles different kinds of vehicles like Car, Bike, etc. by integrating different parts of a vehicle which are created independently of each other.

Builder Pattern Solution



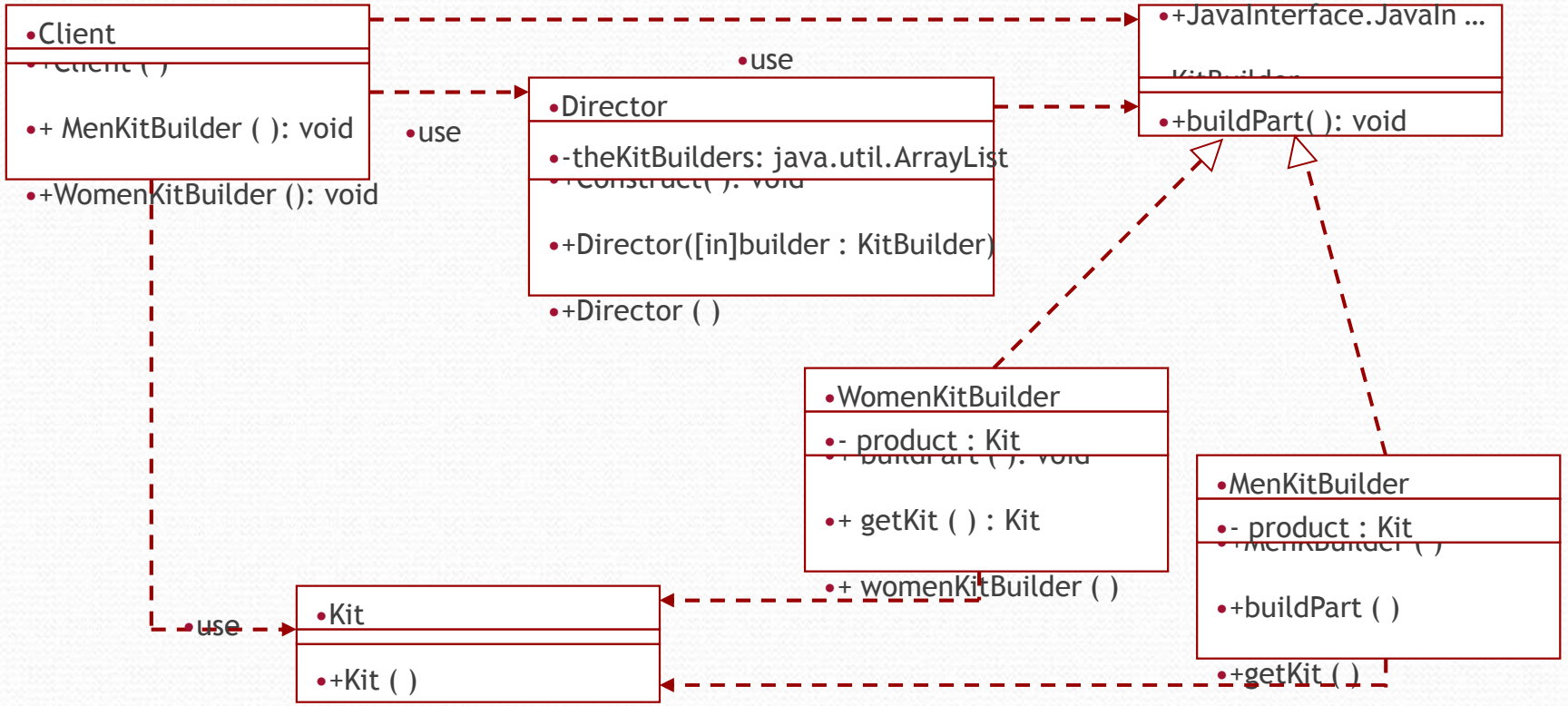
Builder versus Factory and Abstract Factory

- The **builder pattern** actually creates all the subtypes, and the composition of the objects might differ within the same class. The **factory pattern** creates one of the various subtypes of an object.
- **Builder pattern** focuses on constructing a complex object step by step. The **Abstract Factory** focuses on a family of product objects (simple or complex).
- **Builder pattern** returns the product as a final step. The **Abstract Factory** returns the product immediately.

Case Study on Builder Pattern

- We need to create promotion kits for men and women.
- Each kit will have a video, a garment, and a book. Each of these items will be specific to men or women based on for whom the kit is being built.

Case Study Solution: Builder Pattern

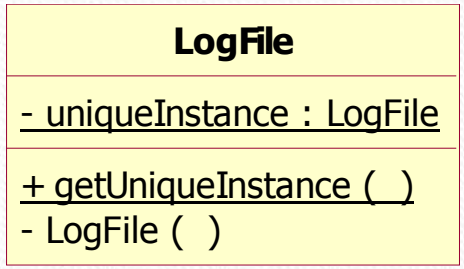


Builder Pattern

- Pros:
  - It lets you vary a product's internal representation.
  - It isolates code for construction and representation.
  - It gives you finer control over the construction process.

Singleton

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



Singleton - Code

```
public class LogFile {
// This attribute stores the instance of the Singleton class.
private static LogFile uniqueInstance;
// We could also initialize static member immediately as: private
   static LogFile uniqueInstance = new LogFile()
//This operation implements the logic for returning the same
   instance of the  Singleton pattern.
public static synchronized LogFile getUniqueInstance() {
// You can customize the operation based on your application needs.
if (uniqueInstance == null) {
uniqueInstance = new LogFile();
}
return uniqueInstance;
}
private LogFile() {
}}
```

Structural Design Patterns

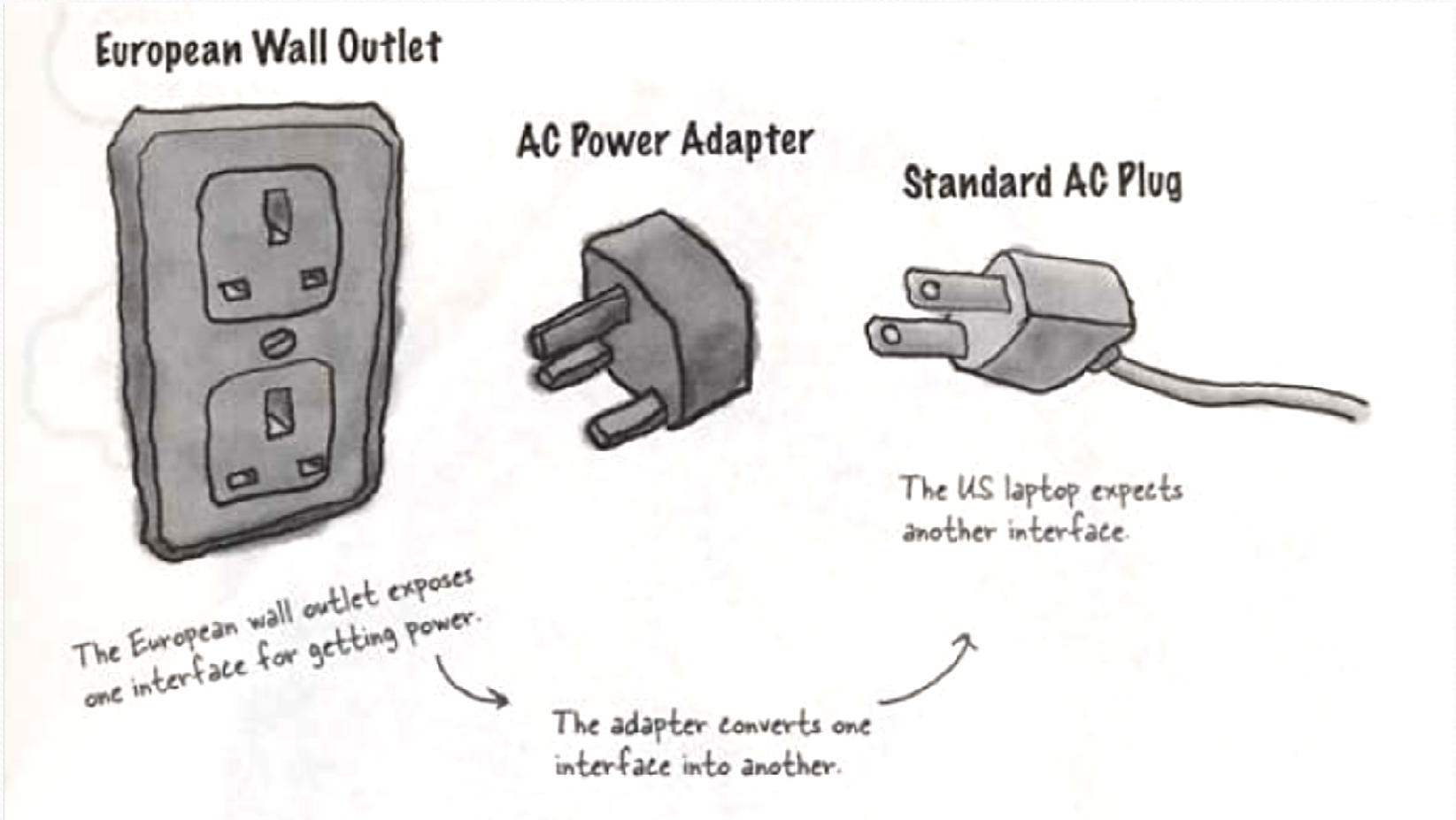
Introducing Structural Patterns

- Structural patterns describe how objects and classes can be combined to form larger structures
  - Structural class patterns use inheritance to compose interfaces or implementations
  - Structural object patterns describe how objects can be organized to work with each other to form a larger structure

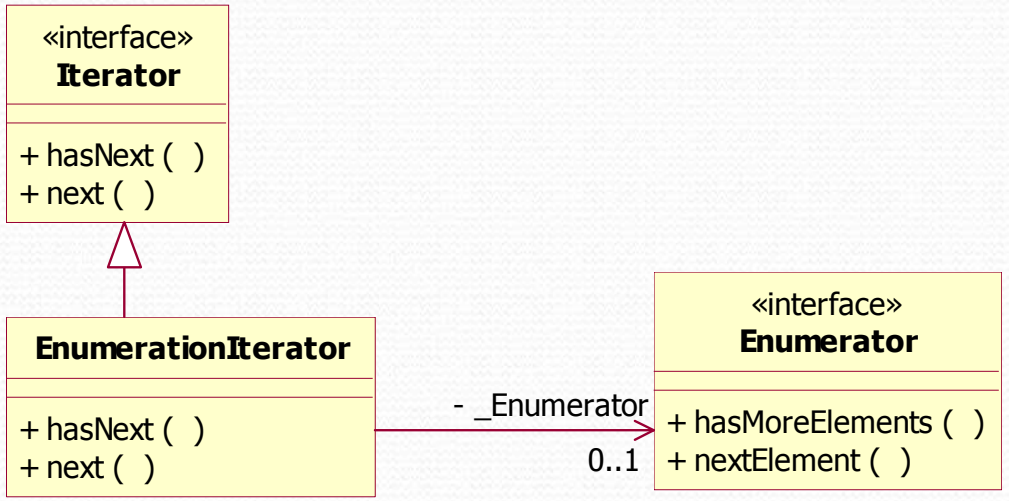
Adapter

- Allows for incompatible interfaces to work with each other by converting the interface of one class to what the client expects

Example of Adapter Pattern



Adapter - Structure



- **Target (Iterator)**: defines interface that Client uses
- **Adapter (EnumerationAdapter):** Adapts interface to target interface
- **Adaptee (Enumerator):** Defines existing interface that needs adaptation
- **Client:** A class that collaborates with objects conforming to the Target interface

Adapter - Code

```
//Target
public interface Iterator {
public boolean hasNext();
public object next();
}

//Adaptee
public interface Enumeration {
// Operations to be adapted by the
   Adapter.
public boolean
   hasMoreElements();
public object nextElement();
}
```

```
//Adapter
public class EnumerationIterator
   implements Iterator {
Enumeration enum;
public EnumerationIterator
   (Enumeration enum) {
   this.enum = enum;
}
public boolean hasNext() {
   return enum.hasMoreElements();
}
public Object next() {
   return enum.nextElement();
}
}
```

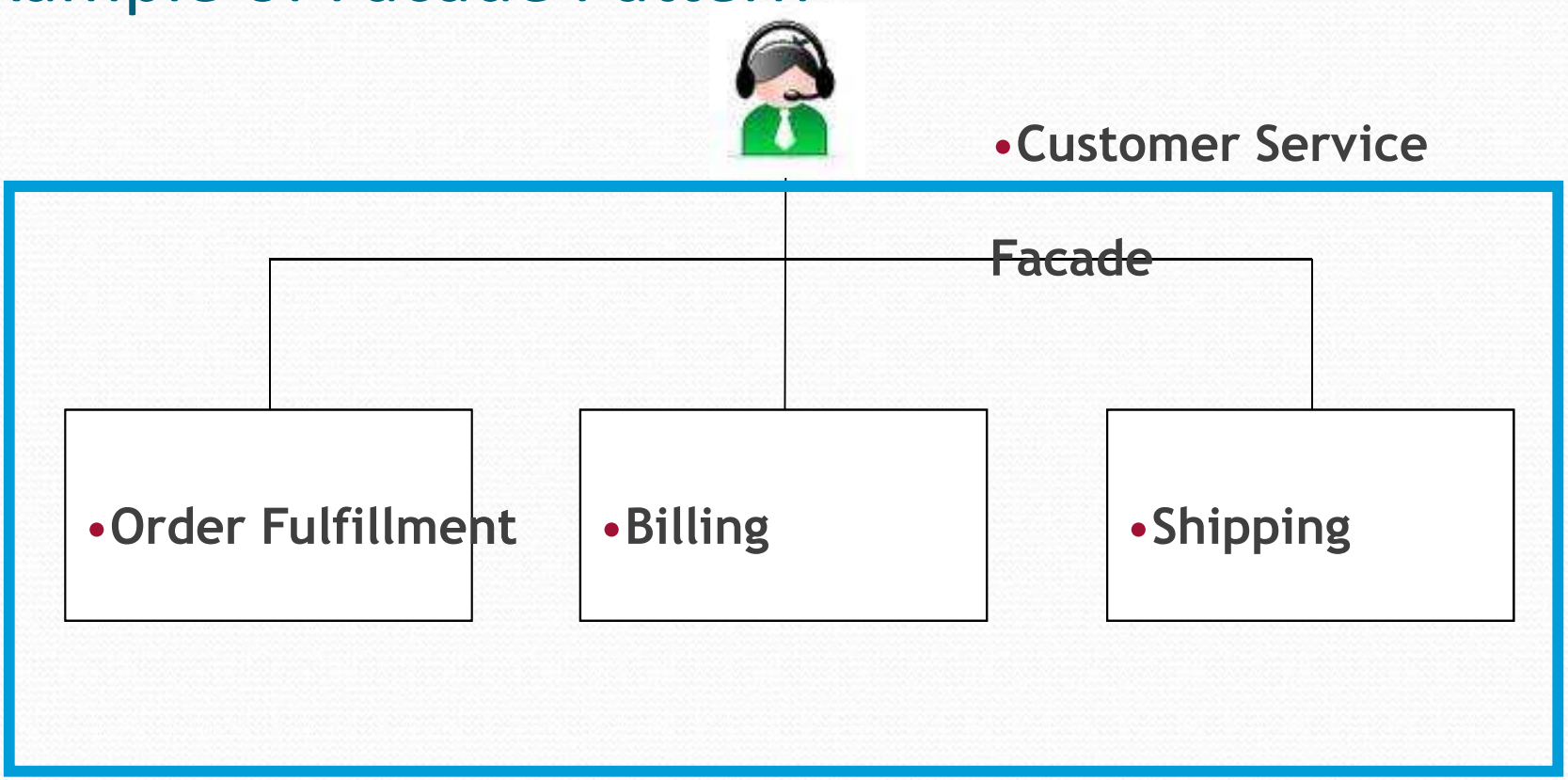
Adapter Pattern

- Pros:
  - It allows two or more incompatible objects to communicate and interact.
  - It improves reusability of older functionality.

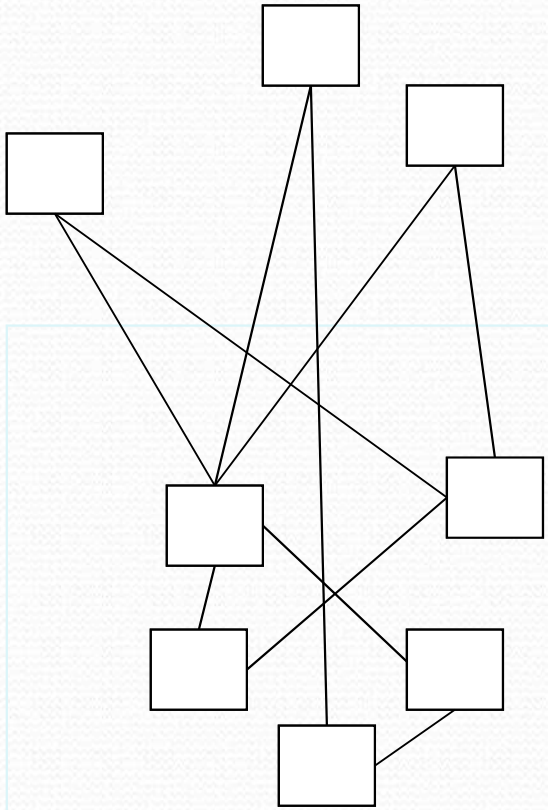
Facade Pattern

- Provides a unified interface to a set of interfaces in a subsystem
  - Wraps a complicated subsystem with a simpler interface.
- You can wrap a complicated subsystem with a simpler interface.

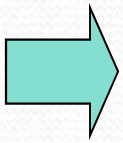
Example of Facade Pattern



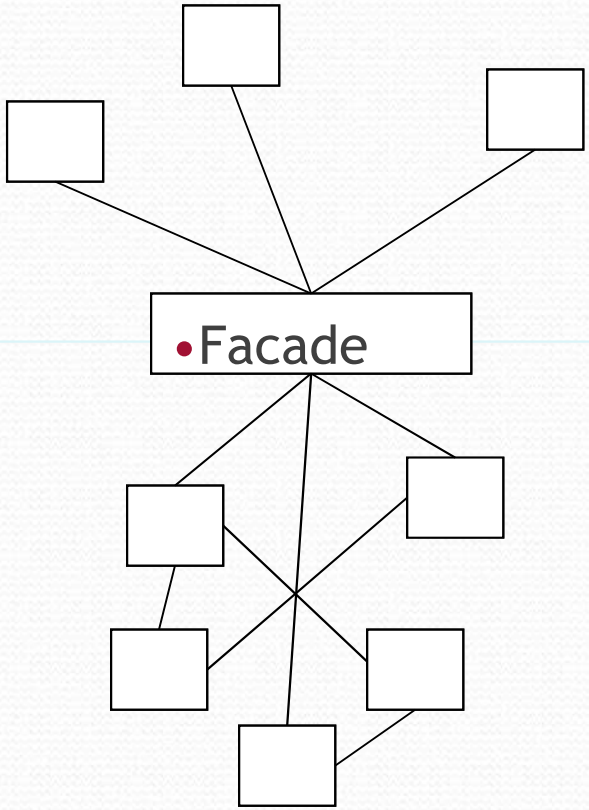
Structure of Facade Pattern



- *client classes*



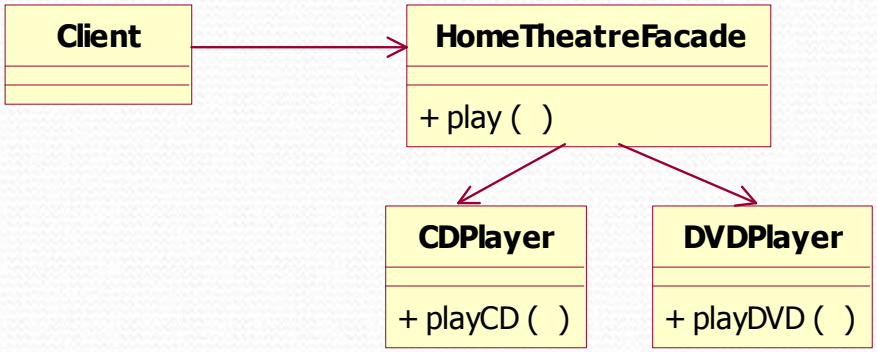
- *subsystem classes*



Facade - Structure

- **Facade (HomeTheatreFacade):** It knows about the subsystem classes and delegates the client request to the appropriate subsystem

- **Subsystem Classes (CDPlayer and DVDPlayer):** Implements the subsystem functionality by handling work assigned by the Facade.



Facade - Code

```
//Facade
public class HomeTheatreFacade {
private CDPlayer aCDPlayer;
private DVDPlayer aDVDPlayer;
public void play() {
/* Route the operation to CDPlayer or
    DVDPlayer depending on the
    requirement i.e. One of the two
    statements below
   aCDPlayer.playCD();
   aDVDPlayer.playDVD();
*/
}
}
```

```
//Subsystem classes
public class CDPlayer {
public void playCD() {
//Specific Code
}
}
//Subsystem classes
public class DVDPlayer {
public void playDVD() {
//Specific Code
}
}
```

Facade Pattern

- Pros:
  - It is a simple interface to a complex system.
  - It shields clients from directly accessing the subsystem components.
  - It promotes weak coupling between the subsystem and its clients.

Introducing Behavioral Patterns

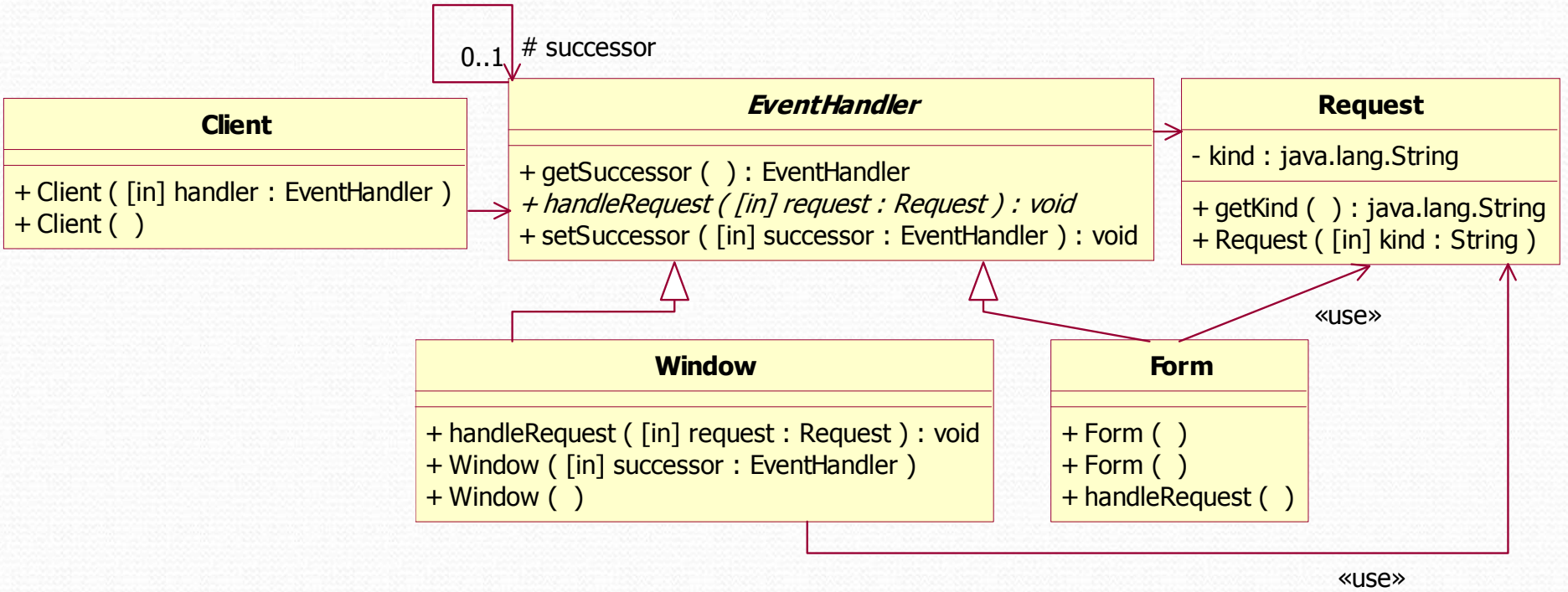
- Behavioral patterns are those that deal with interactions between the objects

- Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it

Chain of Responsibility

- Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
  - Chains the receiving objects and passes the request along the chain until an object handles it.

Chain of Responsibility - Structure



- **Handler (EventHandler):** Defines an interface for handling requests and optionally implements the successor link.
- **ConcreteHandler (Window and Form):** Handles requests it is responsible for and for the rest, it forwards the request to its successor.

Chain of Responsibility - Code

```
//Handler
public abstract class EventHandler {
protected EventHandler successor;
public EventHandler getSuccessor() {
// Return the successor based on application need
 return this.successor;
}
// Operation to be implemented by the ConcreteHandler.
public abstract void handleRequest(Request request);
public void setSuccessor(Handler successor) {
//stores successor
this.successor = successor;
}
}
```

```
//Concrete Handler. Similar code for Form class.
public class Window extends EventHandler {
public Window(Handler successor) {
//Setting reference to successor
this.successor = successor; }
public Window() { }
public void handleRequest(Request request) {
// Handle request else forward the request to successor.
if ( request.getKind().equals("test")) {
// add your codes here to handle the request.}
else { // the successor will handle the request.
this.successor.handleRequest(request);
} }
}
```

Examples

- Real Life: Issue escalation
- IT : Exception Handling in OO languages like Java

Chain of Responsibility Pattern

- Pros:
  - It provides for reduced coupling.
    - The receiver and the sender have no explicit knowledge of each other.
  - It provides for added flexibility in assigning responsibilities to objects.

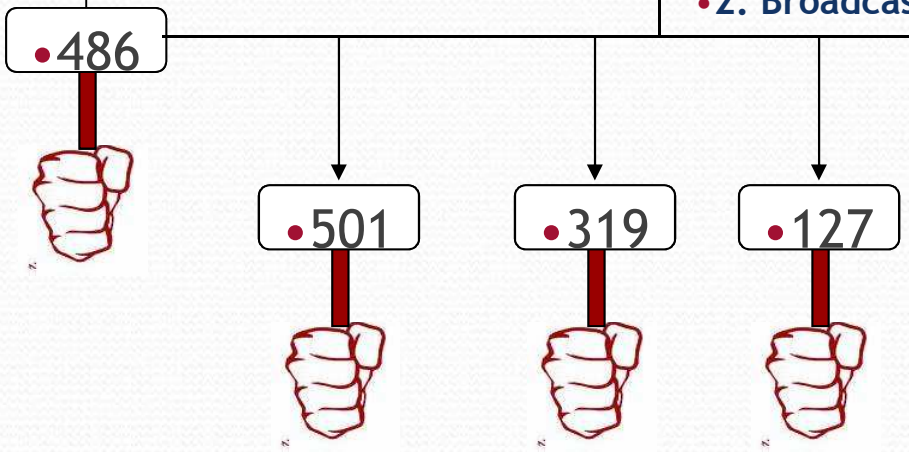
Usage of Observer Pattern

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

- It is also known as “Dependents” and “Publish-Subscribe”.

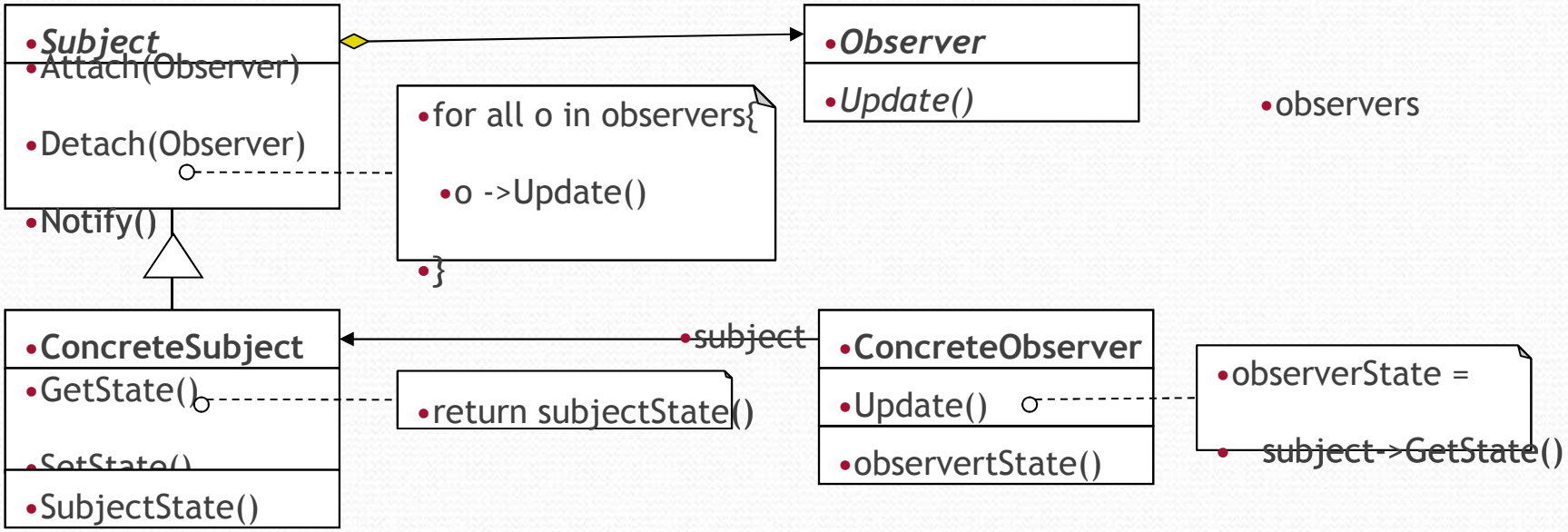
Example of Observer Pa n

- Auctioneer (Subject)
- 1. Accept Bid
- 2. Broadcast New High Bid



- Bidders (Observers)

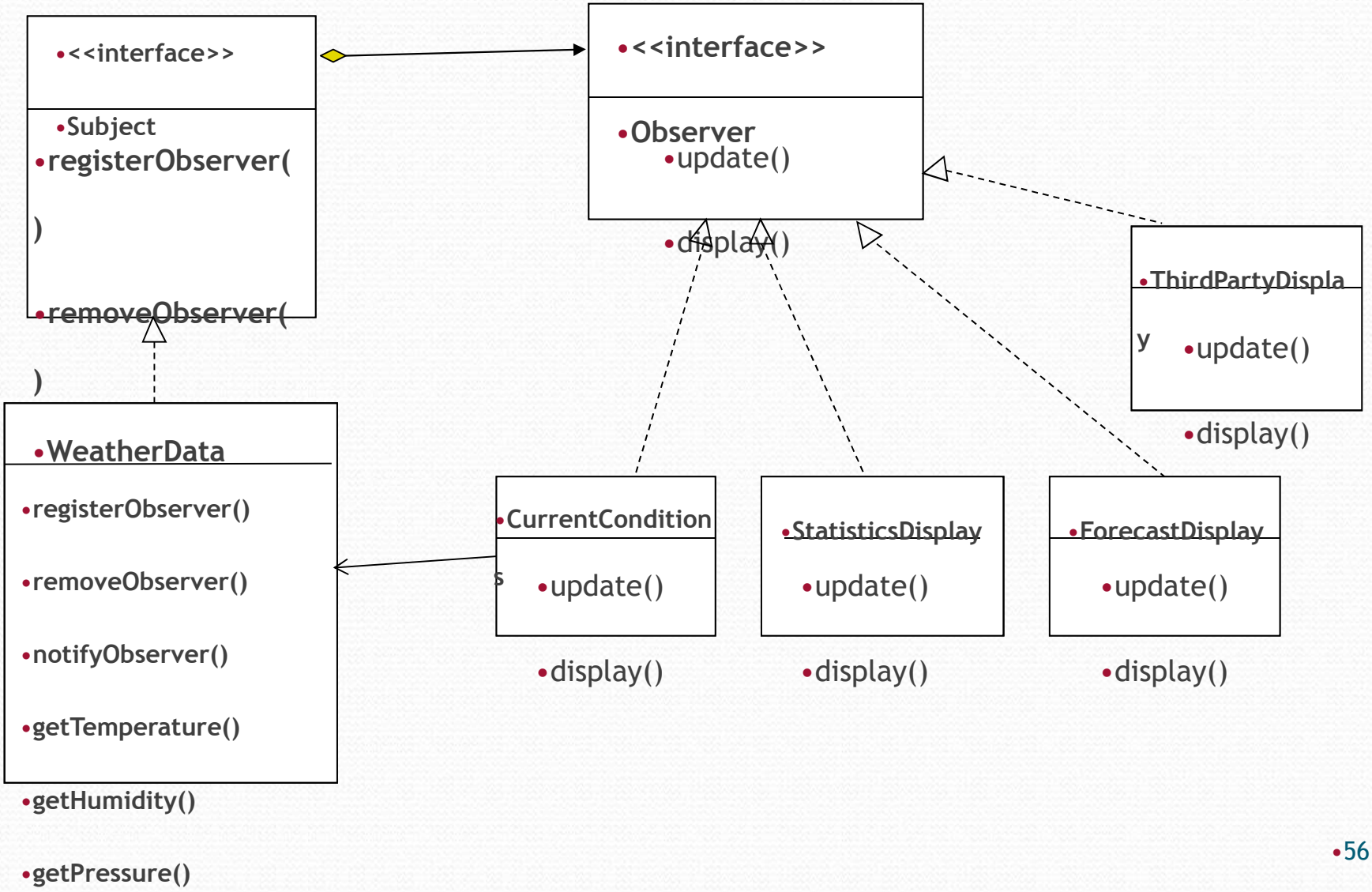
Structure of Observer Pattern



Case Study on Observer Pattern

- Requirement is to build a next generation Internet-based Weather Monitoring Station
  - The weather station will be based on a **WeatherData** object which tracks current weather conditions (temperature, humidity, and barometric pressure).
  - The application should initially provide three display elements: current conditions, weather statistics, and a simple forecast. All these should be updated in real time as the **WeatherData** object acquires the most recent measurements.

Observer Case Study Solution



Observer Pattern

- Pros:
  - It provides abstract coupling between Subject and Observer.
  - It provides support for broadcast communication.

Summary

- You now have an understanding of
  - What design patterns are
  - Gang Of Four Patterns

- Just for laffs

On a Lighter note ☺

Software and Church are much the same: Why?

First we build them then we pray!!! ☺