

GoF Patterns Revisited

DSN Team

Agenda

- Creational Patterns
 - Singleton
 - Factory Pattern
 - Builder
- Structural Patterns
 - Adapter
 - Façade
 - Flyweight
 - Proxy

Agenda - contd

- Behavioural Patterns
 - Command
 - Observer
 - Strategy
 - Template method

Creational Patterns

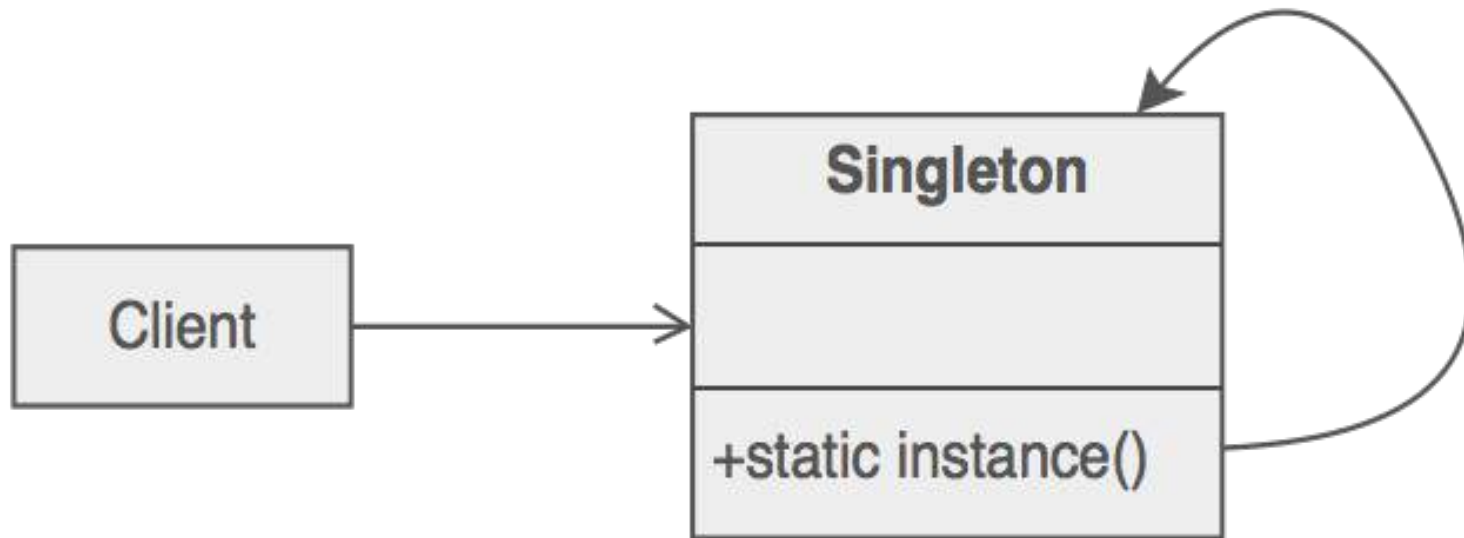
- Singleton
- Factory Pattern
- Builder

Singleton Pattern

Singleton

- Motivation
 - Ensure that only one instance of a class is created.
 - Provides a global point of access to the object.
- Problem / Forces
 - To have only one instance for a class like a global variable.
 - One instance means only one instance for the entire application.
 - The same class itself is responsible for creating only one instance.

UML Representation



Implementation

- Prior to JavaSE5 Singletons are implemented using
 - private constructor
 - private static member variable to hold reference to the instance of the class
 - public static method usually named as `getInstance()` to return the instance of the class

Prior to JavaSE5

- Thread-safe implementation for multi-threading use
 - This implementation ensures that there is only one instance even multiple threads access it.
 - But synchronization at the method level is very expensive when it comes to performance.

Thread safe implementation

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

        return instance;
    }
    ...
    public void doSomething()
    {
        ...
    }
}
```

Lazy instantiation using double locking mechanism

- If the singleton object is already created we just have to return it without using any synchronized block
- This optimization consist of checking in an unsynchronized block if the object is null and if not to check again and create it in an synchronized block. This is called double locking mechanism.

Lazy instantiation using double locking mechanism

```
public static Singleton getInstance()  
{  
    if (instance == null)  
    {  
        synchronized(Singleton.class)  
        {  
            if (instance == null)  
            {  
                System.out.printlnFirst time getInstance was invoked!";  
                instance = new Singleton();  
            }  
        }  
    }  
  
    return instance;  
}
```

Creating Singletons from JaveSE5 and later

- Enum's in JavaSE5 are by default Singleton and thread safe
- It's very easy to write Singleton using Enum.
- Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program.

JavaSE5 Enum is Singleton

```
public enum EnumSingleton {  
  
    INSTANCE;  
  
    public static void doSomething(){  
        //do something  
    }  
}
```

- you can access doSomething() method as EnumSingleton.INSTANCE.doSomething()

Other problem in Traditional Singletons

- Serialization and De-serialization problem.
- JavaSE5 Enum's to the rescue.
- Developers need not have to bother about Serialization problem
- It's automatically taken care by the JVM

Singleton vs Static Class

- Singleton object stores in Heap but, static object stores in stack.
- We can clone the object of Singleton but, we can not clone the static class object.
- Singleton class follow the OOP(object oriented principles) but not static class.
- We can implement interface with Singleton class but not with Static class.
- A singleton can be initialized lazily or asynchronously while a static class is generally initialized when it is first loaded.
- Singleton can maintain state but static class can't.

Bad Practices

- Don't go for Singleton
 - just to avoid creating multiple instances for a class.
 - Instead when you want to maintain a shared state in the entire application go in for Singleton
 - Unless there is no real need to maintain a global state.
- Singletons instances are not garbage collected often as the instance is static.
- Too many singletons may lead to memory leak.

Creational Patterns

- Singleton
- Factory Pattern
- Builder

Factory Pattern

What is wrong with new

What is wrong with **new**.
We aren't supposed to
program to a imp, but every
time I use that's exactly
what I am doing. Right..?



We want to use
interface to keep code
flexible

Cache cache = new CacheLRU

But we have to create
an instance of a
concrete class

What is wrong with new

Remember that design
should be
open for extension and
Closed for modification

But we have to
create an object at
some point and java
gives as one way to
create an object,
right? So what gives ?

Technically there is nothing wrong with new,
after all it's part of java

For flexible, it
should be
interface, but we
can't directly
create either of
those

```
public void putCache(String key, String value) {  
    Cache cache = new CacheLRU();  
    cache.put(key,value);  
}
```




Identifying the aspects that vary

```
public void putCache(String key, String value, String cacheType)
{
    Cache cache = null;

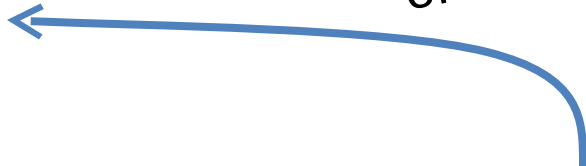
    if(cacheType == "LRU") {
        cache = new CacheLRU();
    } else if(cacheType == "LRUT") {
        cache = new CacheLRUT();
    }

    cache.put(key,value);
}
```

We are passing type
of cache to put cache



Based on the type of cache,
we instantiate the correct
concrete class and assigned
to the cache



But the pressure is on to add more cache types

```
public void putCache(String key, String value, String cacheType) {
```

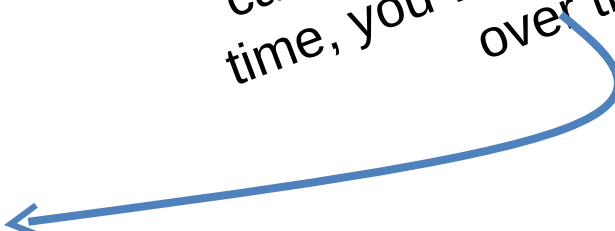
```
    Cache cache = null;
```

```
    if(cacheType == "LRU") {  
        cache = new CacheLRU();  
    } else if(cacheType == "LRUT") {  
        cache = new CacheLRUT();  
    } else if(cacheType == "Week") {  
        cache = new WeekHashCache();  
    } else if(cacheType == "LRUT") {  
        cache = new HashtableImpl();  
    }  
}
```


```
    cache.put(key,value);
```

```
}
```

This is what varies. As the cache section changes over time, you will modify this code over time



The code is not closed for modification, if the cache changes we have to modify it



Encapsulating object creation

```
public void putCache(String key, String value, String cacheType) {
```

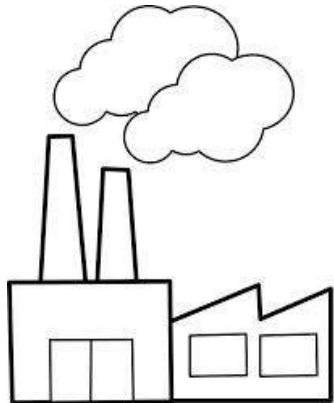
```
    Cache cache = null;
```



```
    cache.put(key,value);
```

```
}
```

Pull the object
creation code out of
putCache



Place that code in an object
that is only going to worry
about how to create cache

```
(if(cacheType == "LRU") {  
    cache = new CacheLRU();  
} else if(cacheType == "LRUT") {  
    cache = new CacheLRUT();  
} else if(cacheType == "Week")  
{  
    cache = new  
    WeekHashCache();  
} else if(cacheType == "LRUT") {  
    cache = new HashtableImpl();  
})
```


Building a Simple Cache factory

Our new class, the CacheFactory. It has to create Cache for it's client

First we define a getcache() method in a factory. This is the method all client will use to instantiate new cache object

```
public class CacheFactory {  
  
    public Cache getCache(String cacheType) {  
        Cache cache = cacheContainer.get(cacheType);  
        return cache;  
    }  
  
    private Map cacheContainer = null;  
    // init cache container separately  
}
```

To avoid if else condition we used map to create cache object. This logic can be changed based on the need

Reworking the putCache

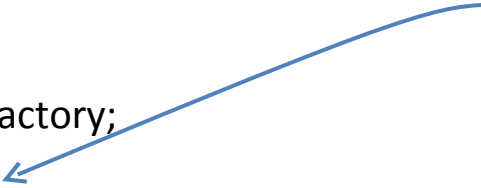
```
public class CondizioneCache {
```

```
    private CacheFactory cacheFactory;
```


```
    public void setCacheFactory(CacheFactory cacheFactory) {  
        this.cacheFactory = cacheFactory;  
    }
```

```
    public void putCache(String key, String value, String cacheType) {  
        Cache cache = cacheFactory.getCache(cacheType);  
        cache.put(key, value);  
    }  
}
```


CondizioneCache get the
factory passed to it in the
setter



putCache method uses
the factory to create it
cache simply by passing
the type of cache



No more concrete instantiate here. We
replaced the **new** with a getCache
method on the **factory** object



Doe's and Don't

```
public class CacheFactory {  
  
    public Cache getLRUCache() {  
        return new CacheLRU();  
    }  
  
    public Cache getLRUTCACHE() {  
        return new CacheLRUT();  
    }  
}
```



Wrong usage of factory class

```
public class CacheFactory {  
  
    public Cache getCache(String cacheType) {  
        Cache cache = cacheContainer.get(cacheType);  
        return cache;  
    }  
  
    private Map cacheContainer = null;  
    // init cache container separately  
}
```



Correct usage of factory class

Conclusion

- The basic principle behind this pattern is that, at run time, we get an object of similar type based on the parameter we pass
- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Forces principle of programming for interface rather than implementation - DIP
- Forces Open for extension and closed for modification - **OCP**
- Instead of having object creation code on client side we encapsulate inside Factory method - **Encapsulation**
- Are you going to use the factory in multiple places? If you are not, you don't need a factory

Creational Patterns

- Singleton
- Factory Pattern
- Builder

Builder

Who is Builder ?



A Person who is taking care building a product.



How to solve this problem?

- I need a immutable object with some mandatory fields and some optional fields.
- How am I going to allow a client to create Pizza?

```
public class Pizza {  
  
    private int size; // mandatory field  
    private boolean tomato; // optional field  
    private boolean pepper; // optional field  
  
}
```


Setter Methods?

- We can go for setter Methods right?



If so.. It is no longer immutable...
Isn't it?

Telescopic Constructor

- We can go for telescopic constructor like

```
public class Pizza {  
  
    private final int size; // mandatory field  
    private boolean tomato; // optional field  
    private boolean pepper; // optional field  
  
    public Pizza(int size) {  
        this.size = size;  
    }  
    public Pizza(int size, boolean tomato) {  
        this(size);  
        this.tomato = tomato;  
    }  
    public Pizza(int size, boolean tomato, boolean pepper) {  
        this(size, tomato);  
        this.pepper = pepper;  
    }  
}
```

Telescopic Constructor

- What is wrong with telescopic constructor?

If I have to create a pizza... I have to remember the parameter order... and that hurts.. Is there any better way?

- Order of parameter is important.
possibility of passing wrong
parameter if same data type is
involved.



Pizza Builder

```
public class PizzaBuilder {  
  
    private final int size; // mandatory field  
    private boolean tomato; // optional field  
    private boolean pepper; // optional field  
  
    public PizzaBuilder(int size) {  
        this.size = size;  
    }  
    public PizzaBuilder tomato(boolean tomato) {  
        this.tomato = tomato;  
        return this;  
    }  
    public PizzaBuilder pepper(boolean pepper) {  
        this.pepper = pepper;  
        return this;  
    }  
    public Pizza build() {  
        return new Pizza(size, tomato, pepper);  
    }  
}
```

Building Pizza

```
Pizza pizza = new PizzaBuilder(10).pepper(true).tomatto(true).build();
```



The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters

Structural Patterns

- Adapter
- Façade
- Flyweight
- Proxy

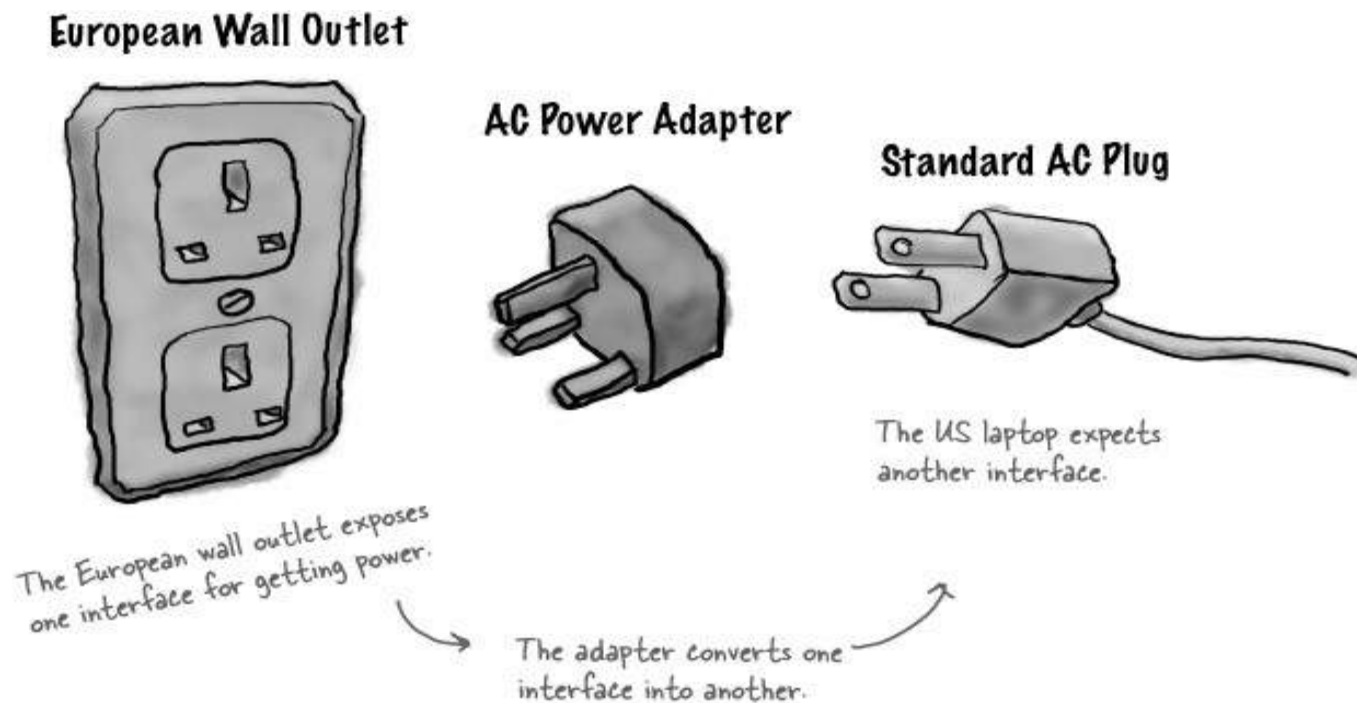
Structural Patterns

- Adapter
- Façade
- Flyweight
- Proxy

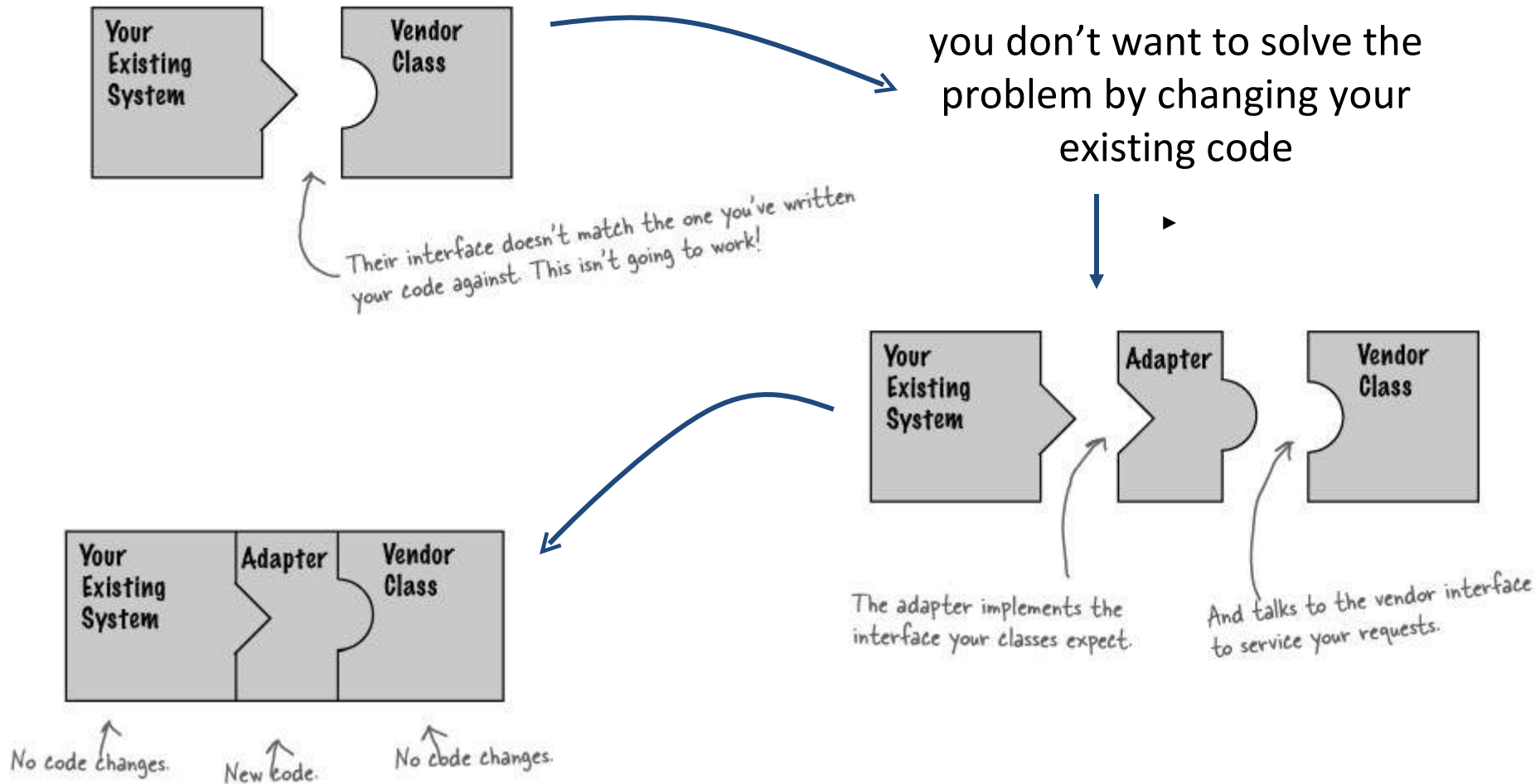
Adapter Pattern

Adapters all around us

Adapter sits in between the plug of your laptop and the European AC

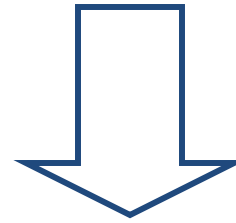


Object oriented adapters



Where we can use adapter

When and what is the possible situation we can use adapter



When a external subsystem class has been designed to accept input in a certain format but our system not providing the input in the required format. we can use an adapter to change the input format to the required one.



Adapter Pattern explained

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class ClassificazioneViewAdapter implements IClassificazioneView {
```

```
    private ClassificazioneView classificazioneAdaptee = null;
```

```
    public ClassificazioneViewAdapter(final ClassificazioneView view) {  
        this.classificazioneView = view;  
    }
```

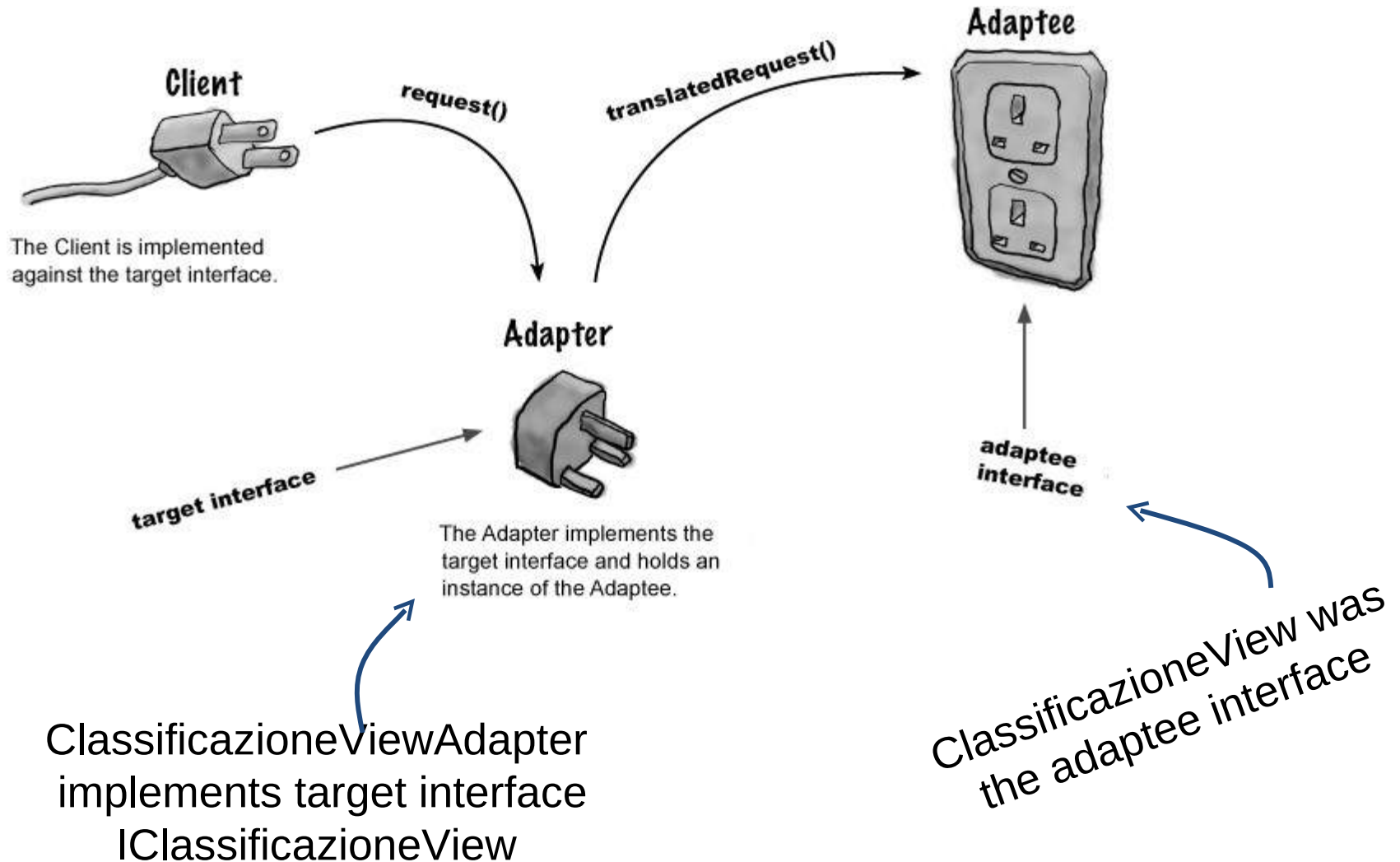
```
    public Long getId() {  
        return classificazioneView.getId();  
    }
```

```
    public String getDescrizione() {  
        return classificazioneView.getDescrizione();  
    }  
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface;

Adapter Pattern explained



Conclusion

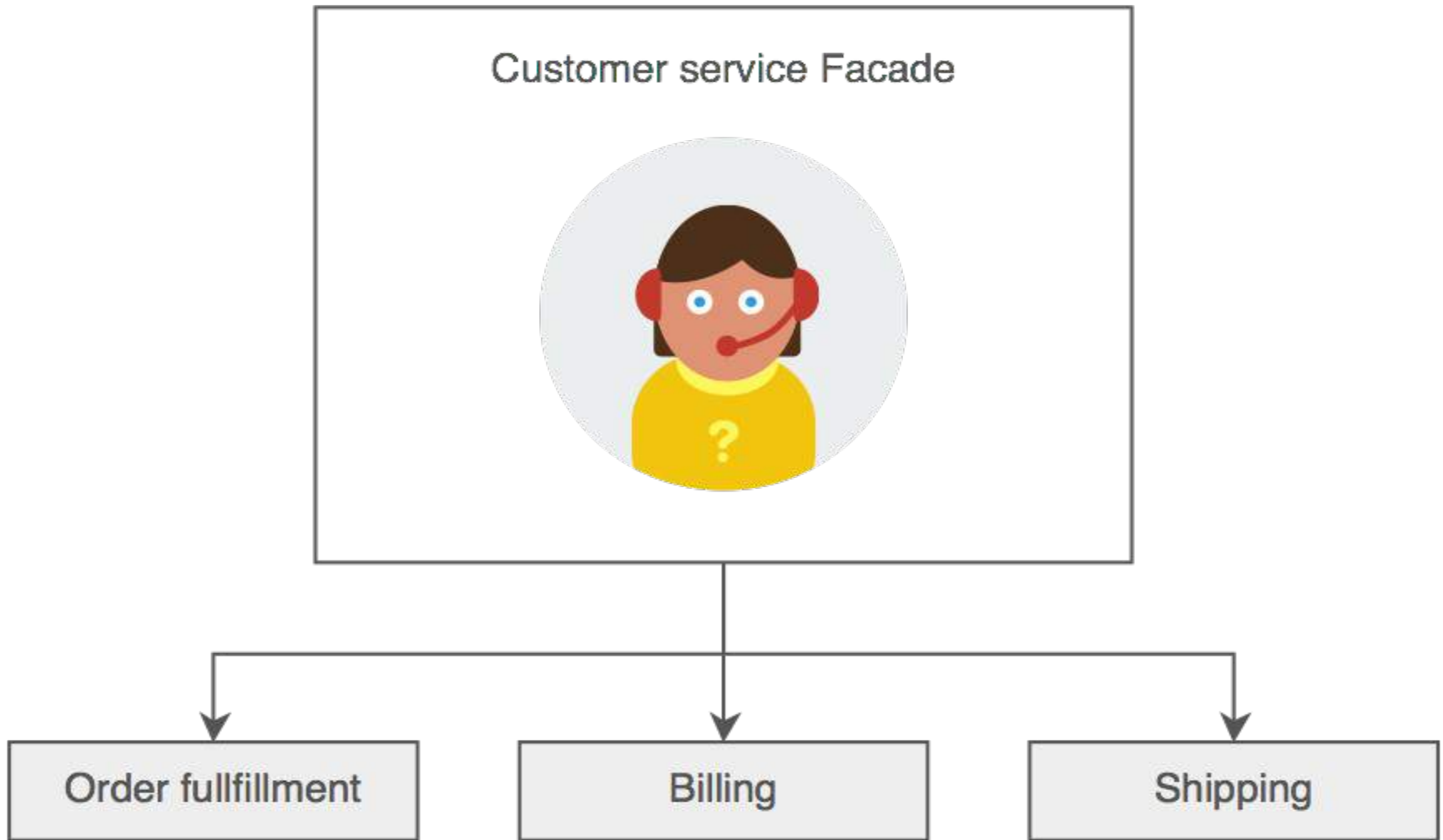
- Adaptor is used to make two parties talk with each other, because their communication channel is not compatible with each other
- Adaptor convert the interface of a class into another interface clients expect
- Instead of touching the client class each time, we can use an adapter to change the format to the required one

Structural Patterns

- Adapter
- Façade
- Flyweight
- Proxy

Facade Pattern

Example



Example

- The consumer calls one number and speaks with a customer service representative.
- The customer service representative acts as a Façade providing an interface to
 - the order fulfilment department
 - the billing department
 - and the shipping department.

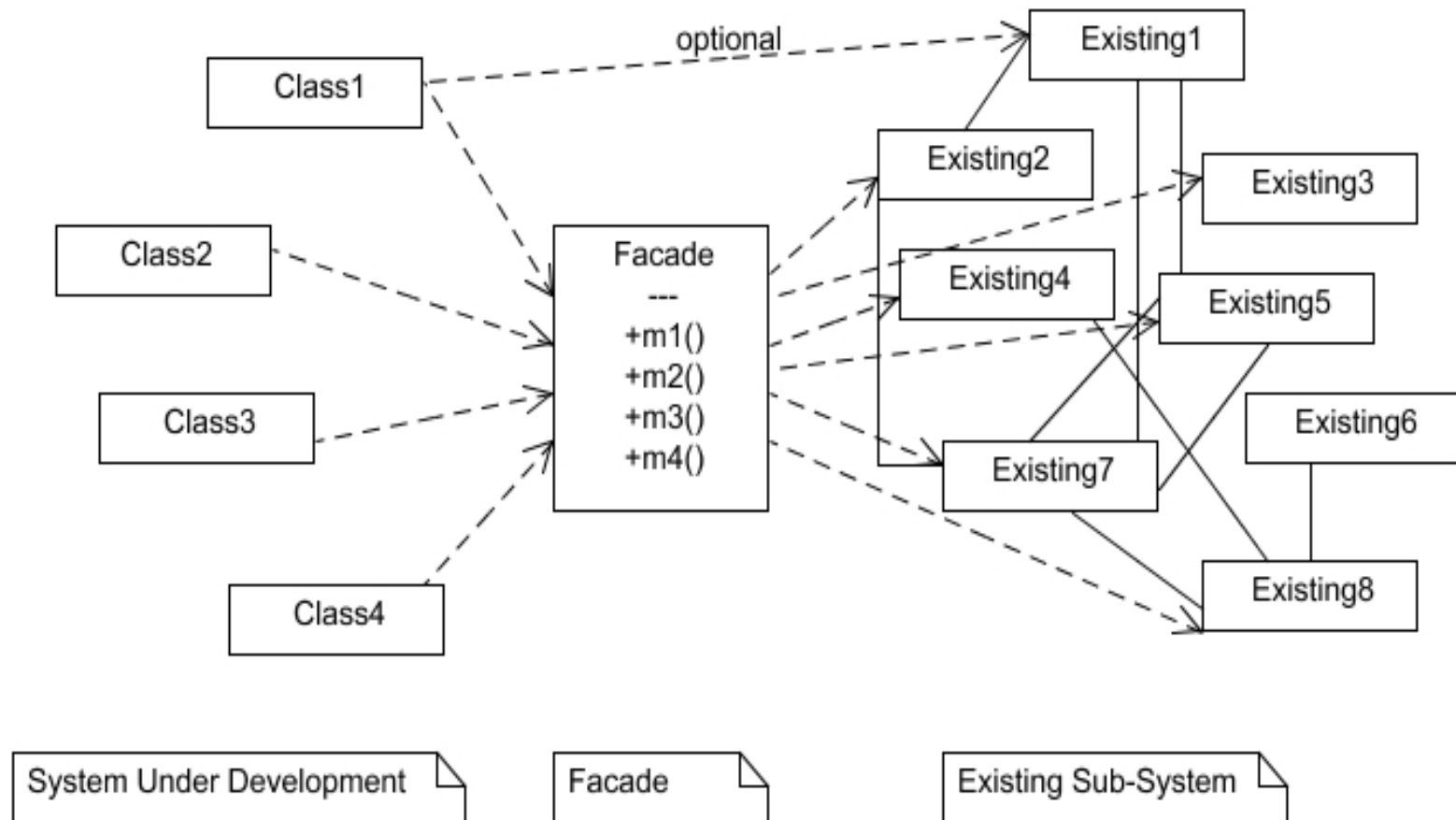
Problem / Forces

- Problem / Forces
 - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.
 - Promotes decoupling the subsystem from its potentially many clients.
 - Facade is the only access point for the subsystem.

Intent

- Intent
 - Provide a unified interface to a set of interfaces in a subsystem.
 - Facade defines a higher-level interface that makes the subsystem easier to use. Wrap a complicated subsystem with a simpler interface.

UML Representation



Rules of thumb

- Facade defines a new interface, whereas Adapter uses an old interface.
- Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Adapter and Facade are both wrappers; but they are different kinds of wrappers.
- The intent of Facade is to produce a simpler interface.
- And the intent of Adapter is to design to an existing interface.

Facade vs Business Delegate vs Session Facade

- Facade Provide a unified interface to a set of interfaces in a subsystem.
- Session Facade resides on the business-tier. It's implemented as SLSB encapsulates complex interactions between business objects participating in a workflow.
- Business Delegate resides on the presentation-tier. It is used to reduce the coupling between the presentation-tier and the business tier

Structural Patterns

- Adapter
- Façade
- Flyweight
- Proxy

Flyweight Pattern

What is Flyweight?



Flyweight

- There are two types of **states** for an object
 - Intrinsic** – things that are constant and will not change and stored in memory.
 - Extrinsic** – that are not constant and calculated using intrinsic state and not stored in memory.
- When our object is immutable and we don't want to create instance if it is already present. (Like our master table Data)

How to create Flyweight?

- Identify shareable state (intrinsic) and non-shareable state (extrinsic)
- Create a Factory that can return an existing object or a new object
- The client must use the Factory instead of "new" to request objects
- The client (or a third party) must provide/compute the extrinsic state

Flyweight

```
public class PaymentStatus {  
    private Long statusId;  
    private String status;  
    private String statusDesc;  
}
```

```
public class PaymentStatusFactory {  
  
    private Map<String, PaymentStatus> cache = new HashMap<String, String>();  
  
    public PaymentStatus getStatus(String statusCode) {  
        PaymentStatus found = cache.get(statusCode);  
        if(found == null) {  
            found = paymentStatusDao.find(statusCode);  
            cache.put(statusCode);  
        }  
        return found;  
    }  
}
```

Benefits & Drawbacks

- Reduces overall memory usage particularly when not all items are used or when items are used by multiple consumers.
- Can cause Memory leak if you wrongly choose the Flyweight object.
- Caution : Flyweight object should not contain any shared state!!

Structural Patterns

- Adapter
- Façade
- Flyweight
- Proxy

Proxy Pattern

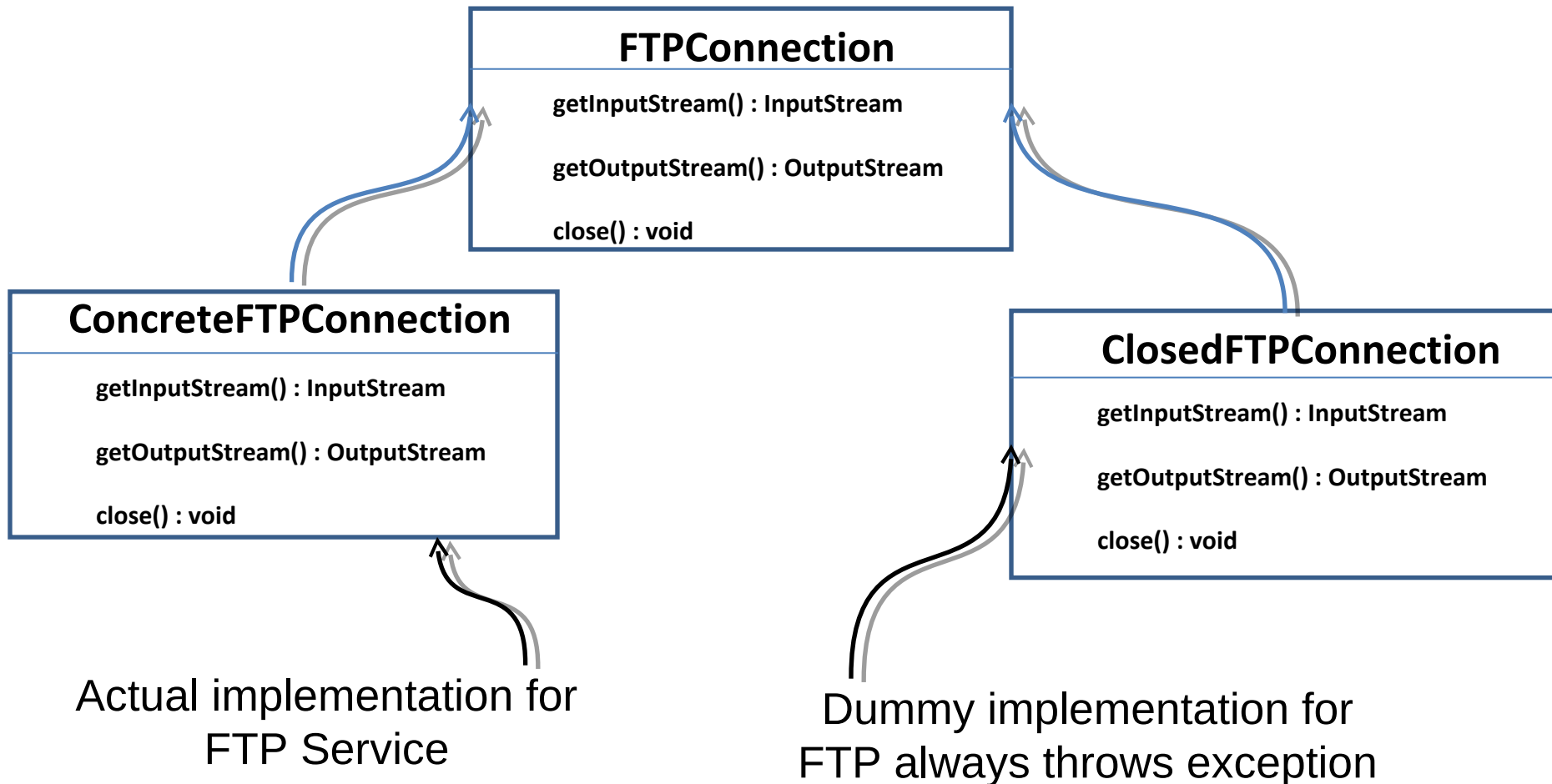
Coding to control access

```
public interface FTPConnection {  
    InputStream getInputStream();  
    OutputStream getOutputStream();  
    void close();  
}
```

FTPConnection provides
FTP client functionality
can fetch files and file
details from the server,
and also upload files to
the server

The Close method should be called
when the required FTP operations have
been completed so that the connection
to the server is released

Coding to control access



Controlling Object Access

Hey team, i'd really like to controls access to real object after client called close method. Can you find a way to control it.



Sounds easy, If you remember we have already two FTPConnection implementation class ConcreteFTPConnection and ClosedFTPConnection. All we need to do is when client invoke close method we need to change object reference from ConcreteFTPConnection to ClosedFTPConnection

Introducing Proxy

Well, we need to find is there any existing solution for this problem, before start into code

Don't worry guys. I've been brushing upon my design pattern. All we need is a proxy



Coding Proxy Class

```
public class ProxyFTPConnection implements FTPConnection {
```

```
    private FTPConnection ftpConnection = null;
```

```
    private ClosedFTPConnection closedFTPConnection = new closedFTPConnection();
```

```
    public ProxyFTPConnection(FTPConnection ftpConnection) {  
        this.ftpConnection = ftpConnection;  
    }
```

```
    public InputStream getInputStream() {  
        return ftpConnection.getInputStream();  
    }
```

```
    public OutputStream getOutputStream() {  
        return ftpConnection.getOutputStream();  
    }
```

```
    public void close() {
```

```
        ftpConnection.close();
```

```
        ftpConnection = closedFTPConnection;
```

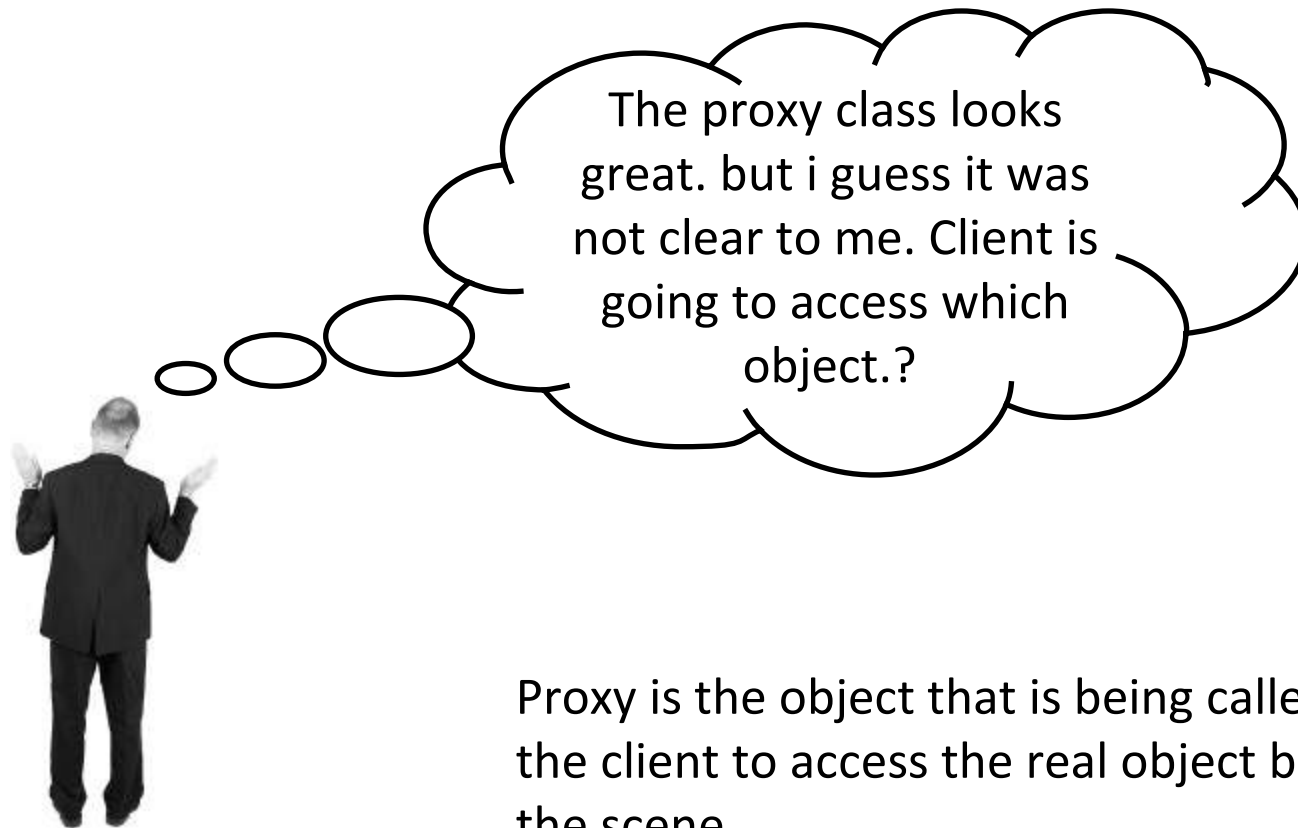
```
    }
```

```
}
```

Proxy class act like a
wrapper class over real
object

Client invoke close method , intern
proxy change object reference from
ConcreteFTPConnection to
ClosedFTPConnection

Creating Proxy object



Proxy is the object that is being called by the client to access the real object behind the scene

Creating Proxy object

```
public class FTPConnectionPool {
```

```
    public FTPConnection getFTPConnection() {  
        FTPConnection ftpConnection = createFTPConnection();  
        return new ProxyFTPConnection(ftpConnection);  
    }
```

```
    // Other method implementation  
}
```

FTPConnection always return
ProxyFTPConnection to client

Adapter vs Proxy vs Decorator

Adapter

Provides two different real object for two different interface which enables the client to interact with each other

Proxy

Provides wrapper over the real object for the same interface

Decorator

Provides two different real object for two different interface which enables the client to interact with each other

Conclusion

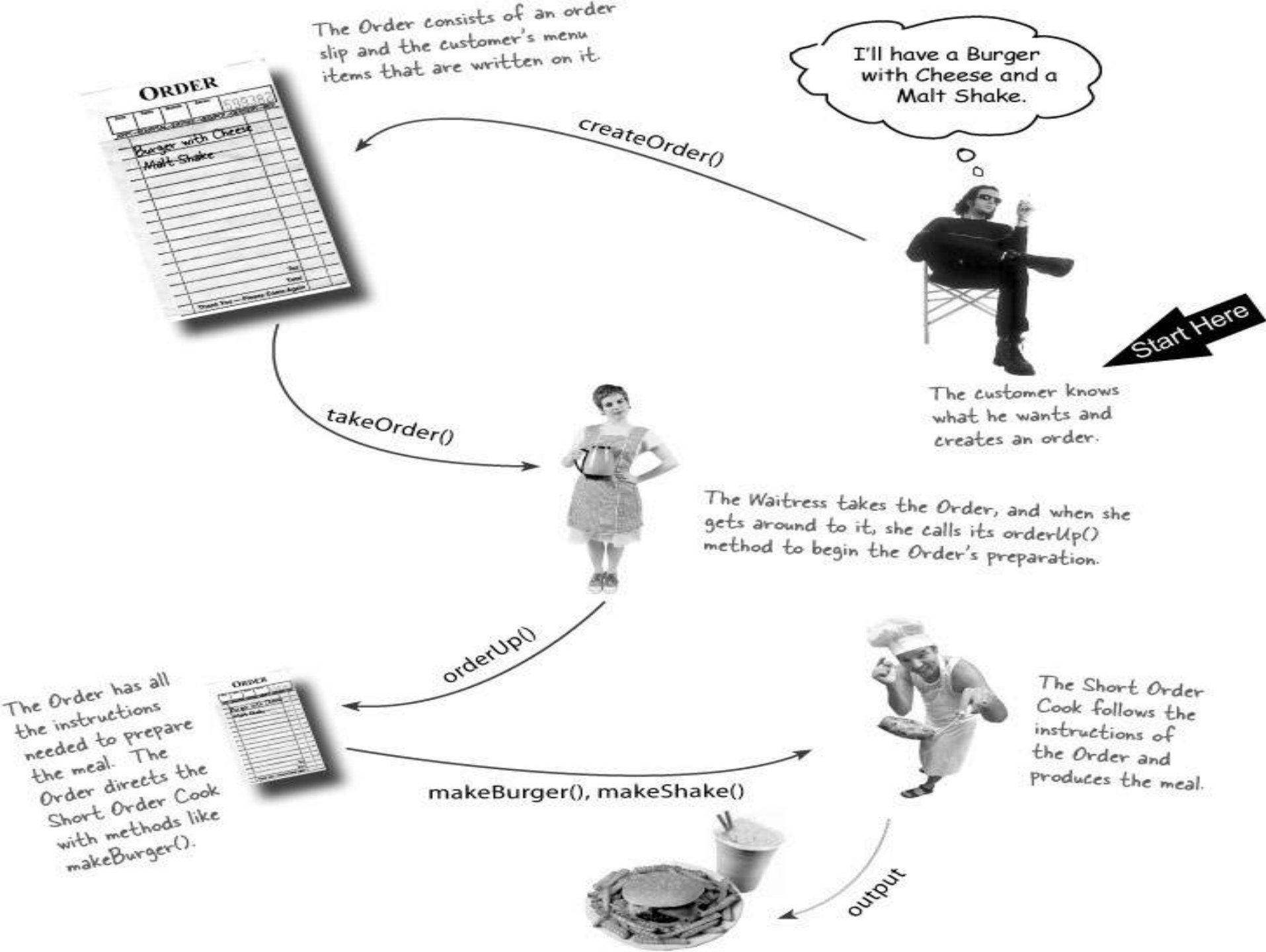
- Proxy design pattern allows you to create a wrapper class over real object.
- Proxy is act like wrapper class, controls access to real object in turn you can add extra functionalities to real object without changing real object's code
- Proxy is the object that is being called by the client to access the real object behind the scene
- In place of or on behalf of are literal meanings of proxy and that directly explains proxy design pattern

Behavioral Patterns

- Command
- Observer
- Strategy
- Template method

Behavioral Patterns

- Command
- Observer
- Strategy
- Template method



Command

- Intent
 - An object-oriented callback
 - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

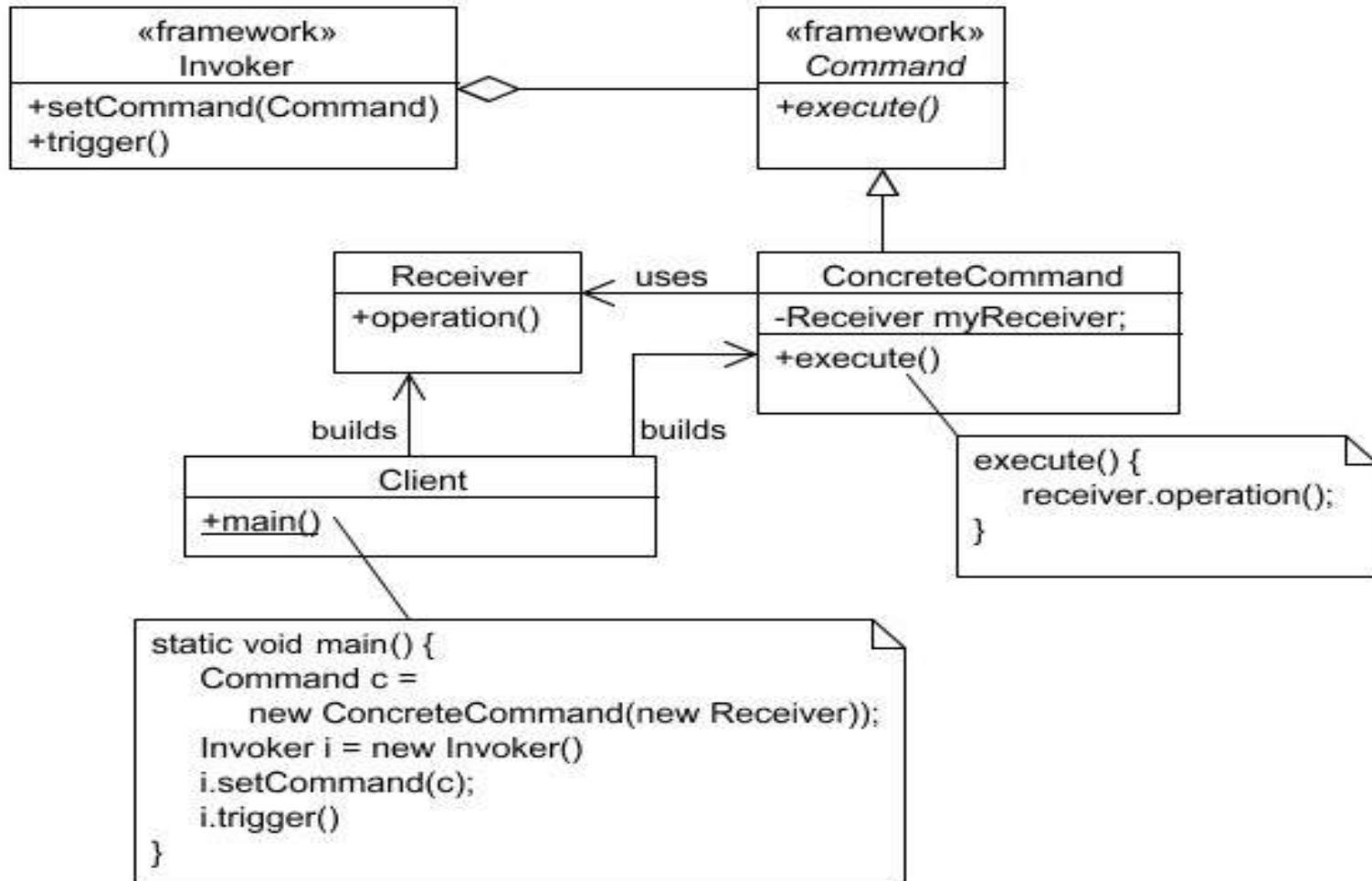
Problem / Forces

- Problem / Forces
 - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

How it works?

- How it works?
 - Command decouples the object that invokes the operation from the one that knows how to perform it.
 - Create an abstract class / interface that contains a method `execute()` or `doSomeAction()`.
 - All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual `execute()`.

UML Representation



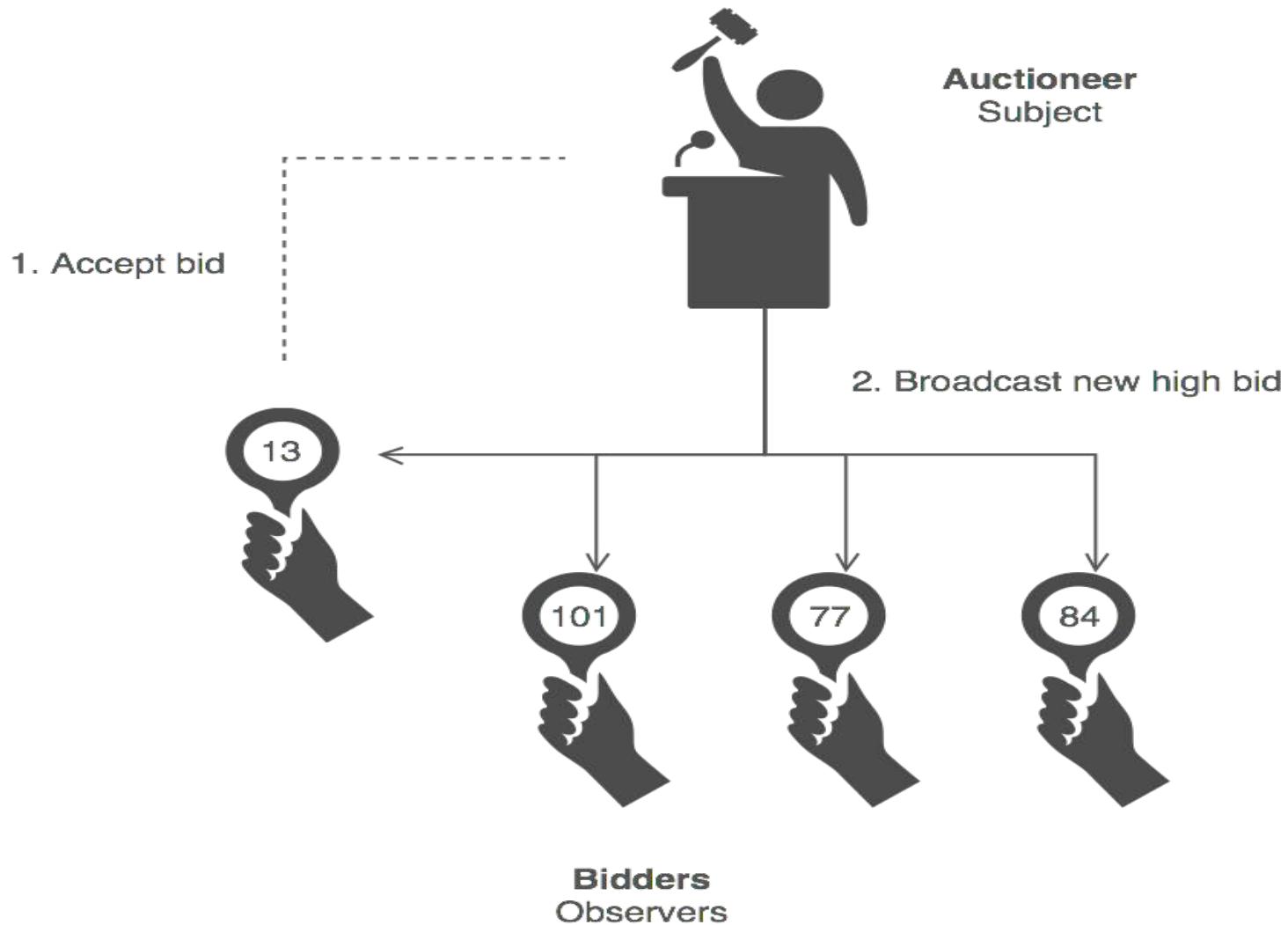
Framework that uses command pattern

- GBS Proprietary frameworks
 - Hibernate wrapper's doInHibernate()
 - OperationTx
- Open source frame works
 - Spring JDBC
 - Apache Common collections
- Java SE
 - Java SE 8 Lamda expressions
 - Concurrency frame work

Behavioral Patterns

- Command
- Observer
- Strategy
- Template method

Example



Example

- Some auctions demonstrate this pattern.
- The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid.
- The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.

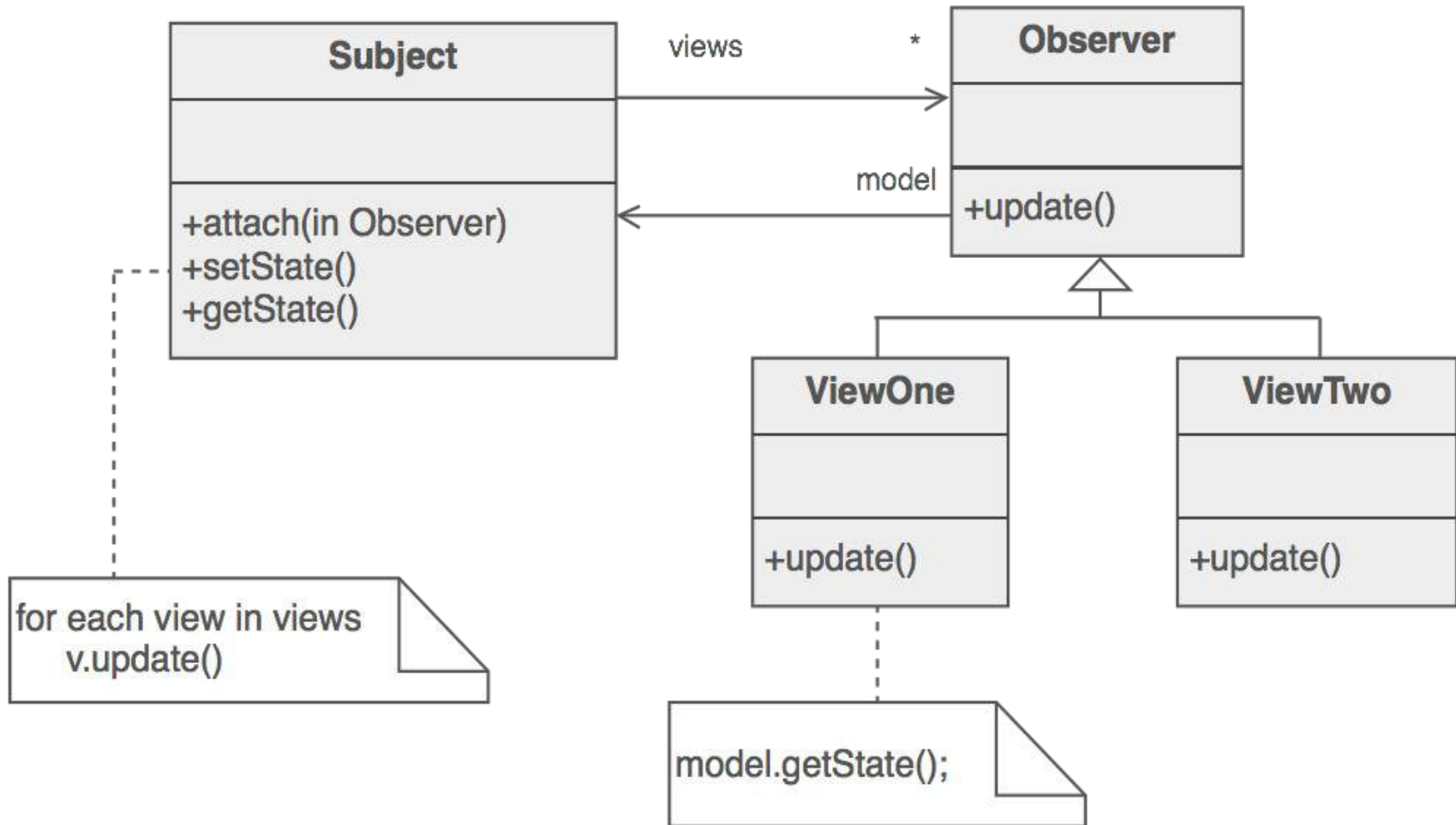
Intent

- Intent
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
 - The "View" part of Model-View-Controller.
 - Observer Pattern is also a good implementation of Hollywood Principle.

Problem / Forces

- Problem / Forces
 - A large monolithic design does not scale well as new graphing or monitoring requirements are needed.

UML Representation



How it works?

- Define an object that is the "Subject" of the data model and/or business logic.
- Delegate all "view" functionality to decoupled and distinct Observer objects.
- Observers register themselves with the Subject as they are created.
- Whenever the Subject changes, it broadcasts to all registered Observers that it has changed.
- Each Observer queries the Subject for that subset of the Subject's state

JEE6 Annotation

- Observer pattern can be implemented in JEE6 using the annotation `@Observes`

Behavioral Patterns

- Command
- Observer
- Strategy
- Template method

Strategy Pattern

Introduction

Hey guys, i got the new CRQ which have several computations / algorithms depending on certain conditions.

I thought to go ahead in implementing by applying either switch case or if else

Initially somehow we manage but if the program demands are too complex then it is difficult to frame such as well as to maintain



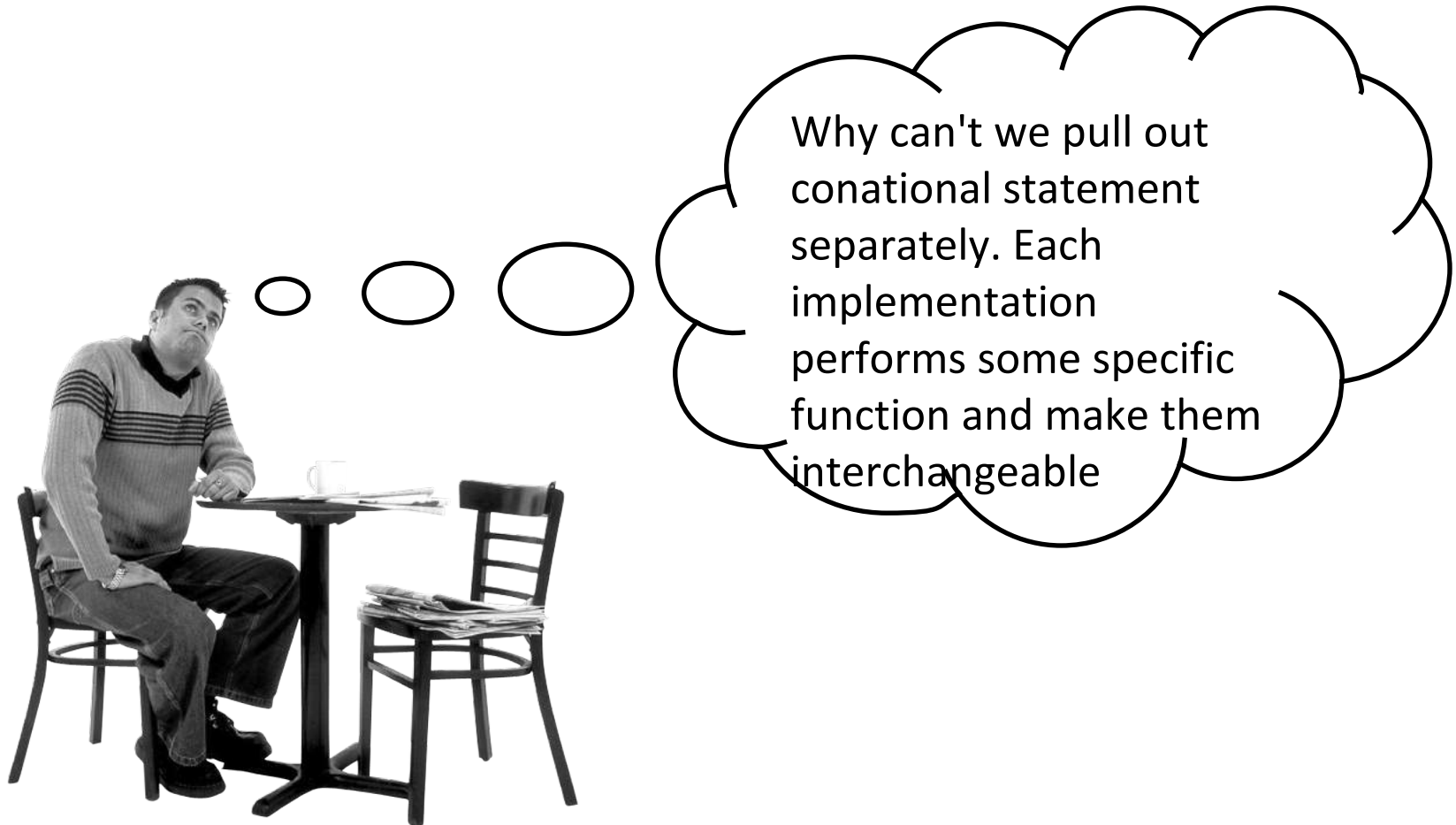
Introduction

In many situations, i
come across the need
to write
computational logic
which i generally
accomplish by using if
else or switch case

but later on it is
difficult to
maintenance and
understand



Rescuer



Identified Solution

It's really great that
the algorithm vary
independently from
clients that use it



This solution we call it as **Strategy**



Coding Strategy Pattern

```
public class CalcolaCondizioneService implements ICalcolaCondizioneService {
```

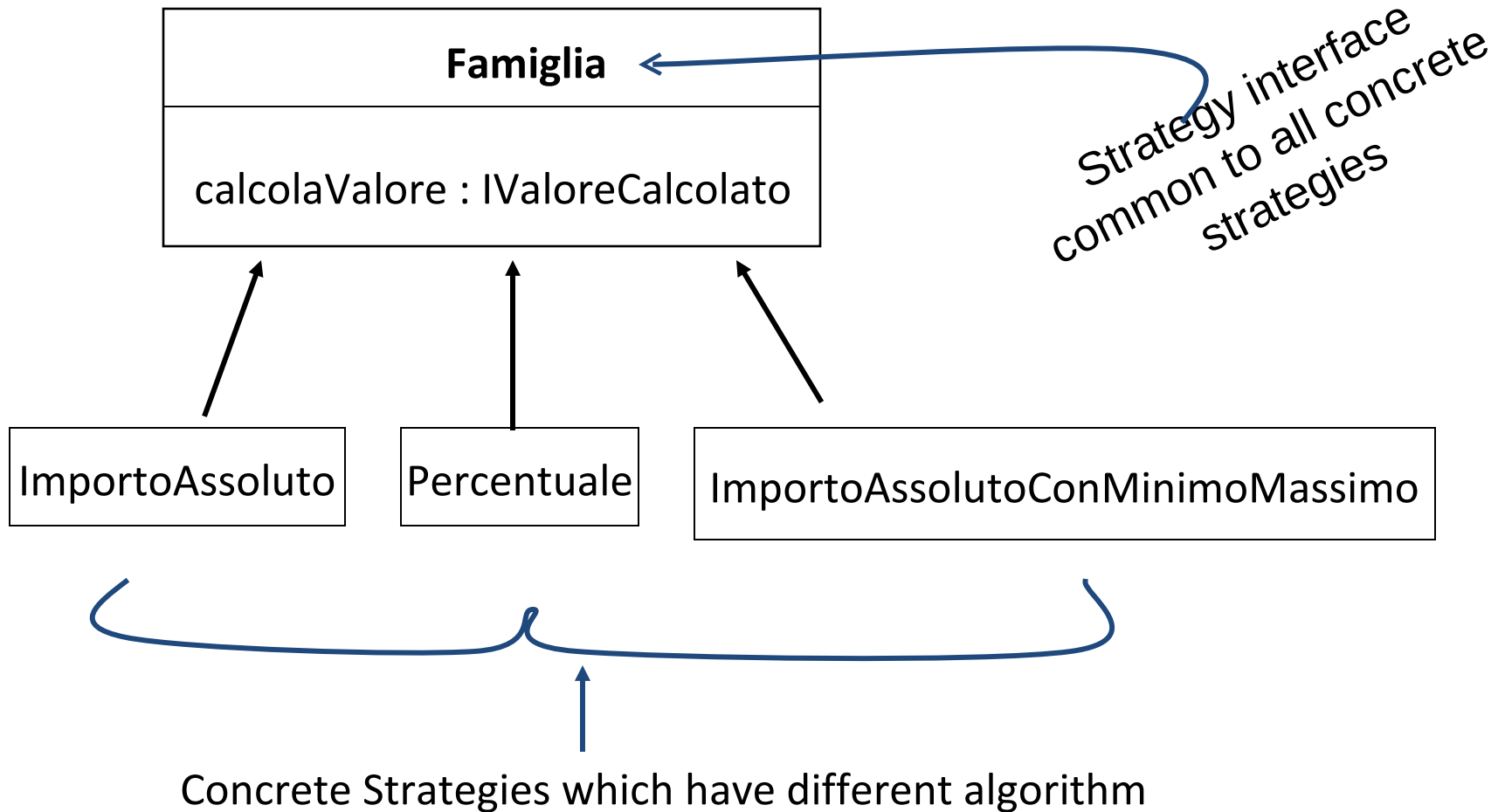
```
    public double calcolaCondizione(String familyName, String serviceName, Date date) {  
        Double calulatedValue = null;
```

```
        if(familyName.equals("ImportoAssoluto")) {  
            // logic goes here  
        } else if(familyName.equals("Percentuale")) {  
            // logic goes here  
        } else if(familyName.equals("ImportoAssolutoConMinimoMassimo")) {  
            // logic goes here  
        } else if(familyName.equals("PercentualeConMinimoMassimo")) {  
            // logic goes here  
        }  
    }
```

```
        return calulatedValue;
```

```
    }  
}
```


Introducing Strategy Interface



Re-writing the CalcolaCondizione Service

```
public class CalcolaCondizioneService implements ICalcolaCondizioneService {
```

```
    private Famiglia famiglia;
```

```
    public void setFamiglia(Famiglia famiglia) {  
        this.famiglia = famiglia;  
    }
```

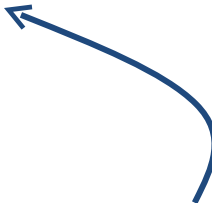
```
    public double calcolaCondizione(String serviceName, Date date) {  
        Double calulatedValue = null;
```

```
        IValoreCalcolato valore = famiglia.calcolaValore(serviceName);
```

```
        // method implementation continued
```

```
        return calulatedValue;
```

```
    }  
}
```



Delegates requests to the indicated
Strategies received from the client

Conclusion

- Strategy lets the algorithm vary independently from clients that use it
- Defines a family of algorithms
- Encapsulates each algorithm
- Makes the algorithms interchangeable within that family
- Forces program to an interface, not an implementation
- Forces polymorphic behavior
- Forces open for extension and closed for modification

Behavioral Patterns

- Command
- Observer
- Strategy
- Template method

Template Method

Template Pattern or Template Method Pattern – Real Time Example



Template Method [prepareCoffee]

- 1.Boil water
- 2.Add milk
- 3.Add Sugar
- 4.Add CoffeePowder

Bru Coffee



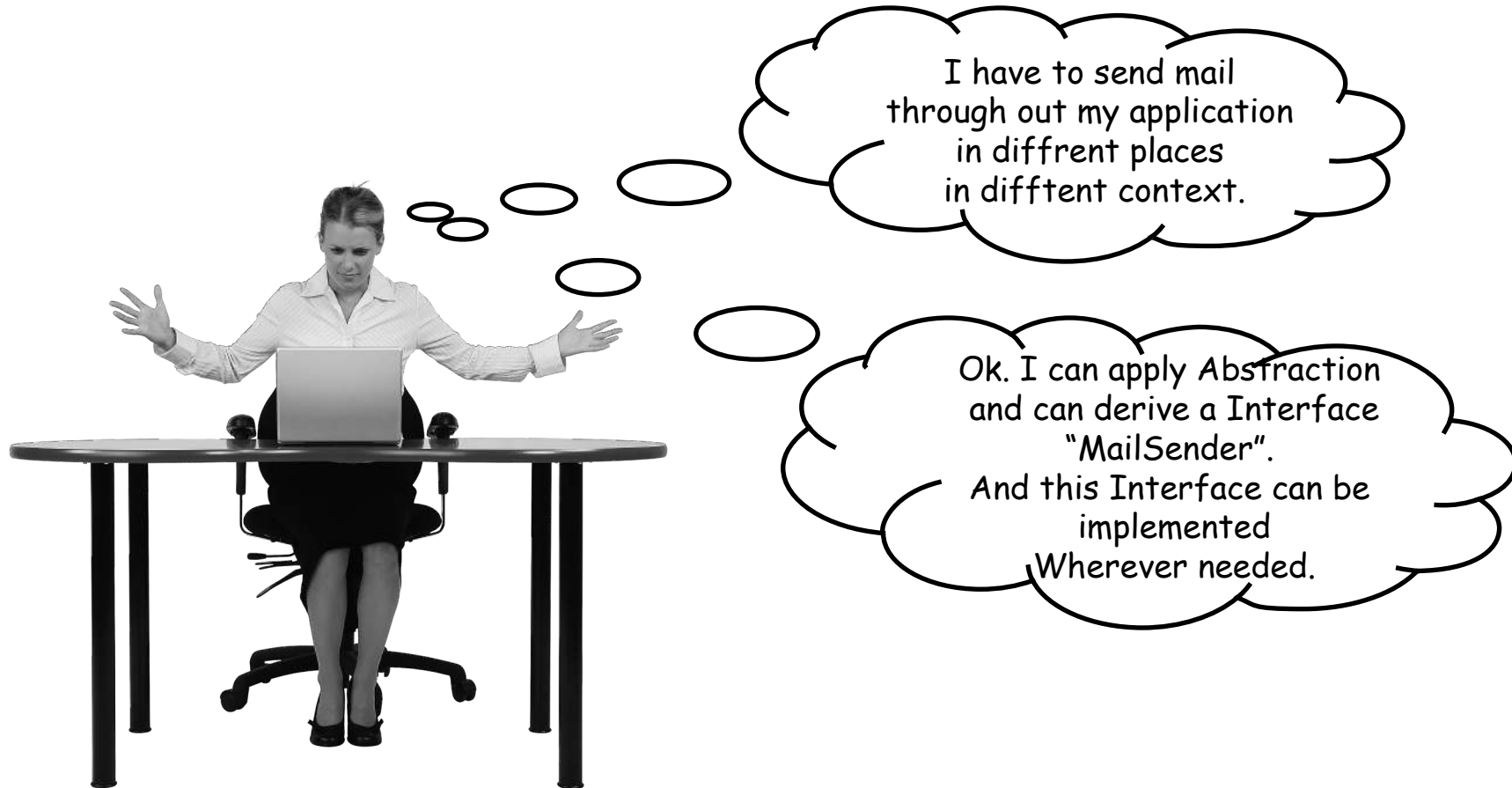
Nescafe Coffee



When to go for Template?

- When you see common step of algorithm repeated in more than places.
- When you want to have the control of algorithm execution order in one place.
- And in those common steps some of the steps are vary.

Mail Sending problem



Mail Sender



I had implemented MailSender
in many classes.
But there are some common steps.
Which is repeated in all the impls.
How do I control the steps?

Template Method

```
public interface MailSender {  
    void sendMail(MailData data);  
}  
  
public abstract class AbstractMailSenderTemplate implements MailSender {  
  
    public abstract AddressDetail address(MailData data);  
    public abstract String subject(MailData data);  
    public abstract String buildMailBody(MailData data);  
  
    public final void sendMail(MailData data) {  
        MailView view = new MailView();  
        view.setAddresses(address(data));  
        view.setSubject(subject(data));  
        view.setBody(buildMailBody(data));  
        h2oMailSender.sendMail(view);  
    }  
}
```

Benefits

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure.

Conclusion

- Standard Steps and some steps vary.
- Hook method(s) (Step)
- Don't call us, we'll call you !!
- Template method invokes the hook method(s).
- Subclasses must have “is-a” relationship with the template class.

BAD Practices

What's Next?

