Unit 4

Design Patterns:

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Creational Patterns:

Singleton

- This is to ensure only one instance of the object exists in the JVM.
- The global access point creates an object the first time and the subsequent calls returns the old object.
- Good for a shared resource like a database object.
- 3 things in the implementation:
    - Static object stored somewhere.
    - The default constructor is private.
    - A static creation method which in turns calls the constructor. Subsequent calls should return the cached object.

This is eager initialization.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

This is a more proper example. Lazy initialization.

```
public class Printer {
    private static Printer printer;

    private Printer() {
    }

    public static Printer getInstance() {
        return printer == null ? printer = new Printer() : printer;
    }
}
```

This is **not thread safe** however.

```
class Singleton
{
    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

This is **thread safe**.

A combo of these gives us Double Checked Locking

```
class Singleton
{
    private static volatile Singleton obj  = null;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj==null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

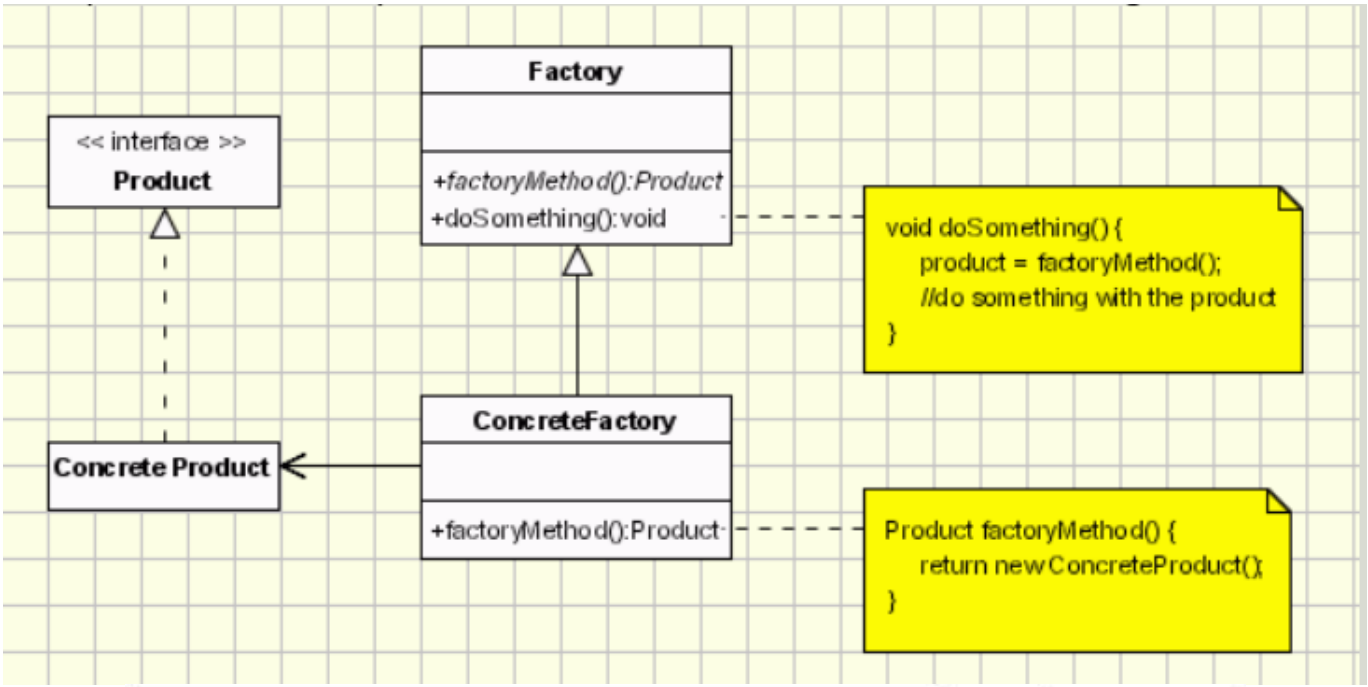
This has a few issues:

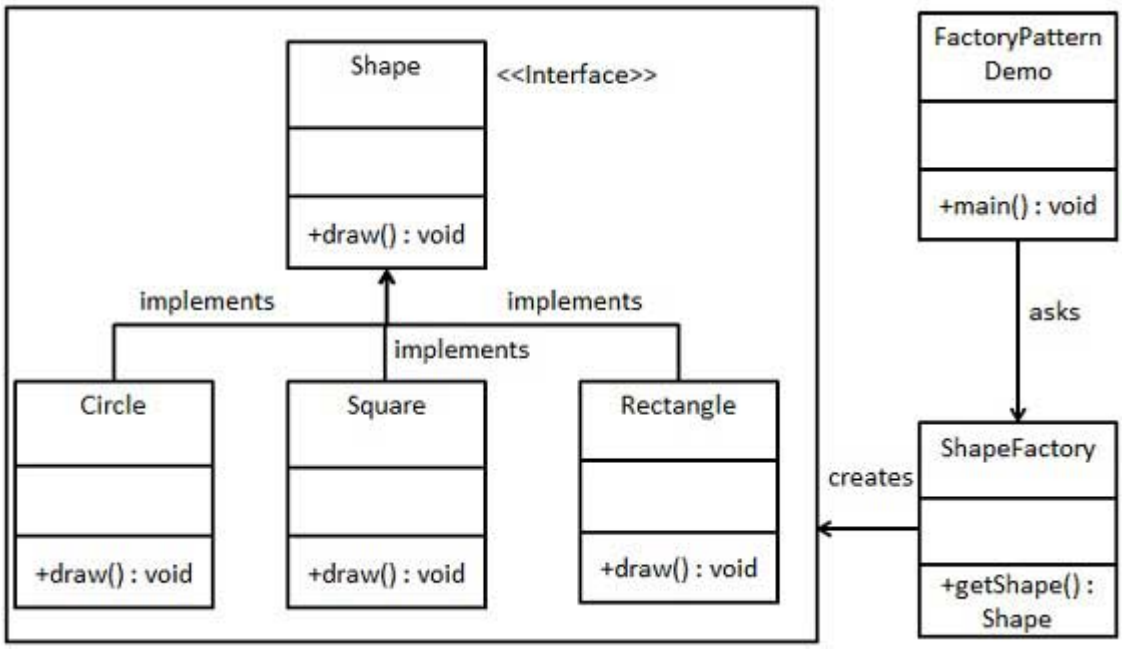
- SRP is usually violated.
- It can potentially mask bad design.
- It needs these extra bits for implementation in multi threaded environment.
- Harder to test

Factory

> defines an interface for creating an object, but let subclasses decide which class to instantiate

- This is for creating objects but we don't specify the exact class. We can instead specify requirements or something like its name.
- Subclasses override the creation method.





Here we can send the shape name to ShapeFactroy and it will return the appropriate Shape based object.

Times to use factory:

- Can't anticipate the class of objects we must create.
- We want the subclasses to specify the objects it creates.

Pros:

- Coupling down
- SRP
- Open / Close

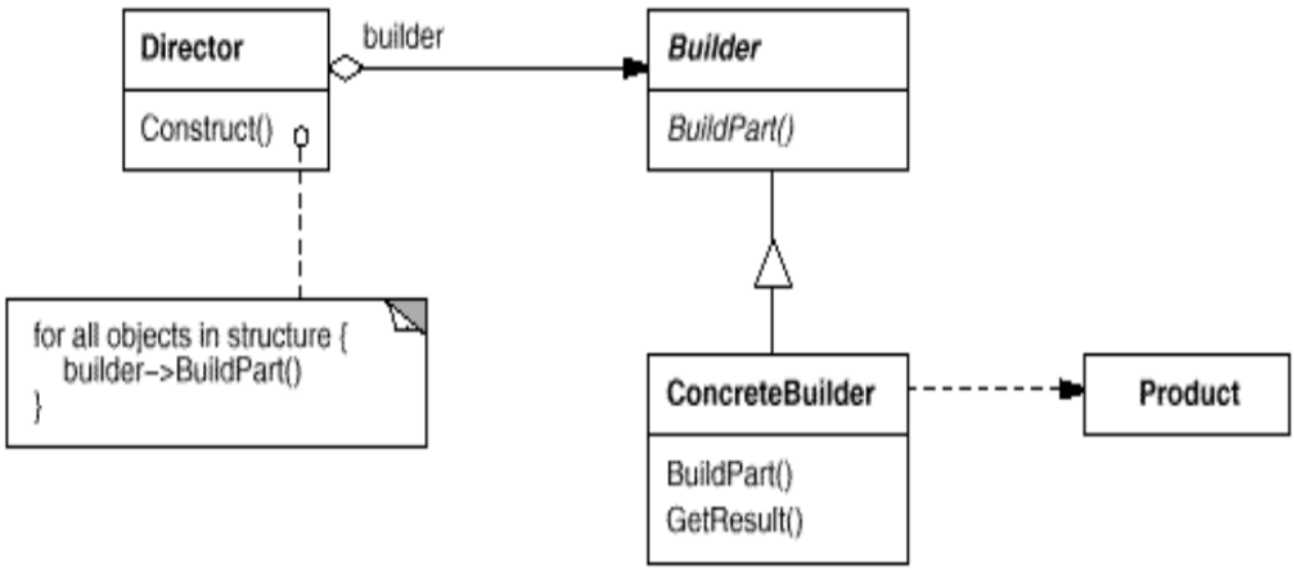
Cons:

- Complex

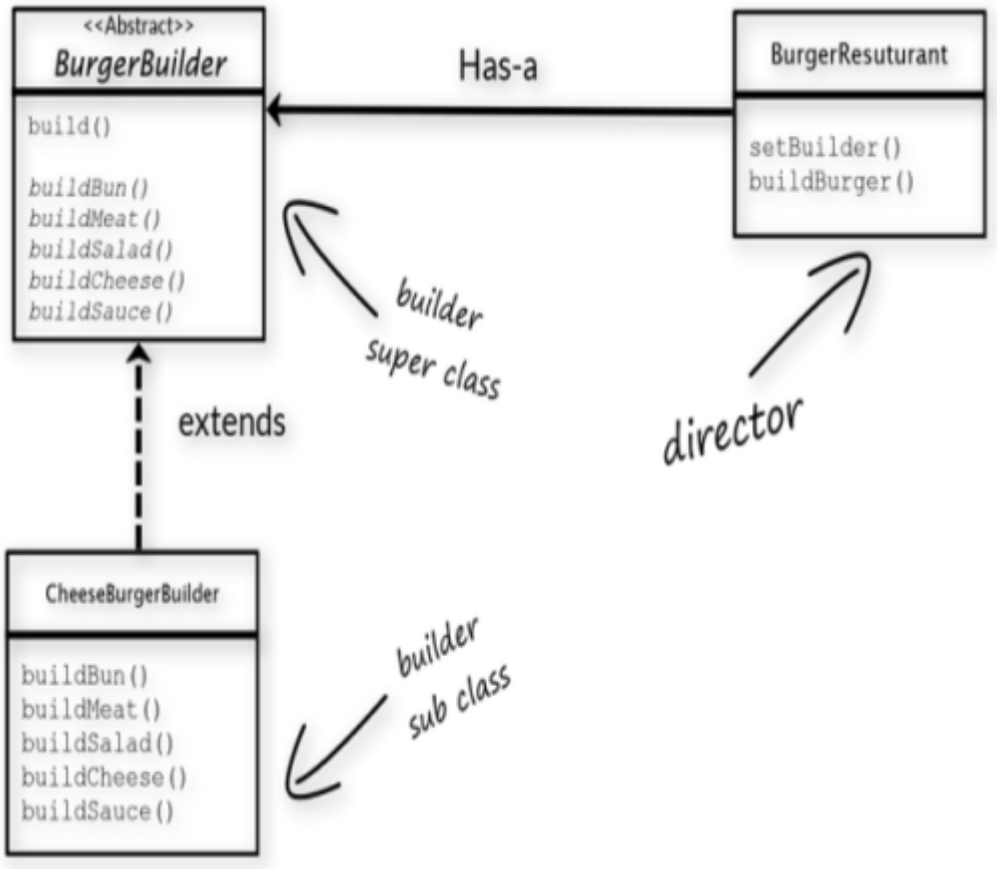
Builder

- Creational Design Pattern. Encapsulation. Maybe by default abstraction also?
- Builder Pattern says that "construct a complex object from simple objects using step-by-step approach"
- Usually good for making SRP type classes. Also to make complex objects.
- Useful for breaking huge constructors with large number of parameters.
- Useful for making diff representations of the same object. Ex: burger thingy below.
- SOLID compliance is improved. => code-reuse.

One stupid implementation is an open ended conversion. Lets say that we are trying convert from one text format to another. A class called DocxToPdf will be converting a Docx object to pdf object.



Let's see an implementation.

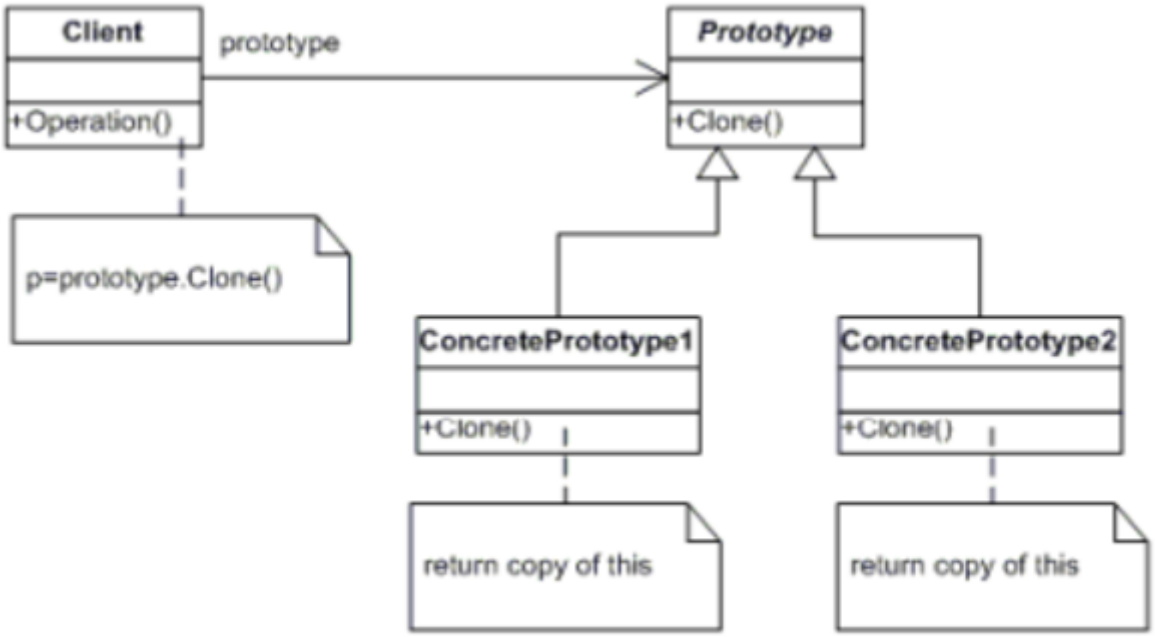


To be more accurate there will be more implementations of BurgerBuilder like HamBurgerBuilder or CornCheeseBuilder etc.

The director type implementation will also have **polymorphism**.

Prototype

- This is for when creating new objects is expensive. So instead we will be cloning.
- Useful when we the required objects are specified at run time.
- When a object has a few different states, it is easier to clone rather than make new one from scratch.
- Easier to add and remove objects.
- Harder to make more complex objects.

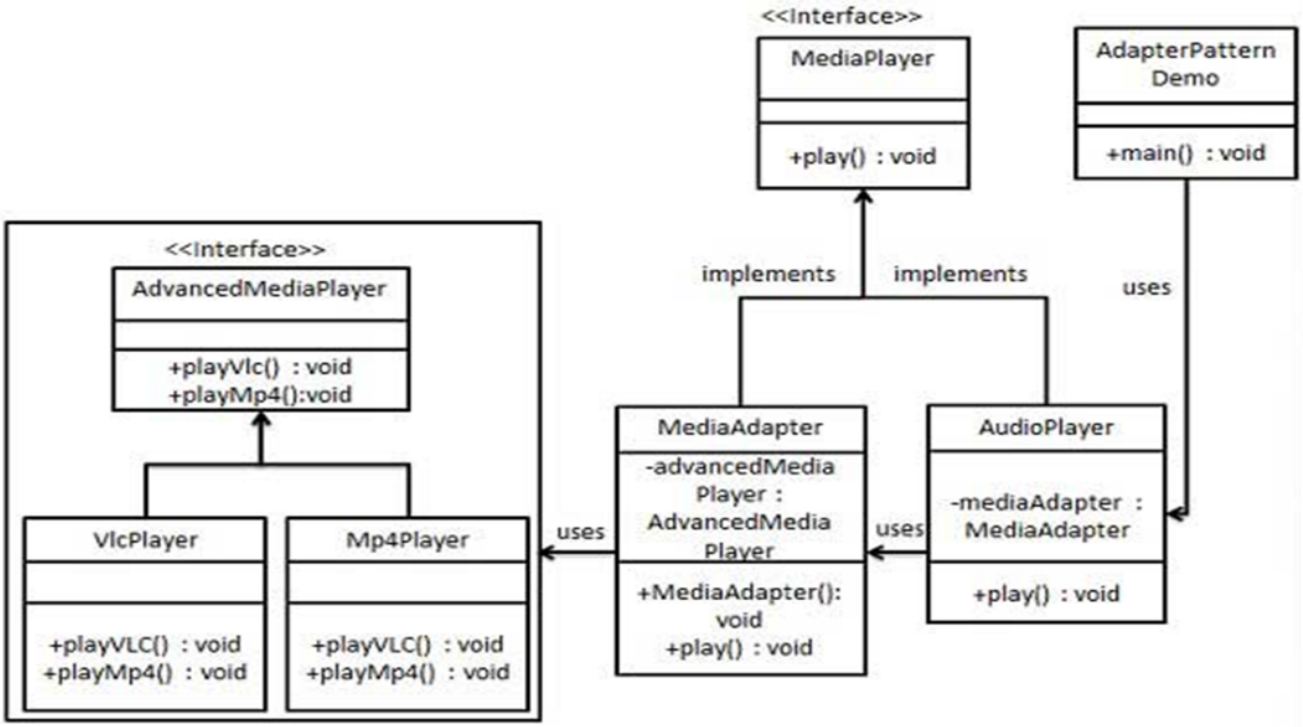


We can have a prototype registry which is kinda like a factory.

Structural Patterns:

Adapter

- A convertor lol. Structural pattern.
- Makes a bridge between objects that need to interact with each other.
- This is sometimes considered like a wrapper.



The MP4 ,VLC problem.

Adaptee is the class getting converted. This is "controversial".

Object Adaptor vs Class Adaptor

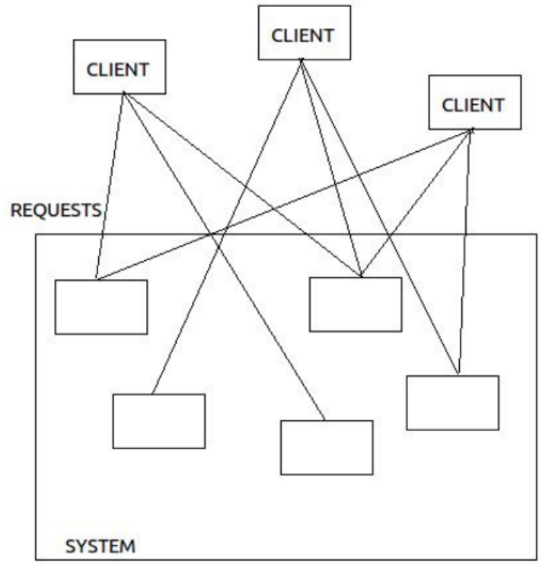
- Objects Adapters uses composition, the Adaptee delegates the calls to Adaptee (opossed to class adapters which extends the Adaptee).
- The main advantage is that the object Adapter adapts not only the Adaptee but all its subclasses. All it's subclasses with one "small" restriction: all the subclasses which don't add new methods, because the used mechanism is delegation. So for any new method the Adapter must be changed or extended to expose the new methods as well.
- The main disadvantage is that it requires to write all the code for delegating all the necessary requests to the Adaptee.

- Class adapter uses inheritance instead of composition. It means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee.
- There are advantages and disadvantages: It adapts the specific Adaptee class. The class it extends. If that one is subclassed it can not be adapted by the existing adapter.
- It doesn't require all the code required for delegation, which must be written for an Object Adapter.
- If the Target is represented by an interface instead of a class then we can talk about "class" adapters, because we can implement as many interfaces as we want.

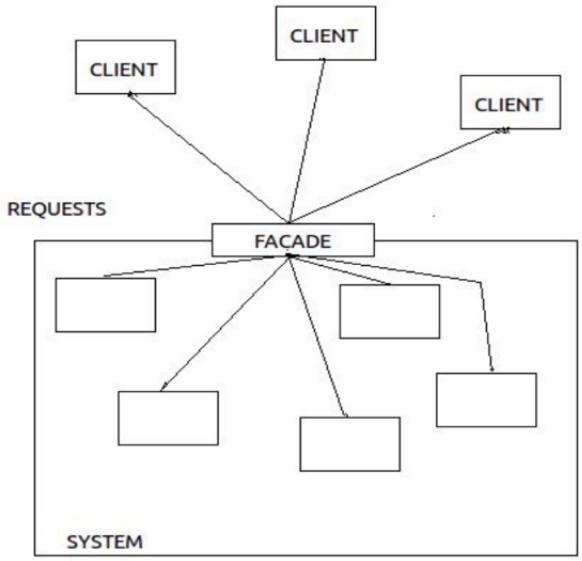
Façade

- It gives a interface to let us use much more complex classes and objects.
- Can be created using an overarching `interface` or a regular `class`

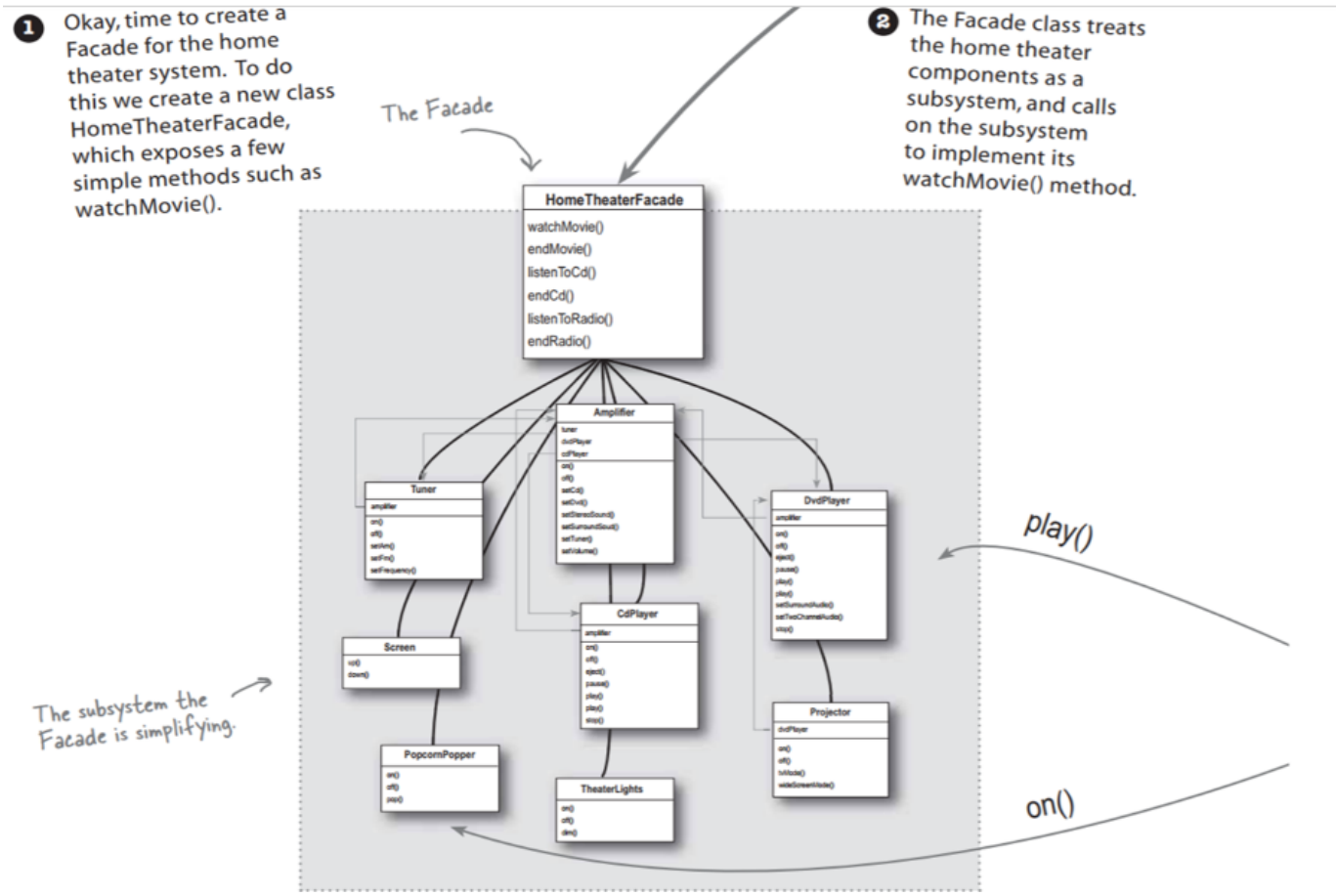
- It defines an entry point.
- It decouples the subsystems from the clients. It can sometimes remove circular dependencies.



Without facade

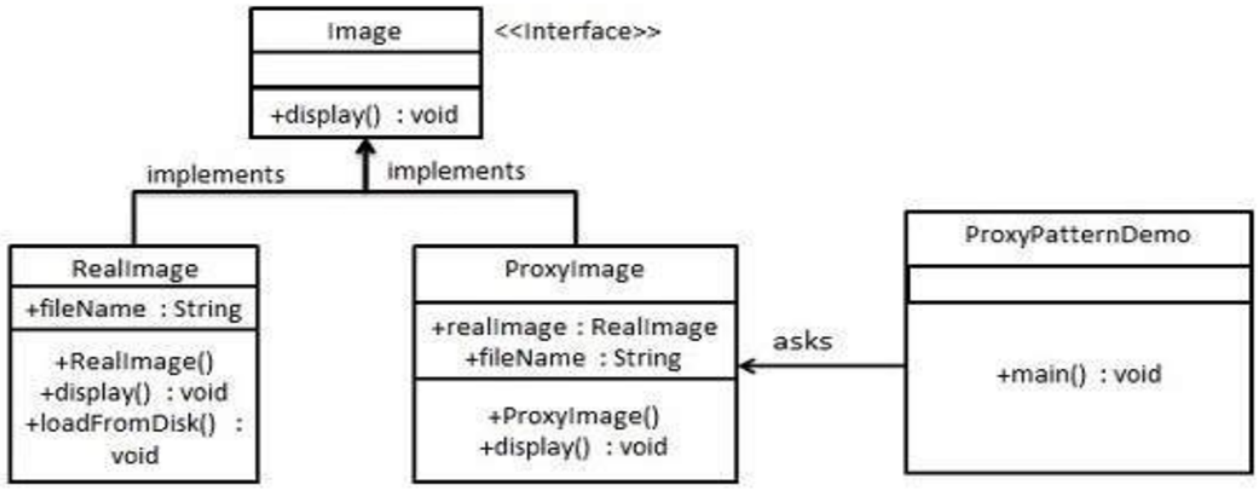


With facade



Proxy

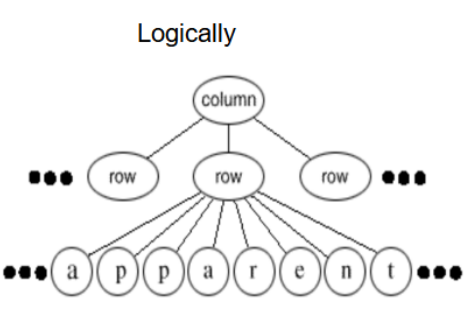
- Provides a surrogate/placeholder for another object.
- Can be used for various things:
    - Lazy loading => Virtual Proxy: Lightweight replacement when the real object is very heavy.
    - Access Control => Protection Proxy
    - Remote loading => Remote Proxy: Has basic stuff of something in a remote location.
    - Caching => Cache Proxy
- It follows open/closed principle

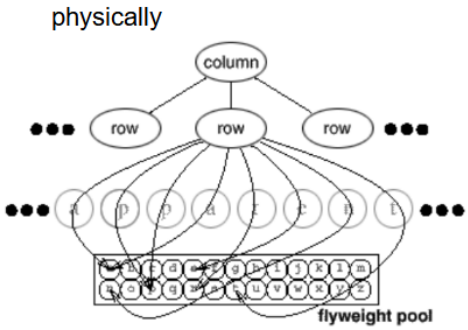


One way to look at this is that proxy image will be shown until the real image gets loaded.

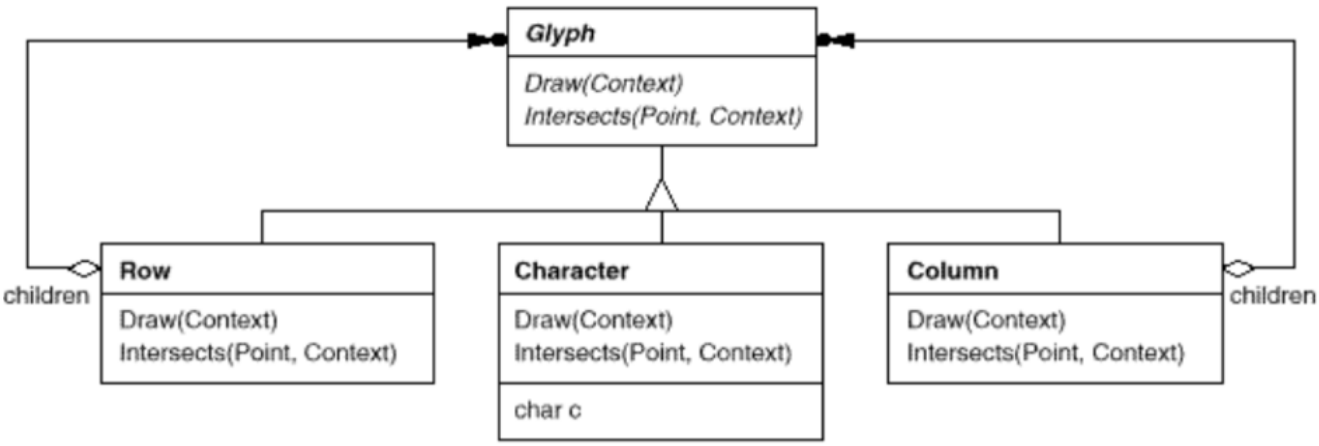
Flyweight

- These objects do not store their state. It is stored outside. Intrinsic and Extrinsic State.
- Good for sharing objects with many instances but a few types.
- "parts of state" can be shared between objects. All the above let us reduce RAM usage.
- This lets us propagate changes more efficiently. Cuz of the kinda centralized nature.
- Extrinsic state calculations can sometimes be very taxing.

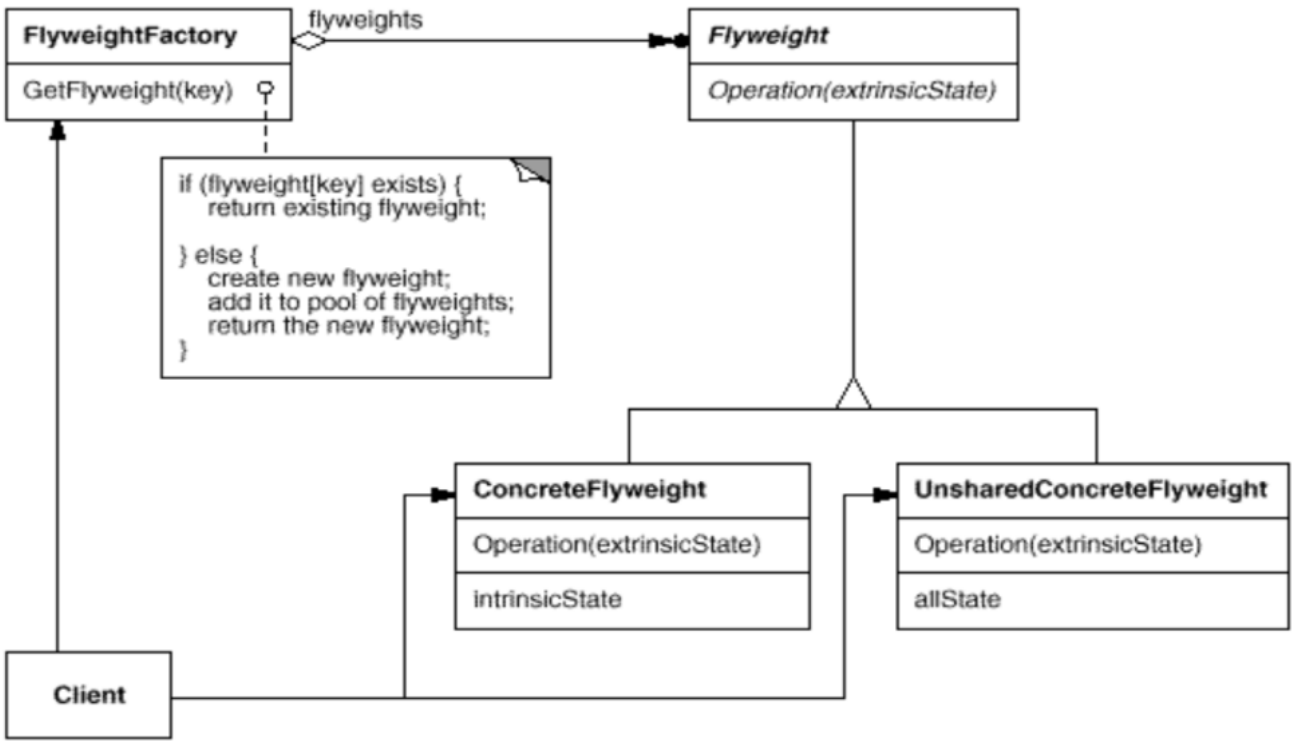




A document editing software.



UML



Class Diagram

Example for flyweight:

> A pen that has various different refill ink colors. Everything else is shared. So intrinsic attributes will be pen nib, body etc. Extrinsic will be refill color.