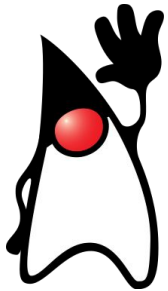
# DESIGN PATTERNS

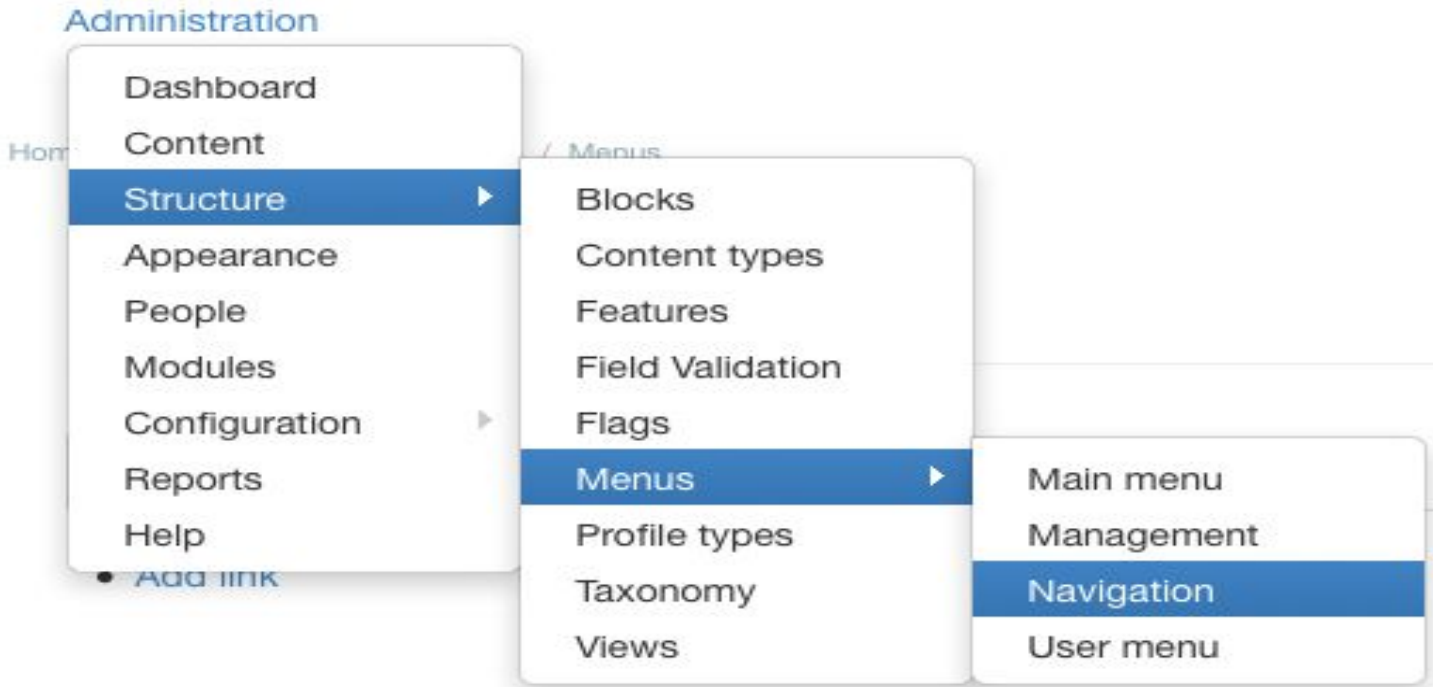
Miguel Bravo Arvelo
Samuel Fumero Hernández

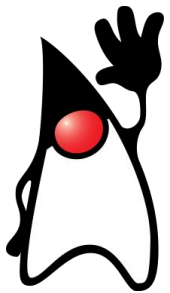
# Index



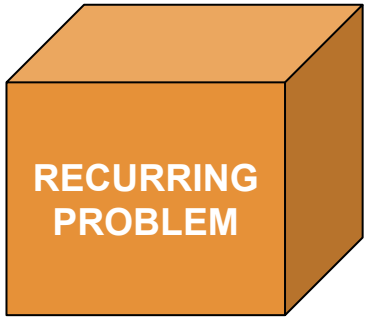


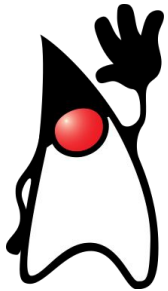
# 1. What is a Design Pattern?



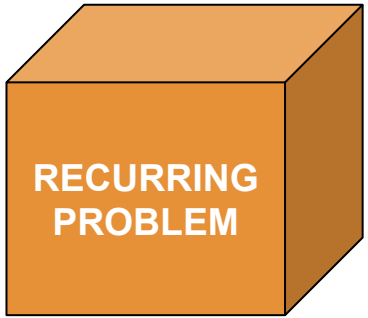



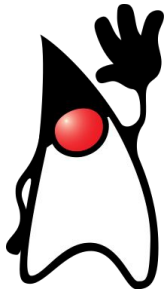
# 1. What is a Design Pattern?



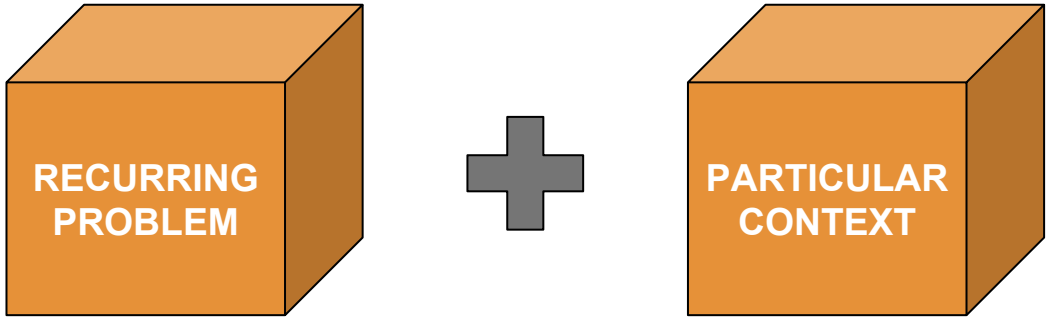



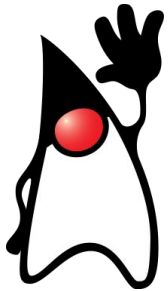
# 1. What is a Design Pattern?



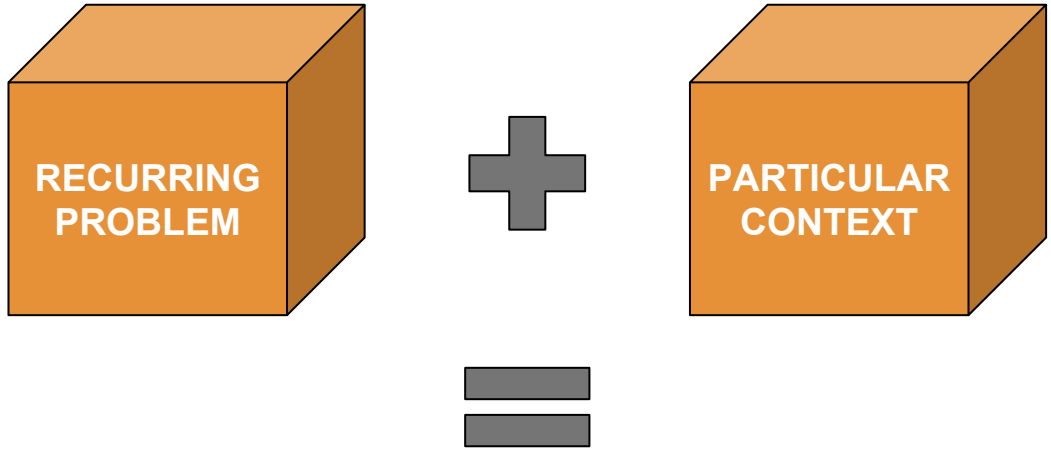



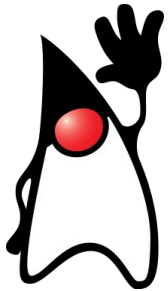
# 1. What is a Design Pattern?



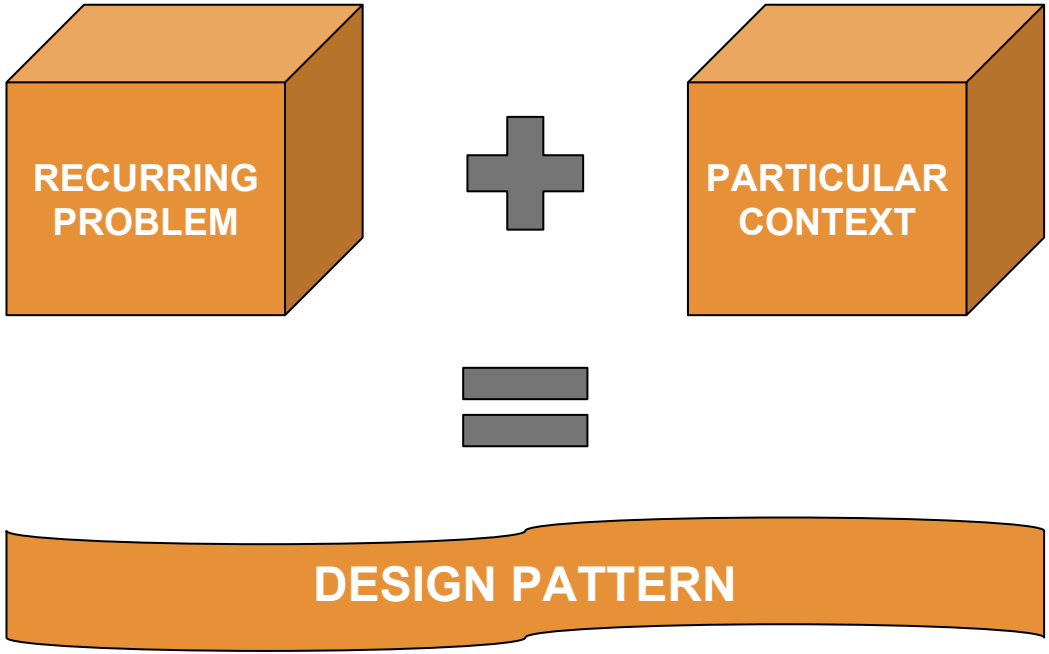



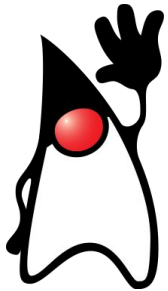
# 1. What is a Design Pattern?



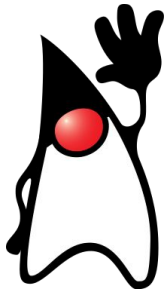
# 1. What is a Design Pattern?



# 1. What is a Design Pattern?

**IS A DESIGN SOLUTION TO A RECURRING PROBLEM IN A PARTICULAR CONTEXT**



# 1. What is a Design Pattern?

## THE IDENTIFICATION

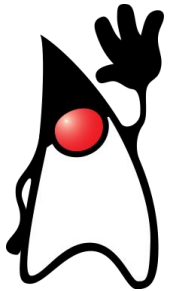
- Name (with synonymous)
- Pattern category

## THE PROBLEM

- Intention
- Motivation
- Applicability

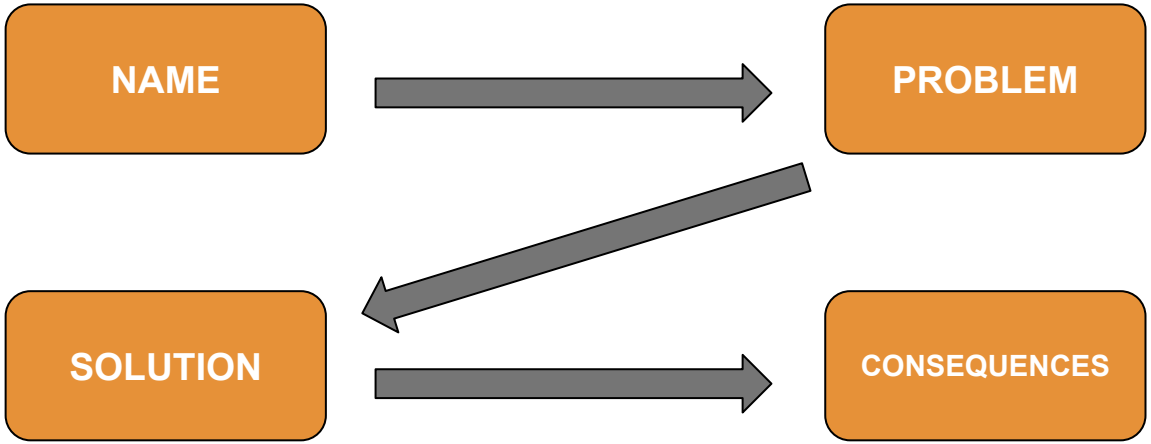
## THE SOLUTION

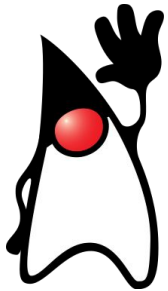
- Structure
- Participants
- Collaborations
- Implementations
- Example code
- Known uses
- Related patterns
- Consequences



# 1. What is a Design Pattern?

## Simplified:

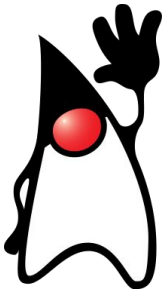


# 2. Objectives of design patterns

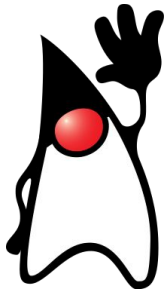
- Avoid repetition in the search for solutions
- Formalize a common vocabulary among designers
- Standardize the way the design is done.

- Impose certain design alternatives over other
- Eliminate the creativity inherent in the design process
- Forcefully implement a pattern in a simple code.



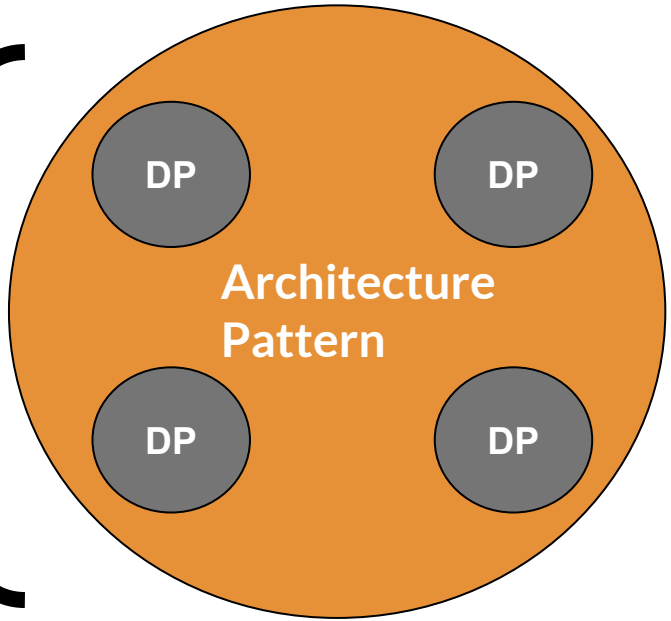
# 3. Why do we need Design Patterns?

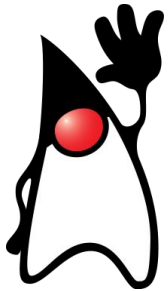
- Robust code
- Code reusability
- High maintainability
- Greater understanding among developers



# 4. Design Pattern Vs. Architecture Pattern

Software application



# 5. Common Design Patterns

## Creational Design

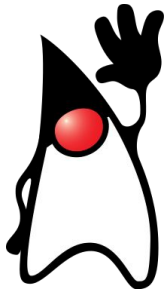
- Singleton
- Factory
- Builder
- Prototype

## Structural Design

- Adapter
- Composite
- Facade
- Bridge
- Decorator
- FlyWeight

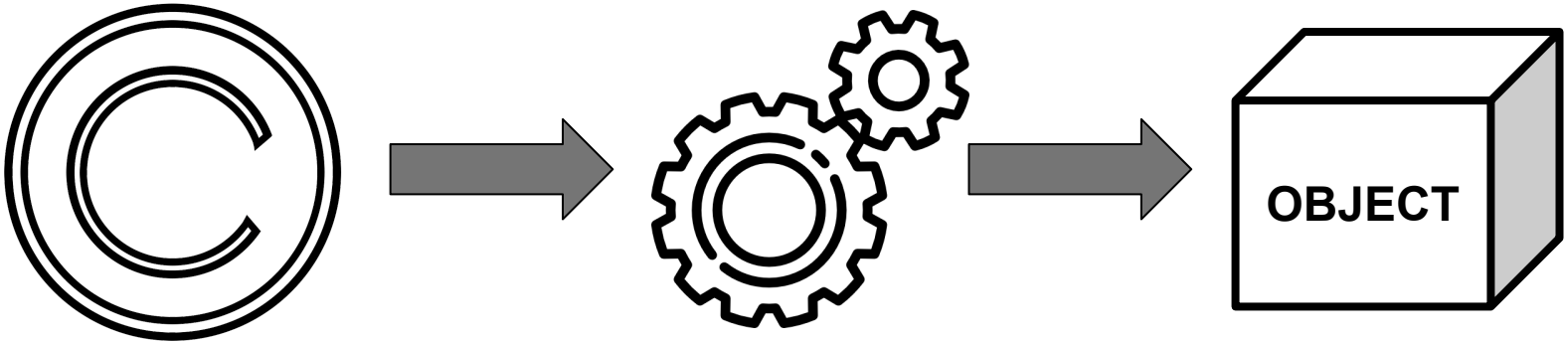
## Behavioral Design

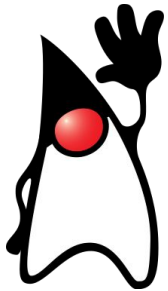
- Observer
- Strategy
- Interpreter
- Iterator
- State Pattern
- Chain of Responsibility



# 5.1 Creational Design Patterns

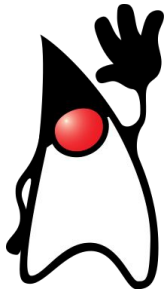
- These design patterns are all about class instantiation.



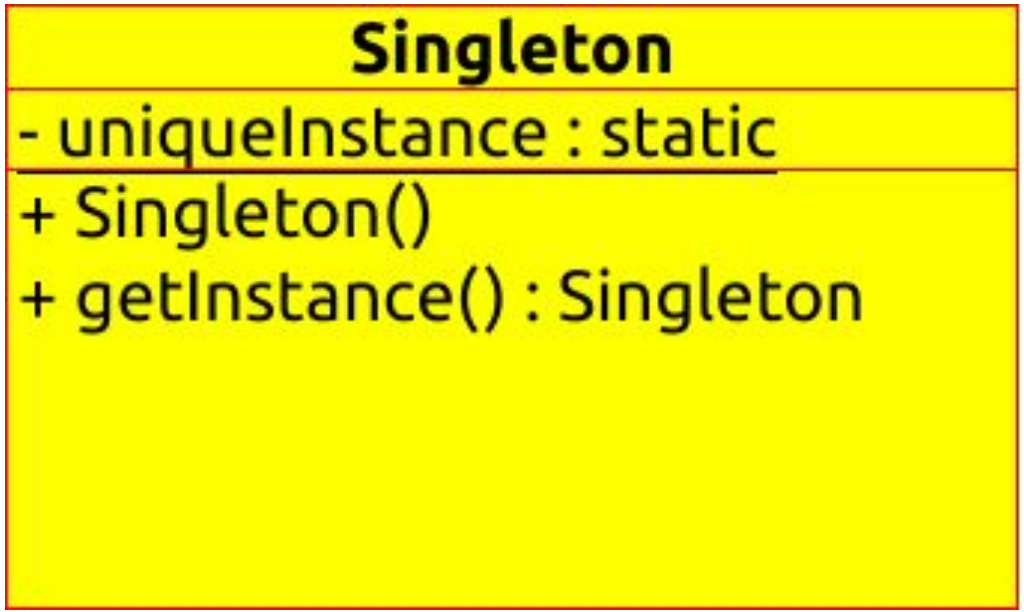



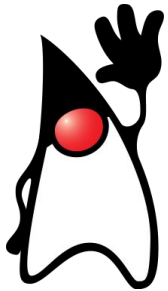
# 5.1.1 Singleton

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.



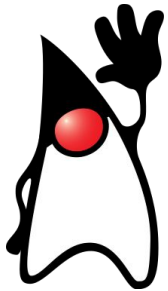
# 5.1.1 Singleton



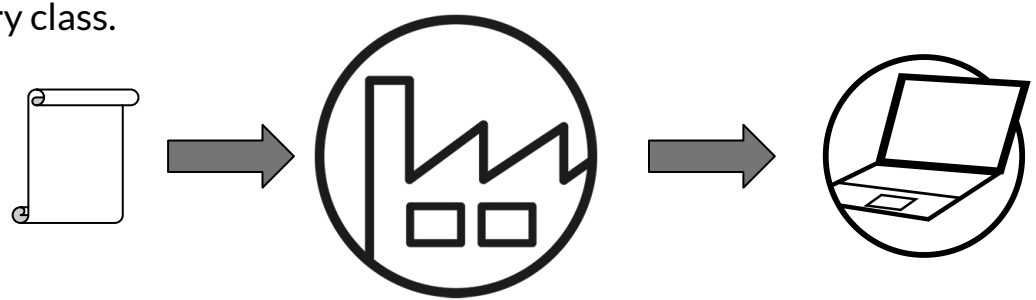
# 5.1.1 Singleton

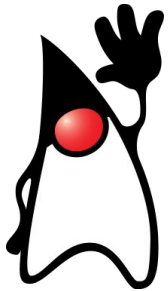
```
1.  public class LazyInitializedSingleton {
2.
3.      private static LazyInitializedSingleton instance;
4.
5.      private LazyInitializedSingleton(){}
6.
7.      public static LazyInitializedSingleton getInstance(){
8.          if(instance == null){
9.                  instance = new LazyInitializedSingleton();
10.         }
11.         return instance;
12.     }
13. }
```



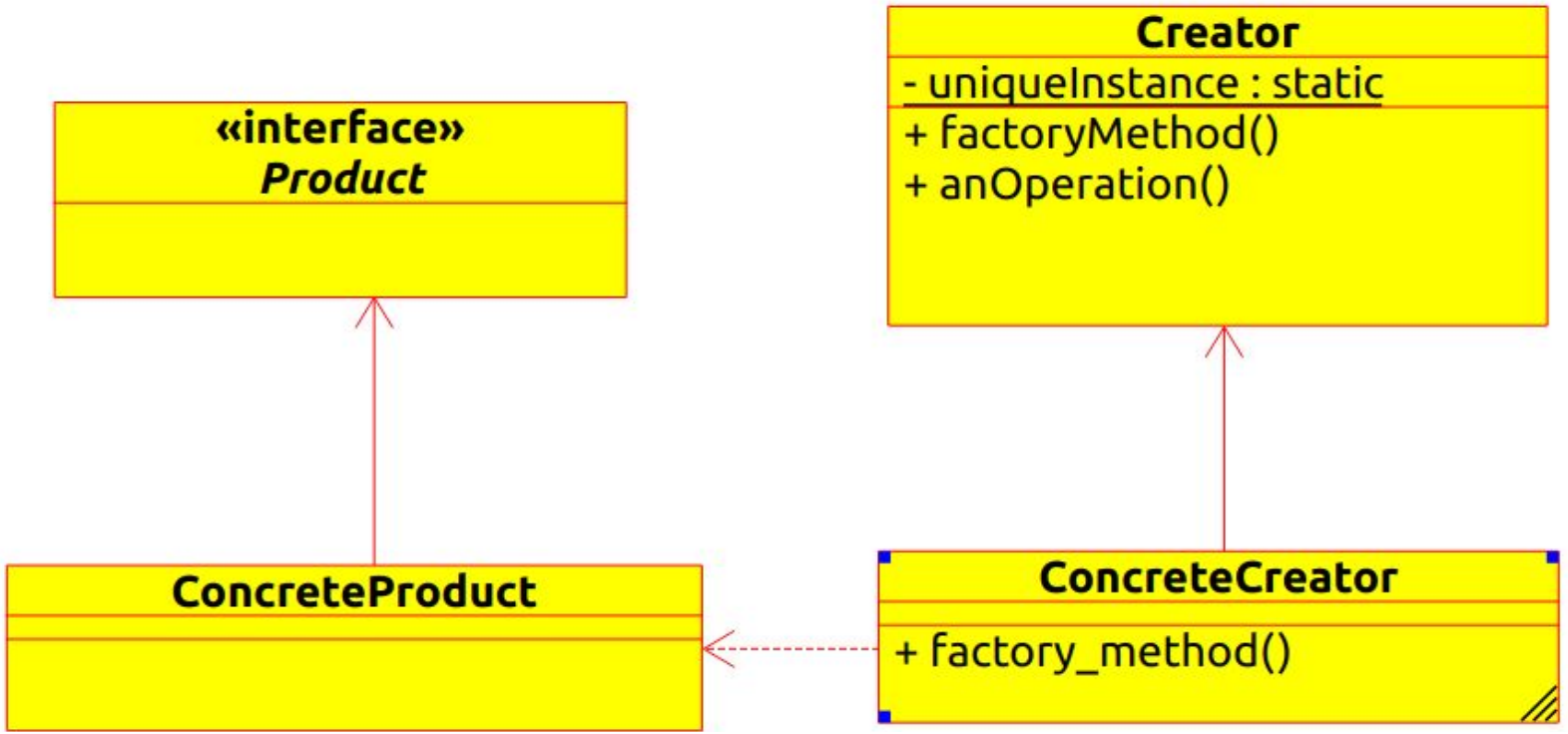
# 5.1.2 Factory Method

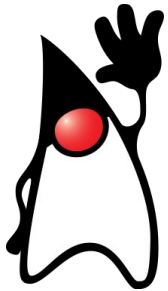
- Factory design pattern is used when we have a super class with multiple sub-classes and based on input.
- Only returns one of the sub-class.
- Takes out the responsibility of instantiation of a class from client program to the factory class.





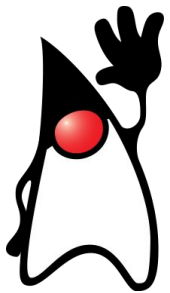
# 5.1.2 Factory Method



# 5.1.2 Factory Method

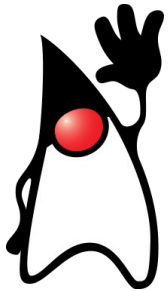
```
1.  import design.model.Computer;
2.  import design.model.PC;
3.  import design.model.Server;
4.
5.  public class ComputerFactory {
6.
7.      public static Computer getComputer(String type, String ram,
8.                                         String hdd, String cpu){
9.          if("PC".equalsIgnoreCase(type))
10.             return new PC(ram, hdd, cpu);
11.         else if("Server".equalsIgnoreCase(type))
12.             return new Server(ram, hdd, cpu);
13.         return null;
14.     }
15. }
```



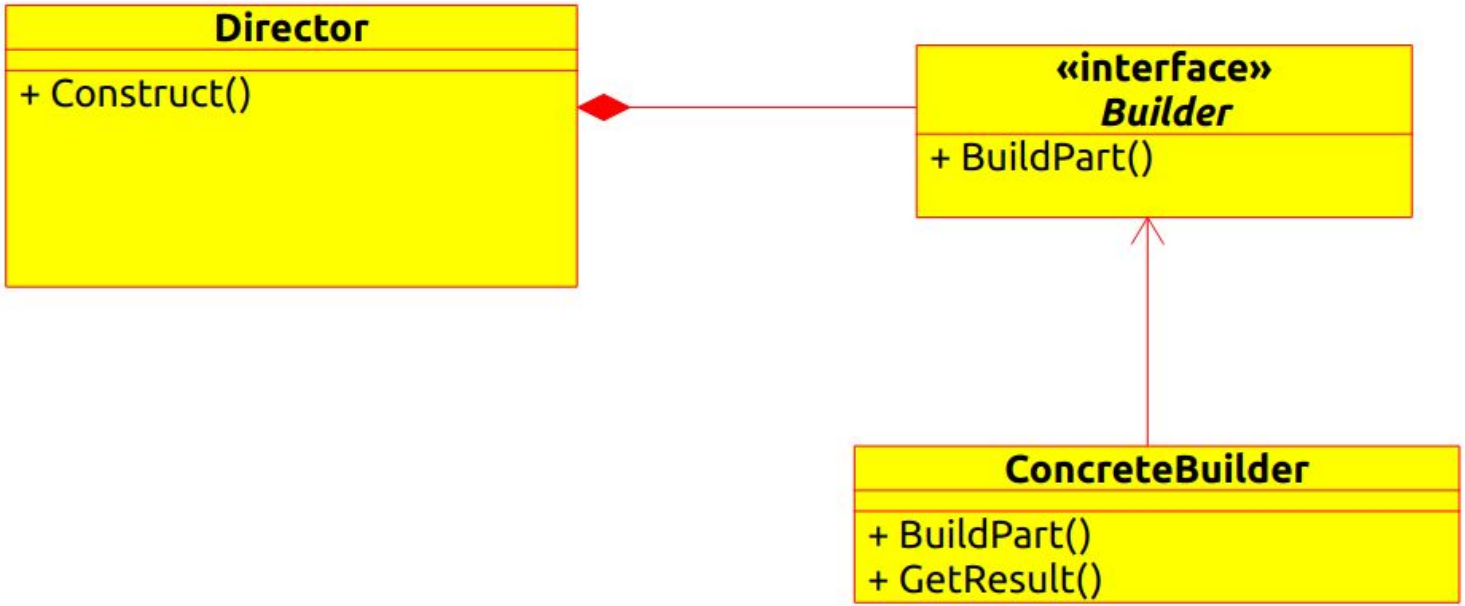
# 5.1.3 Builder

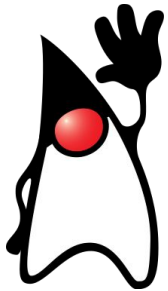
- Too Many arguments to pass from client program to the Factory class
- Some of the parameters might be optional

new ClassName(arg(1), arg(2), arg(3), … ,arg(n));



# 5.1.3 Builder

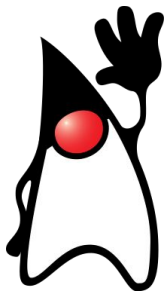

Director
+ Construct()
«interface»
Builder
+ BuildPart()
ConcreteBuilder
+ BuildPart()
+ GetResult()



# 5.1.3 Builder

```
1.  public class User {
2.      private int id;
3.      private int age;
4.      .
5.      .
6.      .
7.      private String name;
8.
9.      public User(int id);
10.     public User(int id, int age);
11.     .
12.     .
13.     .
```

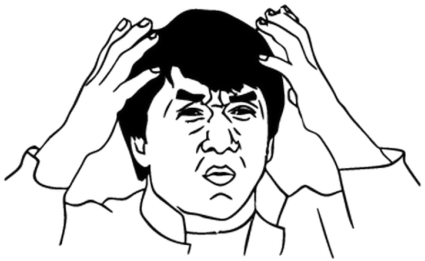
```
1.  User usu1 = new User(2,22);
2.  User usu2 = new User("Pepe",6);
3.  User usu3 = new User(54,22, "Jack");
4.  .
5.  .
6.  .
```

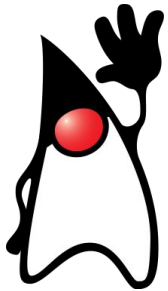


# 5.1.3 Builder

```
1.  public class User {
2.      private int id;
3.      private int age;
4.      .
5.      .
6.      .
7.      private String name;
8.
9.      public User(int id);
10.     public User(int id, int age);
11.     .
12.     .
13.     .
```

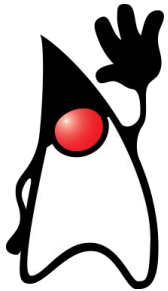
```
1.  User usu1 = new User(2,22);
2.  User usu2 = new User("Pepe",6);
3.  User usu3 = new User(54,22, "Jack");
4.  .
5.  .
6.  .
```





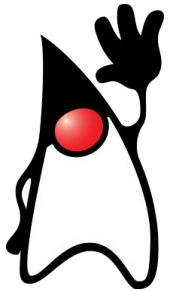
# 5.1.3 Builder

```
1.  public class TestBuilderPattern {
2.
3.      public static void main(String[] args) {
4.
5.          User usu1 = new UserBuilder().name(“Pepe”).
6.                  age(24).id(5).build();
7.      }
8.
9.  }
```

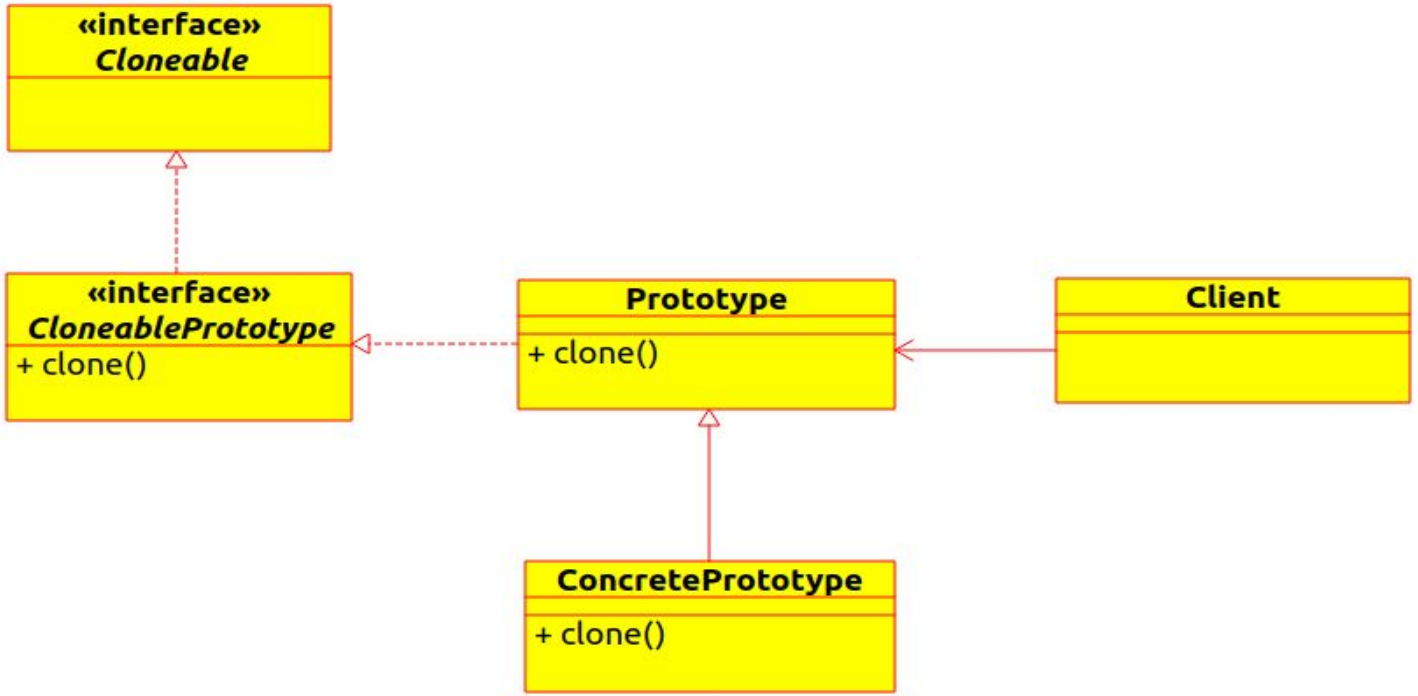


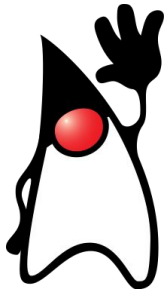
# 5.1.4 Prototype

- Create objects by cloning based on a template of existing objects
- Objects of the same class are very similar to each other
- The initial creation of each object is an expensive operation



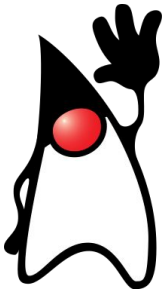
# 5.1.4 Prototype



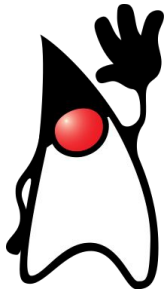
# 5.1.4 Prototype

```
1.  public class client {
2.
3.      public static void main (String[] args) {
4.
5.          Prototype product = new Prototype();
6.          Prototype product2 = product.clone();
7.      }
8.  }
```



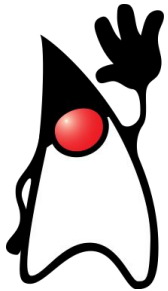
# 5.2 Structural Design Patterns

- These design patterns are all about Class and Object composition.
- Structural class-creation patterns use inheritance to compose interfaces.
- Structural object-patterns define ways to compose objects to obtain new functionality.

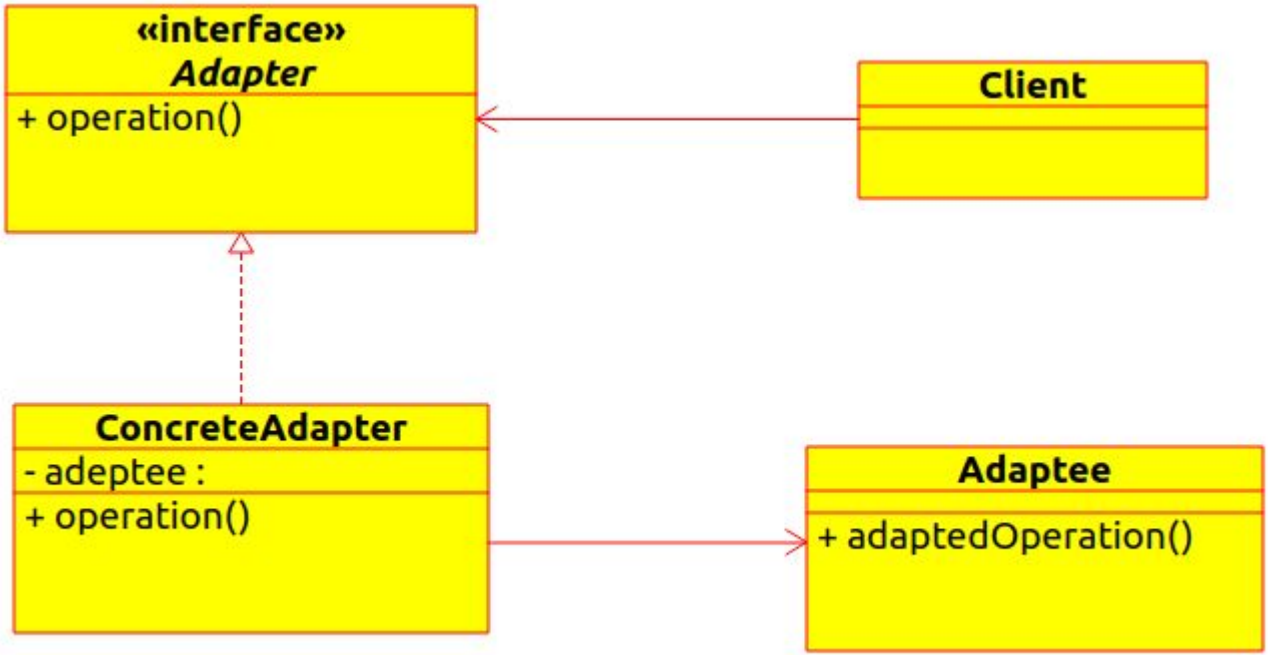


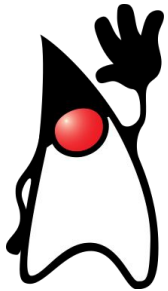
# 5.2.1 Adapter

- When implementing Adapter pattern, there are two approaches, both these approaches producing the same result.
    - Class Adapter – This form uses java inheritance and extends the source interface, in our case Socket class.
    - Object Adapter – This form uses Java Composition and adapter contains the source object.

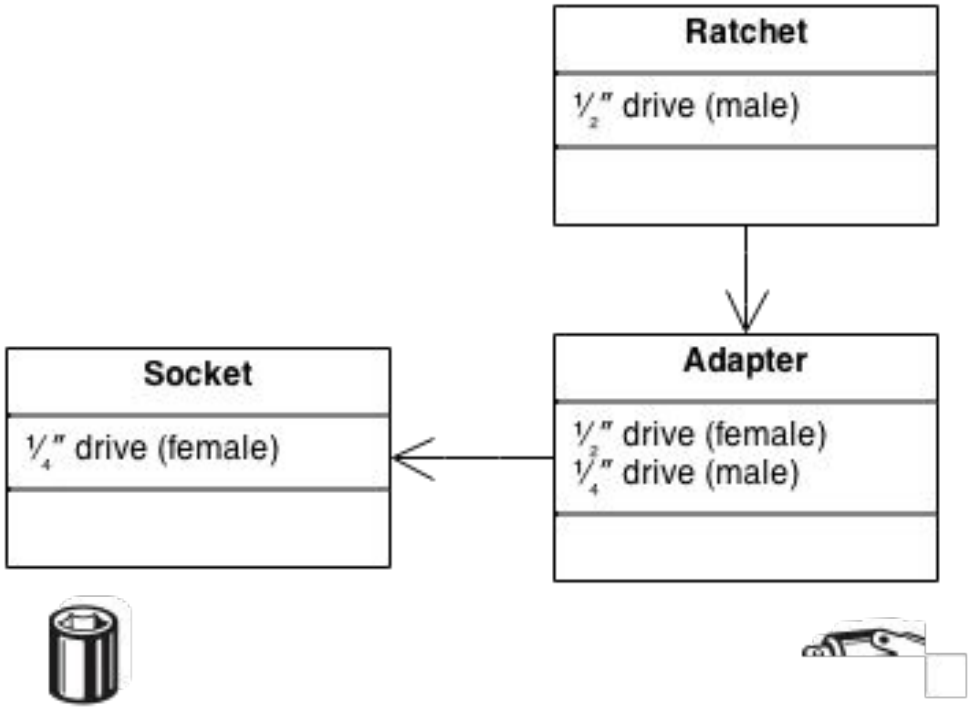


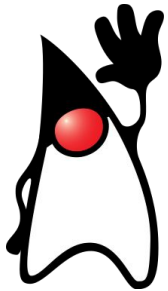
# 5.2.1 Adapter


«interface»
Adapter
+ operation()
Client
ConcreteAdapter
- adeptee :
+ operation()
Adaptee
+ adaptedOperation()



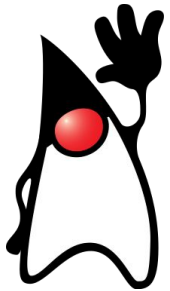
# 5.2.1 Adapter


Ratchet
½" drive (male)
Socket
¼" drive (female)
Adapter
½" drive (female)
¼" drive (male)

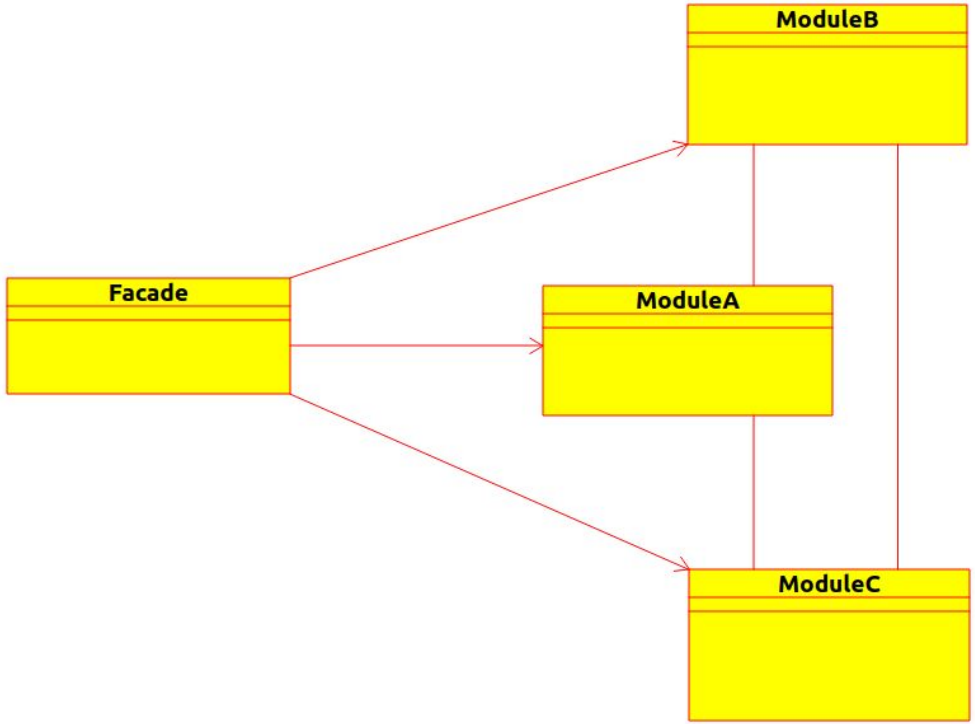


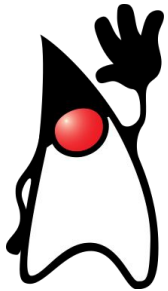
# 5.2.2 Facade

- Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.
- So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports.
- But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it.
- Provides a wrapper interface on top of the existing interface to help client application.

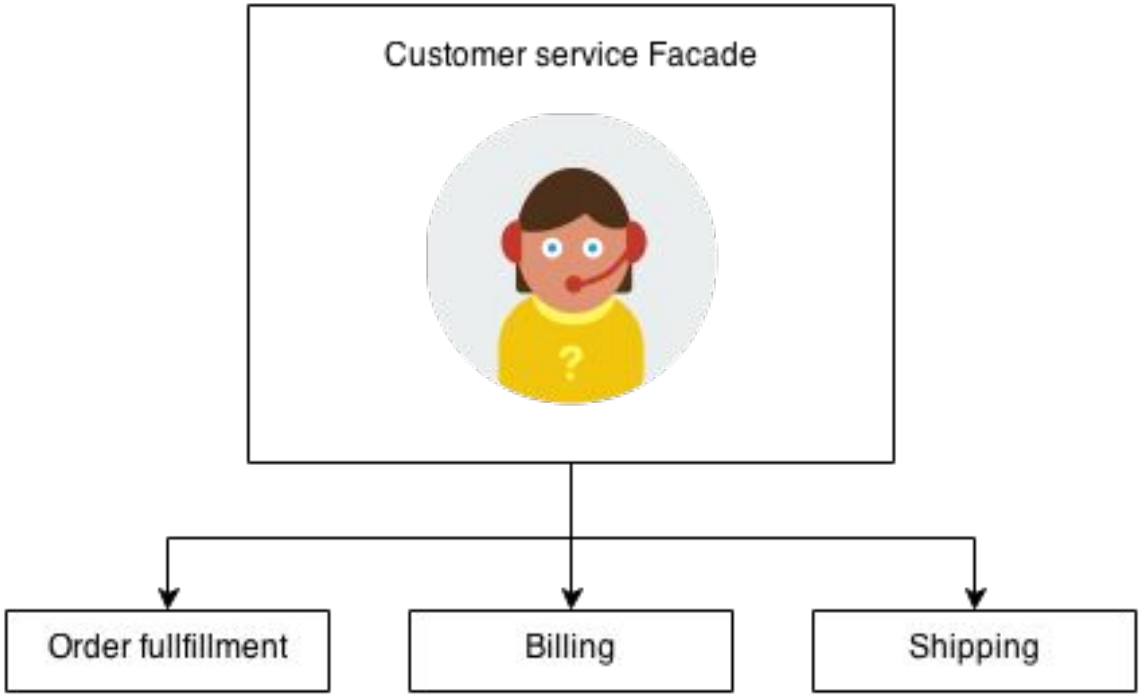


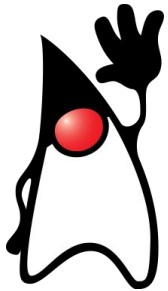
# 5.2.2 Facade



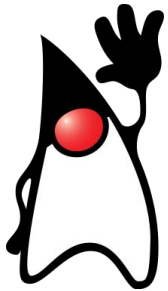
# 5.2.2 Facade


Customer service Facade
Order fullfillment
Billing
Shipping

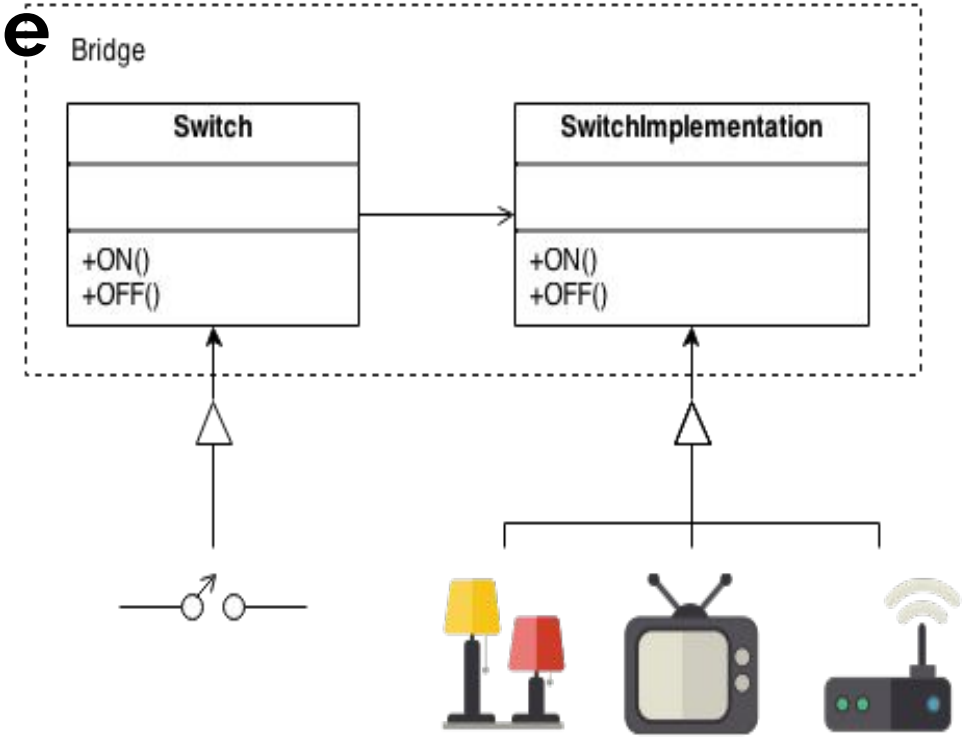


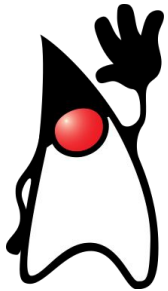
# 5.2.3 Bridge

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.



# 5.2.3 Bridge

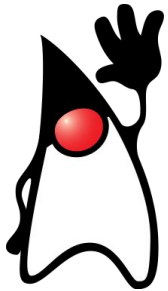


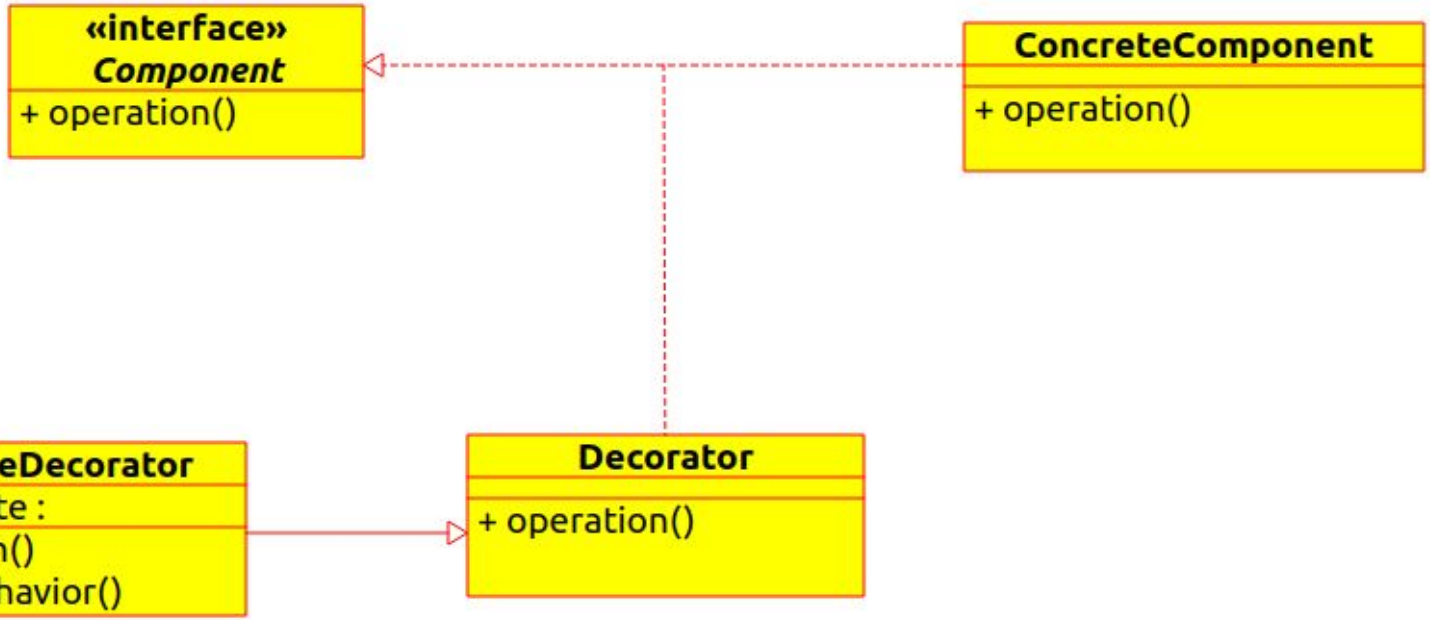
# 5.4 Decorator

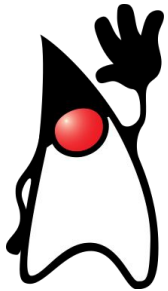
We use inheritance or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class, in order to implement this we will need:

1. Component Interface – The interface or abstract class defining the methods that will be implemented.
2. Component Implementation – The basic implementation of the component interface.
3. Decorator – Decorator class implements the component interface and it has a HAS-A relationship with the component interface.
4. Concrete Decorators – Extending the base decorator functionality and modifying the component behavior accordingly.

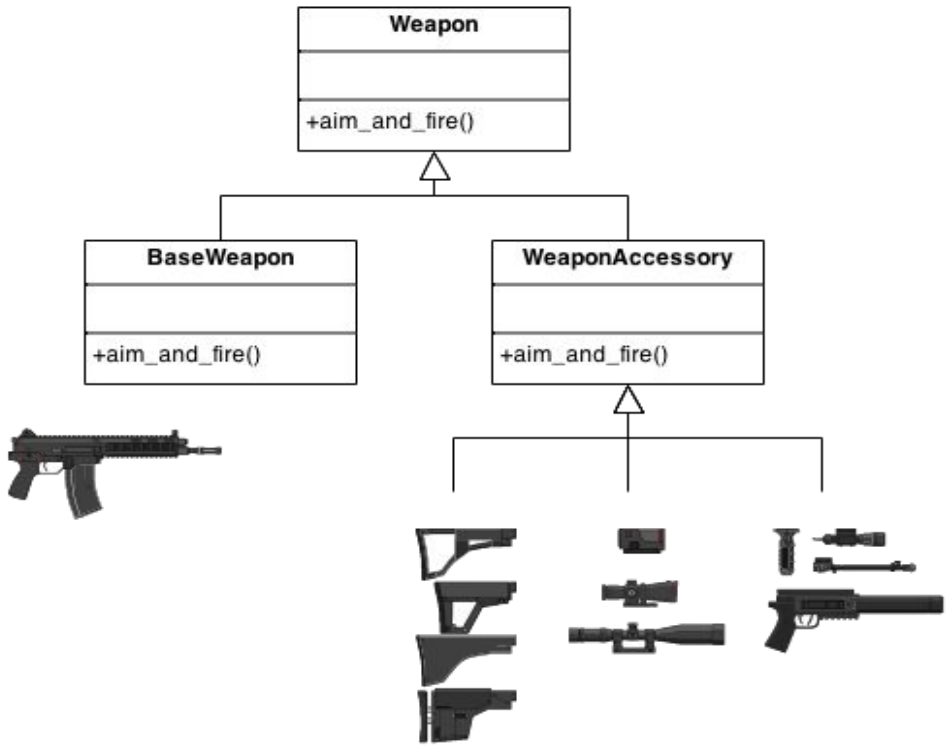


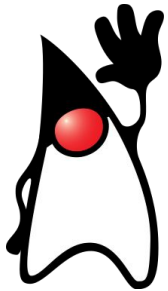
# 5.4 Decorator



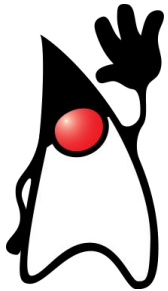
# 5.4 Decorator



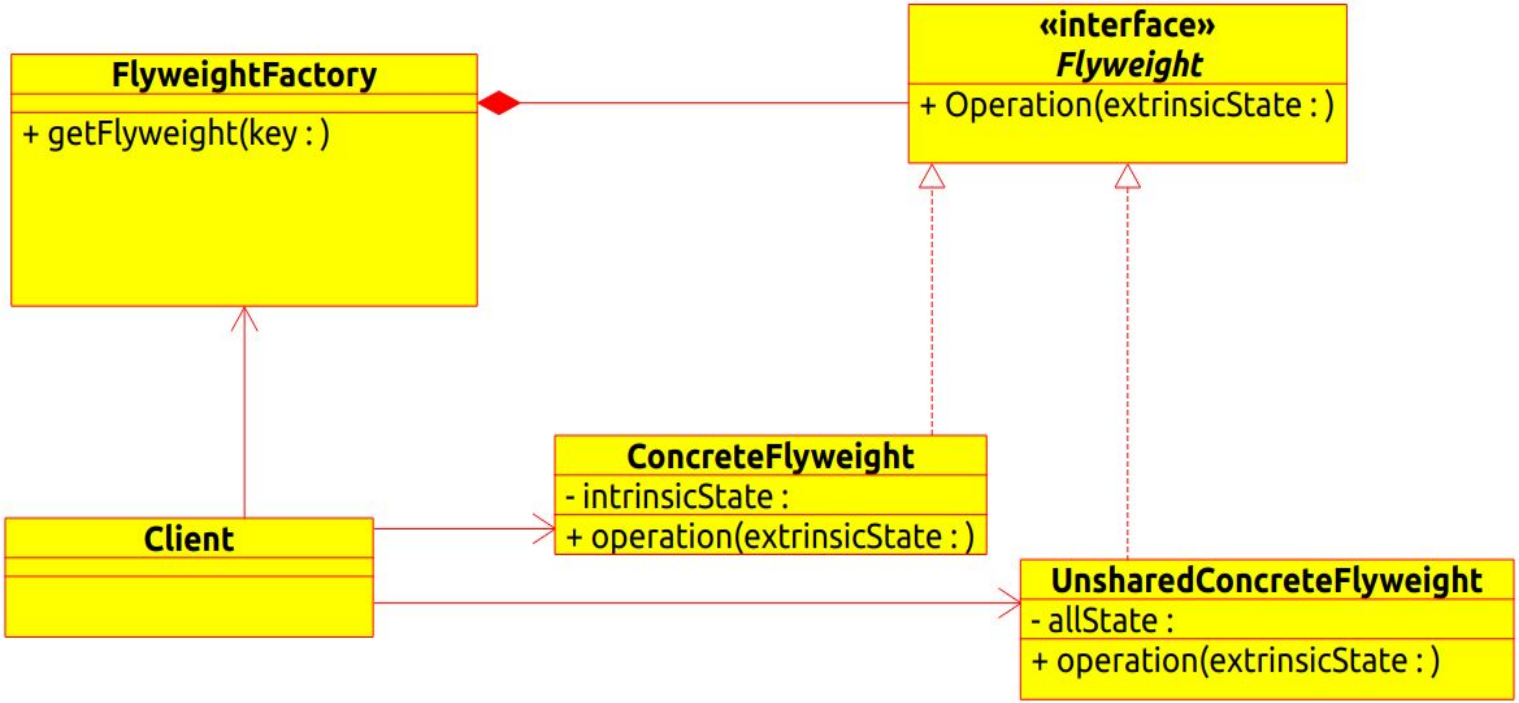



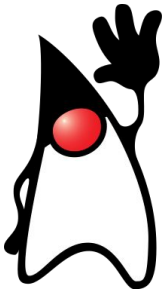
# 5.4 FlyWeight

- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.



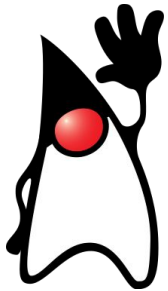
# 5.4 FlyWeight



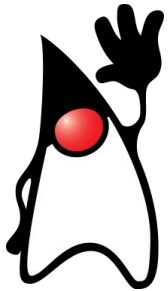
# 5.4 FlyWeight

Browser loads images just once and then reuses them from pool:



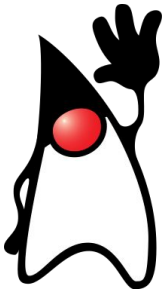
# 5.5 Behavioral Design Patterns

- These design patterns are all about Class's objects communication.
- Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

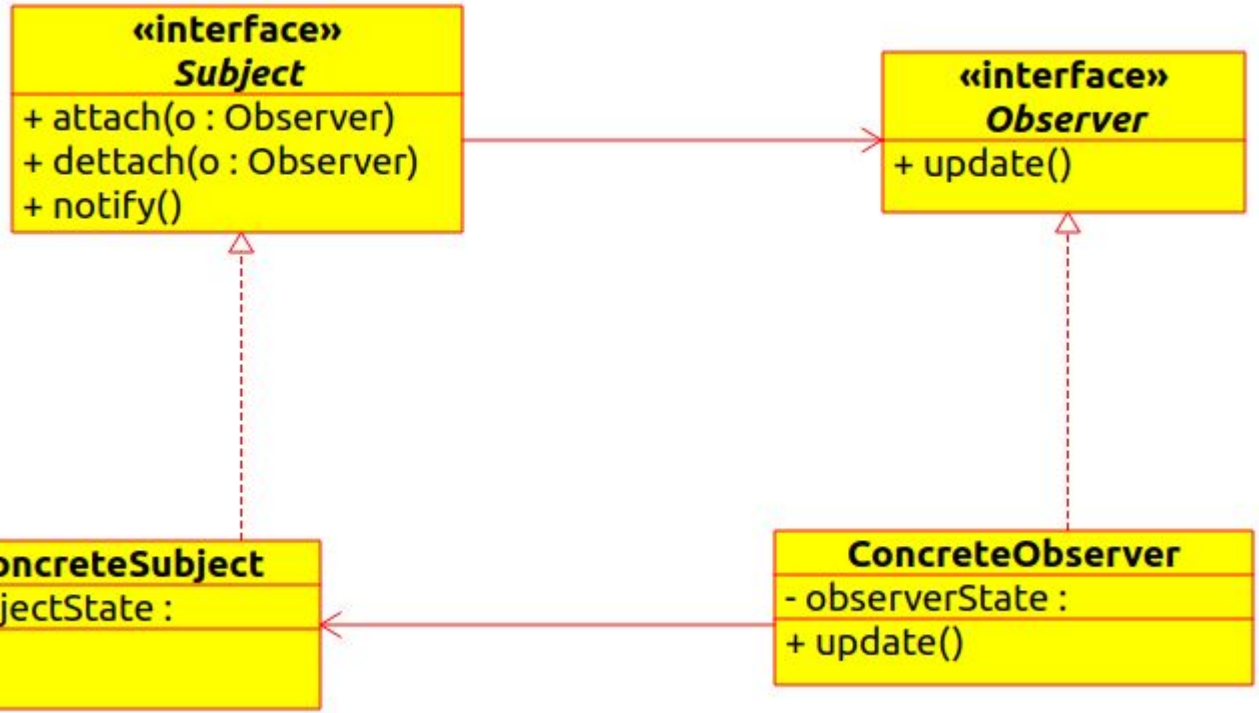


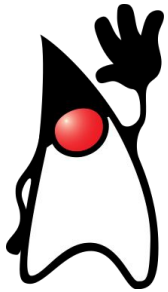
# 5.4 Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.

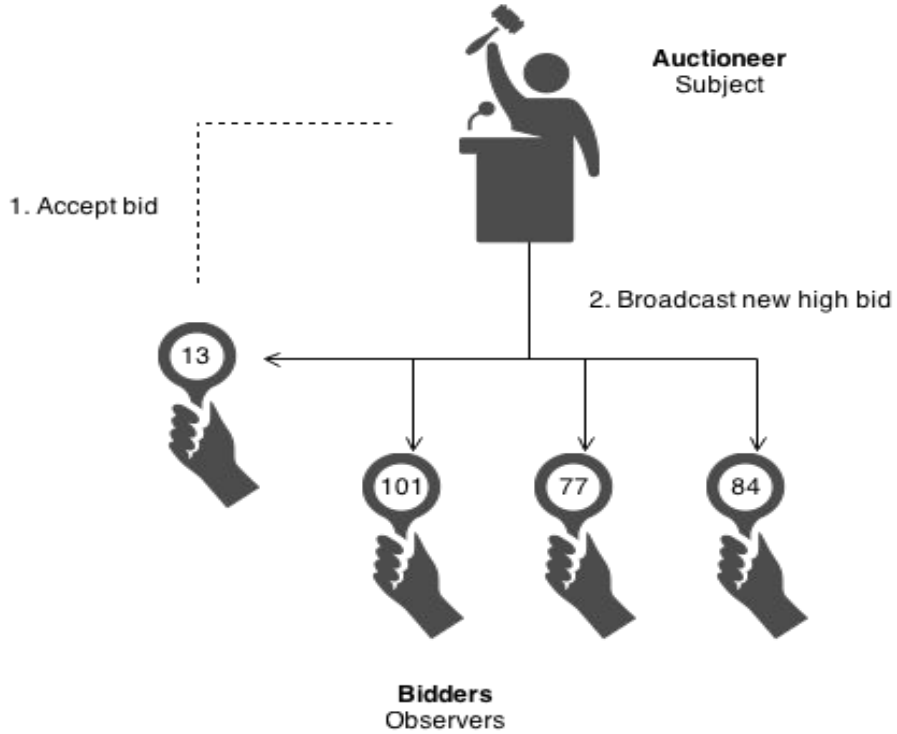


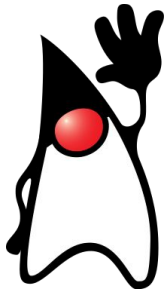
# 5.4 Observer


«interface»
Subject
+ attach(o : Observer)
+ dettach(o : Observer)
+ notify()
«interface»
Observer
+ update()
ConcreteSubject
- subjectState :
ConcreteObserver
- observerState :
+ update()



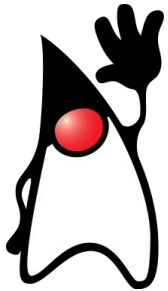
# 5.4 Observer


Auctioneer
Subject
1. Accept bid
2. Broadcast new high bid
13
101
77
84
Bidders
Observers

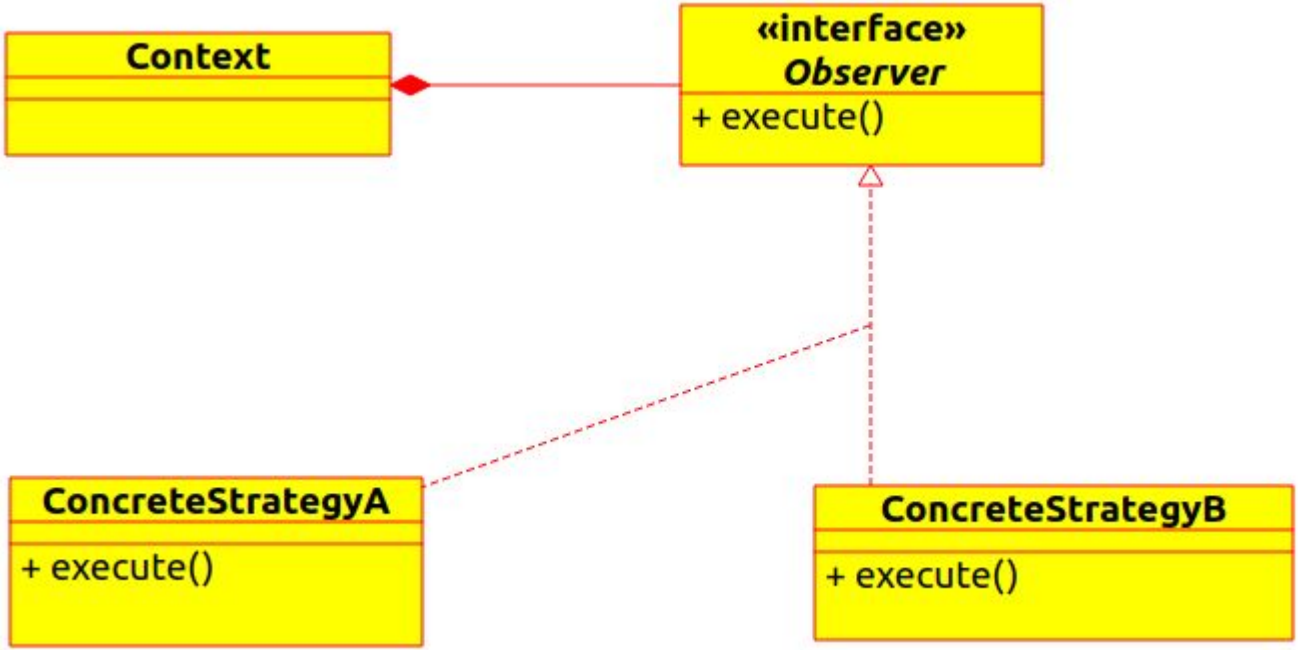


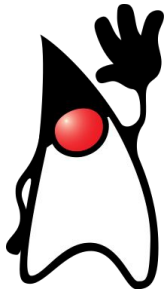
# 5.4 Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

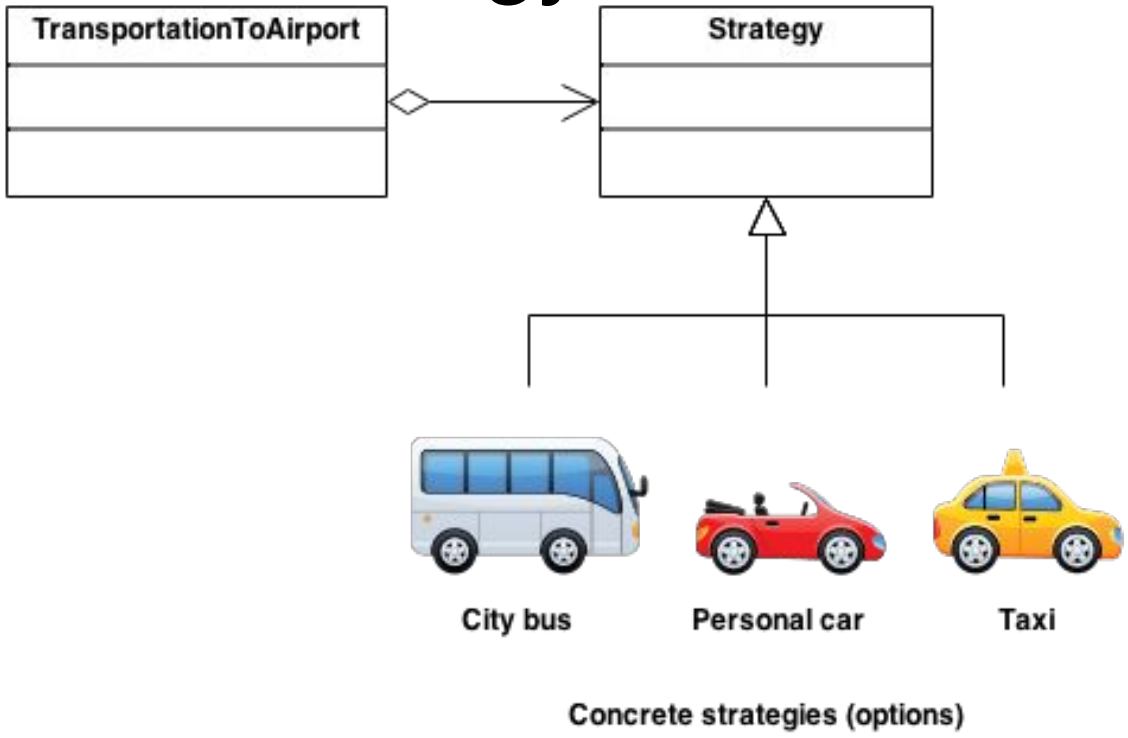


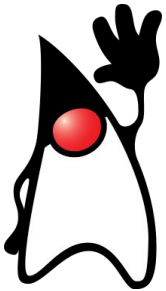
# 5.4 Strategy


Context
«interface»
Observer
+ execute()
ConcreteStrategyA
+ execute()
ConcreteStrategyB
+ execute()



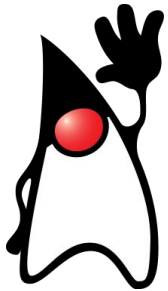
# 5.4 Strategy


TransportationToAirport
Strategy
City bus
Personal car
Taxi
Concrete strategies (options)

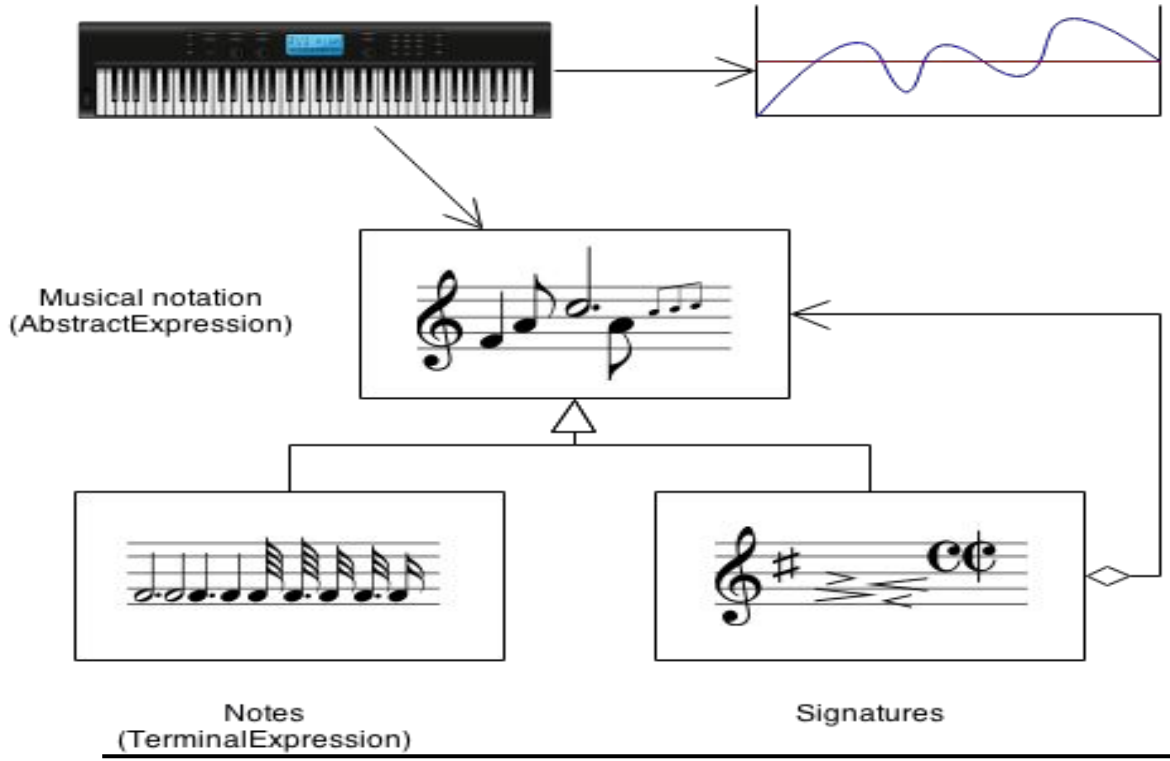


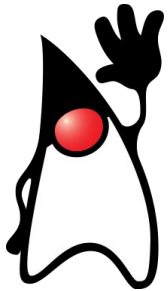
# 5.4 Interpreter

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.



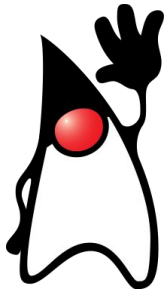
# 5.4 Interpreter



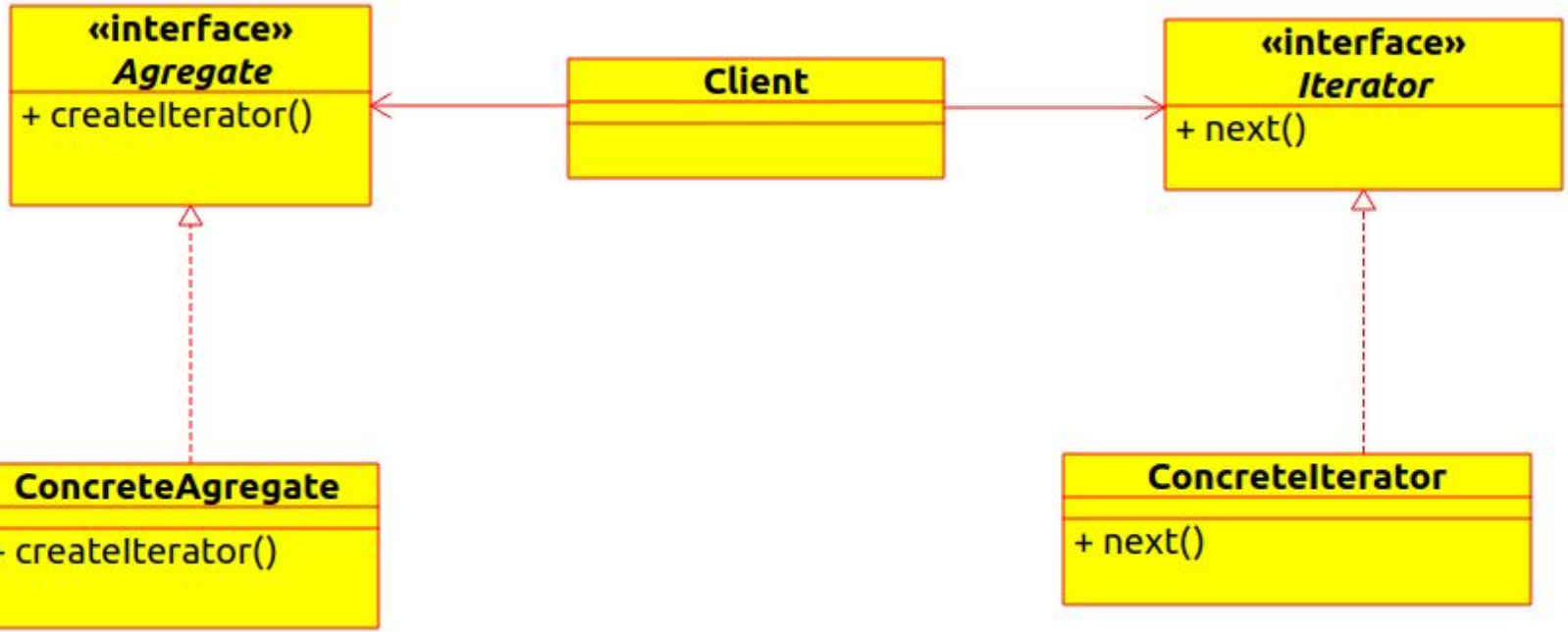



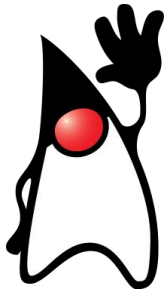
# 5.4 Iterator

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

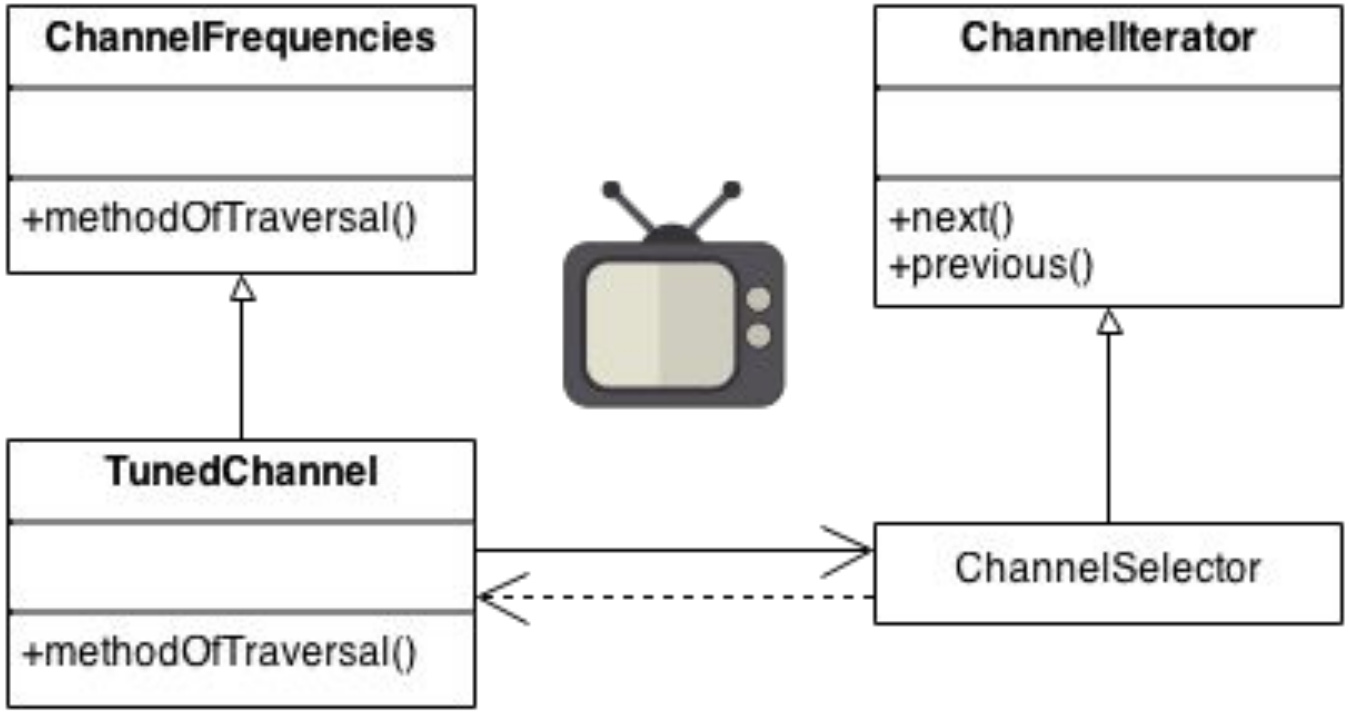


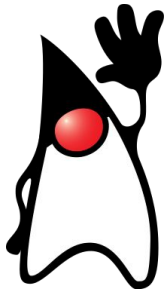
# 5.4 Iterator


«interface»
Agregate
+ createIterator()
Client
«interface»
Iterator
+ next()
ConcreteAgregate
+ createIterator()
ConcreteIterator
+ next()

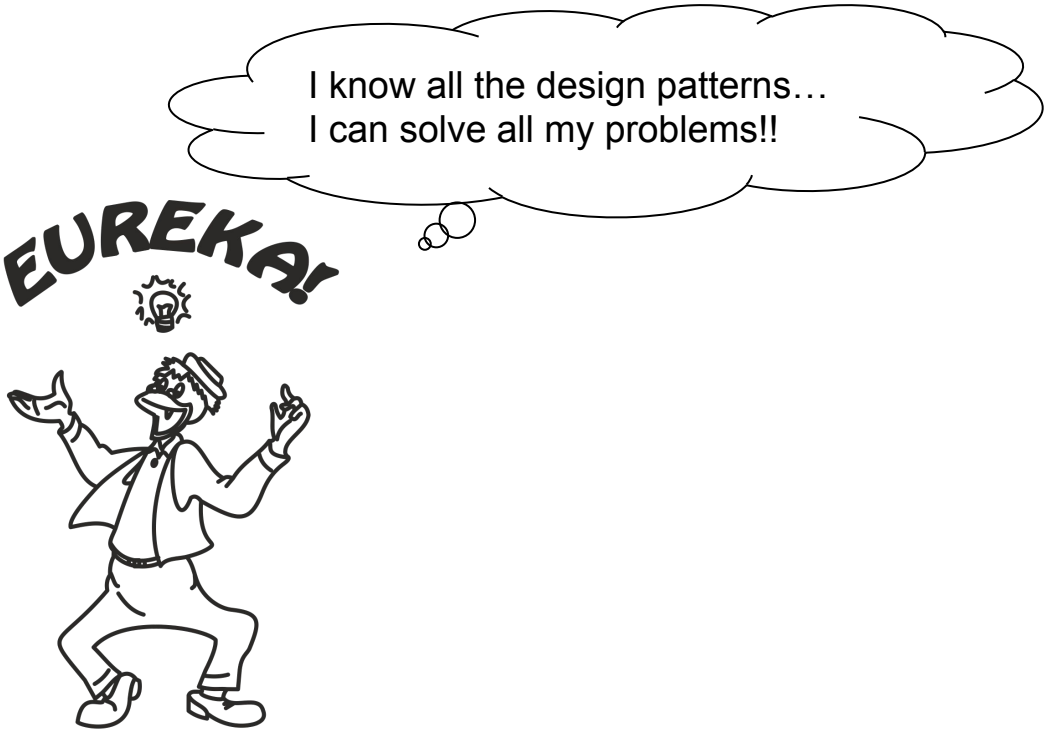


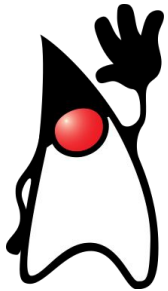
# 5.4 Iterator


ChannelFrequencies
+methodOfTraversal()
ChannelIterator
+next()
+previous()
TunedChannel
+methodOfTraversal()
ChannelSelector

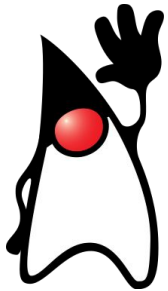


# 6. Conclusions



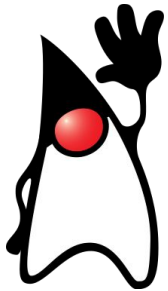
# 6. Conclusions

# 6. Conclusions

Design patterns facilitate the process of designing a software

Do not try to apply the design patterns with a shoehorn



# Biography

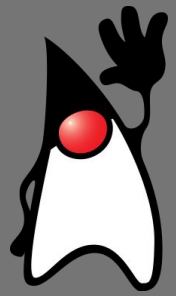

**Wikipedia - Design Patterns**
https://en.wikipedia.org/wiki/Design_Patterns

**Wikipedia - Model View Controller**
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

**Black Wasp - Gang of four design patterns**
http://www.blackwasp.co.uk/gofpatterns.aspx

**Journal Dev - Java Design Patterns**
https://www.journaldev.com/1827/java-design-patterns-example-tutorial

**Source Making - Design Patterns**
https://sourcemaking.com/design_patterns

# Thank you!

ANY QUESTIONS?