# LLD System Design

Author: Ajay Kumar

Github: ajay-panchal-099 (AJAY PANCHAL) (github.com)
LinkedIn:
AJAY KUMAR | LinkedIn

References:

https://www.oodesign.com/

https://refactoring.guru/design-patterns

https://www.youtube.com/playlist?list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW

## SOLID Principle:

**S** → Single Responsibility Principle

**O** → Open Close Principle

**L** → Liskov's Substitution Principle

**I** → Interface Segregation Principle

**D** → Dependency Inversion Principle

### Single Responsibility Principle:

*A class should have only one reason to change.*

This principle states that if we have 2 reasons to change for a class, we have to split the functionality into two classes. Each class will handle only one responsibility and in future, if we need to make one change we are going to make it in the class that handles it.

### Open Close Principle:

*Software entities like classes, modules and functions should be **open for extension** but **closed for modifications**.*

When referring to the classes Open Close Principle can be ensured by the use of Abstract Classes and concrete classes for implementing their behaviour. This will enforce having Concrete Classes extending Abstract Classes instead of changing them.

Some particular cases of this are *Template* Patterns and *Strategy* Patterns.

### Liskov's Substitution Principle:

*Derived types must be completely substitutable for their base types.*

The child should extend the parent class capabilities, not narrow it down.

This principle is just an extension of the Open Close Principle in terms of behaviour meaning that we must make sure that newly derived classes are extending the base classes without changing their behaviour. The new derived classes should be able to replace the base classes without any change in the code.

### Interface Segregation Principle:

*Clients should not be forced to depend upon interfaces that they don't use.*

When we write our interfaces we should take care to add only methods that should be there. Interfaces containing methods that are not specific to them are called polluted or fat interfaces. We should avoid them.

### Dependency Inversion Principle:

*High-level modules should not depend on low-level modules. Both should depend on abstractions.*

*Abstractions should not depend on details. Details should depend on abstractions.*

The class should depend on interfaces rather than concrete classes.

The dependency Inversion Principle states that we should decouple high-level modules from low-level modules, introducing an abstraction layer between the high-level classes and low-level classes. Furthermore, it inverts the dependency: instead of writing our abstractions based on details, we should write the details based on abstractions.

## Design Patterns

1. **Behavioural Design Patterns:**
    a. Chain Of Responsibility
    b. Command
    c. Interpreter
    d. Iterator
    e. Mediator
    f. Observer
    g. Strategy
    h. Template Method
    i. Visitor
    j. Null Object
2. **Creational Design Pattern:**
    a. Singleton
    b. Factory
    c. Factory Method
    d. Abstract Factory Method
    e. Builder
    f. Prototype
    g. Object Pool
3. **Structural Design Patterns:**
    a. Adapter
    b. Bridge
    c. Composite
    d. Decorator
    e. Flyweight

f. Memento

g. Proxy

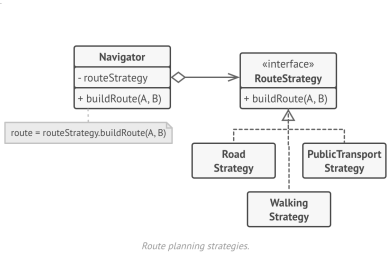
## Behavioural Design Patterns:

### Strategy Pattern:

link: https://refactoring.guru/design-patterns/strategy

**Strategy** is a behavioural design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

When a child is sharing some code that is not present in the parent, it means there is code duplication. In that case, split those functionalities using the interface and concrete class and create a has-a relationship.



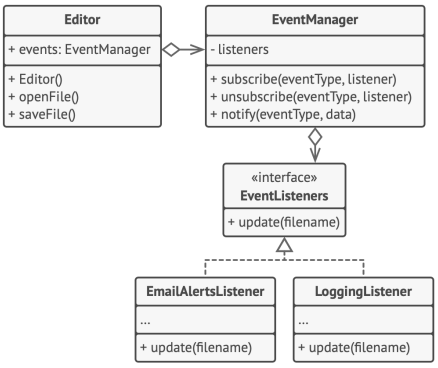

### Observer Design Pattern: Walmart interview question

link: https://refactoring.guru/design-patterns/observer

**The observer** is a behavioural design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Ex: notify-me button, Weather Station etc.

One to Many Relationships. One Observable, many observers. Like pub-sub.

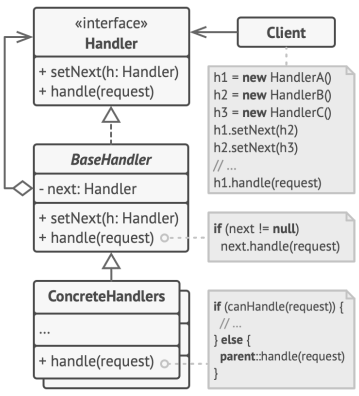



### Chain Of Responsibility:

**A chain of Responsibility** is a behavioural design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

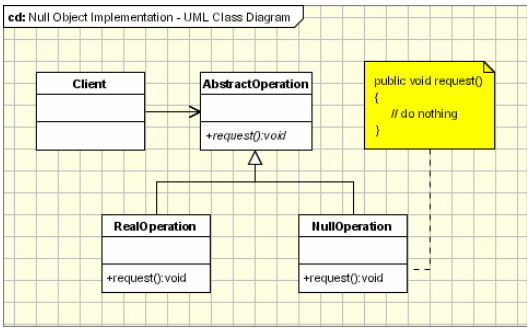
Ex: Logger System, vending machine, ATM (can use state pattern too) etc.

The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having *a slot for each type of coin,* the machine has only one slot for all of them. The dropped coin is routed to the appropriate storage place that is determined by the receiver of the command.



## NULL Object Pattern:

1. A Null Object replaces null type: We create the same class as a real class which inherits the same parent class and defines the function to do nothing or default behaviour. Therefore if a real object is not created, return a null object.
2. No need to out-check for null everywhere.




```
Facotry class:
Vehicle -> Car
                -> Null
public class VehicleFactory {
    static Vehicle getVehicleObject(String typeOf
Vehicle){
        if("Car".equals(typeOfVehicle)) {
            return new Car();
        }
        return new NullVehicle();
    }
}
```

## Command Pattern:

3 parts:  The waiter (Invoker) takes the order from the customer on his pad. The order is then queued for the order cook and gets to the cook (Receiver) where it is processed.

Invoker → It just press the button or give the command

Command Interface → It contains different-different command concrete class, refine/modify the request and forward to receiver

Receiver → which receives the command. It always remains same.

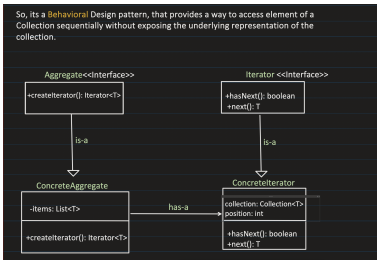
Ex: Undo, Redo, TV remote etc.

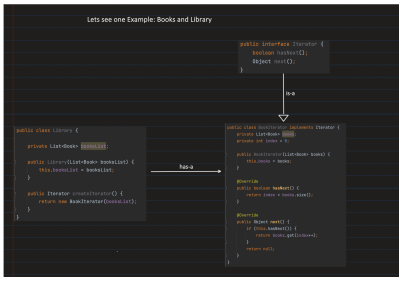
## Iterator Pattern:

**Iterator** is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Two interfaces:

Aggregate iterators will have a createiterator() method which will call



appropriate iterator.

Iterator interface will have concrete implementation of different-different iterators.

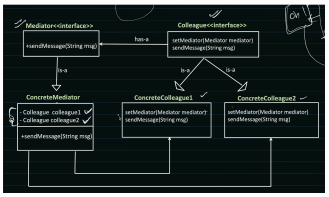
Ex: Iterator.

## Mediator Pattern:

**Mediator** lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

All requests goes via mediator.

Ex: Online auction system, Airline Management system etc.




concrete mediator maintains a list of colleagues.

## Visitor Pattern | Double Dispatch method:

link: https://www.youtube.com/watch?v=pDsz-AuFO0g&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=40 (visit again)

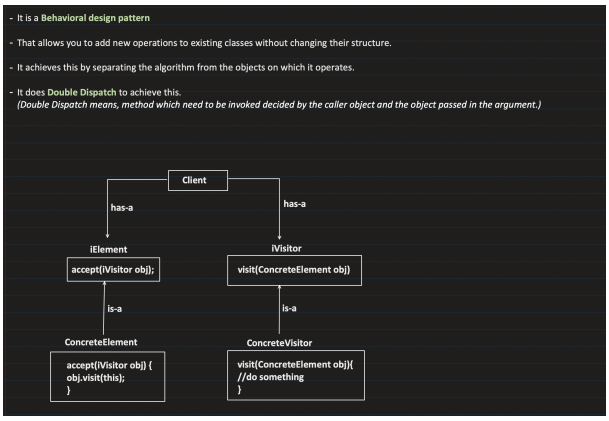
**Visitor** is a behavioral design pattern that lets you separate algorithms from the objects on which they operate. It has 2 parts, elements and visitors.

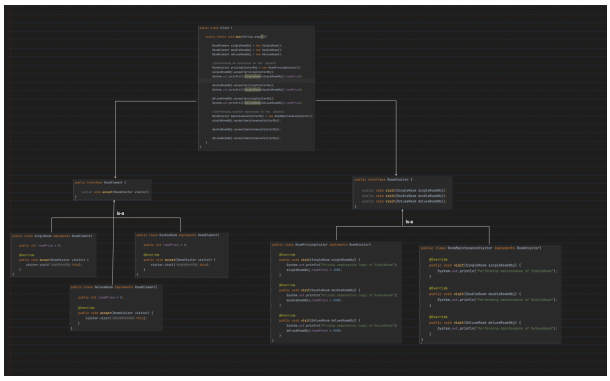
The Visitor pattern suggests that you place the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes. The original object that had to perform the behaviour is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.

Prob: Suppose we have to add xml implementation for some class. If we directly add xml fn in classes, it violets open-closed principle and single responsibility principle. To solve this we create new visitor interface and take those functionality and create concrete class out of them and call their functions at runtime with this object and behaviour object as argument.

We can't use polymorphism because exact class of a node object is unknown in advance, the overloading mechanism won't be able to determine the correct method to execute.

Define all new behaviours as class.



## Memento/Snapshot design pattern:

**Memento** is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.
Three main component:

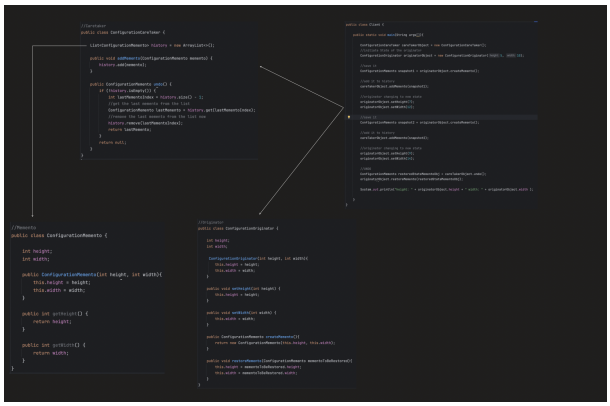
Originator: Object for which state needs to be saved. Expose methods to save and restore its state using memento object.

Memento: It represent the object which holds the state.

Caretaker: Manages the list of states(i.e. list of Memento)

It doesn't eexpose the object internal implementation. (with help of Memento class)

Ex: Undo-Redo etc.



Flow:

1. create new caretaker
2. create new originator
3. call creatememto method of originator class which will return snapshot
4. add snapshot to caretaker

### Template Pattern:

**Template Method** defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

It is like that you define a flow of instructions in a class and every subclass must follow the same flow. Their implementations might be different.

sol: create a new ***final abstract method*** which calls the internal methods in a specific flow. Now each subclass can have different definition but template method defines the flow.

### Interpreter Pattern: Not very common

- 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

The implementation of the Interpreter pattern is just the use of the composite pattern applied to represent a grammar. The Interpreter defines the behaviour while the composite defines only the structure.

little confusing: https://notebook.zohopublic.in/public/notes/74tdo9c0a25404d6d49a5b679cfe90ed19d48

## All Creational Design Pattern:

### Factory Pattern:

**Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

In simple words, when client need to create objects based on conditions, then we provide a common interface to create the objects.

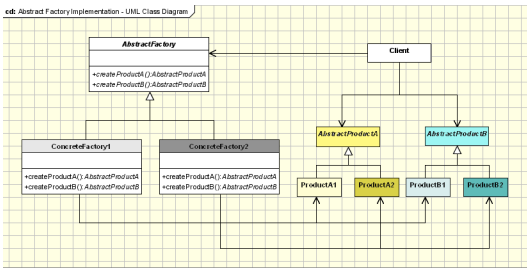
Object ⇒ ShapeFactoryObj = new ShapeFactory(); Shape shapeObj = ShapeFactoryObj.getShape(”CIRCLE”); shapeObj.draw();

```
public Document CreateDocument(String type){
    if (type.isEqual("html"))
        return new HtmlDocument();
    if (type.isEqual("proprietary"))
        return new MyDocument();
    if (type.isEqual("pdf"))
        return new PdfDocument ();
}
```

## Abstract Factory: Factory of Factory

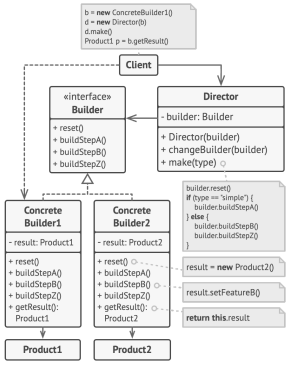
**Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

All parent classes are abstract class.




## Builder Design Pattern:

**Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.




Explanation:

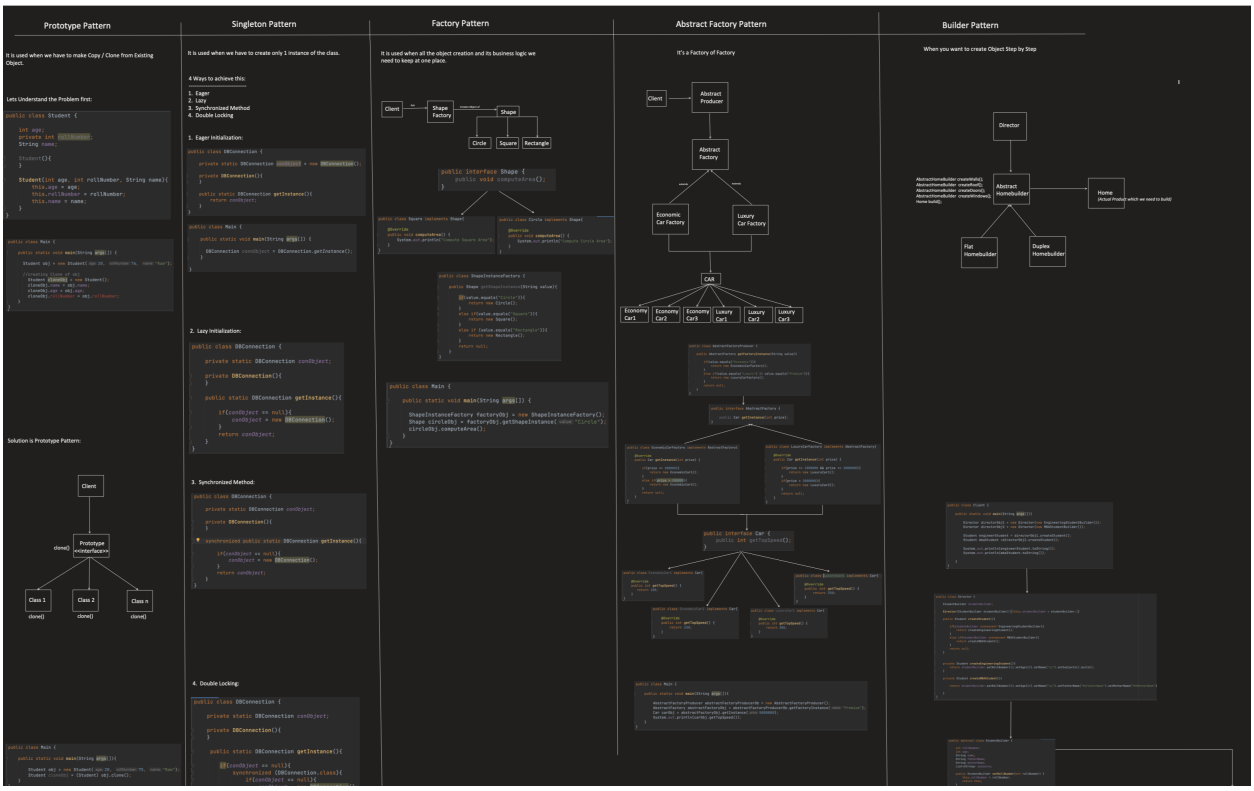
Director *has-a* builder class object and builder class has 2 concrete implementation.

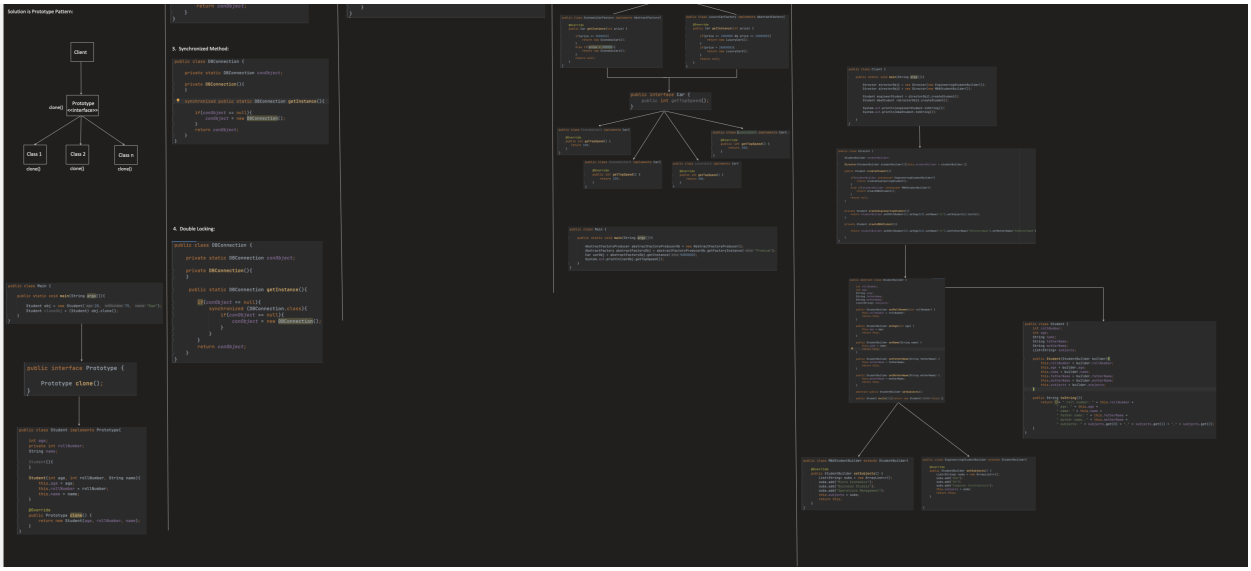
concrete-1/2 *is-a* (extends) builder class.

These concrete class itself *has-a* actual object reference.

So, when director create an object of concrete Impl class, we create the builder1 step by step and at .build we return the actual class object(product obj).

link:
https://gitlab.com/shrayansh8/interviewcodingpractise/-/tree/main/src/LowLevelDesign/DesignPatterns/BuilderDesignPatte



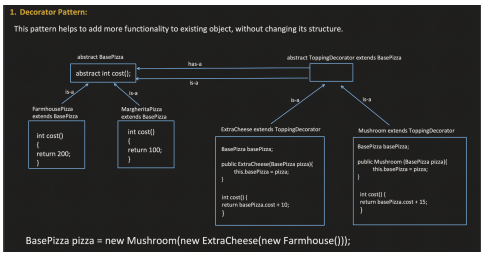
## Structural Design Patterns:

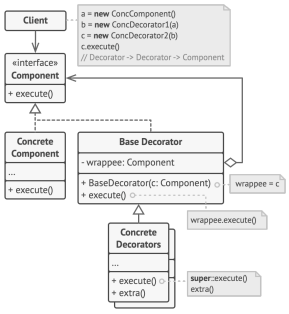
### Decorator Pattern:

**Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Decorator abstract class has has-a and is-a relationship with base component class.

Ex: Pizza (Base pizza +cheese+corn...), Notifier app , coffee machine etc.



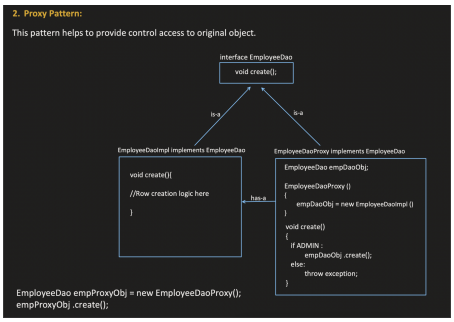




## Proxy Pattern:

**Proxy** is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

Ex: Proxy server, caching system, Preprocessing and Post processing(like logging) etc.

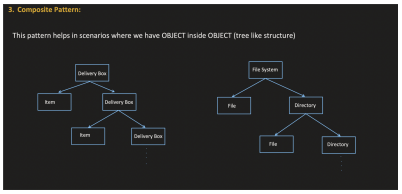


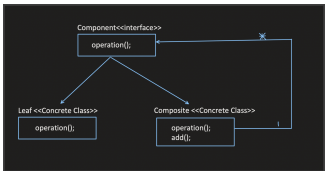

## Composite Pattern:

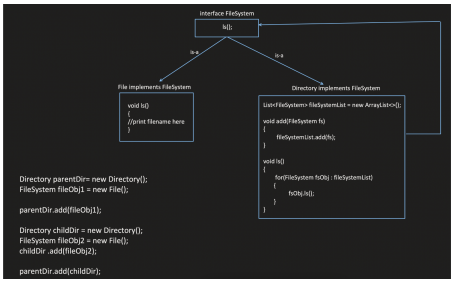
**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

A problem which has tree like structure.

Ex: File design pattern, object inside object(tree like structure) etc.



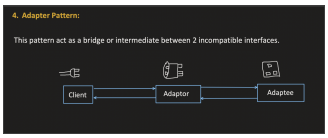



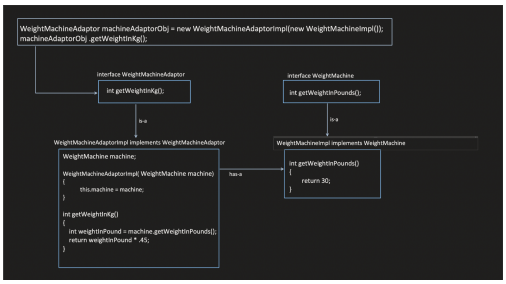
## Adapter Pattern:

**Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Convert the interface of a class into another interface clients expect.

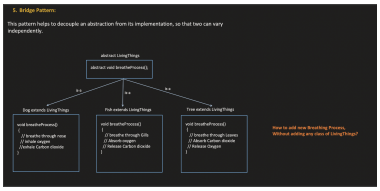
Ex: integration framework, xml to json, kg to pound etc.

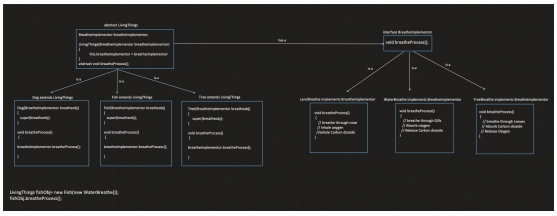



## Bridge Design Pattern:

The intent of this pattern is to decouple abstraction from implementation so that the two can vary independently.



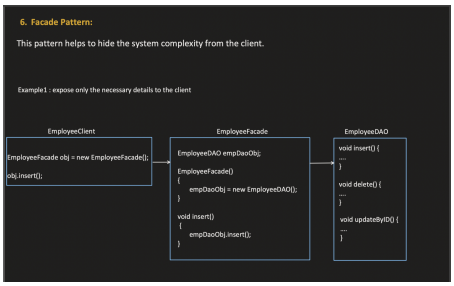

## Facade Design Pattern:

**Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

Facade layer is not mandatory if client want he can directly talk to system without facade layer.

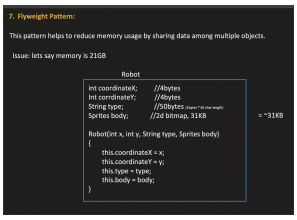
Ex: Exposing only necessary info to 3rd party apps integration etc.



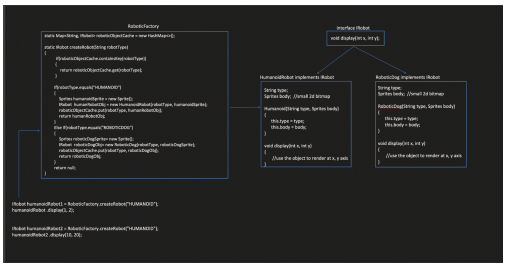

**Facade** is similar to **Proxy** in that both buffer a complex entity and initialize it on its own. Unlike *Facade*, *Proxy* has the same interface as its service object, which makes them interchangeable.

**Facade** defines a new interface for existing objects, whereas **Adapter** tries to make the existing interface usable. *Adapter* usually wraps just one object, while *Facade* works with an entire subsystem of objects.

## Flyweight Pattern:








Ex: Word Processor etc.

## More Examples:

- Design Parking Lot: https://www.youtube.com/watch?v=Sh3HeOMRoTQ
- Tic Tac Toe: https://www.youtube.com/watch?v=x8N3fINNdTE&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=10
- Design Elevator: https://www.youtube.com/watch?v=14Cc8IDWtFM
- Snake and Ladder games: https://www.youtube.com/watch?v=zRz1GPSH50I
- Implement HashMap: https://www.youtube.com/watch?v=AsAymWn7D40&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=15 (Imp)
- String Builder design pattern: https://www.youtube.com/watch?v=qOLRxN5eVC0&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=27
- BUG in Double-Checked Locking of Singleton Pattern & its Fix: https://www.youtube.com/watch?v=upfrQvOgC24&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=32 → Use volatile object type for db.connection
- Design Word Processor using Flyweight Design Pattern: https://www.youtube.com/watch?v=Mwm6tB3×1do&list=PL6W8uoQQ2c61X_9e6Net0WdYZidm7zooW&index=34
-