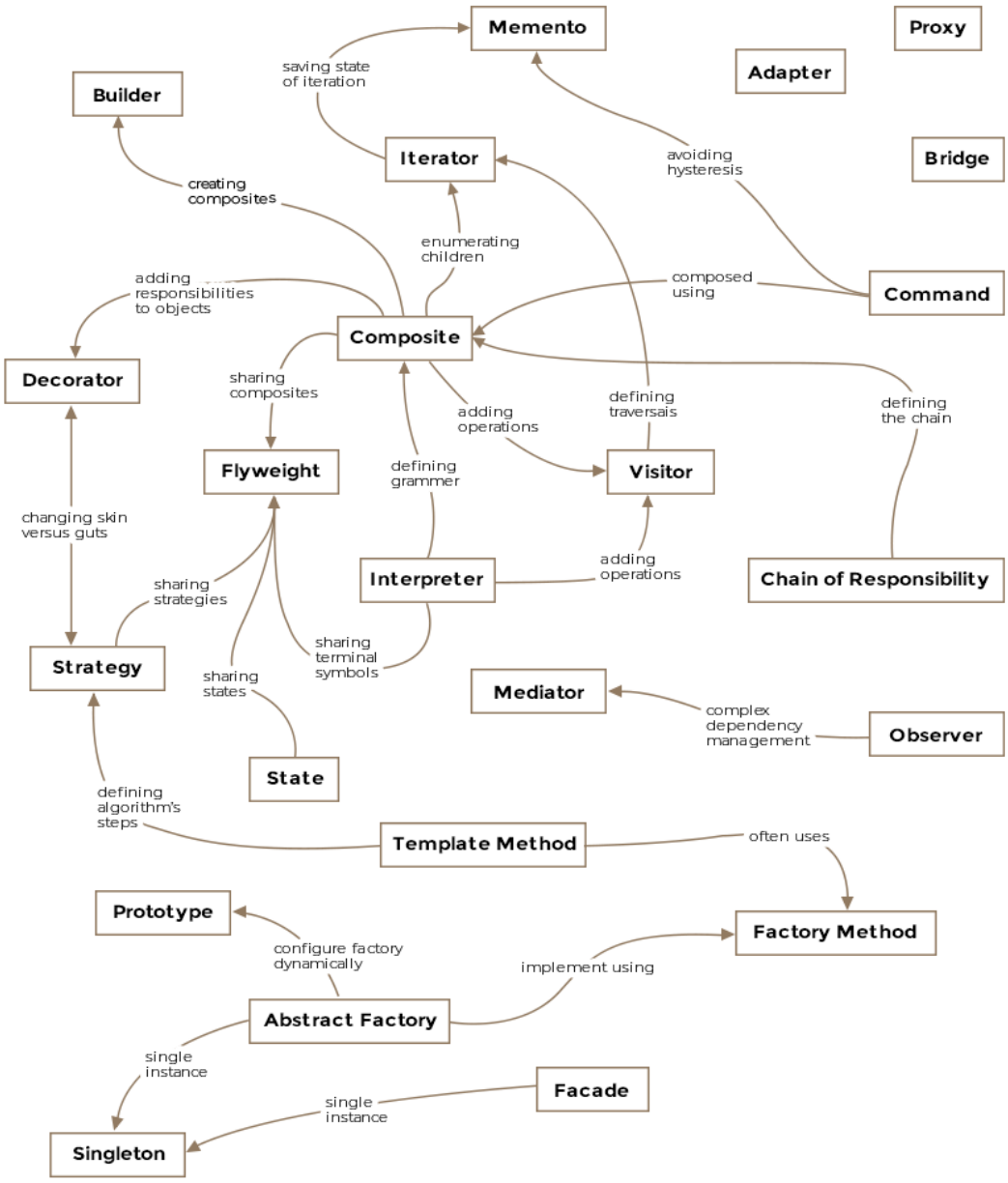
# Introduction

This lesson lays down the groundwork for understanding design patterns

## Why Patterns ?

Why do we need patterns? The blunt answer is ***we don't want to reinvent the wheel!*** Problems that occur frequently enough in tech life usually have well-defined solutions, which are flexible, modular and more understandable. These solutions when abstracted away from the tactical details become design patterns. If you experienced a déjà vu feeling when designing a solution for a problem that felt eerily similar to the solution of a previous problem, albeit in a different domain, then you were probably using a pattern unknowingly.

Below is an image showing the relationship among the various design patterns as explained by the seminal design patterns work done by the gang of four.




**Design Pattern Relationships**

## Example

Let's consider an example to understand what design patterns are and how do they get applied. The class constructor is one of the basic concepts in an object orientated language. The constructors help create objects of the class and can take in parameters. Let us take the following class as an example.

```
public class Aircraft {


    private String type;

    public Aircraft(String type) {
        this.type = type;
    }
}
```

In the above example, we have the default constructor for the class that takes in a single parameter the `type` of the aircraft. Now say after a few days, you realize you want to add additional properties to your `Aircraft` class. Say you want to add the color of the aircraft as a property, but you have already released a version of your library and can't modify the original constructor. The solution is to add another constructor with two parameters like so

```
public class Aircraft {

    private String type;
    private String color;

    public Aircraft(String type) {
        this.type = type;
    }

    public Aircraft(String type, String color) {
        this.type = type;
        this.color = color;
    }
}
```

If you continue this way you'll end up having a bunch of constructors with increasing number of arguments looking like a telescope:

```


Aircraft(String type)
Aircraft(String type, String color)
Aircraft(String type, String color, String prop3)
Aircraft(String type, String color, String prop3, String prop4)


```

The telescoping pattern is called an ***anti-pattern: how NOT to do things!*** The way to approach a class with an increasing number of variables is to use the **Builder Pattern** that we'll discuss in depth in the following chapters.

Seasoned developers are expected to be well-versed in design patterns and applying them makes the code reusable and maintainable for future. Design patterns aren't limited to object orientated languages but also exist for other domains of Computer Science such as distributed systems, big data system or user interface.

## Suggestions for Object Orientated Design

Whenever writing code in an object orientated language, sticking to the following list of suggestions will make your code amenable to changes with the least effort.

- Separate out parts of code that vary or change from those that remain the same.
- Always code to an interface and not against a concrete implementation.
- Encapsulate behaviors as much as possible.
- Favor composition over inheritance. Inheritance can result in explosion of classes and also sometimes the base class is fitted with new functionality that isn't applicable to some of its derived classes.
- Interacting components within a system should be as loosely coupled as possible.
- Ideally, class design should inhibit modification and encourage extension.
- Using patterns in your day to day work, allows exchanging entire implementation concepts with other developers via shared pattern vocabulary.

Some of the above suggestions are embodied in the patterns we'll be discussing in the upcoming lessons. However, remember that making one's design flexible and extensible correspondingly increases the complexity and understandability of the code base. One must walk a fine line between the two competing objectives when designing and writing software.

# Types of Design Patterns

This lesson lists the most used and familiar design patterns

## Introduction

Design patterns for object orientated programs are divided into three broad categories listed below. These are the same categories used by GoF in their seminal work on design patterns.

- Creational
- Structural
- Behavioural

Each of these are explained below

## Creational

Creational design patterns relate to how objects are constructed from classes. New-ing up objects may sound trivial but unthoughtfully littering code with object instance creations can lead to headaches down the road. The creational design pattern come with powerful suggestions on how best to encapsulate the object creation process in a program.

- Builder Pattern
- Prototype Pattern
- Singleton Pattern
- Abstract Factory Pattern

## Structural

Structural patterns are concerned with the composition of classes i.e. how the classes are made up or constructed. These include:

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

## Behavioral

Behavioral design patterns dictate the interaction of classes and objects amongst eachother and the delegation of responsibility. These include:

- Interpreter Pattern
- Template Pattern
- Chain of Responsibility Pattern
- Command Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern

- Strategy Pattern

- Visitor Pattern

## Examples

For most of the patterns, we borrow concepts from the aviation industry to create our examples. You'll find the course regularly talking about F-16s and Boeings to elaborate aspects of the pattern under discussion.

## For Interview Prep

For folks, who are rushing through the course for an upcoming interview, I would suggest going through all the **creational design patterns**, **decorator**, **proxy**, **iterator**, **observer** and **visitor** patterns. As you read through them, be sure to look at the Java framework's api examples pointed out in each lesson.

# Builder Pattern

This lesson discusses how complex objects can be represented and constructed without coupling the two operations.

## What is it ?

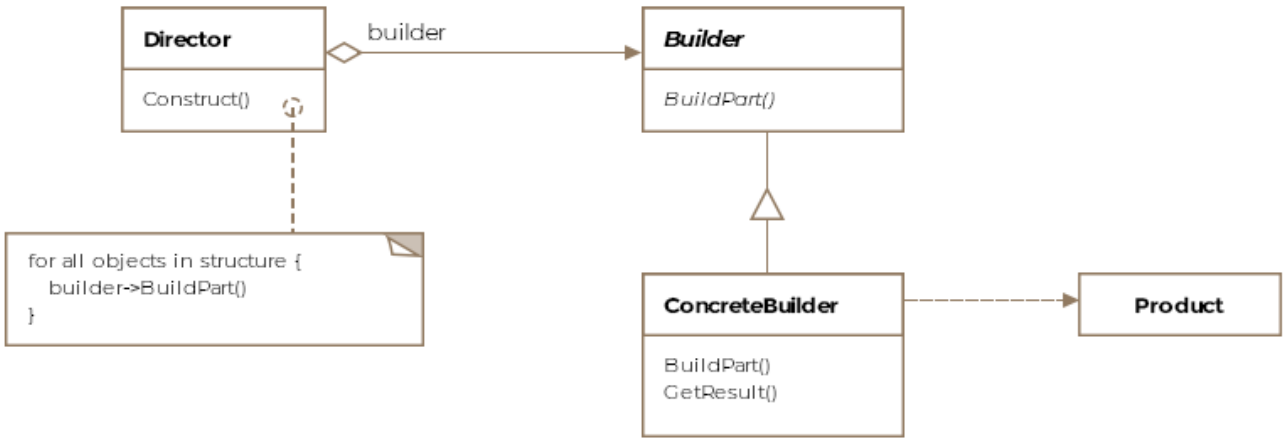
As the name implies, a builder pattern is used to build objects. Sometimes, the objects we create can be complex, made up of several sub-objects or require an elaborate construction process. The exercise of creating complex types can be simplified by using the builder pattern. A *composite* or an *aggregate* object is what a builder generally builds.

Formally, a ***builder pattern encapsulates or hides the process of building a complex object and separates the representation of the object and its construction. The separation allows us to construct different representations using the same construction process.*** In Java speak, different representations implies creating objects of different classes that may share the same construction process.

## Class Diagram

The class diagram consists of the following entities

- **Builder**
- **Concrete Builder**
- **Director**
- **Product**




Class Diagram

## Example

Continuing with our example of airplanes let's say the construction of an aircraft involves the three steps in order:

1. the making of the cockpit
2. then the engine
3. and finally the wings

In our hypothetical world, every aircraft requires at least the above three steps. However, a passenger aircraft can have an added step of making bathrooms in the plane. The steps represent the ***construction*** process from our formal definition. The ***product*** is an aircraft but can have different ***representations*** such as an F-16 or a Boeing-747. Using the same construction process, we should be able to produce both F-16s and Boeings.

Let's see some code now. First we'll start with the abstract interface for our `AircraftBuilder` class. The builder contains a method for each component that can be part of the final product. These methods are selectively overridden by concrete builders depending on if the builders will be including that part in the final product variant that they are responsible for building.

```
public abstract class AircraftBuilder {

    public void buildEngine() {

    }

    public void buildWings() {

    }

    public void buildCockpit() {

    }

    public void buildBathrooms() {

    }

    abstract public IAircraft getResult();
}
```

Now we'll implement two concrete builders, one for F-16 and one for Boeing-747.

```
public class Boeing747Builder extends AircraftBuilder {

    Boeing747 boeing747;

    @Override
    public void buildCockpit() {

    }

    @Override
    public void buildEngine() {

    }

    @Override
    public void buildBathrooms() {

    }

    @Override
    public void buildWings() {
```

```
    }


    public IAircraft getResult() {
        return boeing747;
    }
}

public class F16Builder extends AircraftBuilder {

    F16 f16;

    @Override
    public void buildEngine() {
        // get F-16 an engine
        // f16.engine = new F16Engine();
    }

    @Override
    public void buildWings() {
        // get F-16 wings
        // f16.wings = new F16Wings();
    }

    @Override
    public void buildCockpit() {
        f16 = new F16();
        // get F-16 a cockpit
        // f16.cockpit = new F16Cockpit();
    }

    public IAircraft getResult() {
        return f16;
    }
}
```

For brevity's sake, we have provided the skeleton of the builders and skipped individual implementation of each method. Note the `F16Builder` doesn't override the `buildBathrooms` method, since there are no bathrooms in the F-16 cockpit. The Boeing's builder does override the bathroom's method since a Boeing-747 has bathrooms for passengers.

The process or algorithm required to construct the aircraft which in our case is the specific order in which the different parts are created is

case is the specific order in which the different parts are created is captured by another class called the `Director`. The director is in a *sense* ***directing*** the construction of the aircraft. The final product is still returned by the builders.

```
public class Director {

    AircraftBuilder aircraftBuilder;

    public Director(AircraftBuilder aircraftBuilder) {
        this.aircraftBuilder = aircraftBuilder;
    }

    public void construct(boolean isPassenger) {
        aircraftBuilder.buildCockpit();
        aircraftBuilder.buildEngine();
        aircraftBuilder.buildWings();

        if (isPassenger)
            aircraftBuilder.buildBathrooms();
    }
}
```

Notice how we can pass in the builder of our choice, and vary the *aircraft product* (representation) to be either an F-16 or a Boeing-747. In our scenario, the builders return the same supertype however that may not be the case if the builders return products that aren't very similar.

The client will consume the pattern like so:

```
public class Client {

    public void main() {

        F16Builder f16Builder = new F16Builder();
        Director director = new Director(f16Builder);
        director.construct(false);

        IAircraft f16 = f16Builder.getResult();
    }
}
```

The `AircraftBuilder` interface hides how a given aircraft gets built. The client is unaware of the classes `F16Engine`, `F16Cockpit` and similar classes for Boeing-747.

## Skipping the Director

You may find the builder pattern being used without the **director**. The client can directly instantiate the builder and invoke the required methods to get a product for itself. This is a common antidote for **telescoping constructors**. Imagine a class with too many attributes but some attributes are to be set optionally. In such a case the builder can be invoked to only set the required attributes and create a product

## Other Examples

- The Jave api exposes a `StringBuilder` class that doesn't really conform to the strict reading of the GoF builder pattern but can still be thought of as an example of it. Using the `StringBuilder` instance we can successively create a string by using the `append` method.
- Another hypothetical example could be creating documents of type pdf or html. Consider the snippet below:

```
    public IDocument construct(DocumentBuilder documentBuilder) {

        documentBuilder.addTitle("Why use design patterns");
        documentBuilder.addBody("blah blah blah... more blah bla
h blah");
        documentBuilder.addAuthor("C. H. Afzal");
        documentBuilder.addConclusion("Happy Coding!");

        // Return the document and depending on the concrete
```

```
        // implementation of the DocumentBuilder, we could return
        // either a pdf or html document.

        return documentBuilder.buildDocument();

    }
```

The above method can appear in a director or client code and different document types can be built by varying the concrete type of the DocumentBuilder passed into the method. We could have a `HtmlDocumentBuilder` and a `PdfDocumentBuilder` derive from the abstract class `DocumentBuilder`.

## Caveats

- The builder pattern might seem similar to the abstract factory pattern but one difference is that the builder pattern creates an object step by step whereas the abstract factory pattern returns the object in one go.

# Singleton Pattern

This lesson discusses how the Singleton pattern enforces only a single instance of a class to ever get produced and exist throughout an application's lifetime.

## What is it ?

Singleton pattern as the name suggests is used to create one and only instance of a class. There are several examples where only a single instance of a class should exist and the constraint be enforced. Caches, thread pools, registries are examples of objects that should only have a single instance.

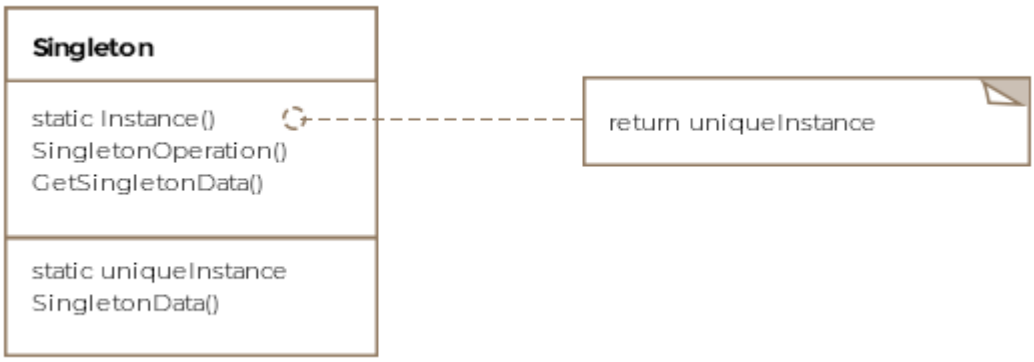
Its trivial to new-up an object of a class but how do we ensure that only one object ever gets created? The answer is to make the constructor private of the class we intend to define as singleton. That way, only the members of the class can access the private constructor and no one else.

Formally the Singleton pattern is defined as ***ensuring that only a single instance of a class exists and a global point of access to it exists***.

## Class Diagram

The class diagram consists of only a single entity

- **Singleton**




Class Diagram

## Example

As an example, let's say we want to model the American President's official aircraft called "Airforce One" in our software. There can only be one instance of Airforce One and a singleton class is the best suited representation.

Below is the code for our singleton class

```
public class AirforceOne {

    // The sole instance of the class
    private static AirforceOne onlyInstance;

    // Make the constructor private so its only accessible to
    // members of the class.
    private AirforceOne() {
    }

    public void fly() {
        System.out.println("Airforce one is flying...");
    }

    // Create a static method for object creation
    public static AirforceOne getInstance() {

        // Only instantiate the object when needed.
        if (onlyInstance == null) {
            onlyInstance = new AirforceOne();
        }
```

```
        return onlyInstance;

    }
}

public class Client {

    public void main() {
        AirforceOne airforceOne = AirforceOne.getInstance();
        airforceOne.fly();
    }
}
```

## Multithreading and Singleton

The above code will work hunky dory as long as the application is single threaded. As soon as multiple threads start using the class, there's a potential that multiple objects get created. Here's one example scenario:

- Thread **A** calls the method `getInstance` and finds the `onlyInstance` to be null but before it can actually new-up the instance it gets context switched out.
- Now thread **B** comes along and calls the `getInstance` method and goes on to new-up the instance and returns the `AirforceOne` object.
- When thread **A** is scheduled again, is when the mischief begins. The thread was already past the if null condition check and will proceed to new-up another object of `AirforceOne` and assign it to `onlyInstance`. Now there are two different `AirforceOne` objects out in the wild, one with thread A and one with thread B.

There are two trivial ways to fix this race condition.

- One is to add `synchronized` to the `getInstance()` method.

  ```
  synchronized public static AirforceOne getInstance()
  ```

- The other is to undertake static initialization of the instance, which is guaranteed to be thread-safe.

```
    // The sole instance of the class
    private static AirforceOne onlyInstance = new AirforceOne();
```

The problem with the above approaches is that synchronization is expensive and static initialization creates the object even if it's not used in a particular run of the application. If the object creation is expensive then static intialization can cost us performance.

## Double-Checked Locking

The next evolution of our singleton pattern would be to synchronize only when the object is created for the first time and if its already created, then we don't attempt to synchronize the accessing threads. This pattern has a name called **"double-checked locking"**.

```
public class AirforceOneWithDoubleCheckedLocking {

    // The sole instance of the class. Note its marked volatile
    private volatile static AirforceOneWithDoubleCheckedLocking onlyI
nstance;

    // Make the constructor private so its only accessible to
    // members of the class.
    private AirforceOneWithDoubleCheckedLocking() {
    }

    public void fly() {
        System.out.println("Airforce one is flying...");
    }

    // Create a static method for object creation
    synchronized public static AirforceOneWithDoubleCheckedLocking ge
tInstance() {

        // Only instantiate the object when needed.
        if (onlyInstance == null) {
            // Note how we are synchronizing on the class object
```

```
            synchronized (AirforceOneWithDoubleCheckedLocking.class)
{
                if (onlyInstance == null) {
                    onlyInstance = new AirforceOneWithDoubleCheckedLo
cking();
                }
            }
        }

        return onlyInstance;
    }
}
```

The above solution marks the singleton instance volatile however the JVM `volatile` implementation for Java versions 1.4 will not work correctly for double checked locking and you'll need to use another way to create your singletons.

The *double checked locking* is now considered an antipattern and its utility has largely passed away as JVM startup times have sped up over the years.

## Other Examples

In the Java API we have the following singletons:

- java.lang.Runtime
- java.awt.Desktop

## Caveats

- Its possible to subclass a singleton class by making the constructor

protected instead of private. It might be suitable under some circumstances. An approach taken in these scenarios is to create a

**register of singletons** of the subclasses and the `getInstance` method can take in a parameter or use an environment variable to return the desired singleton. The registry maintains a mapping of string names to singleton objects.

# Prototype Pattern

This lesson discusses how new objects can be created from existing objects using the prototype pattern.

## What is it ?

Prototype pattern involves creating new objects by copying existing objects. The object whose copies are made is called the **prototype**. You can think of the prototype object as the seed object from which other objects get created but you might ask why would we want to create copies of objects, why not just create them anew? The motivations for prototype objects are as follows:

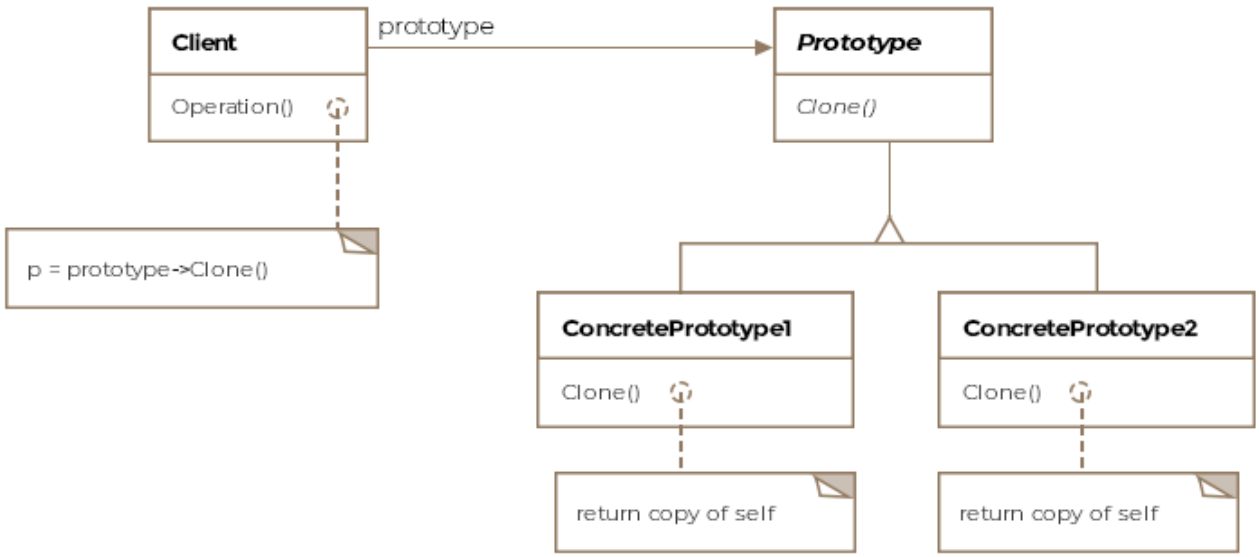
- Sometimes creating new objects is more expensive than copying existing objects.
- Imagine a class will only be loaded at runtime and you can't access its constructor statically. The run-time environment creates an instance of each dynamically loaded class automatically and registers it with a **prototype manager**. The application can request objects from the prototype manager which in turn can return clones of the prototype.
- The number of classes in a system can be greatly reduced by varying the values of a cloned object from a prototypical instance.

Formally, the pattern is defined as ***specify the kind of objects to create using a prototypical instance as a model and making copies of the prototype to create new objects***.

## Class Diagram

The class diagram consists of the following entities

- **Prototype**
- **Concrete Prototype**
- **Client**




Class Diagram

## Example

Let's take an example to better understand the prototype pattern. We'll take up our aircraft example. We created a class to represent the F-16. However, we also know that F-16 has a handful of variants. We can subclass the F16 class to represent each one of the variants but then we'll end up with several subclasses in our system. Furthermore, let's assume that the F16 variants only differ by their engine types. Then one possibility could be, we retain only a single F16 class to represent all the versions of the aircraft but we add a setter for the engine. That way, we can create a single F16 object as a prototype, clone it for the various versions and compose the cloned jet objects with the right engine type to

represent the corresponding variant of the aircraft.

First we create an interface

```
public interface IAircraftPrototype {

    void fly();

    IAircraftPrototype clone();

    void setEngine(F16Engine f16Engine);
}
```

The F-16 class would implement the interface like so:

```
public class F16 implements IAircraftPrototype {

    // default engine
    F16Engine f16Engine = new F16Engine();

    @Override
    public void fly() {
        System.out.println("F-16 flying...");
    }

    @Override
    public IAircraftPrototype clone() {
        // Deep clone self and return the product
        return new F16();
    }

    public void setEngine(F16Engine f16Engine) {
        this.f16Engine = f16Engine;
    }
}
```

And the client can exercise the pattern like so:

```
public class Client {

    public void main() {

        IAircraftPrototype prototype = new F16();
```

```
        // Create F16-A
        IAircraftPrototype f16A = prototype.clone();
        f16A.setEngine(new F16AEngine());

        // Create F16-B
        IAircraftPrototype f16B = prototype.clone();
        f16B.setEngine(new F16BEngine());
    }
}
```

Note that the interface `IAircraftPrototype` clone method returns an abstract type. The client doesn't know the concrete subclasses. The `Boeing747` class can just as well implement the same interface and be on its way to produce copies of prototypes. The client if passed in the prototype as `IAircraftPrototype` wouldn't know whether the clone's concrete subclass is an F16 or a Boeing747.

The prototype pattern helps eliminate subclassing as the behavior of prototype objects can be varied by composing them with subparts.

## Shallow vs Deep Copy

The prototype pattern requires that the prototype class or interface implements the `clone()` method. Cloning can be either ***shallow*** or ***deep***. Say our F-16 class has a member object of type `F16Engine`. In a shallow copy, the cloned object would point to the same F16Engine object as the prototype. The engine object would end up getting shared between the two. However, in a deep copy, the cloned object would get a copy of its own engine object as well as any of the nested objects within it. There will be no sharing of any fields, nested or otherwise between the prototype and the clone.

## Dynamic Loading

The prototype pattern also helps with dynamic loading of classes. Language frameworks which allow dynamic loading will create an

instance of the loaded class and register it in a managing entity. The application can at runtime request the object of the loaded class from the manager. Note, the application can't access the class's constructor statically.

## Other examples

- In Java the root `Object` class exposes a `clone` method. The class implements the interface `java.lang.Cloneable`.

## Caveats

- Implementing the `clone` method can be challenging because of circular references.

# Factory Method Pattern

This lesson discusses how derived classes can be given the responsibility of creating appropriate objects.

## What is it ?

A factory produces goods, and a software factory produces objects. Usually, object creation in Java takes place like so:

```
SomeClass someClassObject = new SomeClass();
```

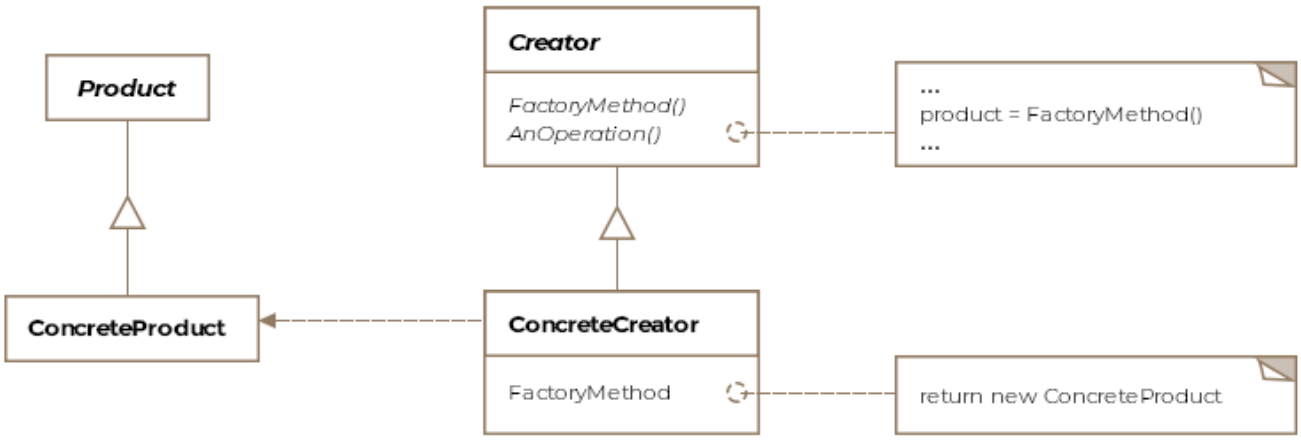
The problem with the above approach is that the code using the `SomeClass`'s object, suddenly now becomes dependent on the concrete implementation of `SomeClass`. There's nothing wrong with using `new` to create objects but it comes with the baggage of tightly coupling our code to the concrete implementation class, which is a violation of ***code to an interface and not to an implementation***.

Formally, the factory method is defined as ***providing an interface for object creation but delegating the actual instantiation of objects to subclasses***.

## Class Diagram

The class diagram consists of the following entities

- **Product**
- **Concrete Product**
- **Creator**
- **Concrete Creator**




Class Diagram

## Example

Continuing with our aircraft example scenario, let's assume we are trying to model the F-16 fighter jet. The client code needs to construct the engine object for the fighter jet and fly it. The naive implementation for the class would be something like below:

```
public class F16 {

    F16Engine engine;
    F16Cockpit cockpit;

    protected void makeF16() {
        engine = new F16Engine();
        cockpit = new F16Cockpit();
    }

    public void fly() {
        makeF16();
        System.out.println("F16 with bad design flying");
    }
}

public class Client {

    public void main() {
```

```
        // We instantiate from a concrete class, thus tying
        // ourselves to it

        F16 f16 = new F16();
        f16.fly();
    }
}
```

In the above code, we have committed ourselves to using a concrete implementation of the `F16` class. What if the company comes up with newer versions of the aircraft and we are required to represent them in the program? That would make us change the client code where we *new-up* the F16 instance. One way out, is to encapsulate the object creation in another object that is solely responsible for new-ing up the requested variants of the F-16. For starters, let's say we want to represent the A and B variants of F16, then the code would look like:

```
public class F16SimpleFactory {

    public F16 makeF16(String variant) {

        switch (variant) {
        case "A":
            return new F16A();
        case "B":
            return new F16B();
        default:
            return new F16();
        }
    }
}
```

The above is an example of a **Simple Factory** and isn't really a pattern but a common programming idiom. You could also mark the `make` method static to skip the factory object creation step. However, since static methods can't be overridden in subclasses because they are unique to a class, we won't be able to subclass the **Static Factory**. Remember simple and static factories aren't the same as the factory method pattern.

However, if we want to keep the creation of the F16 object parts within the same class and still be able to introduce new F16 variants as they come along, we could subclass F16 and delegate the creation of the right

F16 variant object to the subclass handling that variant. This is exactly the factory method pattern! The ***method*** here is the `makeF16()` which we'll make behave like a factory that produces the appropriate F16 variants. Proceeding forward we introduce two subclasses like so

```
public class F16 {

    IEngine engine;
    ICockpit cockpit;

    protected F16 makeF16() {
        engine = new F16Engine();
        cockpit = new F16Cockpit();
        return this;
    }

    public void taxi() {
        System.out.println("F16 is taxing on the runway !");
    }

    public void fly() {
        // Note here carefully, the superclass F16 doesn't know
        // what type of F-16 variant it was returned.
        F16 f16 = makeF16();
        f16.taxi();
        System.out.println("F16 is in the air !");
    }
}

public class F16A extends F16 {

    @Override
    public F16 makeF16() {
        super.makeF16();
        engine = new F16AEngine();
        return this;
    }
}

public class F16B extends F16 {

    @Override
    public F16 makeF16() {
        super.makeF16();
        engine = new F16BEngine();
```

```
        return this;
    }
}
```

We used inheritance to subclass and specialize the engine object. **A factory method may or may not provide a default or generic implementation but lets subclasses specialize or modify the product by overriding the create/make methods. In our example the variant models only have a different engine but the same cockpit.** The client code can now use the newer models like so:

```
public class Client {
    public void main() {
        Collection<F16> myAirForce = new ArrayList<F16>();
        F16 f16A = new F16A();
        F16 f16B = new F16B();
        myAirForce.add(f16A);
        myAirForce.add(f16B);

        for (F16 f16 : myAirForce) {
            f16.fly();
        }
    }
}
```

Note that the factory method pattern, returns an abstract type, be it a Java interface or a Java abstract class. The superclass, in our case F16 doesn't know what variant of the F16 it was returned from the `makeF16()` method. The general setup is that the superclass has the implementation for all the methods other than the creation methods. A create method is either abstract or comes with a default implementation and in turn is invoked by the other methods of the superclass. The creation of the right objects is the responsibility of the subclasses.

## Differences with Simple/Static Factory

The factory method pattern might seem very similar to the simple or static factory, however, the primary difference is that simple factories can't produce varying products through inheritance as a factory method

pattern can.

## Other Examples

- The factory method pattern pervades toolkits and frameworks. The pattern can be used whenever a class doesn't know ahead of time what subclass objects it would need to instantiate. This is common problem in designing frameworks, where the consumers of the framework are expected to extend framework provided abstract classes and hook-in functionality or object creations.
- The Java API exposes several factory methods:
    - `java.util.Calendar.getInstance()`
    - `java.util.ResourceBundle.getBundle()`
    - `java.text.NumberFormat.getInstance()`

## Caveats

- The pattern can result in too many subclasse with very minor differences.
- If the subclass extends the functionality, then the superclass can't use it unless it downcasts it to the concrete type. The downcast may fail at runtime.

# Abstract Factory Pattern

This lesson details the working of yet another creational design pattern, that allows us to create a family of products in a flexible and extensible manner.

## What is it ?

In the previous lesson, we learned the factory method pattern. We saw how we were able to model the variants of the F-16 using the factory method. But there are numerous airplanes other than F-16 that we'll need to represent. Say the client buys a Boeing-747 for the CEO to travel and now wants your software to provide support for this new type of aircraft.

The abstract factory pattern solves the problem of creating *families of related products*. For instance, F-16 needs an engine, a cockpit, and wings. The Boeing-747 would require the same set of parts but they would be specific to Boeing. Any airplane would require these three *related* parts but the parts will be plane and vendor specific. Can you see a pattern emerge here? We need a framework for creating the related parts for each airplane, a family of parts for the F-16, a family of parts for the Boeing-747 so on and so forth.

Formally, the abstract factory pattern is defined as ***defining an interface to create families of related or dependent objects without specifying their concrete classes***.

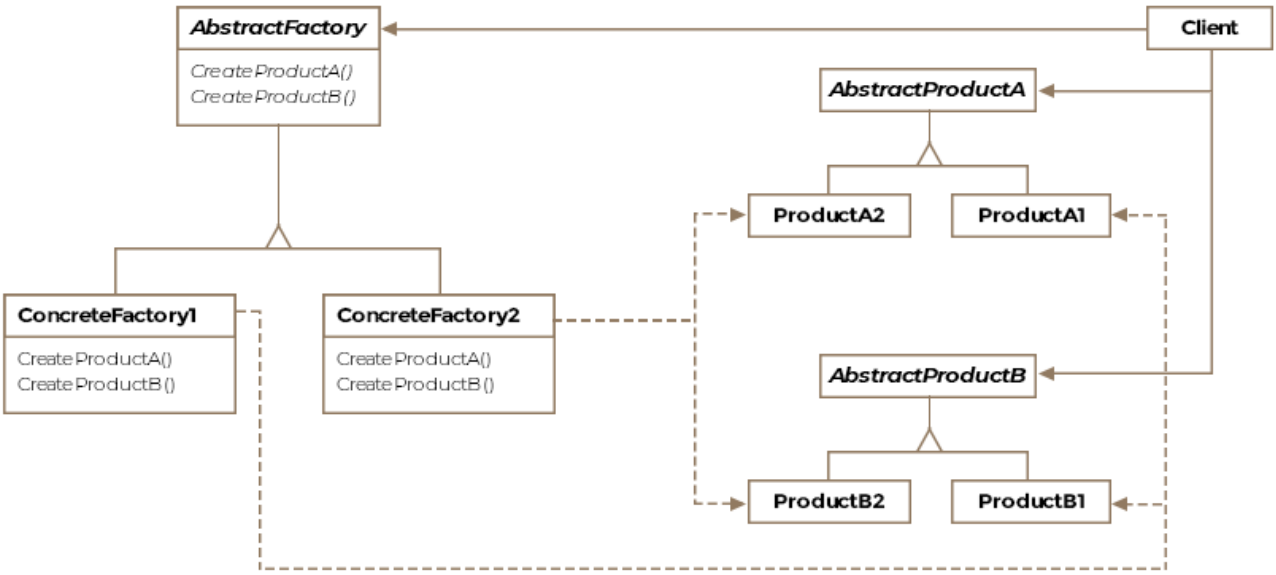
## Class Diagram

The class diagram consists of the following entities

- **Abstract Factory**

- **Concrete Factory**

- **Abstract Product**
- **Concrete Product**
- **Client**




Class Diagram

## Example

An abstract factory can be thought of as a super factor or a factory of factories. The pattern achieves the creation of a family of products without revealing concrete classes to the client. Let's consider an example. Say, you are creating a simulation software for the aviation industry and need to represent different aircraft as objects. But before you represent an aircraft, you also need to represent the different pieces of an aircraft as objects. For now let's pick three: the cockpit, the wings, and the engine. Now say the first aircraft you want to represent is the mighty F-16. You'll probably write three classes one for each piece specific to the F-16. In your code you'll likely consume the just created three classes as follows:

```
    public void main() {
```

```
        F16Cockpit f16Cockpit = new F16Cockpit();
        F16Engine f16Engine = new F16Engine();
        F16Wings f16Wings = new F16Wings();

        List<F16Engine> engines = new ArrayList<>();
        engines.add(f16Engine);
        for (F16Engine engine : engines) {
            engine.start();
        }
    }
```

This innocuous looking snippet can cause severe headaches down the road if your simulation software takes off and you need to expand it to other aircraft. Below is a list of what is wrong with the above snippet:

- The concrete classes for the three parts have been directly exposed to the consumer.
- F-16 has several variants with different engines and say in future if you want to return engine object matching to the variant, you'll need to subclass `F16Engine` class and that would necessitate a change in the consumer snippet too.
- The List in the code snippet is parametrized with a concrete class, in future if you add another aircraft engine object then the new engine can't be added to the list, even though engines for all aircraft are somewhat similar.

We'll fix these issues one by one and see how the abstract factory pattern would emerge.

## Code to an interface not to an implementation

One of the fundamental principles of good object orientated design is to hide the concrete classes and expose interfaces to clients. An object responds to a set of requests, these requests can be captured by an interface which the object's class implements. The client should know

what requests an object responds to rather than the implementation.

In our example, we can create an interface `IEngine`, which exposes the method `start()`. The F16Engine class would then change like so:

```
public interface IEngine {

    void start();
}

public class F16Engine implements IEngine {

    @Override
    public void start() {
        System.out.println("F16 engine on");
    }
}
```

With the above change see how the corresponding consumer code changes

```
    public void main() {
        IEngine f16Engine = new F16Engine();
        List<IEngine> engines = new ArrayList<>();
        engines.add(f16Engine);
        for (IEngine engine : engines) {
            engine.start();
        }
    }
```

Suddenly the consumer code is free of the implementation details of what class implements the F-16 engine and works with an interface. However, we would still like to hide the `new F16Engine()` part of the code. We don't want the consumer to know what class we are instantiating. This is discussed next.

## Creating a factory

Instead of new-ing up objects in client code, we'll have a class responsible for manufacturing the requested objects and returning them to the client

We'll call this class `F16Factory` since it can create the various parts of the

F16 aircraft and deliver them to the requesting client. The class would take the following shape.

```
public class F16Factory {

    public IEngine createEngine() {
        return new F16Engine();
    }

    public IWings createWings() {
        return new F16Wings();
    }

    public ICockpit createCockpit() {
        return new F16Cockpit();
    }
}
```

Suppose we pass in the `F16Factory` object in the constructor to the client code and it would now create objects like so:

```
    public void main(F16Factory f16Factory) {
        IEngine f16Engine = f16Factory.createEngine();
        List<IEngine> engines = new ArrayList<>();
        engines.add(f16Engine);
        for (IEngine engine : engines) {
            engine.start();
        }
    }
```

Note how this setup allows us the freedom to change the concrete class representing the F16Engine as long as it commits to the `IEngine` interface. We can rename, enhance or modify our class without causing a breaking change in the client. Also note that by just differing the factory class passed into the client constructor, we are able to provide the client with the same parts for a completely new aircraft. This is discussed next.

## Factory of Factories

Wouldn't it be great if we could use the same client snippet for other aircraft such as Boeing747 or a Russian MiG-29? If we could have all the factories being passed into the client agree to implement the `createEngine()` method, then the client code will keep working for all kinds of aircraft factories. But all the factories must commit to a common interface whose methods they'll implement and this common interface will be the **abstract factory**.

## Implementation

Let's start with an interface that would define the methods the factories for different aircraft would need to implement. The client code is written against the abstract factory but composed at runtime with a concrete factory.

```
public interface IAircraftFactory {

    IEngine createEngine();

    IWings createWings();

    ICockpit createCockpit();
}
```

Note that we mean a Java abstract class or a Java interface when referring to "interface". In this instance, we could have used an abstract class if there were a default implementation for any of the products. The create methods don't return concrete products rather interfaces to decouple the factory consumers from the concrete implementation of parts.

The formal definition of the abstract factory pattern says abstract factory pattern defines an interface for creating families of related products without specifying the concrete classes. Here the `IAircraftFactory` is *that* interface in the formal definition and note how its create methods are not

returning concrete parts but rather interfaces that'll be implemented by the concrete parts' classes.

Next let's define our factories for the two aircraft.

```
public class F16Factory implements IAircraftFactory {

    @Override
    public IEngine createEngine() {
        return new F16Engine();
    }

    @Override
    public IWings createWings() {
        return new F16Wings();
    }

    @Override
    public ICockpit createCockpit() {
        return new F16Cockpit();
    }
}

public class Boeing747Factory implements IAircraftFactory {

    @Override
    public IEngine createEngine() {
        return new Boeing747Engine();
    }

    @Override
    public IWings createWings() {
        return new Boeing747Wings();
    }

    @Override
    public ICockpit createCockpit() {
        return new Boeing747Cockpit();
    }
}
```

The concrete factories will be responsible for creating F-16 or Boeing

specific engine, cockpit and wings. Each part has a corresponding product

interface that we don't list for brevity's sake. The interfaces representing the parts would be:

- `IEngine`
- `ICockpit`
- `IWings`

All the **create** methods are actually factory methods that have been overridden. Indeed, the factory method pattern is utilized when implementing the abstract factory pattern. For the sake of brevity, we have skipped listing the concrete classes for engine, wings, and cockpit.

In the previous lesson, we created a class for F-16 which included a public method `fly()`. This method internally invoked the `makeF16()` method and after the aircraft was manufactured, it invoked the `taxi()` method before printing a fly statement. In our scenario, all aircrafts are expected to follow the same pattern. They first get manufactured, then taxi on the runway and then fly away. We can thus create a class for an aircraft that does these three tasks. Note, how we aren't creating separate classes to represent the two aircraft i.e. the F-16 and Boeing-747 rather a single `Aircraft` class that can represent both.

```
// Incomplete skeleton of the class.
public class Aircraft {

    IEngine engine;
    ICockpit cockpit;
    IWings wings;

    protected Aircraft makeAircraft() {
        //TODO: provide implementation
    }

    private void taxi() {
```

```
        System.out.println("Taxing on runway");
    }

    public void fly() {
        Aircraft aircraft = makeAircraft();
        aircraft.taxi();
        System.out.println("Flying");
    }
}
```

For now we'll keep the `makeAircraft` method empty. Let's first see how a client will request F-16 and Boeing-747 objects.

```
public class Client {

    public void main() {

        // Instantiate a concrete factory for F-16
        F16Factory f16Factory = new F16Factory();

        // Instantiate a concrete factory for Boeing-747
        Boeing747Factory boeing747Factory = new Boeing747Factory();

        // Lets create a list of all our airplanes
        Collection<Aircraft> myPlanes = new ArrayList<>();

        // Create a new F-16 by passing in the f16 factory
        myPlanes.add(new Aircraft(f16Factory));

        // Create a new Boeing-747 by passing in the boeing factory
        myPlanes.add(new Aircraft(boeing747Factory));

        // Fly all your planes
        for (Aircraft aircraft : myPlanes) {
            aircraft.fly();
        }

    }
}
```

We'll need to add a constructor to our `Aircraft` class, which will store the passed-in factory object and create the aircraft parts using the factory. **Just by composing the aircraft object with a different factory we are able to get a different aircraft**. The complete version of the aircraft class

would look like below:

```
public class Aircraft {

    IEngine engine;
    ICockpit cockpit;
    IWings wings;
    IAircraftFactory factory;

    public Aircraft(IAircraftFactory factory) {
        this.factory = factory;
    }

    protected Aircraft makeAircraft() {
        engine = factory.createEngine();
        cockpit = factory.createCockpit();
        wings = factory.createWings();
        return this;
    }

    private void taxi() {
        System.out.println("Taxing on runway");
    }

    public void fly() {
        Aircraft aircraft = makeAircraft();
        aircraft.taxi();
        System.out.println("Flying");
    }
}
```

The client just needs to instantiate the right factory and pass it in. The consumer or client of the factory is the `Aircraft` class. We could have created an interface `IAircraft` to represent all the aircraft that the class `Aircraft` in turn would implement but for our limited example it's not necessary.

The resulting code is easily extensible and flexible.

To tie our current example with the example discussed in the factory method pattern lesson, we have the option of either subclassing the `F16Factory` further to create factories for the **A** and **B** variants of F-16. We could also parametrize the existing `F16Factory` factory to take in an enum specifying the variant model and accordingly return the right part in a switch statement.

## Other Examples

- The abstract factory is particularly useful for frameworks and toolkits that work on different operating systems. For instance, if your library provides fancy widgets for the UI, then you may need a family of products that work on MacOS and a similar family of products that work on Windows. Similarly, themes used in IDE can be another example. If your IDE supports light and dark themes then it may use the abstract factory pattern to create widgets that belong to the light or dark theme just by varying the concrete factory that creates the widgets.
- `javax.xml.parsers.DocumentBuilderFactory.newInstance()` will return you a factory.
- `javax.xml.transform.TransformerFactory.newInstance()` will return you a factory.

## Caveats

- It might appear to the naive reader that the factory method pattern and the abstract factory pattern are similar. The difference between

the two lies in their motivations. The factory method pattern is usually responsible for creating a single product whereas an abstract factory pattern creates entire families of related products. Furthermore, in the factory method pattern, we use inheritance to create more specialized products whereas, in an abstract factory pattern, we practice object composition by passing in factories that are consumed to create the desired products.

- In our aircraft example, we can add a new aircraft simply by creating a concrete factory for it. However, note that if a helicopter is added to the fleet and requires a part that an aircraft doesn't have, then we'll need to extend the `IAircraftFactory` interface with another create method for the part required only by the helicopter. This will cascade the change to existing factories that'll need to return null since the new component isn't part of the jets.
- Concrete factories can be best represented as a singleton object

# Adapter Pattern

Adapter pattern is similar to how an electrical adapter lets your laptop work both in the US or UK even though voltages are different.

## What is it ?

When two heads of states who don't speak a common language meet, usually a language interpreter sits between the two and translates the conversation, thus enabling communication. The Adapter pattern is similar in that it sits between two incompatible classes that otherwise can't work with eachother and lets them work together. Another example to consider is when one buys electronics from USA and tries to use them in India. The two countries have different power voltages being distributed to consumers and using an electronic appliance from one country in another requires a physical adapter which steps up or down the voltage appropriately. The concept of the software adapter pattern is similar.

Formally, the adapter pattern is defined as ***allowing incompatible classes to work together by converting the interface of one class into another expected by the clients***

.

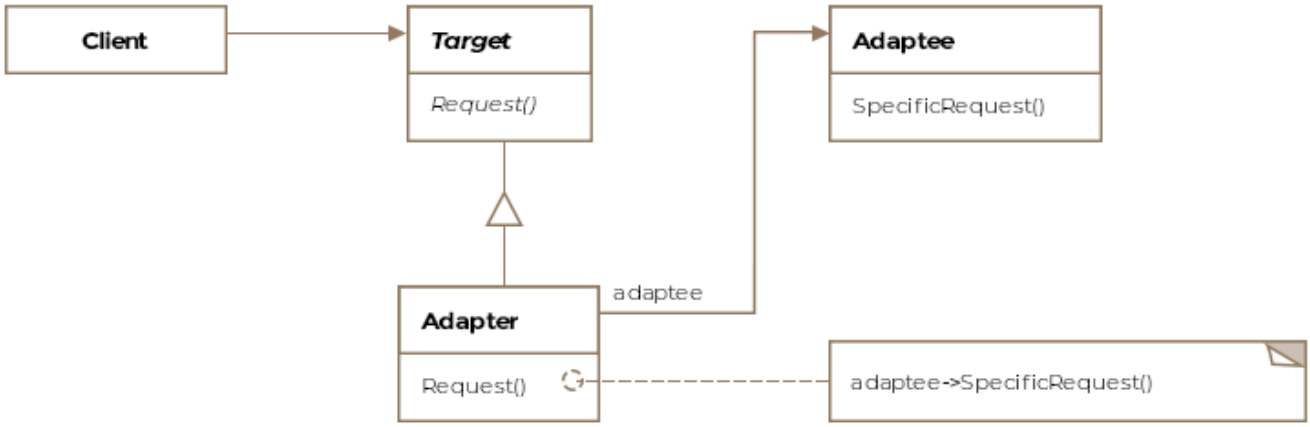
## Class Diagram

The class diagram consists of the following entities

- **Target**
- **Client**

- **Adaptee**

- **Adapter**




Class Diagram

## Example

Let's take our aircraft example again. Your software only deals with fancy jets but suddenly you are required to *adapt* your software to cater to a local hot air balloon company. Rewriting your software from scratch is not feasible. To complicate matters the balloon company already provides you with classes that represent hot air balloons which are incompatible with your `IAircraft` interface, which you use to represent modern aircraft. We'll use the adapter pattern to make the hot air balloon classes work with our existing infrastructure for aircraft. Let's see what the balloon class looks like:

```
public class HotAirBalloon {

    String gasUsed = "Helium";

    void fly(String gasUsed) {
        // Take-off sequence based on the kind of feul
        // Followed by more code.
    }

    // Function returns the gas used by the balloon for flight
```

```
    String inflateWithGas() {
        return gasUsed;

    }
}
```

Unfortunately, the `fly` method for the `HotAirBalloon` class is parametrized and can't work with the `IAircraft` interface. We'll need an adapter here that can make the `HotAirBalloon` class work with the `IAircraft` interface. The **adapter** in pattern-speak should implement the client interface, which is the `IAircraft` interface.

```
public interface IAircraft {
    void fly();
}
```

The adapter implementation would be the following:

```
public class Adapter implements IAircraft {

    HotAirBalloon hotAirBalloon;

    public Adapter(HotAirBalloon hotAirBalloon) {
        this.hotAirBalloon = hotAirBalloon;
    }

    @Override
    public void fly() {
        String feulUsed = hotAirBalloon.inflateWithGas();
        hotAirBalloon.fly(feulUsed);
    }
}
```

The important things to note about the adapter are:

- The adapter is ***composed*** with the **Adaptee** object, which in our case is the HotAirBalloon object.
- The adapter implements the interface the client knows about and consumes. In this case, it is the `IAircraft`.

Let's see the client code now

```
    public void main() {

        HotAirBalloon hotAirBalloon = new HotAirBalloon();
        Adapter hotAirBalloonAdapter = new Adapter(hotAirBalloon);

        hotAirBalloonAdapter.fly();
    }
```

Note the client is manipulating objects that implement the `IAircraft` interface. It doesn't know anything about the `HotAirBalloon` class and the adapter is responsible for masking the gory details for the client. The client can now make a hot air balloon fly even though it deviates from the `fly()` method enforced by the `IAircraft` interface.

## Object Adapter

The hot air balloon example that we just discussed is really an *object adapter* example. We *composed* the adapter with the adaptee object to make incompatible classes work together. In the case of Java, we can only practice object adaptation for reasons you'll learn shortly.

Using objects for adaptation gains us the usual benefits of object composition, The design becomes flexible and the adapter can stand in place of the adaptee or any of its subclassed-objects.

## Class Adapter

The complementary concept to object adapter is the *class adapter*. The class adapter works via multiple inheritance which isn't supported in Java. However, the idea is that the adapter extends both, the interface in use by the client, as well as, the adaptee class. Adaptation works via inheritance instead of composition.

One benefit of the adaptation via inheritance is that behavior can be overridden or new functionality can be added in the adapter.

### Other Examples

- If you have two applications, one spits out output as XML and the other takes in input as JSON then you'll need an adapter between the two to make them work seamlessly.
- `Enumeration` is a read-only interface from early days of Java which had only two methods `hasMoreElements` and `nextElement`. Later on, when Sun released Collections, it introduced the `Iterator` interface which also allows to remove elements. To support legacy code, we can create an adapter class to translate between the two interfaces and since enumeration is read-only, it can throw a runtime exception, when an item removal is requested.
- In the Java API, one can find `java.io.InputStreamReader` and `java.io.OutputStreamWriter` as examples of the adapter pattern

# Bridge Pattern

This lesson discusses how parallel class hierarchies or layers can be decoupled from one another using the bridge pattern.

## What is it ?

A physical bridge provides connectivity between two points. The bridge pattern describes how to pull apart two software layers fused together in a single class hierarchy and change them into parallel class hierarchies connected by a bridge.

The bridge pattern can be applied to scenarios where the class and what it does changes often. Think of it as two layers of abstraction. The class itself becomes one layer and what it does i.e. the implementation becomes another layer. This setup allows us to extend the two layers independently of each other. In Java, both the layers would be represented by two separate class hierarchies. The bridge sits between these two class hierarchies, allowing the class abstraction to configure itself with the implementation abstraction.

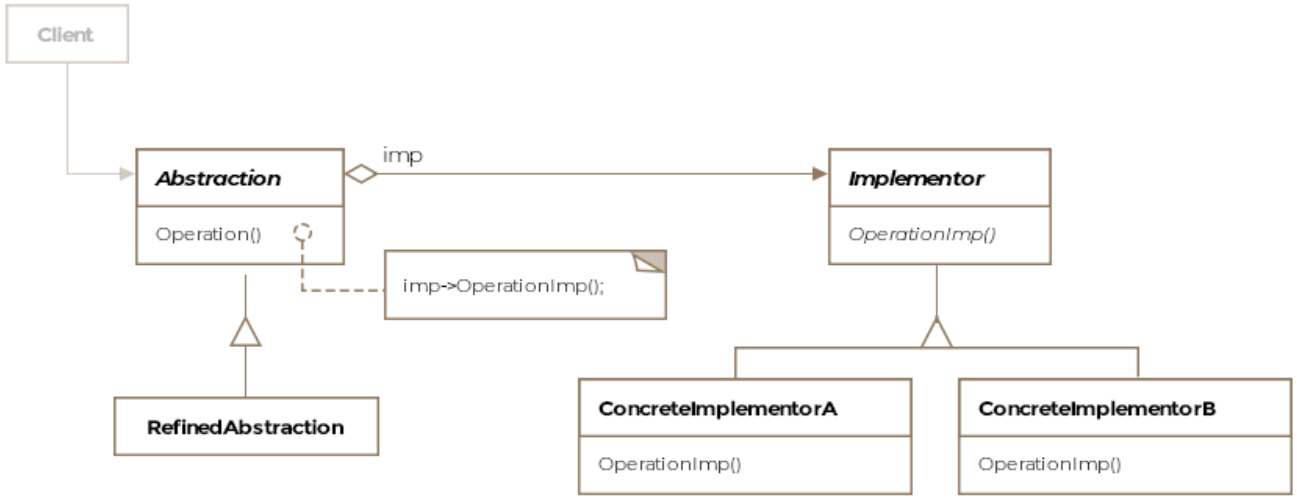
Formally, ***the bridge pattern lets you vary the abstraction independently of the implementation, thus decoupling the two in the process***. However, the *abstraction* and the *implementation* in the definition don't mean Java's abstract class/interface and concrete classes respectively.

## Class Diagram

The class diagram consists of the following entities

- **Abstraction**

- **Refined Abstraction**
- **Implementor**
- **Concrete Implementor**

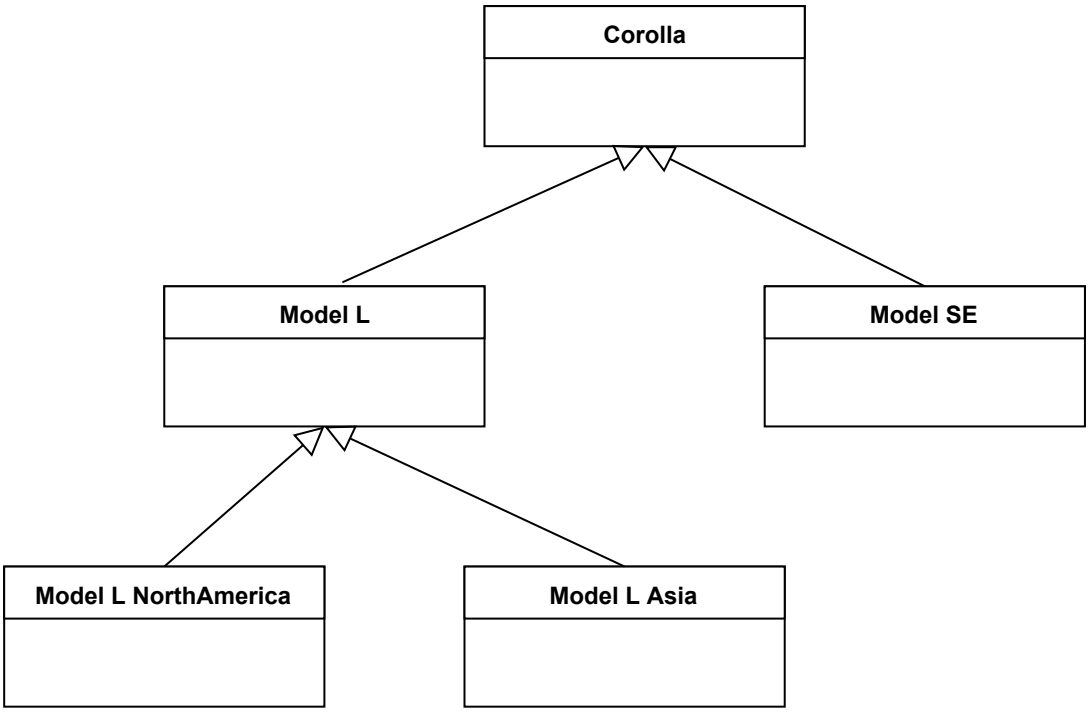



Class Diagram

## Example

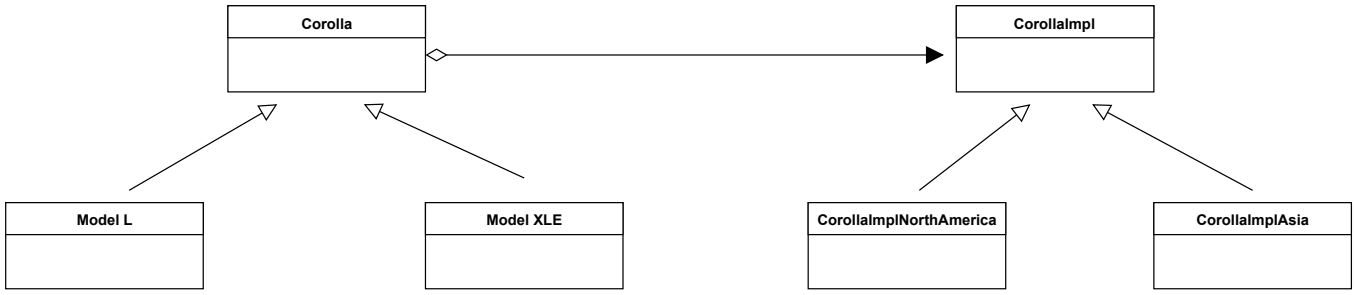
Suppose you are writing software for Toyota Motors and need to represent the most sold car in the world - the Toyota Corolla. We'll use an abstract class `Corolla` to represent the car. The concrete classes would represent each of the different models of the car. So far so good. However, the same model could be built to different standards for different locales. For instance, the North American model may have different safety requirements than an Asian model. The same model could be left or right handed depending on which country it is being shipped to. Corolla has several models and the models have different names in different countries. In the US, the different models include L, LE, XLE etc. For our purposes we'll consider only one model **L**.

Let's see how the class hierarchy would look like.




Without the Bridge Pattern

We can divide the above class structure into two hierarchies. One that just represents the models of the car and another that represents the location-specific variations for each model of the car. After applying the pattern the class diagram would look like below:




So you can see there's one hierarchy of class the `Corolla` which would have subclasses for each of the models, however, the actual car produced by the factory may be different from another car of the same model that is destined for a different country. The *implementation* of the car would make up a separate class hierarchy `CorollaImpl` which will have implementation subclasses for each of the Corolla models and have information such as safety equipment installed, whether the car is left or

right handed drive etc.

You would have guessed by now that the class `Corolla` (the abstraction) would hold a reference to an object of the class `CorollaImpl` (the implementation) and invoke method calls on the implementation object. We are using object composition to add location dependent behavior to each model of the car.

Let's examine the first class hierarchy that represents the car Corolla and acts as the abstraction.

```
public abstract class AbstractCorolla {

    protected AbstractCorollaImpl corollaImpl;

    public AbstractlCorolla(AbstractCorollaImpl corollaImpl) {
        this.corollaImpl = corollaImpl;
    }

    abstract void listSafetyEquipment();

    abstract boolean isCarRightHanded();
}
```

The `AbstractCorolla` holds a reference to the *implementation* class `AbstractCorollaImpl` object. The abstract class `AbstractCorollaImpl` forms a parallel class hierarchy and is defined below.

```
public abstract class AbstractCorollaImpl {

    abstract void listSafetyEquipment();

    abstract boolean isCarRightHanded();
}
```

The class that represents the model **L** is shown below:

```
public class Corolla_L extends AbstractCorolla {
```

```
    public Corolla_L(AbstractCorollaImpl corollaImpl) {
        super(corollaImpl);

    }

    @Override
    void listSafetyEquipment() {
        corollaImpl.listSafetyEquipment();

    }

    @Override
    boolean isCarRightHanded() {
        return corollaImpl.isCarRightHanded();
    }
}
```

We'll have corresponding *implementation* classes for the **L** model. They are given below:

```
public class Corolla_L_Impl_AsiaPacific extends AbstractCorollaImpl {

    @Override
    void listSafetyEquipment() {
        System.out.println("Not so safe.");
    }

    @Override
    boolean isCarRightHanded() {
        return false;
    }
}

public class Corolla_L_Impl_NorthAmerica extends AbstractCorollaImpl
{

    @Override
    void listSafetyEquipment() {
        System.out.println("High safety standards.");
    }

    @Override
    boolean isCarRightHanded() {
        return true;
    }
}
```

The client can use the classes like so:

```
public class Client {

    public void main() {

        AbstractCorolla myCorolla = new Corolla_L(new Corolla_L_Impl_
AsiaPacific());
        System.out.println(myCorolla.isCarRightHanded());

        myCorolla.setCorollaImpl(new Corolla_L_Impl_NorthAmerica());
        System.out.println(myCorolla.isCarRightHanded());
    }
}
```

Note how the client can switch out the implementation class object at runtime and make the model behave for an entirely different location. Using the bridge pattern we have avoided permanent binding of models and their intended locations of operation. The client can continue to work with the objects of the *abstraction layer* without noticing any changes to *implementation layer* classes. New safety rules or regulations would only affect the implementation layer classes.

## Other Examples

- Consider the development of a GUI toolkit. The toolkit will likely use system level api calls specific to the operating system. If you design a widget menu then you'll need to subclass the menu for both Windows and Linux. If you have several dozen widgets then each widget class will end up with subclasses for each operating system you intend to target. The code slowly becomes one giant monolith that is hard to change or debug.

  The bridge pattern suggest to create two class hierarchies. One captures the widget operations and another which encapsulates the system specific api details. Before the change we could have the following classes:

- `Menu` and its derived classes `MenuWindows` and `MenuLinux`

After applying the bridge pattern, the resulting classes would be

- `Menu`
- `AbstractMenuImpl` and its derived classes `MenuImplWindows` and `MenuImplLinux`

When a `Menu` object is instantiated we can compose it with either of the implementation classes to target different operating systems.

## Caveats

- The bridge pattern may be confused with the adapter pattern but one difference between the two is that the adapter pattern is usually applied after a system is designed whereas the bridge pattern is intentionally applied as part of the design process to decouple the two layers.

# Composite Pattern

This lesson discusses the composite pattern that lets us treat individual elements and group of elements as one.

## What is it ?

Composite literally means ***made up of various elements or parts***. The pattern allows you to treat the whole and the individual parts as one. The closest analogy you can imagine is a tree. The tree is a *recursive* data-structure where each part itself is a sub-tree except for the leaf nodes. The root is the top-level composite and its children are either composites themselves or just leaf nodes. The leaf itself can be thought of as a tree with just a single node.

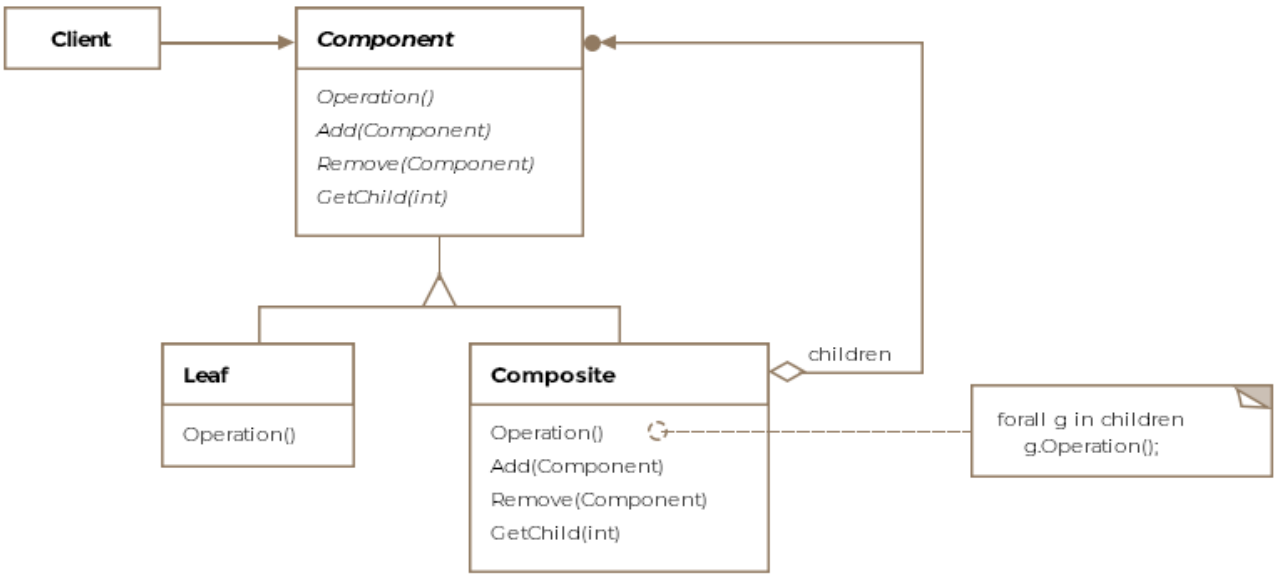
Formally, the composite pattern is defined as ***composing objects into tree structures to represent part-whole hierarchies, thus letting clients uniformly treat individual objects and composition of objects.***

The pattern allows the clients to ignore the differences between the whole and the part.

## Class Diagram

The class diagram consists of the following entities

- **Component**
- **Leaf**
- **Composite**
- **Client**



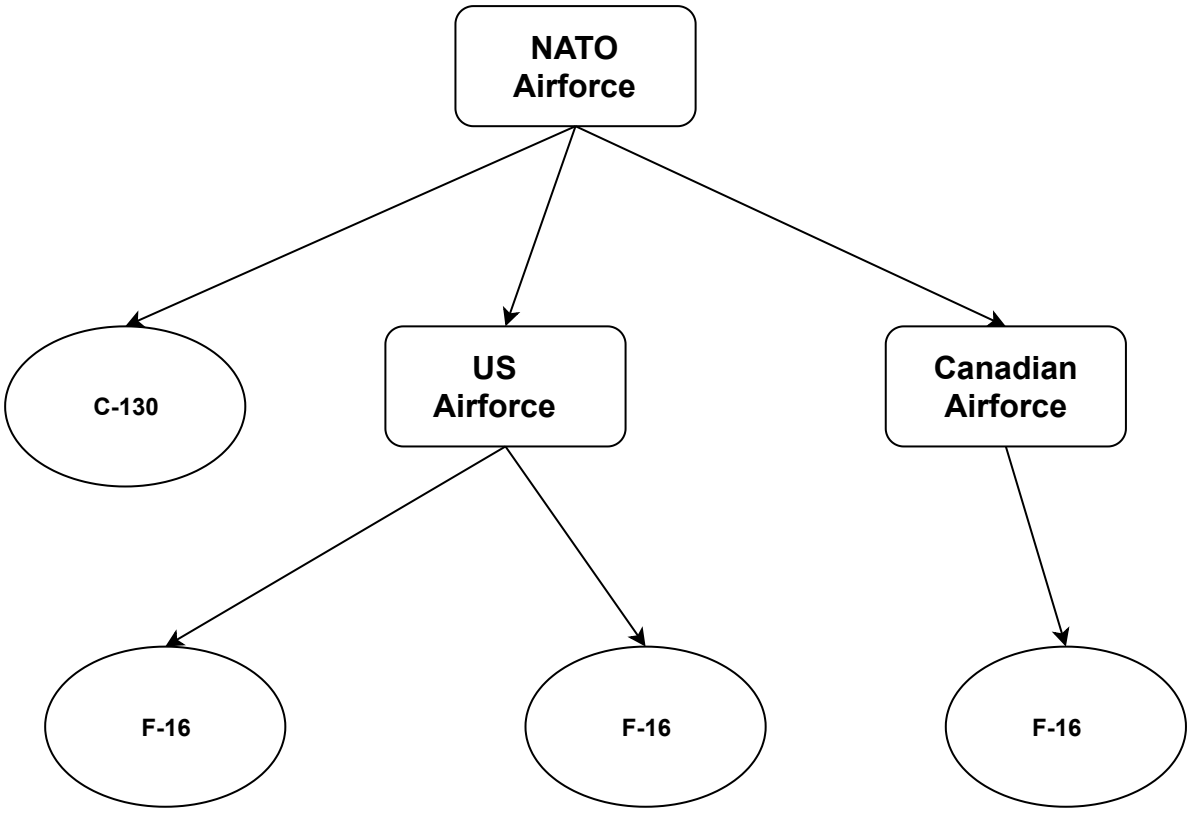

Class Diagram

## Example

Assume, that we now want to represent all the aircraft in the combined air forces of the NATO alliance. An air force is primarily made up of several aircraft but it can also have sub-air forces. For instance, the US has the 1st Air Force, 2nd Air Force so on and so forth. Our NATO alliance can consist of air forces from multiple countries including individual planes.

If we want to treat the composite and each part as the same, we would need both the *part* (the aircraft) and the *whole* (the airforce) to implement the same interface. In our scenario we'll create three classes:

- `Airforce`
- `F16`
- `C130Hercules`




Tree Depiction of Composite

The class `Airforce` will represent the **composite** and the other two classes the **part**. Furthermore, we'll create an interface `IAlliancePart` that will allow us to treat the objects from each of the three classes as one type.

Each aircraft requires some number of pilots to operate and maybe peripheral staff for maintainance. The classes would implement the funtionality to return the number of personnel required for the aircraft to operate. The `Airforce` class should return the total number of personnel required to operate all the aircraft composed of the airforce.

```
public interface IAlliancePart {

    // Any implementing class should return the
    // the number of personnel or staff required
    // to operate the aircraft or the airfoce
    int getPersonnel();
}
```

The classes implementing the above interface appear below:

```
public class F16 implements IAircraft, IAlliancePart {

    @Override
    public int getPersonnel() {
        // We need 2 pilots for F-16
        return 2;
    }
}

public class C130Hercules implements IAircraft, IAlliancePart {

    @Override
    public int getPersonnel() {
        // This cargo plane, needs 5 people
        return 5;
    }
}
```

The above two classes act as parts, now we'll write the composite class `Airforce`.

```
public class Airforce implements IAlliancePart {

    ArrayList<IAlliancePart> planes = new ArrayList<>();

    public void add(IAlliancePart alliancePart) {
        planes.add(alliancePart);
    }

    @Override
    public int getPersonnel() {

        // We iterator over the entire air force which can
        // contain leaf nodes (planes) or other composites
        // (air forces). This iteration is an example of an
        // internal iterator.
        Iterator<IAlliancePart> itr = planes.iterator();
        int staff = 0;

        while (itr.hasNext()) {
            staff += itr.next().getPersonnel();
        }
```

```
        return staff;

    }
}
```

Pay attention to the `getPersonnel` method for the `Airforce` class. It is an example of an ***internal iterator***. It is called internal because the `Airforce` assumes the responsibility of iterating over itself and its subparts. The iteration can also be extracted out into a separate class and would make an example of an ***external iterator***.

The internal iterator will recursively call the `getPersonnel` method on the nested air force objects. The leaves would actually be the planes and will return a number. The personnel count for the root air force object will be the sum of all the people required to operate all the planes.

The client can invoke the `getPersonnel` method on the root object and get a total count. Note how transparency is created by treating the composite and the part as same. The client code or the internal iterator code doesn't need conditional if-else statements to check for the type of the object and then call the appropriate method on it. The client code appears below:

```
public class Client {

    public void main() {

        Airforce NatoAllaiance = new Airforce();

        // The nested methods aren't listed for brevity's sake
        NatoAllaiance.add(createCanadaAirForce());
        NatoAllaiance.add(createUSAAirForce());

        F16 frenchF16 = new F16();
        C130Hercules germanCargo = new C130Hercules();

        NatoAllaiance.add(frenchF16);
        NatoAllaiance.add(germanCargo);

        System.out.println(NatoAllaiance.getPersonnel());
```

```
    }
}
```

Note ours is a simple example with a single method and we chose an `interface` instead of an `abstract class` to represent the whole and the part. If we opted for the latter, we could have provided default implementations for some methods.

The composite pattern allows a client to work seamlessly with a composite object. The client doesn't need to distinguish between the composite and the part. To make this happen, the composite, as well as, the part needs to implement a common interface or inherit from a common abstract class. This will let the client invoke common methods on both. However, it is possible that the common super-type has methods which make sense for the part and not for the composite or vice versa. Say our interface `IAlliancePart` could have a method `fire()` which would be applicable to the part, i.e. the plane but not to the composite, i.e. the air force. In such a scenario, it is ok to put in a default implementation or throw an `UnSupportedOperationException`.

## Other Examples

- In Java, the class `javax.faces.component.UIComponent` is an example of the composite.
- Imagine a UI Menu widget which can have sub-menus and menu items that perform some action when clicked on by the user. The menus would form the composite while the menu-items would form the part.

## Caveats

- references to parents: Since the composite is a tree structure, one may or may not need to store references to the parent.
- ordering of children in a composite: In some scenarios, it might be required to store or traverse the children in a certain order within the composite.
- cacheing part of the composite for traversal: For complex composites, it may make sense to cache part of the composite object to speed up traversal or search.

# Decorator Pattern

This lesson discusses how objects can be enhanced with new behavior using the decorator pattern.

## What is it ?

A decoration is added to something to make it more attractive, in the same spirit, the decorator pattern adds new functionality to objects without modifying their defining classes.

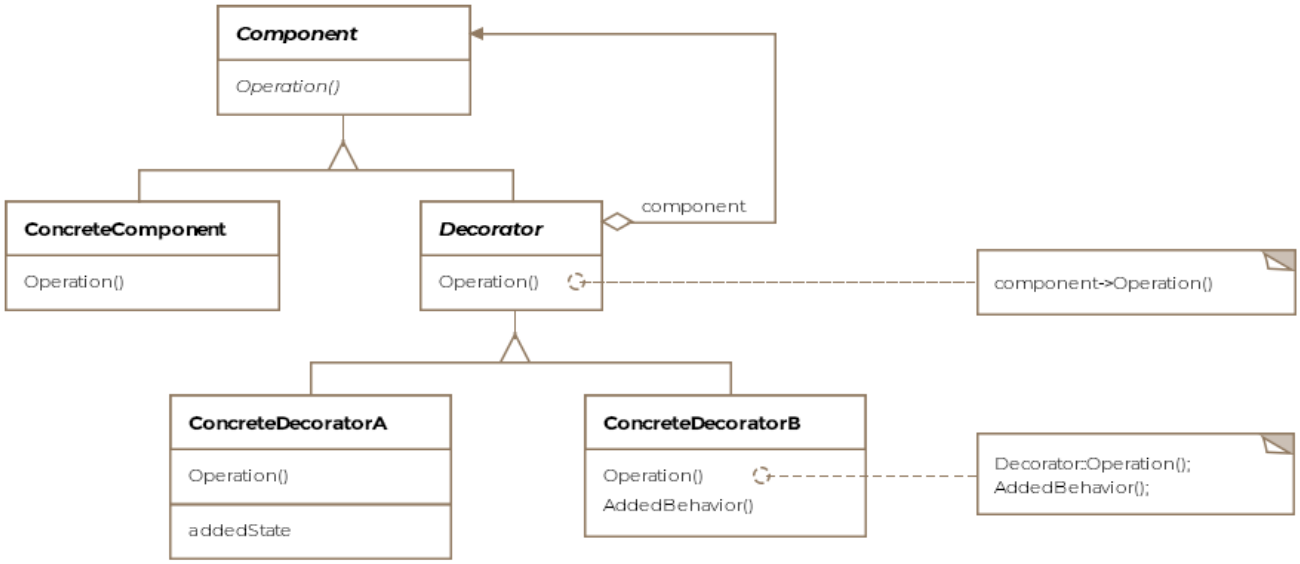
The decorator pattern can be thought of as a wrapper or more formally a way to ***enhance or extend the behavior of an object dynamically***. The pattern provides an alternative to subclassing when new functionality is desired.

The strategy is to wrap the existing object within a decorator object that usually implements the same interface as the wrapped object. This allows the decorator to invoke the methods on the wrapped object and then add any additional behavior. Usually, the decorator adds behavior to the existing functionality of the wrapped object i.e. the decorator takes action either before or after invoking some method on the wrapped object.

## Class Diagram

The class diagram consists of the following entities

- **Component**
- **Concrete Component**
- **Decorator**
- **Concerete Decorator**




Class Diagram

## Examples

To better understand the decorator pattern, let's go back to our aircraft example. The aircraft are produced somewhat similar to cars. There's the base model of a car and then there are optional packages that a customer can request to be added to the car. For instance a car can have a technology package added to the base model, then maybe the sports package so on and so forth. Similarly, our base Boeing-747 model can have two additional properties added to it. One is to let the plane be fitted with luxury fittings and the other is making the plane bullet-proof.

Say we are interested in the weight of our plane, which can be important in determining the fuel required for flights. Adding either or both of the options would make the plane heavier. We would want an extensible way of adding properties to the plane object and still be able to know its weight with the additional packages installed on the plane.

Let's first see how the aircraft interfaces and classes look like:

```
public interface IAircraft {

    float baseWeight = 100;

    void fly();
```

```
    void land();


    float getWeight();

}

public class Boeing747 implements IAircraft {

    @Override
    public void fly() {
        System.out.println("Boeing-747 flying ...");
    }

    @Override
    public void land() {
        System.out.println("Boeing-747 landing ...");
    }

    @Override
    public float getWeight() {
        return baseWeight;
    }
}
```

The decorator pattern requires an abstract decorator class that implements the abstract interface for the object being wrapped. In this case, we call our decorator `BoeingDecorator` and have it implement the `IAircraft` interface.

```
public abstract class BoeingDecorator implements IAircraft {

}
```

We'll have two concrete decorators, one for the luxury fittings and the other for bullet proofing the plane.

```
public class LuxuryFittings extends BoeingDecorator {

    IAircraft boeing;

    public LuxuryFittings(IAircraft boeing) {
```

```
        this.boeing = boeing;
    }

    @Override
    public void fly() {
        boeing.fly();
    }

    @Override
    public void land() {
        boeing.land();
    }

    @Override
    public float getWeight() {
        return (30.5f + boeing.getWeight());
    }
}

public class BulletProof extends BoeingDecorator {

    IAircraft boeing;

    public BulletProof(IAircraft boeing) {
        this.boeing = boeing;

    }

    @Override
    public void fly() {
        boeing.fly();
    }

    @Override
    public void land() {
        boeing.land();
    }

    @Override
    public float getWeight() {
        return 50f + boeing.getWeight();
    }
}
```

See how the concrete decorators save a reference to the object that they wrap. The `getWeight` method in each decorator calls the base model's

wrap. The `getWeight` method in each decorator calls the base model's `getWeight` to get the base model's weight and then adds the weight added to the plane because of itself. The client can use the decorators like so:

```
public class Client {

    public void main() {
        IAircraft simpleBoeing = new Boeing747();
        IAircraft luxuriousBoeing = new LuxuryFittings(simpleBoeing);
        IAircraft bulletProofBoeing = new BulletProof(luxuriousBoeing
);
        float netWeight = bulletProofBoeing.getWeight();
        System.out.println("Final weight of the plane: " + netWeight
);
    }
}
```

From the client code, one can observe how the plane's behavior is extended at runtime. Note that the decorator's abstract class implements the same interface as the Boeing747. This is so that the concrete decorator object can stand in place of the Boeing object. From the client code, one can see how we wrap the boeing object in successive decorators and are able to retrieve the net weight.

## Other Examples

- A prominent example of this pattern is the `java.io` package, which includes several decorators. Look at the snippet below:

```
    public void main() {
        // FileInputStream is responsible for reading the file
        FileInputStream fileInputStream = new FileInputStream("my
File.txt");
        // BufferedInputStream extends FilterInputStream and no
t FileInputStream, it is
        // a decorator which enhances the functionality of the or
iginal object by wrapping over it.
        BufferedInputStream bufferedInputStream = new BufferedInp
```

```
utStream(fileInputStream);
        // The read operation becomes buffered now

        bufferedInputStream.read();
    }
```

The `FileInputStream` is the object actually responsible for reading the text file. If we wanted to introduce buffered read functionality, one way of doing it would be to subclass `FileInputStream` and add the new functionality. The other is to use the decorator pattern which is how it is implemented in the Java framework. The `BufferedInputStream` wraps the `FileInputStream` to provide buffering capabilities.

## Caveats

- One of the issues with the decorator pattern is that we may end up with too many classes as the number of decorators grows. The `java.io` package suffers from the same issue, as it makes extensive use of the decorator pattern.
- Also, if we want to take a specific action based on the concrete type of the plane, we may not be able to do so. Once the concrete object is wrapped inside a decorator the reference to the object is through the abstract type and not the concrete type anymore.

# Facade Pattern

This lesson discusses how the interface to a complex system of interacting entities can be simplified by providing a front that hides the subsystem intricacies from the client.

## What is it ?

A facade literally means the front of a building or an outward appearance to hide a less pleasant reality. The facade pattern essentially does the same job as the definition of the word facade. Its purpose is to hide the complexity of an interface or a subsystem.

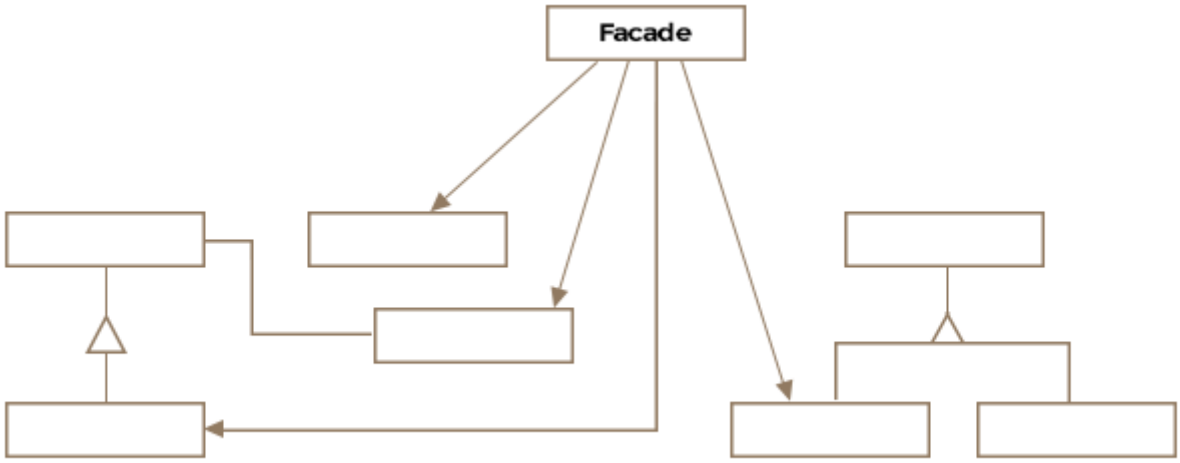
If you take a look around the amenities of current life, almost everything is a facade. When you press a button to turn on the room lights. The button is a facade that hides from you the complexities of electric power generation and distribution and magically lights up your room. The facade makes complex systems easier to use.

Formally the facade pattern is defined as ***a single uber interface to one or more subsystems or interfaces intending to make use of the subsystems easier***.

## Class Diagram

The class diagram consists of the following entities

- **Facade**
- **Subsystem Classes**




Class Diagram

## Example

Modern aircrafts have a feature called autopilot that allows the airplane to fly to its destination in an automated fashion without much interference from human pilots. The autopilot feature needs to juggle with all the subsystems and health-checks of the aircraft to ensure a smooth flight. It can hide away all the underlying complexity of automated flight from a pilot. Let's look at how the Boeing autopilot facade would be created:

```
public class AutopilotFacade {

    private BoeingAltitudeMonitor altitudeMonitor;
    private BoeingEngineController engineController;
    private BoeingFuelMonitor feulMonitor;
    private BoeingNavigationSystem navigationSystem;

    public AutopilotFacade(BoeingAltitudeMonitor altitudeMonitor, Boe
ingEngineController engineController,
                           BoeingFuelMonitor feulMonitor, BoeingNavig
ationSystem navigationSystem) {
        this.altitudeMonitor = altitudeMonitor;
        this.engineController = engineController;
        this.feulMonitor = feulMonitor;
        this.navigationSystem = navigationSystem;
    }
```

```
    public void autopilotOn() {
        altitudeMonitor.autoMonitor();

        engineController.setEngineSpeed(700);
        navigationSystem.setDirectionBasedOnSpeedAndFeul(
                engineController.getEngineSpeed(),
                feulMonitor.getRemainingFeulInGallons());
    }

    public void autopilotOff() {
        altitudeMonitor.turnOff();
        engineController.turnOff();
        navigationSystem.turnOff();
        feulMonitor.turnOff();
    }
}
```

The facade is encapsulating the logic required to activate and deactivate the autopilot in the `autopilotOn` and `autopilotOff` methods. It is also being passed in all the subsystems required in its constructor. We have intentionally left out the implementation for the subsystems `BoeingAltitudeMonitor`, `BoeingEngineController`, `BoeingFuelMonitor` and `BoeingNavigationSystem` for brevity. The key point to understand here is that these subsystems expose certain operations that are hidden behind an interface that lumps their execution in the right sequence.

You may argue that the class `Boeing747` could have just as well invoked the required methods on the subsystem objects and you are right. The intent is not to hide the subsystems but to make it easier to use the collection of subsystems. The `Boeing747` class only works with the facade. It codes against the facade rather than individual subsystems so that tomorrow if any subsystem is switched out for a better one the change is quarantined to the facade and doesn't cascade across the code base.

The facade pattern shields the client from having to deal with all the complex subsystem classes, thus creating a loose coupling between the subsystem and its clients. Upon receiving a request, the facade forwards the request to the appropriate subsystem and may do any necessary translation inbetween.

## Other Examples

- Class `javax.faces.context.FacesContext` internally uses other types like `LifyCycle` and `ViewHandler`, so that the end user doesn't deal with them directly.
- `javax.faces.context.ExternalContext` internally uses the `HttpSession`, `HttpServletRequest` and other classes. It acts as a facade for the consumers of the underlying classes.

## Caveats

- Usually a single facade object is needed and is implemented as a singleton.

# Flyweight

This lesson discusses how the flyweight pattern can be applied to reduce memory requirements.

## What is it ?

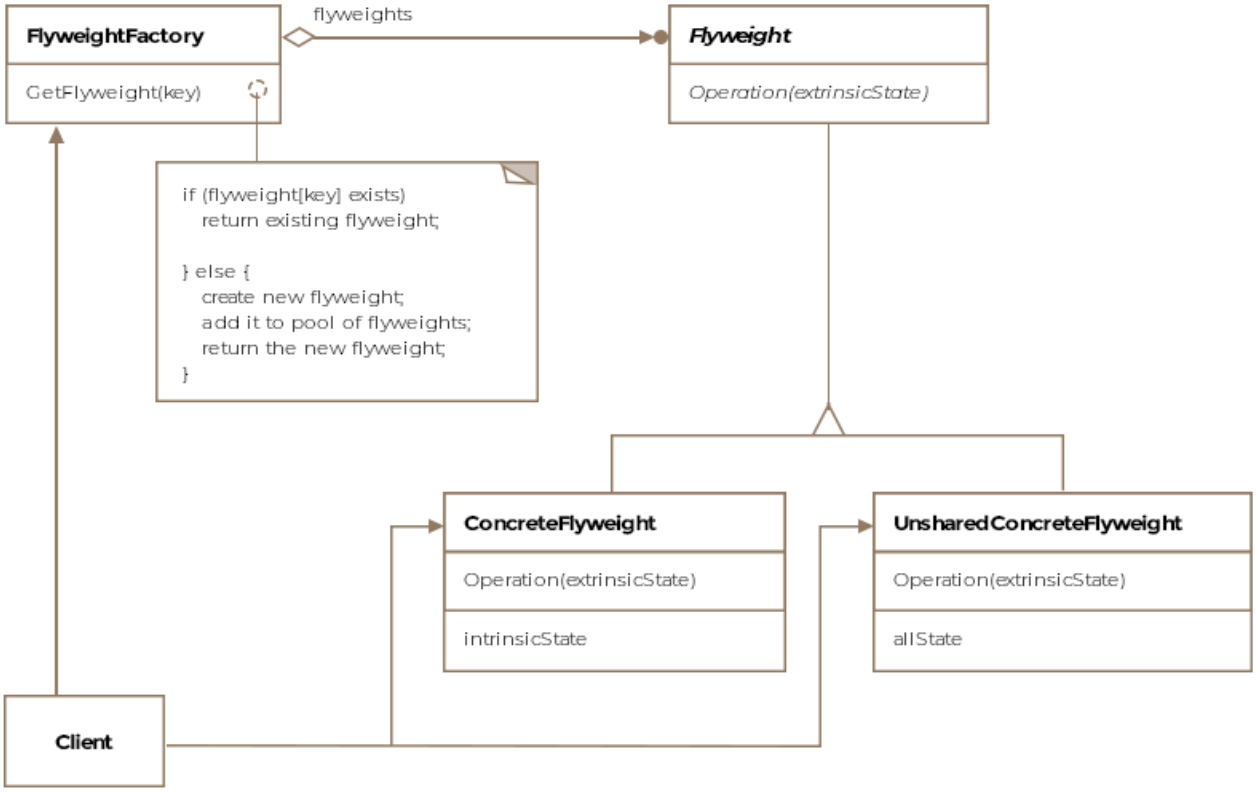
Flyweight is a category in boxing competitions for light weight boxers. The intent of the pattern is somewhat similar in that it tries to reduce bloated code to a more compact and lean representation, which uses less memory.

Formally, the pattern is defined as ***sharing state among a large number of fine-grained objects for efficiency***.

## Class Diagram

The class diagram consists of the following entities

- **Flyweight**
- **Concrete Flyweight**
- **Unshared Concrete Flyweight**
- **Flyweight Factory**
- **Client**




## Example

Following OO principles to the core may lead you to create too many objects in your application that have part of their state shared. For instance, continuing with our aircraft scenario, if you are designing a global radar that tracks all the planes currently airborne in the world at any time then your radar screen will show thousands of airplanes represented as objects in memory. If your hardware is limited in memory then you have a problem.

Each object would have some shared state that is independent of where the plane is flying in the world. This state which is independent of the *context* of the plane is called **intrinsic state** and can be factored out and shared amongst all similar planes. The state of the plane which changes with the context is called the **extrinsic state**. In this case, the coordinates of the plane will change for each plane and can be thought of as the extrinsic state. The remaining amount of fuel for each airplane is another piece of information that is extrinsic. However, the number of crew required to fly a particular variant of the F-16 would be the same across all the F-16s of that variant that are airborne. This would be an example

of **intrinsic state**. The crew number isn't dependent on the context, i.e. which part of the world is the plane flying in, which country does it belong to, is it on a patrolling mission or a combat mission etc - none of that affects or changes the number of people required to operate the aircraft.

Using the flyweight pattern, we can move the extrinsic state of the object outside of the class and only keep the intrinsic state within the class. This change would allow us to reuse the same F-16 object for all the F-16s that are currently airborne and show up on the radar. The number of objects required to represent the flying F-16s would drastically reduce. The extrinsic state of the planes can always be passed-in to the methods that use it.

The astute reader would immediately question where do we keep the *extrinsic state*? That can be kept in a separate ***context*** object. But then you may retort that it is like going back to square one, for now, we are creating more objects and the whole purpose of the pattern was to reduce the number of objects. Your observation is correct, however, note that the context object only contains that information which varies per instance of F-16. The information which is common across the F-16 instances is stored in a flyweight object. Earlier, we had a single *heavyweight* object that contained both kinds of information and was needed per instance of F-16. With the break-up, the single heavyweight object becomes a flyweight object used by all F-16s consisting of information that'll not change and an additional context object per F-16 instance consisting of information that'll change for each instance of the F-16. Since we are keeping a single copy of the non-varying information, we'll witness memory savings.

Let's see how the F-16 flyweight class would look like

```
public class F16 implements IAircraft {

    // Intrinsic state
    private final String name = "F16";
    private final int personnel = 2;
    private final String dimensions = "15m long 3m wide";
    private final String wingspan = "33 feet";
```

```
    // Extrinsic state includes the current position and current spee
d

    // of the aircraft that is being passed in for computing remainin
g
    // time to destination
    public double getTimeToDestination(int currX, int currY, int dest
X, int destY, int currSpeed) {

        // algorithm to calculate the remaining time to reach
        // destination.

        return 1;
    }
}
```

The client code can take advantage of the flyweight like so:

```
public class Client {

    public void main(int[][] coordsF16) {

        F16 flyweightF16 = new F16();

        for (int i = 0; i < coordsF16.length; i++) {
            int currX = coordsF16[i][0];
            int currY = coordsF16[i][1];

            // We are passing in the extrinsic state to the flyweigh
t object. Note we are storing the
            // extrinsic state of the airborne f16s in a 2-dimensiona
l array.
            System.out.println("time to destination = " +
                                flyweightF16.getTimeToDestination(curr
X, currY, 10, 10, 200));
        }

    }
}
```

Note, how the client is receiving the extrinsic state for each of the F-16 in a two-dimensional array. The flyweight F16 class has information specific to a F-16 plane that won't change. For brevity's sake the getters for the private fields are skipped.

## Other Examples

- GoF discusses a text editor example. In the extreme case, each character can be represented as an object. However, any reasonably sized document would then become bloated with character objects. Each object would contain the font, style, color and the character encoding. For simplicity, if the document is limited to ASCII characters then we can have flyweight objects represent each character in the ASCII table.
- Methods `java.lang.Boolean.valueOf` and `java.lang.Integer.valueOf` both return flyweight objects.

## Caveats

- Usually, we don't want the clients to creat the flyweight objects directly. A flyweight factory is used to manage the flywieght objects.
- It might appear that flyweight pattern is maybe similar to the singleton pattern, however there are some important differences. Flyweights are immutable whereas a singleton can undergo changes. Also, a singleton can only have a single copy whereas flyweights can have more than one object of their type.
- *State* and *Strategy* objects are often implemented as flyweights.
- In practice, composite pattern can be combined with flyweight to create a hierarchical structure, where the leaves are implemented as flyweight objects and are shared.
- Since flyweight objects are shared, identity tests for conceptually different objects would return true.

- Memory savings increase if the extrinsic state can be computed rather than being stored. However, the calculation or lookup of the extrinsic state trades execution time increase in lieu of memory savings.

# Proxy Pattern

This lesson discusses how objects can act on behalf of other objects without the clients knowing they aren't talking to the intended object.

## What is it ?

The literal definition of proxy is ***the authority to represent someone else***. In a proxy pattern setup, a proxy is responsible for representing another object called the **subject** in front of clients. The real subject is shielded from interacting directly with the clients. There could be several reasons why one would want to front a subject with a proxy, some are listed below:

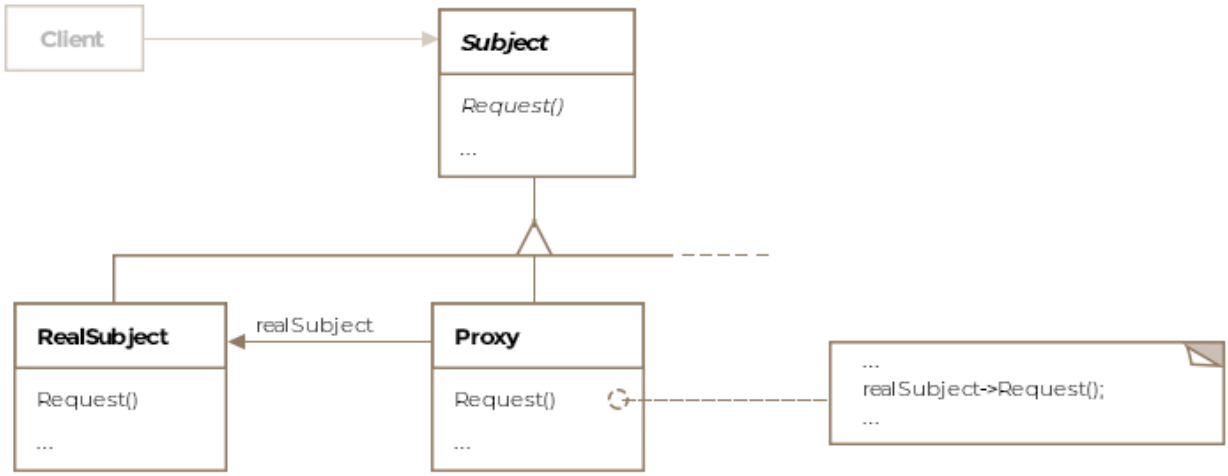
- To access remote objects over the internet, running in another JVM or another address space
- To protect a subject from clients not authorized to access it
- To stand in place of an object that may be expensive to create and delay the object's creation till it is accessed
- To cache queries or results from subject for clients
- There are a number of other use cases such as the firewall proxy, synchronization proxy etc.

Formally, the pattern is defined as ***a mechanism to provide a surrogate or placeholder for another object to control access to it***.

## Class Diagram

The class diagram consists of the following entities

- **Proxy**
- **Subject**
- **Real Subject**




Class Diagram

## Remote Proxy

An ambassador appointed by a country to another acts like a proxy for his/her country. He or she acts as the communication channel between the host country and the ambassador's country. A remote proxy acts in a somewhat similar fashion and facilitates interaction between the client and the subject. A remote proxy stands in place of an object which isn't running on the same machine or JVM. The client doesn't know that it is not talking to the subject but a proxy. The proxy and the subject implement the same interface allowing the client to invoke the same methods on the proxy as it would have invoked on the subject. The proxy then sends the request for the method invocation along with the method arguments over the network to the remote subject.

The subject doesn't have the intelligence built into it to receive the method invocation request over the network and usually, a helper entity runs alongside the remote subject and handles network communication

on behalf of the subject. Once the method invocation request is received,

the helper entity forwards it to the subject, which in turn executes the method with the passed-in arguments. The results are then ferried back over the network to the proxy. The proxy in turn returns the results to the client.

## Virtual Proxy

A virtual proxy creates objects on demand. Sometimes creation of an object is expensive and complex and should only be created when required. A virtual proxy delays creation of its subject until required and stands in its place before and during its construction.

Think of instagram running on a very weak internet connection on a mobile phone. The pictures aren't loaded quickly enough but to ensure a pleasant user experience, a *loading widget* can appear on the mobile screen while the image gets downloaded in the background. The *loading widget* would act as a proxy for the picture and might have related information about the picture such as the dimensions of the picture, which it can provide to the display frame before the picture download is complete.

## Protection Proxy

A protection proxy or authorization proxy controls access to the real subject. Requests are vetted for authorization before being forwarded to the real subject. These proxies are useful when clients should have different access rights to the subject.

## Example

We'll show a simple example of a remote proxy.

Consider a drone that is being flown by a pilot on the ground. The drone can be thought of as the subject, while the ground cockpit can consist of a proxy that receives actions from the physical controls in the cockpit and forwards them to the remote drone's software.

The `DroneProxy` and the actual subject `Drone` will both implement the same interface `IDrone`. The client will talk with the drone proxy without knowing that it is not talking to the real subject. The drone proxy would in turn forward requests from the client to the real drone object running in the flying drone's computer memory over a wireless connection. The action taken on the ground by the pilot against the drone proxy will be mimicked by the flying drone.

The simplistic drone interface can be:

```
public interface IDrone {

    void turnLeft();

    void turnRight();

    void firstMissile();
}
```

The code for the `DroneProxy` appears below:

```
public class DroneProxy implements IDrone {

    @Override
    public void turnLeft() {
        // forward request to the real drone to
        // turn left over the internet
    }

    @Override
    public void turnRight() {
        // forward request to the real drone to
        // turn right over the internet
    }

    @Override
```

```
    public void firstMissile() {
        // forward request to the real drone to
        // fire missile
    }
}
```

The client code will be like so:

```
public class Client {

    public void main(DroneProxy droneProxy) {

        // perpetual loop that received pilot actions
        while (true) {

            Scanner scanner = new Scanner(System.in);
            String action = scanner.nextLine();

            switch (action) {
            case "left": {
                droneProxy.turnLeft();
                break;
            }

            case "right": {
                droneProxy.turnRight();
                break;
            }

            case "fire": {
                droneProxy.firstMissile();
                break;
            }

            default:
                System.out.println("Invalid Action");
            }
        }
    }
}
```

The object of class `Drone` which will be part of the code that runs on the hardware of the flying drone and controls its flight may look like below:

```
public class Drone implements IDrone {
```

```
public class Drone implements IDrone {

    @Override
    public void turnLeft() {
        // receives the request and any method parameters
        // over the internet to turn the drone to its left.
    }

    @Override
    public void turnRight() {
        // receives the request and any method parameters
        // over the internet to turn the drone to its right.
    }

    @Override
    public void firstMissile() {
        // receives the request and any method parameters
        // over the internet to fire a missile
    }
}
```

Note we have skipped the technological details for transmitting the request and the method parameters over the wire as they may vary from language to language and don't affect the spirit of the pattern.

## Other Examples

- `java.lang.reflect.Proxy` is an example of the proxy pattern.
- `java.rmi.*` package contains classes for creating proxies. RMI is Remote Method Invocation. It is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine. RMI uses *marshalling* to send method parameters over the wire and the subject *unmarshalls* them back into objects.

## Caveats

- A proxy may also be responsibile for additional house-keeping tasks such as deleting the subject, keeping a reference count to the real subject or encoding requests before sending them to the real subject over the wire.

# Chain of Responsibility Pattern

This lesson discusses how a request can travel down a chain of handlers till an appropriate handler is found.

## What is it ?

In a ***chain of responsibility*** pattern implementation, the sender's request is passed down a series of handler objects till one of those objects, handles the request or it remains unhandled and falls off the chain. Multiple objects are given a chance to handle the request. This allows us to decouple the sender and the receiver of a request.

The requestor has no knowledge of the object that will eventually handle its request nor does it have a reference to the handling object. Similarly, the object eventually handling the request isn't aware of the requestor.

Each object in the chain should implement a common supertype and have a reference to its successor. The handler objects can be added to or removed from the chain at runtime.

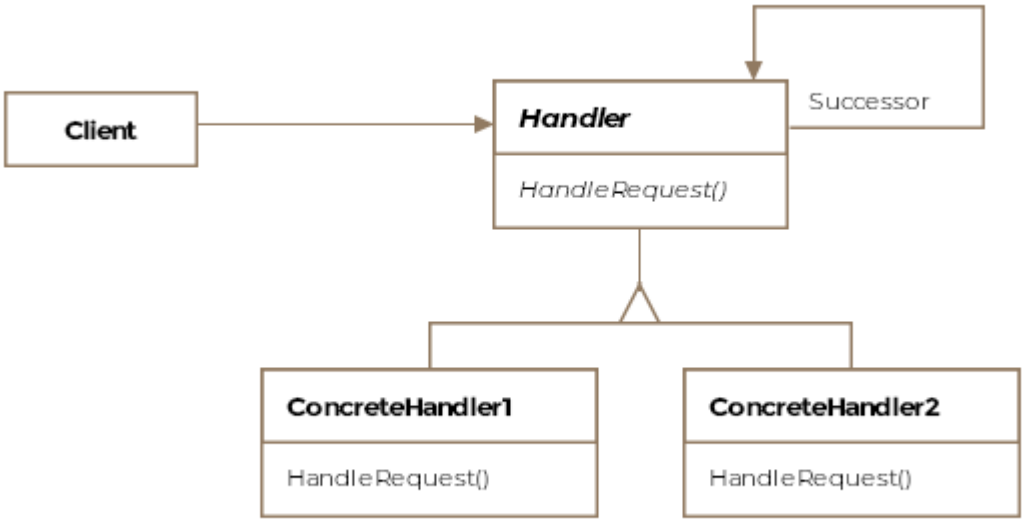
Formally, the pattern is defined as ***decoupling the sender of a request from its receiver by chaining the receiving objects together and passing the request along the chain until an object handles it***.

Usually the pattern is applied when the request can be handled by multiple objects and it is not known in advance which object will end up handling the request.

## Class Diagram

The class diagram consists of the following entities

- **Handler**
- **Concrete Handler**
- **Client**




Example

Imagine an aircraft's cockpit. It would be running some software that would indicate to the pilot about equipment failure, engine temperature, or something as disastrous as a fire. Let's say when some failure happens, the hardware sends an error code to the cockpit's computer which then takes some corrective action based on the error code it receives.

We can model the error codes as requests that require handling by appropriate components. Say, if an engine failure happens, there might be a series of corrective actions that can try to fix the problem successively. For instructional purposes, say the hardware can send out either a ***fire detected*** or a ***low on fuel*** request, which have corresponding handlers. Let's see the listing for `AbstractHandler` and `AbstractRequest` classes first.

```
abstract public class AbstractRequest {

    // Each request is identified by a an integer
    // FireRequest: 1
```

```
    // LowFuelRequest: 2
    private int requestCode;

    public AbstractRequest(int requestCode) {
        this.requestCode = requestCode;
    }

    public int getRequestCode() {
        return requestCode;
    }
}

abstract public class AbstractHandler {

    private AbstractHandler next;

    public AbstractHandler(AbstractHandler next) {
        this.next = next;
    }

    public void setNext(AbstractHandler next) {
        this.next = next;
    }

    public void handleRequest(AbstractRequest request) {
        if (next != null) {
            next.handleRequest(request);
        }
    }
}
```

The naive implementations of the classes `LowFuelRequest` and `FireDetectedRequest` appear below alongside the implementation for the concrete handler `FireHandler`. We skip the implementation for `LowFuelHandler` as it would be similar to `FireHandler`.

```
public class FireDetectedRequest extends AbstractRequest {

    // Fire request is assigned code of 1
    public FireDetectedRequest() {
        super(1);
    }
}

public class LowFuelRequest extends AbstractRequest {
```

```
    // Low on fuel request is assigned code of 2
    public LowFuelRequest() {
        super(2);
    }
}

public class FireHandler extends AbstractHandler {

    // Only interested in handling requests with code 1
    private static int code = 1;

    public FireHandler(AbstractHandler successor) {
        super(successor);
    }

    @Override
    public void handleRequest(AbstractRequest request) {
        if (code == request.getRequestCode()) {
            // Handle the request here.
        } else {
            // If the handler, doesn't handle these type of
            // requests, it can just call the super class's
            // forward request method.
            super.handleRequest(request);
        }
    }
}
```

Finally the interaction between these different classes is exhibited in the client code.

```
public class Client {

    public void main() {

        // Setup the chain like so:
        // FireHandler --> LowFuelHandler --> null
        // The chain has just two handlers with the firstHandler
        // at the head of the chain.
        AbstractHandler lowFuelHandler = new LowFuelHandler(null);
        FireHandler fireHandler = new FireHandler(lowFuelHandler);

        // Create a emergency request that the airplane is running lo
w
        // on fuel.
```

```
        LowFuelRequest lowFuelRequest = new LowFuelRequest();


        // Let the chain handle the request
        fireHandler.handleRequest(lowFuelRequest);
    }
}
```

Notice, how in our example, the request moves along the chain till a handler that is capable of addressing the request receives it. The chain also defines an order in which the request gets handled. If more than one handler can serve the request, then it'll get handled by the one occurring first in the chain.

## Other Examples

- Frontend developers might recall how event handlers work in javascript. If a button is clicked and its event handler method is implemented then the event would get handled and can be stopped from bubbling up to the parent element's event handler. The event handlers can be thought of as forming a chain starting from the leaf (innermost) HTML element all the way to the root element. Note that browsers also support ***event capturing*** where the event travels from the outermost HTML element to the innermost. One departure from the textbook definition of the pattern is that stopping the event from propagation to the next successor is optional.

- Another example is how a web request passes through several filters. `javax.servlet.Filter.doFilter()` follows the chain of responsibility pattern. The `doFilter` method of the Filter is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain. The FilterChain passed into this method allows the Filter to pass on the request and response to the next entity in the chain.

## Caveats

- To form the handler chain, individual objects might need to store references to their successors. However, in certain cases the links might already exist such as that in a composite structure like a tree.

# Observer Pattern

This lesson discusses how updates from an object of interest can be communicated efficiently to interested parties using the observer pattern.

## What is it ?

Social media helps us immensely in understanding the observer pattern. If you are registered on Twitter then whenever you follow someone, you are essentially asking Twitter to send you (**the observer**) tweet updates of the person (**the subject**) you followed. The pattern consists of two actors, *the observer* who is interested in the updates and *the subject* who generates the updates.

A subject can have many observers and is a one to many relationship. However, an observer is free to subscribe to updates from other subjects too. You can subscribe to news feed from a Facebook page, which would be the subject and whenever the page has a new post, the subscriber would see the new post.

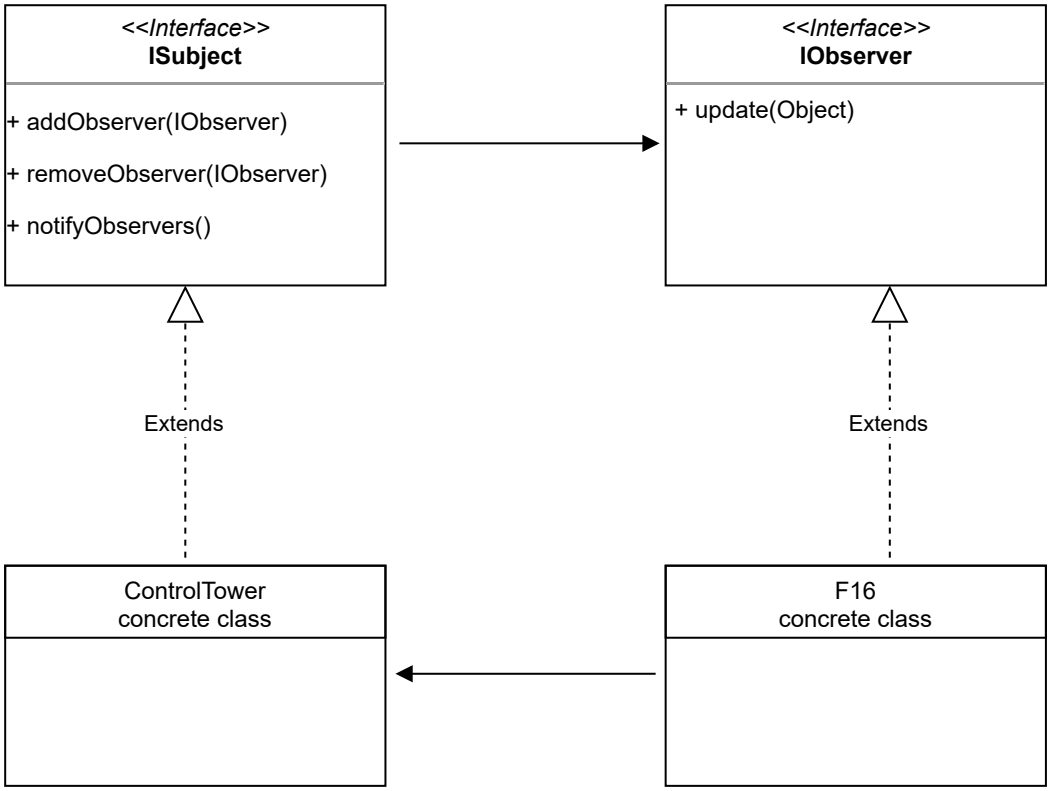
The pattern is formally defined as ***a one to many dependency between objects so that when one object changes state all the dependents are notified***.

## Class Diagram

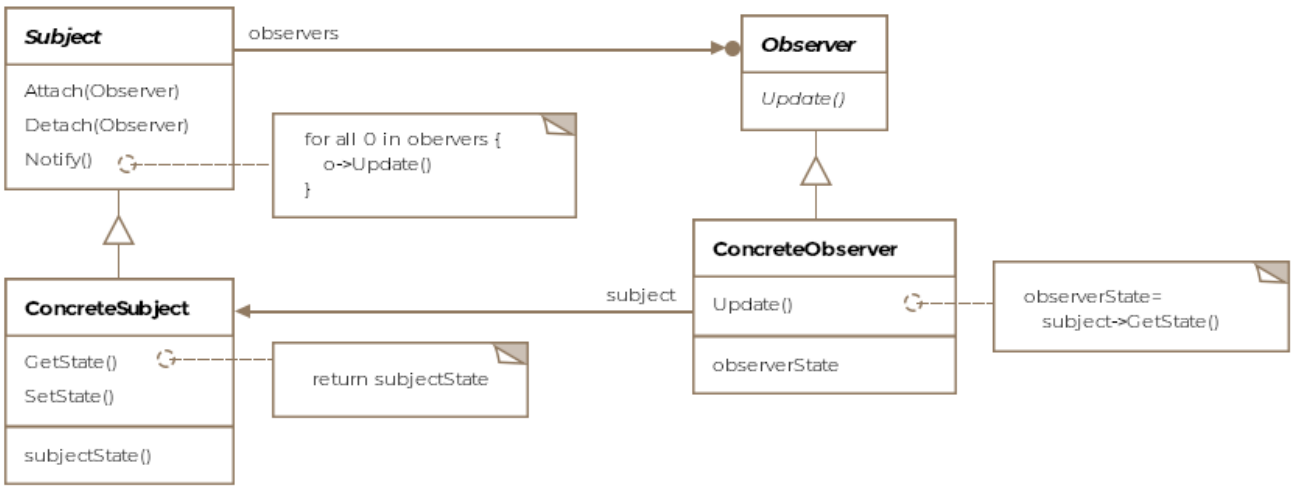
The class diagram consists of the following entities:

- **Subject**
- **Observer**
- **Concrete Subject**

- **Concrete Subject**
- **Concrete Observer**




The class diagram in general is shown below.




To ensure loose coupling we'll define an interface for the subject and one for the observer.

```
public interface ISubject {

    void addObserver(IObserver observer);

    void removeObserver(IObserver observer);

    void notifyObservers();
}

public interface IObserver {

    void update( Object newState);
}
```

The interfaces are almost self-explanatory. A concrete subject or publisher needs to implement `ISubject` and an observer needs to implement `IObserver`.

## Example

Going back to our aircraft example, we can say that any aircraft in flight would be interested in updates from the air-traffic controller. We can imagine that an aircraft, as soon as, it is airborne would want to subscribe to updates from the air traffic controller and unsubscribe when it lands.

## Publisher Code

We create a control tower class which acts as a publisher for all aircraft.

```
public class ControlTower implements ISubject {

    // The ControlTower maintains a list of
    Collection<IObserver> observers = new ArrayList<>();

    @Override
    public void addObserver(IObserver observer) {
        observers.add(observer);

    }
```

```
    }

    @Override
    public void removeObserver(IObserver observer) {
        // Logic to remove the observer goes in here
    }

    @Override
    public void notifyObservers() {
        for (IObserver observer : observers) {
            // We are passing null for state here but we
            // could pass 'this' the subject itself or a
            // type representing the state. These two options
            // represent the Pull vs Push models
            observer.update( null);
        }
    }

    /**
     * This is hypothetical function that runs perptually, gathering
     * runway and weather conditions and notifying all observers of
     * them periodically.
     */
    public void run() {

        while (true) {
            // get new runway/weather conditions and update observers
            // every five minutes
            // Thread.sleep(1000 * 60 * 5)
            notifyObservers();
        }
    }
}
```

Observer Code

The F-16 class would implement the `IObservable` as objects of the F-16 class would want updates from the `ControlTower` class.

```
public class F16 implements IObserver, IAircraft {

    ISubject observable;
```

```
    public F16(ISubject observable) {
        this.observable = observable;

        observable.addObserver(this);
    }

    @Override
    public void fly() {
        System.out.println("F16 is flying ...");

    }

    @Override
    public void land() {

        // Don't forget to unsubscribe from control tower updates
        observable.removeObserver(this);
    }

    @Override
    public void update(Object newState) {
        // Take appropriate action based on newState
    }
}
```

## Push vs Pull

Note how the F-16 class receives the new state as a type of class `Object`. We can pass in a more specific type if we agree on what information gets passed. The way the code is structured, it represents the **Push Model** where the subject is responsible for pushing the new state. Say if the aircraft is a helicopter, is it really interested in the runway conditions? It's supposed to land on a helipad and may not use all the information that it gets passed for the runway.

A flip solution is that in the `update` method, instead of passing in the changed state, we pass in the *subject* object itself. The subject-object in turn exposes getter methods for individual pieces of information it is willing to share with the observers. The observer is now able to lookup only that information which it finds interesting. This is called the **Pull Model.**

## Other Examples

- Frontend frameworks often involve the publisher-subscriber model, where a change in a DOM element on the webpage by a user causes a data-structure held in the browser's memory (think javascript code) gets updated. AngularJS and KnockoutJS frameworks are examples of this pattern.
- In Java, implementations of `java.util.EventListener` are examples of the observer pattern.

## Caveats

Some issues one needs to keep in mind while working with the observer pattern.

- In case of many subjects and few observers, if each subject stores its observers separately, it'll increase the storage costs as some subjects will be storing the same observer multiple times.
- A small change in the subject, may lead to a cascade of updates for the observers and their dependent objects. If clients invoke notify on the subject after each change, it can overwhelm the observers with updates, whereas another option can be to batch the changes and then invoke notify on the subject.
- Usually, another entity *Change Manager* can sit between the observers and the subject in case there are complex dependencies between the subject and the observers.

# Interpreter Pattern

This lesson delves into the interpreter pattern, which allows us to simplify representation and implementation of a new programming language albeit with limited syntax.

## What is it ?

The interpreter literally means a ***translator***, someone who can convert from one form of speech to a another. The interpreter pattern converts a language's sentences into its grammar and interprets them.

Understanding the interpreter pattern requires background knowledge in automata and theory of computation. We'll briefly go over some of the concepts required to understand the pattern.

## Grammar

Every human language has an associated grammar that defines what constructs are legal or illegal. Similarly, computer languages are defined by *grammar* too. Given a snippet of code, the language defined by the grammar would determine if the code is syntactically correct or not. There are four types of grammar known as the Chomsky's hierarchy.

- Regular
- Context Free
- Context Sensitive
- Recursively Enumerable

## Context Free Grammar

We'll be interested in ***context free grammar*** for the purposes of this lesson. Most programming languages use context free grammars to specify the syntax of a language. A syntactically correct program however may or may not compile.

A CFG consists of four components:

- start symbol
- a set of terminal symbols
- a set of non-terminal symbols
- a set of productions (rules)

Let's immediately see an example to understand what is meant by each of the above terms. Consider the below *productions* or rules for an arithematic expression

1. <*expression*> --> *number*
2. <*expression*> --> <*expression*> + <*expression*>
3. <*expression*> --> <*expression*> - <*expression*>
4. <*expression*> --> <*expression*> * <*expression*>
5. <*expression*> --> <*expression*> / <*expression*>

The above rules say that the left hand side expression, which is a non-terminal symbol, can be expanded into the values on the right hand side. A non-terminal symbol is nothing but a variable or a placeholder which can be expanded into the right hand side values. One can recursively keep expanding the non-terminal symbols till a terminal symbol is reached. This is similar to how a recursive algorithm stops recursion once it reaches the base case, otherwise the program would continue in an infinite loop.

The arithematic CFG will have <*expression*> as the start symbol and + - *

/ ***number*** will form the set of terminals, where number is any valid number.

As an example, we can create the following expression using the CFG

- **<expression>** using the start symbol
- **<expression>** + **<expression>** using the second production rule
- **7** + **<expression>** * **<expression>** using the first and fourth production rules
- **7** + **4** * **3** using the first production rule

The string ***7*** + ***4*** * ***3*** is said to be ***in the language of the grammar*** that we defined.

## Connecting back

With the above discussion, now we are in a better position to define the interpreter pattern. The Interpreter pattern uses a class to represent each grammar rule. Symbols on the right-hand side of the rule are instance variables of these classes.

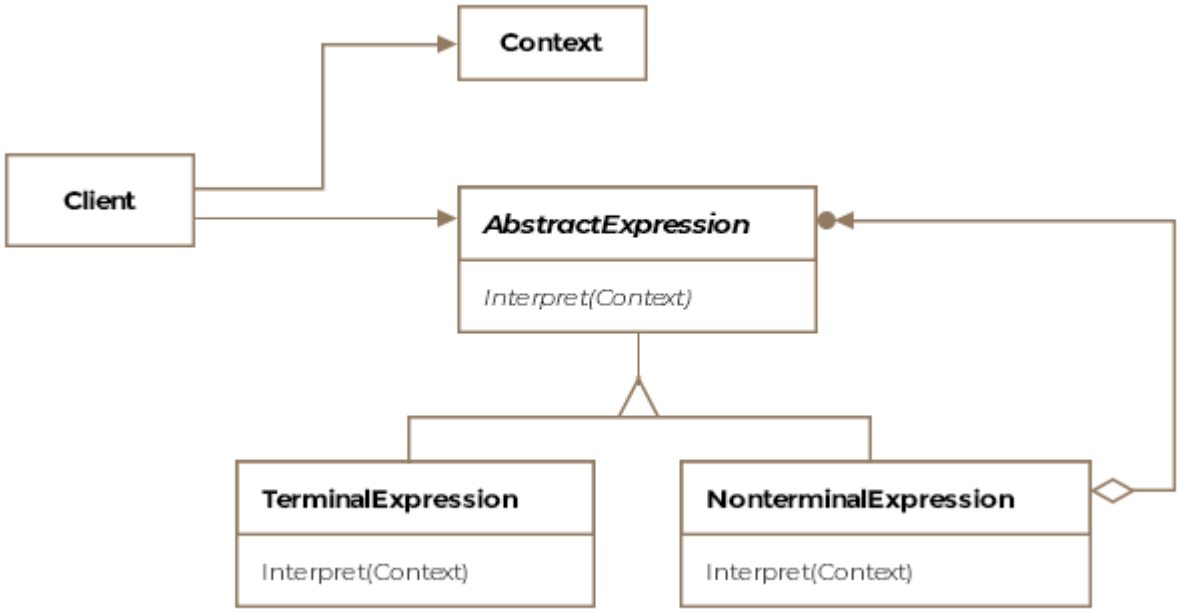
Formally, the pattern is defined as ***describe a way to represent the grammar of a language along with an interpreter that uses the representation to interpret sentences in the language***.

## Class Diagram

The class diagram consists of the following entities

- **Abstract Expression**
- **Terminal Expression**
- **Nonterminal Expression**
- **Context**

- **Client**




Class Diagram

## Example

Let's say you are writing an educational programming language for kids who aspire to be pilots someday. Your language would be very simple and will allow kids to control a plane object on-screen using the following keywords, which make up your programming language:

- *Glide*
- *SplitS*
- *BarrelRoll*

The plane object on the screen will perform one of the three actions when reading the program script. However the restriction is that a plane must start and end with a glide operation and can't perform stunts consecutively, i.e. the splitS and barrelRoll must be separated by a glide operation.

The above language can be defined by the grammar below:

The above language can be defined by the grammar below:

- ***Flight*** is the start symbol and represents the program a child wrties.
- Terminal symbols includes: **glide, splitS, barrelRoll**
- Non-terminal symbols include: <**Flight**> and <**ShowOff**>
- The production rules are:

    1. **<Flight> --> <Flight> <ShowOff> <Flight>**
    2. **<Flight> --> glide**
    3. **<ShowOff> --> splitS**
    4. **<ShowOff> --> barrelRoll**

    Or we can express the same as below when using BNF form:

    1. **<Flight> ::= glide | <Flight> <ShowOff> <Flight>**
    2. **<ShowOff> ::= splitS | barrelRoll**

As an example we can generate a string from the following sequence of operations using the grammar rules:

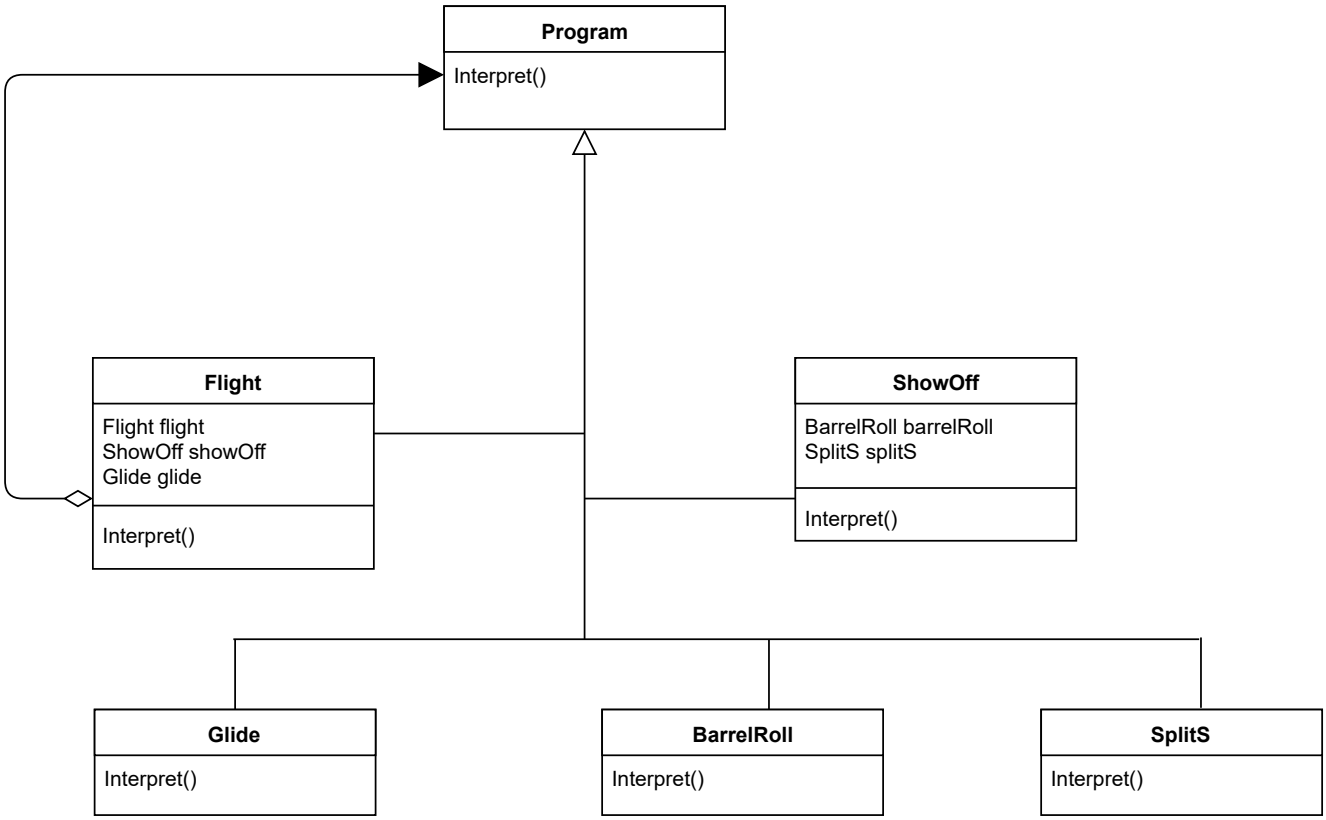
1. <Flight>
2. <Flight> <ShowOff> <Flight>
3. glide splitS <Flight> <ShowOff> <Flight>
4. **glide splitS glide barelRoll glide** This sequence of commands would be considered syntactically correct for our educational programming language.

The recursion ends with the terminal symbols. Applying the interpreter pattern, we model each of the grammar rules as a class. Symbols on the right-hand side of the rule are instance variables of these classes.

Since we use *Flight* as both the start symbol as well as a non-terminal symbol, we can introduce an abstract `Program` class from which all the other classes derive. The `Program` class has an abstract method

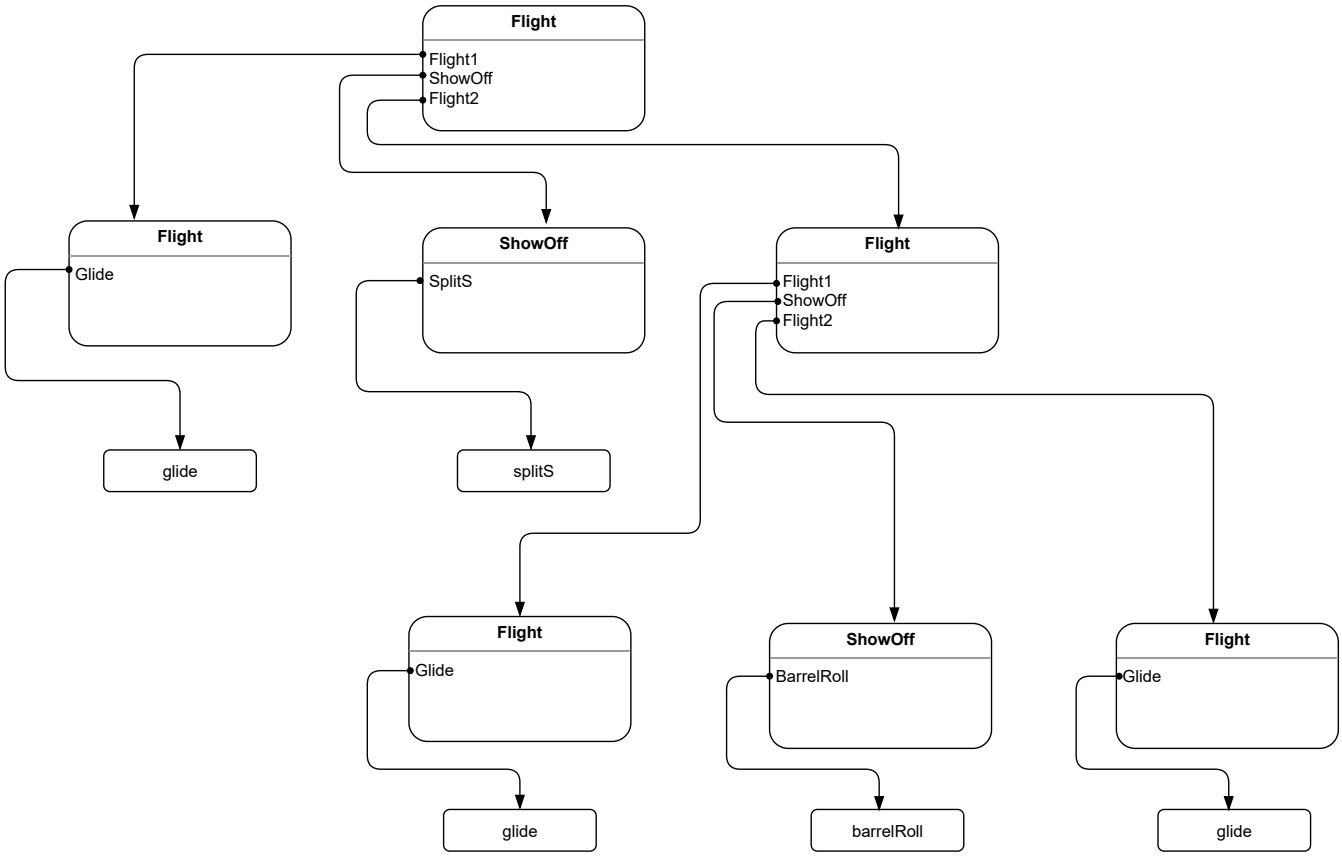
`interpret()` that all derived classes implement and is required by the pattern.

The class diagram for the language defined by the grammar would look like below:




## Abstract Syntax Tree

Abstract syntax trees (AST) are widely used in compilers to represent the structure of a program. Don't confuse the *abstract* to mean an abstract class or interface. Without getting into too much detail, the takeaway is that all the language strings produced by a grammar can be represented as an abstract syntax tree. The tree nodes would be the classes we created from the grammar rules. The internal nodes will be non-terminal symbols and the leaves must necessarily be terminal symbols. Lets take the string we generated earlier ***glide splitS glide barelRoll glide*** and see how its AST would look like.




## Tying it All Together

Each of our subclass implements the `interpret(Context)` method. Interpret takes as an argument the context in which to interpret the expression. Each interpret operation of the terminal symbols defines the base case for recursion. The context contains the input string and information on how much of it has been matched so far. Each subclass of `Program` implements Interpret to match the next part of the input string based on the current context. For instance:

- The classes `Glide`, `BarrelRoll` and `SplitS` will check to see if the input matches any of those words.
- The class `ShowOff` will check if the input matches either of the two values it can take on.
- The class `Flight` class will check if the input matches the terminal symbol or expands into another concatenation of non-terminal symbols.

We'll skip the implementation that'll contain the logic for matching the input stream as its not necessary to understand the pattern.

The listing for the classes appears below

```
public abstract class Program {
    public void interpret(Context context) {}
}

public abstract class Flight extends Program {

    Flight flight;
    ShowOff showOff;
    Glide glide;

    @Override
    public void interpret(Context context) {}
}

public class ShowOff extends Program {

    BarrelRoll barrelRoll;
    SplitS splitS;

    @Override
    public void interpret(Context context) {

    }
}

public class BarrelRoll extends Flight {

    @Override
    public void interpret(Context context) {

    }
}

public class Glide extends Program {
```

```
    @Override
    public void interpret(Context context) {

    }
}

public class SplitS extends Program {

    @Override
    public void interpret(Context context) {

    }
}
```

The client will use pattern like below:

```
public class Client {

    public void main(AbstractSyntaxTree ast) {

        Context context = new Context("glide splitS glide barelRoll g
lide");

        while (ast.hasNext()) {
            Program node = ast.getNextNode();
            node.interpret(context);
        }
    }
}
```

## Other Examples

- `java.util.Pattern` is a compiled representation of a regular expression.
- `java.text.Normalizer` provides functionality to transform Unicode text.

Caveats

- The `interpret()` method can also be put in a visitor object instead of putting it in the expression classes.
- The terminal symbols can also implemented as flyweight objects.
- Its easy to implement, extend and change grammars with limited rules that are implemented using the interpreter pattern. However grammars will lots of rules become hard to manage since there's one class per rule in the interpreter pattern.

# Command Pattern

This lesson discusses how actions and requests can be encapsulated as objects to act as callbacks and in the process allow logging, queueing and undo of commands.

## What is it ?

The command pattern's intention is to decouple the consumers of an action and the object which knows how to perform the action. Let me present an example for clarity. Suppose you are designing a framework for UI, and you add the ability for the users of the framework to add a menu bar. The menu bar will consist of menu-items. When someone clicks on the menu-item some action will be performed. Since you are only building the framework, you don't know what actions the users of the framework can have the menu-item perform. It may vary from opening a document to restarting the application. The command pattern allows us to encapsulate the desired action in an object and the object becomes responsible for invoking the action with the appropriate arguments.

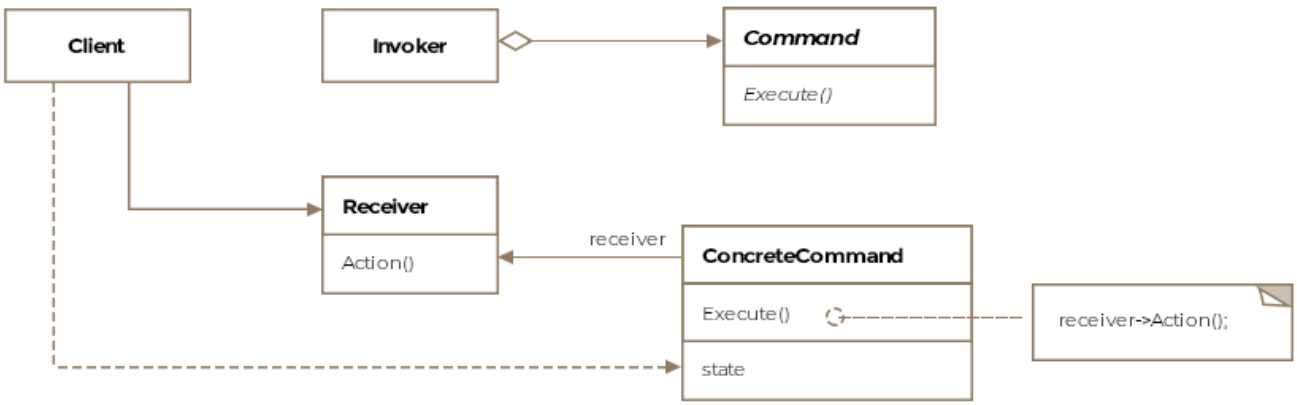
Formally, the pattern is defined as ***representing an action or a request as an object that can then be passed to other objects as parameters, allowing parameterization of clients with requests or actions. The requests can be queued for later execution or logged. Logging requests enables undo operations***.

## Class Diagram

The class diagram consists of the following entities

- **Command**

- **Concrete Command**
- **Client**
- **Invoker**
- **Receiver**




Class Diagram

## Example

Going back to our aircraft example, imagine the cockpit of the Boeing-747. It has a multitude of instrument panels with knobs and buttons. For simplicity's sake let's say the plane has a button for the landing gear (the wheels of the aircraft), which allows the landing gear to be lowered or retracted. **The button shouldn't need to know *how* the landing gear works, it just needs to know *who* has the knowledge to operate the landing gear**. The *who* part will implement the `Command` interface and the button will know it needs to invoke the `execute` method on the *who* object.

Let's look at the code implementation of our simplistic scenario. First up is the command interface:

```
public interface Command {
    void execute();
}
```

When the button is pressed to say lower the landing gear. The button code should only have to deal with a command object that implements the `Command` interface. The button code simply calls `execute` on the command object. Let's look at the `LandingGearDownCommand` class.

```
public class LandingGearDownCommand implements Command {

    // This is called the receiver of the request and
    // actually has the logic to perform the action
    LandingGear landingGear;

    public LandingGearDownCommand(LandingGear landingGear) {
        this.landingGear = landingGear;
    }

    @Override
    public void execute() {
        landingGear.up();
    }
}
```

You'll see the variable `landingGear` in the `LandingGearDownCommand` class. This is called the **Receiver**. It is the object that actually knows how to lower the landing gear and does the job, which is why it is called the receiver because it *receives* the request and processes it. The LandingGearDownCommand is the **Command** in pattern-speak. **The command is composed with the receiver that actually contains the logic to perform the requested action**

You must be wondering that `LandingGearDownCommand` object needs to be instantiated somewhere in the code. The **Invoker** takes on the responsibility of creating the command object and invoking it. In our case, we can imagine a class representing the instrument panel which holds all the commands for the physical buttons on the panel. It may look something like:

```
public class InstrumentPanel {

    // Only two commands for now
    Command[] commands = new Command[2];

    public InstrumentPanel() {
```

```
    }

    public void setCommand(int i, Command command) {
        commands[i] = command;
    }

    public void lowerLandingGear() {
        // Assuming that the client correctly sets the first
        // index to be the landing gear lower command
        commands[0].execute();
    }

    public void retractLandingGear() {
        commands[1].execute();
    }
}
```

Notice how the Invoker is simply setting up the commands and then invoking the `execute` method on the command objects. We can very well replace the command object with an instance of a different implementation and the invoker would still work correctly. This allows decoupling between the invoker and the receivers. The Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it.

The last piece to the command pattern is the client which sets up the invoker with the right commands and the commands with the right receiver objects.

```
    public void main() {

        LandingGear landingGear = new LandingGear();
        LandingGearDownCommand landingGearDownCommand = new LandingGe
arDownCommand(landingGear);
        LandingGearUpCommand landingGearUpCommand = new LandingGearUp
Command(landingGear);

        // Create the instrument panel
        InstrumentPanel instrumentPanel = new InstrumentPanel();

        // Assign the commands
        instrumentPanel.setCommand(0, landingGearUpCommand);
```

```
        instrumentPanel.setCommand(1, landingGearDownCommand);

        // Lower the landing gear
        instrumentPanel.lowerLandingGear();

        // Retract the landing gear
        instrumentPanel.retractLandingGear();
    }
```

## Macro Command

A series of commands can be strung together and executed in a sequence by another command object, sometimes called a *macro command*. It has no explicit receiver as the commands it sequences define their own receivers. The macro command is an example of the *composite pattern*.

## Other Examples

- `java.lang.Runnable` defines the interface implemented by classes whose instances are executed by threads.
- Implementations of `javax.swing.Action` also conform to the command pattern.

## Caveats

- The command pattern is equivalent of a callback function in procedural languages as we parametrize objects with an action to perform
- The command objects can also be queued for later execution.
- The command interface can introduce an `unexecute` method, which

The command interface can introduce an `unexecute` method, which reverses the actions of the `execute` method. The executed commands

can then be stored in a list and traversing the list forwards and backwards while invoking execute or unexecute can support redo and undo respectively. The *memento pattern* can be helpful in storing the state a command needs to undo its effects.

- The command interface can add methods to save and read from disk allowing logging of commands. In case of a crash the log can be read and the commands re-executed in the same sequence to get the system back to the state just before the crash.
- The command pattern offers a way to model *transactions*. A transaction consists of finer grained operations applied to data.

# Iterator Pattern

This lesson discusses how the underlying elements that make up an aggregate object can be looped through without exposing implementation details to clients.

## What is it ?

*Iterate* literally means to perform repeatedly. A *for* loop iterates over an array i.e. it accesses the array repeatedly. If you are familiar with Java then you would already have come across this pattern while working with Java Collections. A brief demonstration is below.

```
        ArrayList<String> companiesIWantToInterviewFor = new ArrayLis
t<>();
        companiesIWantToInterviewFor.add("SnapChat");
        companiesIWantToInterviewFor.add("Twitter");
        companiesIWantToInterviewFor.add("Tesla");

        Iterator<String> it = companiesIWantToInterviewFor.iterator
();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
```

The iterator allows a consumer to go over the elements of a collection without knowing how the collection is implemented. A collection can be a list, array, arraylist, hashmap or any other fancy datastructure. A collection signifies a bunch of objects. It doesn't specify any ordering or properties about the objects it holds. An iterator is formally defined as ***a pattern that allows traversing the elements of an aggregate or a collection sequentially without exposing the underlying implementation***.

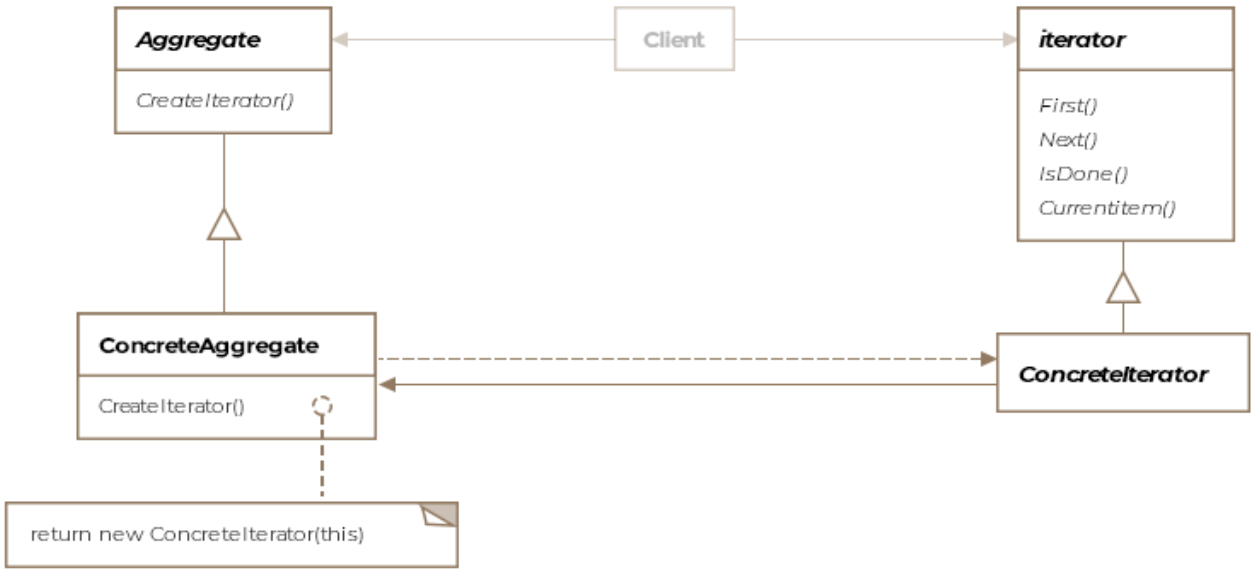
The iterator pattern extracts out the responsibility of traversing over an aggregate out of the aggregate's interface and encapsulates it in the

iterator class, thereby simplifying the aggregate's interface.

## Class Diagram

The class diagram consists of the following entities

- **Iterator**
- **Concrete Iterator**
- **Aggregate**
- **Concrete Aggregate**




## Example

Java already has an interface `Iterator` but for the purposes of learning, we'll create it from scratch. Our interface will expose only two methods `next()` and `hasNext()`. It'll look something like below:

```
public interface Iterator {

    void next();

    boolean hasNext();
```

```
}
```

Lets say we have a class `AirForce` and it contains different kinds of aircraft, some are cargo airplanes, others are fighter jets and a few helicopters. Objects for each of these aircraft types are stored in different collection types. Take a minute to go over the `Airforce` class listing below:

```
public class AirForce {

    List<IAircraft> jets = new ArrayList<>();
    IAircraft[] helis = new IAircraft[1];
    LinkedList<Boeing747> cargo = new LinkedList<>();

    public List<IAircraft> getJets() {
        return jets;
    }

    public IAircraft[] getHelis() {
        return helis;
    }

    public LinkedList<Boeing747> getCargo() {
        return cargo;
    }

    public AirForce() {
        jets.add(new F16());
        helis[0] = new CobraGunship();
        cargo.add(new Boeing747());
    }

    // This method returns a concrete iterator that
    // traverses over the entire airforce planes.
    public Iterator createIterator() {
        return new AirForceIterator(this);

    }

    // This method returns a different concerete iterator
    // that traverses over only the jets in the airforce.
    public Iterator createJetsIterator() {
        return new JetsIterator(jets);

    }
}
```

Note the methods `createIterator()` and `createJetsIterator()` return objects that implement the `Iterator` interface. Notice how each kind of aircraft is stored in a different type of collection. If a client needs to list all the aircraft in an airforce object, it'll have a hard time invoking getters and then going over each individual collection. We mask this complexity by creating an iterator class whose sole job is to list all the aircraft held by the airforce. Look at the implementation below:

```
public class AirForceIterator implements Iterator {

    List<IAircraft> jets;
    IAircraft[] helis;
    LinkedList<Boeing747> cargo;
    int jetsPosition = 0;
    int helisPosition = 0;
    int cargoPosition = 0;

    /**
     * The iterator is composed with the object it'll be iterating ov
er
     */
    public AirForceIterator(AirForce airForce) {
        jets = airForce.getJets();
        helis = airForce.getHelis();
        cargo = airForce.getCargo();

    }

    @Override
    /**
     * We provide our own custom logic of returning aircraft in a
     * sequence. Note we are returning IAircraft interface object whi
ch
     * every plane in our airforce implements. We also design the fun
ction
     * to throw a runtime exception if next is invoked when no more e
lements
     * are left to iterate over
     */
    public IAircraft next() {

        // return helis first
```

```
        if (helisPosition < helis.length) {
            return helis[helisPosition++];

        }

        // return jets second
        if (jetsPosition < jets.size()) {
            return jets.get(jetsPosition++);
        }

        // return cargos last
        if (cargoPosition < cargo.size()) {
            return cargo.get(cargoPosition++);
        }

        throw new RuntimeException("No more elements");

    }

    @Override
    public boolean hasNext() {

        return helis.length > helisPosition ||
               jets.size() > jetsPosition ||
               cargo.size() > cargoPosition;
    }
}
```

Let's look at how the client uses the iterator

```
public class Client {

    public void main() {

        AirForce airForce = new AirForce();

        Iterator jets = airForce.createJetsIterator();

        while (jets.hasNext()) {
            jets.next();
        }

        Iterator allPlanes = airForce.createIterator();

        while (allPlanes.hasNext()) {
            allPlanes.next();
```

```
    }

}
```

Notice, the client has no idea about how the different airplane types are held in the airforce object nor does it know what type of aircraft they are. It simply gets to work with the `IAircraft` interface.

The Java API has its own iterator interface which also includes `remove()` and `forEachRemaining()` methods that we have not included in our in-house iterator.

## Internal vs External Iterator

When the iteration control rests with the client using the iterator, that is, the client is responsible for advancing the traversal and explicitly requesting the next element from the iterator, it is an example of an ***external iterator***. Iterators in our aircraft example are external iterators. On the other hand, when the client hands the iterator an operation to perform and the iterator performs the operation on each element of the aggregate, it is an example of an ***internal iterator***.

## Other Examples

- In Java several classes directly or indirectly implement the `Iterator` interface. By indirect implemenation, it is implied that a class implements an interface that extends the `Iterator` interface.
- Another example one can think of is in case of tree datastructure. One can parametrize the `createIterator()` method to yield iterators which traverse the tree in preorder, inorder or postorder.
- `java.util.Scanner` class is another example of the iterator pattern.

- `java.util.Enumeration` interface although deprecated but is an early example of the iterator pattern in Java.

Caveats

- Note that there could be more than one pending traversals on the aggregate. Each iterator would store its own traversal state. The iterator can apply the memento pattern to store the traversal state.
- By moving the traversal outside of the aggregate, it's easier to provide variations on the traversal algorithm to the client.
- Special care needs to be taken care when insertions or deletions are allowed to an aggregate amidst an on-going traversal. The iterator can either skip over a new element or iterate over the same element twice.
- For composites as described in the *Composite Pattern*, it might make sense to create internal iterators instead of external ones, reason being that the composite can have several levels of aggregate structures and its easier for the composite to internally keep track of the traversal position by calling itself recursively and implicitly storing the traversal path in the call stack.

# Mediator Pattern

This lesson discusses how the mediator pattern centralizes interaction amongst a number of interacting objects reducing coupling and dependence among them.

## What is it ?

A mediator is defined as *a person who makes people involved in a conflict come to an agreement*. The pattern definition however, isn't violent and says the pattern is applied to encapsulate or centralize the interactions amongst a number of objects. Object orientated design may result in behavior being distributed among several classes and lead to too many connections among objects. The encapsulation keeps the objects from referring to each other directly and the objects don't hold references to each other anymore.

Formally, the pattern is defined as ***encouraging loose coupling among interacting objects by encapsulating their interactions in a mediator object, thus avoiding the need for individual objects to refer to each other directly and allowing to vary object interactions independently.***

A mediator controls and coordinates the behavior for a group of objects. The objects only communicate with the mediator or the director and don't know of the other objects.

The mediator pattern should be applied when there are many objects communicating in a well-structured but complex manner resulting in interdependencies that are hard to understand. The participant objects in such a scheme don't lend themselves for reuse because of dependence on so many other objects.

Since the interaction rules or behavior is isolated in a single mediator class, it becomes easier to change. Also note that in the absence of the

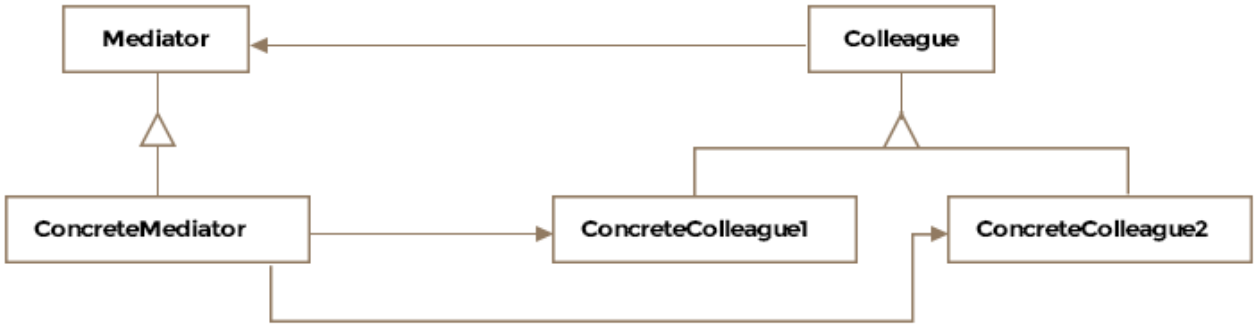
mediator, the objects are engaged in many-to-many interactions but when

the mediator is introduced the interactions become one-to-many between the mediator and the other participating objects.

## Class Diagram

The class diagram consists of the following entities

- **Mediator**
- **Concrete Mediator**
- **Colleague Classes**




Class Diagram

## Example

Imagine an airport without an air-traffic control tower. All the airplanes in flight, about to land and about to take-off need to be aware of every other airplane to coordinate the use of the runways as well as the airspace. There are in a sense dependent on each other to avoid a disaster but this leads to having too many *interconnections* among objects. With a single runway, the worst case is every airplane has to know about every other airplane about to land or take-off to avoid collisions. The control tower object can act as a *mediator* and communication hub for all the

airplanes and ensure a smooth working of the airport.

For instructional purposes, we'll implement the coordination logic required for an airplane to land safely. The `ControlTower` class appears below:

```
public class ControlTower {

    List<IAircraft> queuedForLanding = new ArrayList<>();

    // An aircraft just notifies the control tower that it wants to
    // land and doesn't coordinate with other aircraft
    synchronized public void requestToLand(IAircraft aircraft) {
        queuedForLanding.add(aircraft);
    }

    public void run() {

        // perpetual loop
        while (true) {

            // inefficient busy wait till aircraft requests to land
            while(queuedForLanding.size() == 0);

            IAircraft aircraft;
            synchronized (this) {
                aircraft = queuedForLanding.remove(0);
            }
            // We have only one runway available so only allow a sing
le
            // aircraft to land.
            aircraft.land();
        }
    }
}
```

The aircraft interface and classes are:

```
public interface IAircraft {

    public void land();
}

public class F16 implements IAircraft {
```

```
    ControlTower controlTower;

    public F16(ControlTower controlTower) {
        this.controlTower = controlTower;
    }

    @Override
    public void land() {
        System.out.println("F16 landing...");
    }

    public void requestControlTowerToLand() {
        controlTower.requestToLand(this);

    }
}
```

In our naive example the aircraft object communicates with the `ControlTower` class and requests to land. The control tower object maintains a queue of planes wishing to land and runs a perpetual thread-safe loop that allows each plane to land one after the other. If the number of runways available increases or close down for maintenance, the changes are localized to the `ControlTower` class.

We used the control tower example also for the *observer pattern* and it applies in the mediator pattern too as the interaction between the mediator and the colleagues can be modeled on the observer pattern.

## Other Examples

- `java.util.concurrent.ExecutorService` an Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.
- `java.util.Timer` A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

## Caveats

- The communication between the mediator and other objects, called colleagues, can follow the observer pattern. The mediator acts as the observer and receives notifications from colleagues, which act as subjects. The effects of the change are then forwarded to other colleagues by the mediator.
- The pattern trades complexity of interactions for complexity in the mediator. The mediator itself can become very complex and hard to maintain.

# Memento Pattern

This lesson discusses how the state of an object can be exported as a snapshot without exposing the internals of the object.

## What is it ?

The literal meaning of *memento* is *an object kept as a reminder or souvenir of a person or an event*. The memento pattern ***let's us capture the internal state of an object without exposing its internal structure so that the object can be restored to this state later***. In some sense we are saving a token or a memento of the original object and then recreating the object's state using the memento at a later time.

## Details

The object whose state we capture is called the ***Originator***. The originator's snapshot is called the ***memento***. The memento object is held by another object called the ***Caretaker***. The interaction between these three entities happens as follows:

1. The caretaker requests the originator for a snapshot of its internal state.
2. The originator produces a memento.
3. The memento is held by the caretaker and passed back to the originator when required to revert its state to that captured in the memento. If that need doesn't arise, the memento is eventually discarded by the caretaker.

In the absence of the memento pattern, the originator would need to expose its complete internal state to outside classes who can then create snapshots of the internal state of the originator at any time. However, this approach is brittle, breaks encapsulation and any future changes in the originator's state would require corresponding changes in the classes that consume the originator's internal state.

With the introduction of the memento, the originator itself creates a snapshot of its state and is free to store as much or as little information as it pleases in the memento object. The memento's interface to outside classes is limited while the originator has full access to memento's state. This prohibits external classes including the caretaker from manipulating memento's state but allows the originator to fully access memento's state so that it can restore itself to the checkpoint represented by the memento.

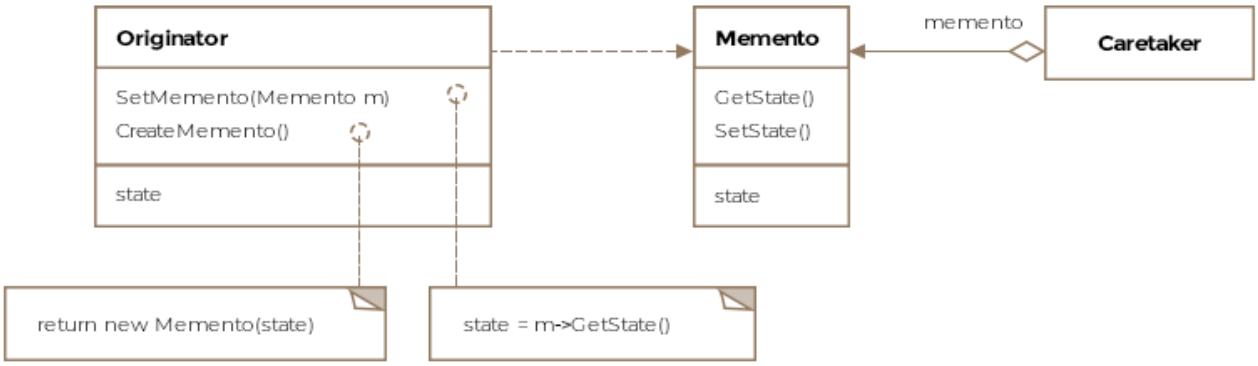
The challenge to limit memento's interface to caretaker and other external classes while at the same time making it completely accessible to the originator may be hard in certain languages. In case of Java, `static classes` can be used to achieve this effect.

Memento lets the originator entrust other objects with information it'll need to revert to a previous state without exposing its internal structure or representations.

## Class Diagram

The class diagram consists of the following entities

- **Memento**
- **Originator**
- **Caretaeker**




Class Diagram

## Example

Modern airplanes are equipped with a device called a blackbox that stores important flight data and helps investigators in case of crashes. Due to the sensitivity of the information, the blackbox contains, we wouldn't want to expose the internal state of the blackbox to clients. We'll write a class mimicking the blackbox that can produce its state as a byte stream for clients. The blackbox code appears below:

```
public class BlackBox implements Serializable {

    private long altitude;
    private double speed;
    private float engineTemperature;
    private static final long serialVersionUID = 1L;

    public BlackBox(long altitude, double speed, float engineTemperat
ure) {
        this.altitude = altitude;
        this.speed = speed;
        this.engineTemperature = engineTemperature;
    }

    // Saving the state of the object as a byte stream
    public byte[] getState() throws IOException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutput out = null;
        byte[] memento = null;
        try {
```

```
            out = new ObjectOutputStream(bos);
            out.writeObject(this);
            out.flush();
            memento = bos.toByteArray();
        } finally {
            try {
                bos.close();
            } catch (IOException ex) {
                // ignore close exception
            }
        }
        return memento;
    }

    // Restoring state from memento
    public BlackBox setState(byte[] memento) throws Exception {
        ByteInputStream bis = new ByteInputStream(memento, memento.le
ngth);

        ObjectInputStream objectInputStream
                = new ObjectInputStream(bis);
        BlackBox blackBox = (BlackBox) objectInputStream.readObject
();
        objectInputStream.close();
        return blackBox;
    }

}
```

In Java, we can use serialization to save the state of the object. We are simply required to mark our class with the `Serializable` interface, which has no methods to implement. Note that in our `setState` method, we are returning an object of type `BlackBox` that the client can then assign. In Java, we can't assign to the `this` keyword, whereas in C++ we can. If it were a C++ implementation, we could have simply assigned the `this` variable the deserialized object, instead of returning it.

The client will work as follows:

```
public class Client {

    public void main(BlackBox blackBox) throws Exception{

        // Save the state of the memento as a byte stream.
```

```
        byte[] memento = blackBox.getState();

        // Do some work.

        // Now restore the blackbox to the previous state
        blackBox = blackBox.setState(memento);

    }

}
```

## Other Examples

- `java.io.Serializable` all implementations of this interface would be examples of the memento pattern.
- `javax.faces.component.StateHolder` this interface is implemented by classes that need to save their state between requests.

## Caveats

- The memento pattern might not be appropriate if large amounts of information need to be copied by the originator.
- Note that using the pattern, the originator is relieved of the responsibility to save its state for the client. Instead, the onus is on the client to request a memento from the originator and manage it. The client can at a later point request the originator to restore itself to the state represented by the memento it holds.

  Imagine a simplistic video game delivered via a web-browser. The game state can be saved by the user. One possibility is for the game to save its state per user of the game and store it on the webservers, the other is to flip the responsibility and store the state on the user's

computer. The user can request a restore of the gaming session by

loading the right memento. This allows the game's codebase to free itself of managing game states per user.

- Sometimes is possible to store incremental changes or the differential between the current and previous states rather than the entire state in the memento. This helps to reduce the space required for storing mementos.

# State Pattern

This lesson discusses how an object exhibits very different behavior and appears to be an instance of a different class when its internal state changes under the state pattern.

## What is it ?

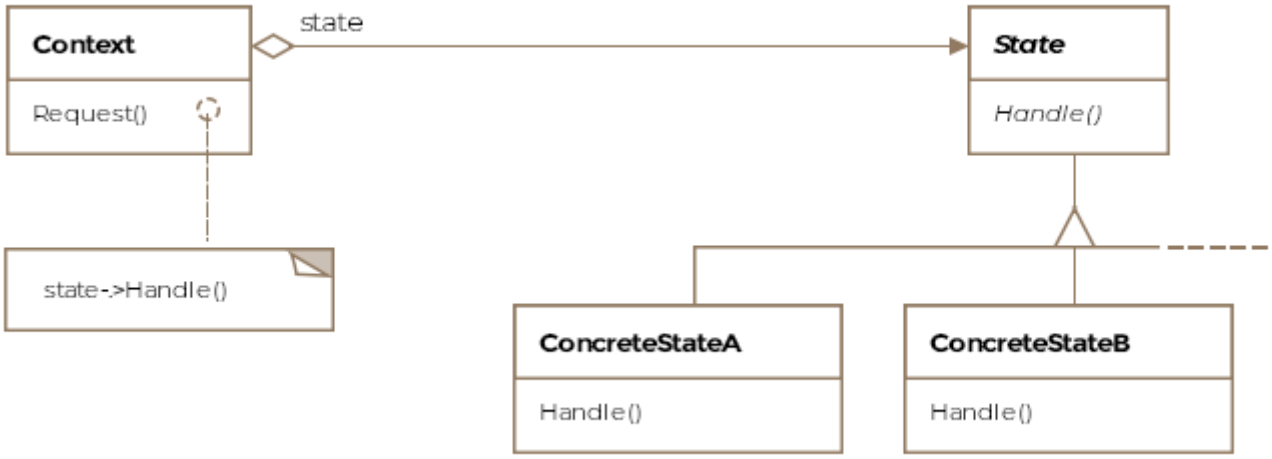
The state pattern will be reminiscent of automata class from your undergraduate degree as it involves state transitions for an object. The state pattern encapsulates the various states a machine can be in. The machine or the ***context***, as it is called in pattern-speak, can have actions taken on it that propel it into different states. Without the use of the pattern, the code becomes inflexible and littered with if-else conditionals.

Formally, the pattern is defined as ***allowing an object to alter behavior when its internal state changes so that it appears to change its class***.

## Class Diagram

The class diagram consists of the following entities

- **Context**
- **State**
- **Concrete State Subclasses**

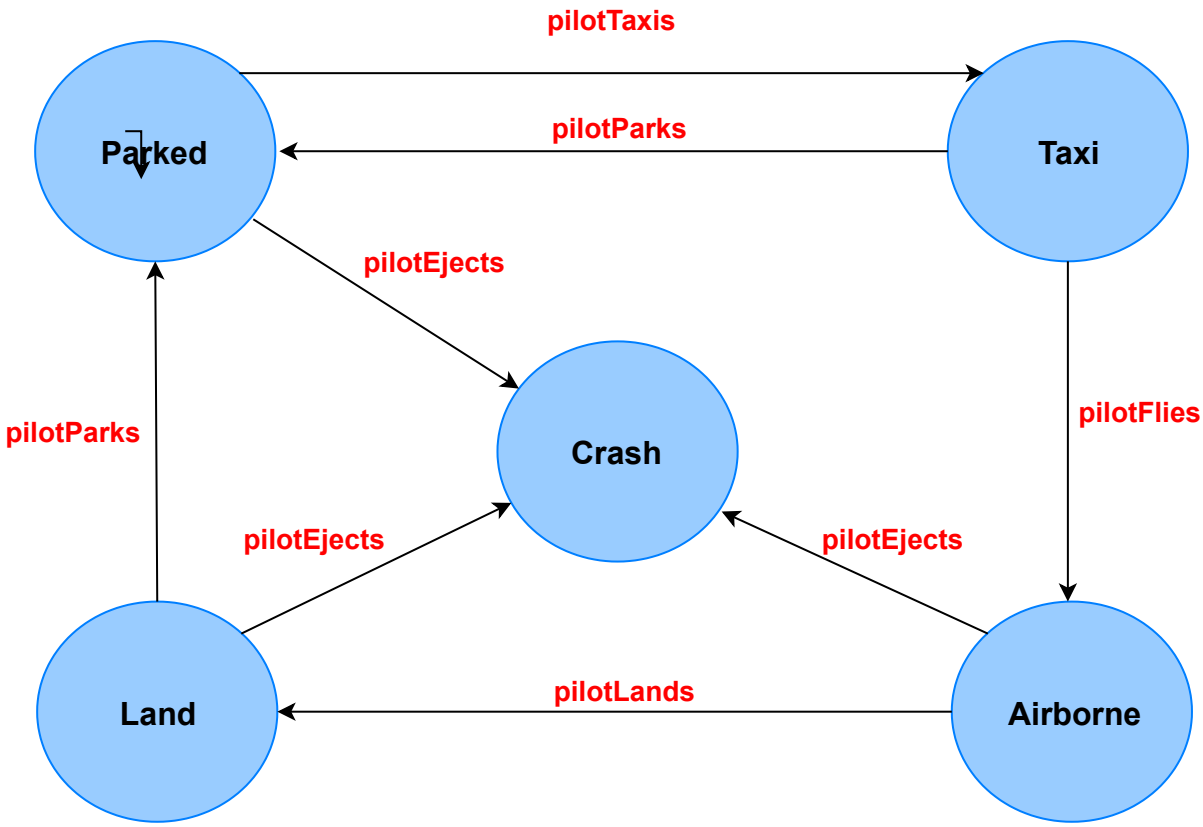



Example

Let's take the case of our F-16 class. An instance of the class can be in various states. Some possible states and transitions to other states are listed below:

| Current State | Possible Transitions to Other States |
| --- | --- |
| Parked | Crash or Taxi |
| Taxi | Airborne or Parked |
| Airborne | Crash or Land |
| Land | Taxi |
| Crash | No transition out of this state |

The state transitions are depicted in the picture below:




The verbs in red in the state diagram are *actions* that propel the aircraft into different states.

Let's see how we'll write the code in the absence of the state pattern. We'll code the state transition function when the pilot takes the *taxi* action.

```
    // Pilot takes the taxi action
    public void pilotTaxies(String currentState) {

        if (currentState == "Parked") {

            currentState = "Taxi";
            System.out.println("Plane is taxing on the runway.");

        } else if (currentState == "Airborne") {

            System.out.println("This is an invalid operation for thi
s state");

        } else if (currentState == "Land") {

            System.out.println("This is an invalid operation for thi
s state");
```

```
        } else if (currentState == "Crashed") {


            System.out.println("This is an invalid operation for thi
s state");

        } else if(currentState == "Taxi"){
            System.out.println("Plane is already taxing on the runwa
y.");
        }

    }
```

The method `pilotTaxies` captures the state transitions once the *taxi* action is taken by the pilot. One can see the labyrinth of if-else conditions will only grow bigger, once we start adding more states for the plane.

Now let's see how the state pattern fixes the above problem. The state pattern will create classes for each of the possible states and each class implements the state-specific behavior. This will result in more number of classes but the design will become flexible and extensible as you'll shortly see. First let's capture the actions a pilot can take, in an interface, which is the **State Interface**. This interface would then be implemented by the different states a F-16 can be in.

```
/**
 * This interface defines the actions a pilot
 * can take against the aircraft object. Each
 * action will move the aircraft into a different
 * state
 */
public interface IPilotActions {

    void pilotTaxies(F16 f16);

    void pilotFlies(F16 f16);

    void pilotEjects(F16 f16);

    void pilotLands(F16 f16);
```

```
    void pilotParks(F16 f16);
}
```

Now let's see how a state would implement this interface. Let's start with the `ParkedState`. For brevity, we'll only show the parked state class. Each of the other states will have a corresponding class that'll code behavior for the F-16 in that state.

```
public class ParkedState implements IPilotActions {

    F16 f16;

    // Notice, how the state class is composed with the context objec
t
    public ParkedState(F16 f16) {
        this.f16 = f16;
    }

    @Override
    public void pilotTaxies(F16 f16) {
        f16.setState(f16.getTaxiState());
    }

    @Override
    public void pilotFlies(F16 f16) {
        System.out.println("This is an invalid operation for this sta
te");
    }

    @Override
    public void pilotEjects(F16 f16) {
        f16.setState(f16.getCrashState());

    }

    @Override
    public void pilotLands(F16 f16) {
        System.out.println("This is an invalid operation for this sta
te");
    }

    @Override
    public void pilotParks(F16 f16) {
```

```
        System.out.println("This is an invalid operation for this sta
te");
    }
}
```

Note how the constructor accepts an instance of the context and saves a reference to it. The plane can only transition to **TaxiState** or **CrashedState** from the **ParkedState**. Either the pilot successfully revs up the engine and takes the plane on the runway or he presses eject if say a fire breaks out on ignition. In our state transition model, you can see that **CrashedState** is a terminal state and there are no transitions out of this state. One can't park, fly, taxi, land or crash again a crashed plane.

The client will use our new set of classes like so:

```
public class Client {

    public void main() {

        F16 f16 = new F16();
        f16.startsEngine();
        f16.fliesPlane();
        f16.ejectsPlane();

    }
}
```

For completeness, the F16 class and the associated interface appear below:

```
public interface IAircraft {
   //Empty interface
}

public class F16 implements IAircraft {

    private ParkedState parkedState = new ParkedState(this);
    private CrashState crashState = new CrashState(this);
    private LandState landState = new LandState(this);
    private TaxiState taxiState = new TaxiState(this);
    private AirborneState airborneState = new AirborneState(this);
```

```
    IPilotActions state;

    public F16() {
        state = parkedState;
    }

    void startsEngine() {
        state.pilotTaxies(this);
    }

    void fliesPlane() {
        state.pilotFlies(this);
    }

    void landsPlane() {
        state.pilotLands(this);
    }

    void ejectsPlane() {
        state.pilotEjects(this);
    }

    void parksPlane() {
        state.pilotParks(this);
    }

    void setState(IPilotActions IPilotActions) {
        state = IPilotActions;
    }

    ParkedState getParkedState() {
        return parkedState;
    }

    CrashState getCrashState() {
        return crashState;
    }

    LandState getLandState() {
        return landState;
    }

    TaxiState getTaxiState() {
        return taxiState;
    }
```

```
    public AirborneState getAirborneState() {
        return airborneState;

    }

}
```

We have delegated the transitions to the state classes. They decide what would be the next state depending on the action the pilot takes.

This pattern may seem very similar to the Strategy Pattern, however, the intent of the two patterns is very different. In strategy, the client is actively composing the context with the strategy object whereas, in the state pattern, the client has no view of the state the context is currently in. The context may show a different behavior for being in a different state. It might appear to the client that the context is a different class altogether when in fact it's only in a different state.

For our example when the F-16 is airborne it becomes capable of using its weapons, firing missiles, destroying tanks etc, all of which it can't do in the parked state. The alteration in the behavior that comes with the state change, almost makes the object appear to belong to a different class.

## Other Examples

- `javax.faces.lifecycle.LifeCycle.execute()` method is dependent on current state of JSF lifecycle.

## Caveats

- In our example, the F-16 object is assumed to be in parked state when

- In our example, the F16 object is assumed to be in parked state when new-ed up however a client can configure the context with a state at instantiation time if the context can be in more than one states when new-ed up. But once configured, the clients don't deal with context states again.
- If the state types only contain behavior and no data then their instances can be expressed as flyweight objects and shared amongst multiple context objects. In our example the `ParkedState` only contains methods and no data particular to an object of F16. This allows us to have multiple F16 objects use the same parked state object to be in the parked state.

# Template Method

This lesson discusses how algorithms with multiple steps can be made configurable by allowing subclasses to provide behavior for some of the steps.

## What is it ?

A template can be thought of as a general or abstract structure that can be customized for specific situations. You may have used a *template* for writing your resume. The template would define the overall structure of the document and leave the details to be added in by the template user. The template method pattern is similar, it defines the skeleton or steps of an algorithm but leaves opportunities for subclasses to override some of the steps with their own implementations.

Formally, the pattern is defined as ***allowing subclasses to define parts of an algorithm without modifying the overall structure of the algorithm.***

The template method pattern factors out the common code among its subclasses and puts them into the abstract class. The variable parts of the algorithm are left for the subclasses to override. These parts are template methods. A ***template method*** defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. The ordering of the steps is fixed by the abstract class. Usually, the algorithm is represented as a series of methods which are then invoked in the desired sequence in another method. Note that the classes may choose to ignore overriding certain steps or choose to rely on the default implementation provided by the abstract class. The abstract class may want to forbid the subclasses from overriding behavior for some steps, it can enforce this constraint by marking the methods implementing those

can enforce this constraint by marking the methods implementing those steps as `final`.

The opportunity afforded to subclasses for overriding some steps of the algorithm is through methods called **hooks**. A hook denotes an optional step for the subclass to override whereas a method marked as `abstract` forces the subclasses to provide an implementation for the step.

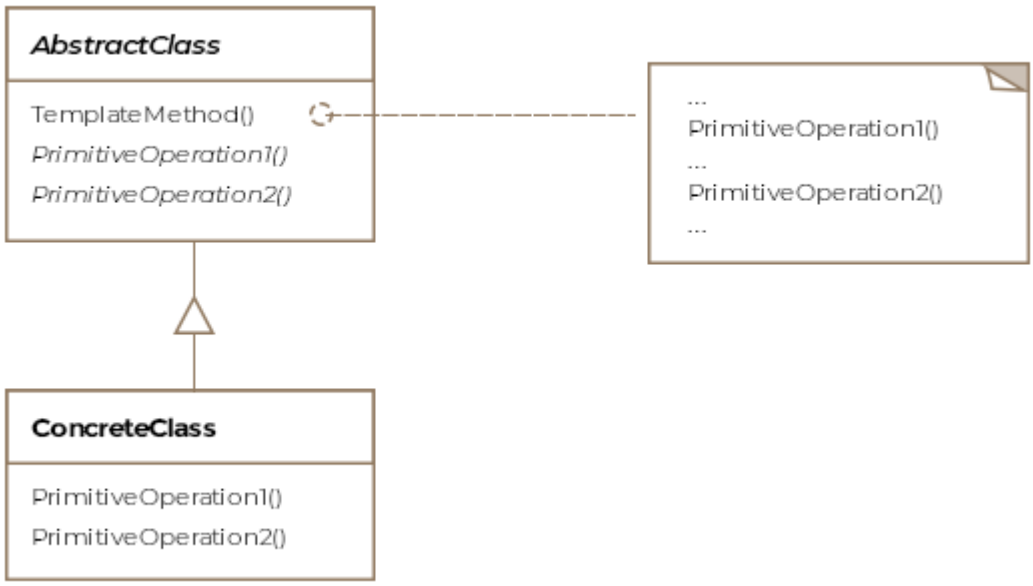
The template pattern method is very suitable for frameworks. A framework generally defines the application control flow and gives developers the opportunity to override certain methods to customize the flow of their application needs. For instance, the very popular message queuing framework Kafka allows developers to set the `UncaughtExceptionHandler` method and give their application a chance to react to an uncaught exception. The developer can choose to ignore and not set any handler. The template method pattern doesn't need to follow the textbook description of the pattern and can deviate to fit in the given context but the underlying spirit should conform to the intent of the pattern description.

One of the benefits apart from code reuse of the template method pattern is that the higher level components don't depend on lower level components and call the lower level components as and when required. When entities at various levels depend horizontally and vertically on various other entities, it becomes what is called a **Dependency Rot**. The pattern helps avoid the dependency rot by making the lower level components (subclasses) depend on the higher level abstract class.

## Class Diagram

The class diagram consists of the following entities

- **Abstract Class**
- **Concrete Class**




## Example

Let's take our aircraft example. Before each flight there's a list of tasks the pilots must go through that is called the "pre-flight checklist". You can well imagine that for most of the aircrafts this list would have a lot of commonalities. Therefore, it makes sense to model an abstract class that captures all the tasks in the preflight checklist in the order they should be performed.

```
public abstract class AbstractPreFlightCheckList {

    // This method captures the template or the skeleton
    // of the algorithm for the pre-flight checklist.
    final public void runChecklist() {

        // Check fuel guage
        IsFuelEnough();

        // Check doors are locked
        doorsLocked();

        // Check air pressure
        checkAirPressure();
    }

    // Don't let subclasses override this method, this is a
```

```
    // mandatory check
    final protected void IsFuelEnough() {
        System.out.println("check fuel gauge");
    }

    // Some airplanes may or may not have doors so allow this
    // method to be overridden by subclasses
    protected boolean doorsLocked() {
        return true;
    }

    // Force subclasses to provide their own way of checking for
    // cabin or cockpit air pressure
    abstract void checkAirPressure();
}
```

The F-16 preflight checklist would then look like

```
public class F16PreFlightCheckList extends AbstractPreFlightCheckLis
t {

    @Override
    void checkAirPressure() {
        // Implement the custom logic for checking cockpit
        // air pressure for F-16
    }

    @Override
    protected boolean doorsLocked() {
        // F-16 unlike a Boeing-747 has no doors
        // so always return true;
        return true;
    }
}
```

Since an F-16 doesn't have doors that need to be locked, it conveniently ignores providing an implementation for the method `doorsLocked` and relies on the default implementation of the abstract class to take the right action.

Other Examples

- Java's applets have gone down in popularity but the applet framework exposed a number of hooks for the developers. For instance, the `start` method gave the application a chance to take action before the applet just got displayed in the browser.
- The java.io package has an abstract `read()` method in `InputStream` that subclasses must implement and is in turn invoked by the method `read(byte arg1[], int offet, int length)`.
- The class `javax.servlet.http.HttpServlet` has a bunch of methods `doGet`, `doPost` and `doPut` etc, that can be overriden by implementing classes.

## Caveats

- Don't confuse the template method pattern with the *strategy pattern*. Strategy pattern uses composition by accepting objects that define the entire algorithm, whereas the template pattern method uses inheritance to vary parts of the algorithm by subclasses but the outline and structure of the algorithm is still the realm of the abstract class.
- Factory method pattern is a specialization of the template method pattern.
- Ideally, the number of methods for which the subclasses need to provide implementation should be minimized when applying the template method pattern.

# Strategy Pattern

This lesson discusses how a set of policies, algorithms or strategies can be made interchangeable without affecting the clients using them.

## What is it ?

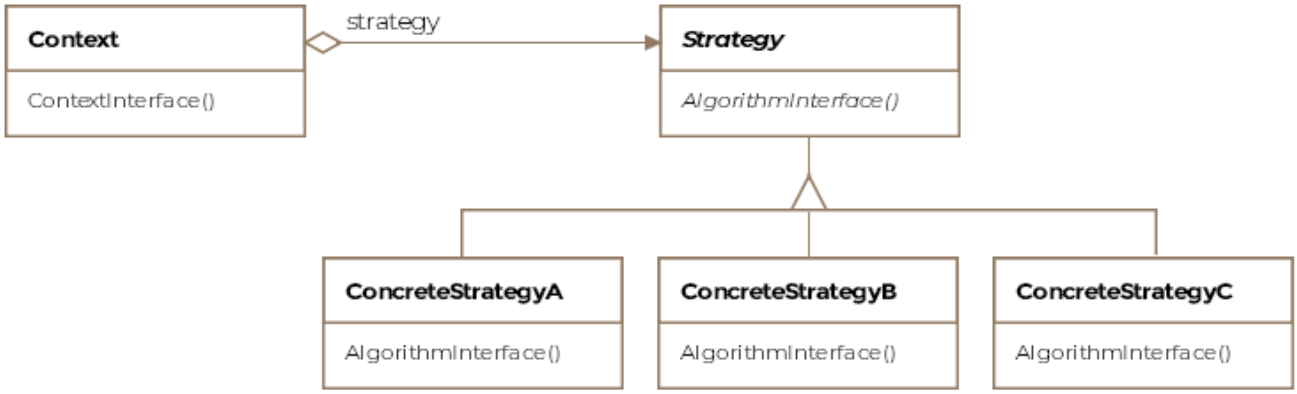
The strategy pattern is one of the simpler patterns to comprehend. It allows grouping related algorithms under an abstraction, which the client codes against. The abstraction allows switching out one algorithm or policy for another without modifying the client.

The strategy pattern is formally defined as ***encapsulating algorithms belonging to the same family and making them interchangeable. The consumers of the common interface that the algorithms implement allow switching out one algorithm for another seamlessly.***

## Class Diagram

The class diagram consists of the following entities

- **Strategy**
- **Concrete Strategy**
- **Context**




## Examples

Concrete algorithms implement the same interface. The **context** has the data the algorithm will act on. Together the context and the strategy interact to implement the chosen algorithm. Usually, clients instantiate the context and pass in the strategy object and then, only interact with the context object.

The most trivial example one can think of is the family of *sorting algorithms*. Say our application is only concerned with sorting integer arrays. All the sorting algorithms can implement a common interface, that we call `ISort`

```
public interface ISort {

    void sort(int[] input);
}
```

The concrete implementations of the interface appear below:

```
public class BubbleSort implements ISort {

    @Override
    public void sort(int[] input) {
        // Do inefficient sorting in order n squared
    }
}
```

```
public class MergeSort implements ISort {


    @Override
    public void sort(int[] input) {
        // Do efficient sorting in nlogn
    }
}
```

The `Context` class holds a reference to the strategy object and when it receives requests from its clients, it forwards them to the strategy object along with the required data.

```
public class Context {

    private ISort howDoISort;

    public Context(ISort howDoISort) {
        this.howDoISort = howDoISort;
    }

    // Context receives the data from its client
    // and passes it on to the strategy object.
    void sort(int[] numbers) {
        howDoISort.sort(numbers);
    }

    // We can change the sorting algorithm using this setter
    public void setHowDoISort(ISort howDoISort) {
        this.howDoISort = howDoISort;
    }
}
```

The client can use the context like so:

```
public class Client {

    private int[] numbers = new int[1000];

    void crunchingNumbers() {

        // Choose the sorting strategy
        BubbleSort bubbleSort = new BubbleSort();
```

```
        // Context receives the strategy object
        Context context = new Context(bubbleSort);

        // Sort the numbers
        context.sort(numbers);

        // Do remaining work
    }
}
```

Notice how the context keeps an interface reference and can be configured with any concrete sort implementation. The implementation can also be changed at runtime through the setter. This decoupling of the client and the algorithms, allows us to vary the algorithms independent of the client.

## Other Examples

- For our aircraft scenario, we can think about the different ways an F-16 can be equipped with weapons before each mission. An F-16 can go for reconnaissance without carrying any weapons, it can be loaded with (God forbid) nuclear weapons or it can carry Sidewinder missiles to intercept incoming enemy fighter jets. When modeling this scenario in our code, we could create a `ArmingStrategy` interface which will have concrete implementations of `NoWeapons`, `NuclearWeapons` and `AirToAirWeapons` as arming strategies for the plane. Before the F16 flies each mission we can set the `armingStrategy` variable held in the `F16` class with the desired arming strategy for the mission.

- `java.util.Comparator` has the method `compare` which allows the user to define the algorithm or strategy to compare two objects of the same type.

- Think how a text editor such as Microsoft Word can make use of the strategy pattern when a client chooses the paragraph alignment options. The strategies could be ***justify text***, ***left-align***, ***right-align*** or ***center-align***.

## Caveats

- The context can either pass the required data or itself to the strategy object. In the latter case, the context would expose methods on itself so that the strategy object can retrieve the required data.
- Strategy objects are good candidates to be implemented as ***flyweight*** objects. This can reduce the memory requirements for the application.
- The context class can be simplified by providing a default strategy and only burdening the clients to provide a strategy object, when the default doesn't meet their requirements.

# Visitor Pattern

This lesson discusses the visitor pattern which adds new behavior to composites without modifying the composite's or it's underlying elements' classes.

## What is it ?

The visitor pattern allows us to define an operation for a class or a class hierarchy without changing the classes of the elements on which the operation is performed.

Recall the `Airforce` class example from the *Composite Pattern* lesson. The `Airforce` class is a composite consisting of several different kinds of airplanes. It can be thought of as the *object structure* on whose elements we want to conduct operations. The *elements* would be the individual planes that make up the airforce object structure.

Say if we are tasked with monitoring of various metrics for each aircraft such as remaining fuel, altitude and temperature then one option would be to build this functionality inside the abstract class of all the airplanes. The consequence would be that we'll need to implement the new methods in all the airplane subclasses. Now imagine, a few days later we are tasked with calculating the total price tag for the Airforce. We will now add another method to the abstract airplane class or interface that'll return the price for each individual plane and sum it across all the airplanes.

There are several problems in our scenario, first the airplane class shouldn't be responsible for monitoring or pricing data. It should just represent the aircraft. With each additional functionality, we'll end up bloating our aircraft classes with new unrelated methods. The visitor patterns lets us out of this dilemma by suggesting to have a separate class that defines the new functionality related to the aircraft. The methods in the `AircraftVisitor` class would take the aircraft object as an argument

the `AircraftVisitor` class would take the aircraft object as an argument and work on it. This saves us from changing our aircraft classes each time we need to support a new functionality relating to the `Airforce` class.

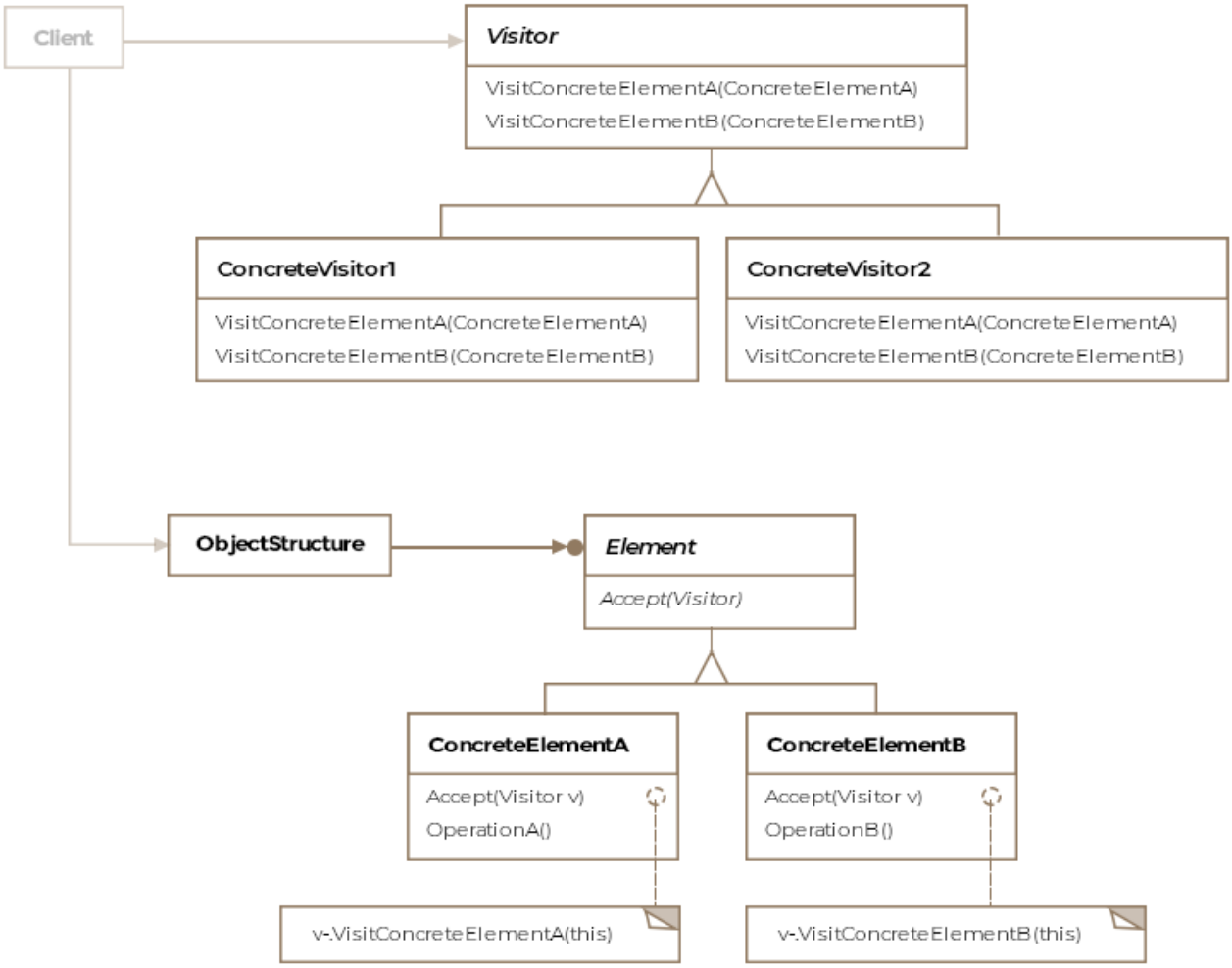
Formally, the pattern is defined as ***defining operations to be performed on elements of an object structure without changing the classes of the elements it works on.***

The pattern is suitable in scenarios, where the object structure class or the classes that make up its elements don't change often but new operations over the object structure are desired.

## Class Diagram

The class diagram consists of the following entities

- **Visitor**
- **Concrete Visitor**
- **Element**
- **Concrete Element**
- **Object Structure**




Class Diagram

## Example

Let's revisit our airforce example. The airforce class is the object structure on which we desire to introduce two new operations, one for collecting metrics for all the planes and two the cost of each aircraft. Let's see the `Airforce`

```
public class Airforce {

    // Holds a collection of planes
    private Collection<IAircraft> planes = new ArrayList<>();

    // Returns an iterator to its collection of planes
    public Iterator<IAircraft> getIterator() {
        return planes.iterator();
    }
}
```

We'll consider two types of airplanes, the `F16` and `Boeing747`. The interface defines an abstract `accept(IAircraftVisitor visitor)` method that must be implemented by all derived classes. This method allows the visitor to access the concrete class's interface, as you'll shortly see. The listing comes below:

```
public interface IAircraft {

    // Each concrete element class is expected to
    // define the accept method
    public void accept(IAircraftVisitor visitor);

}

public class F16 implements IAircraft {

    @Override
    public void accept(IAircraftVisitor visitor) {
        visitor.visitF16(this);
    }
}

public class Boeing747 implements IAircraft{

    @Override
    public void accept(IAircraftVisitor visitor) {
        visitor.visitBoeing747(this);
    }
}
```

Now we'll define the interface `IVisitor` and the two concrete visitors .

```
public interface IAircraftVisitor {

    void visitF16(F16 f16);

    void visitBoeing747(Boeing747 boeing747);
}
```

Notice how the visitor interface defines a visit method for each of the concrete types that make up the object structure. Say if a new airplane C-

130 was added to the object structure then the `IAircraftVisitor` would need to introduce a new method `visitC130`. The visitor interface allows each aircraft to pass itself to the visitor by calling the correponding visit method for its class on the visitor object. The visitor classes are given below:

```
public class MetricsVisitor implements IAircraftVisitor {

    public void visitF16(F16 f16){
        // Logic to get metrics for F16
    }

    public void visitBoeing747(Boeing747 boeing747){
        // Logic to get metrics for Boeing 747
    }

    public void printAccumulatedResults(){

    }
}

public class PriceVisitor implements IAircraftVisitor{

    @Override
    public void visitF16(F16 f16) {
        // Logic to get price for F16
    }

    @Override
    public void visitBoeing747(Boeing747 boeing747) {
        // Logic to get price for Boeing 747
    }

    public void printAccumulatedResults(){

    }
}
```

Note that each visitor can invoke methods specific to each concrete class. Even though the two airplane classes share the same interface, the pattern allows us to work with classes that are unrelated or don't share a common interface. Finally, the client code will look like below:

```
public class Client {
```

```
    public void main(Airforce airforce) {

        Iterator<IAircraft> planes = airforce.getIterator();
        MetricsVisitor aircraftVisitor = new MetricsVisitor();

        while (planes.hasNext()){
            planes.next().accept(aircraftVisitor);
        }

        aircraftVisitor.printAccumulatedResults();
    }
}
```

If we want to define a new operation on the object structure, then it is as easy as adding a new visitor class.

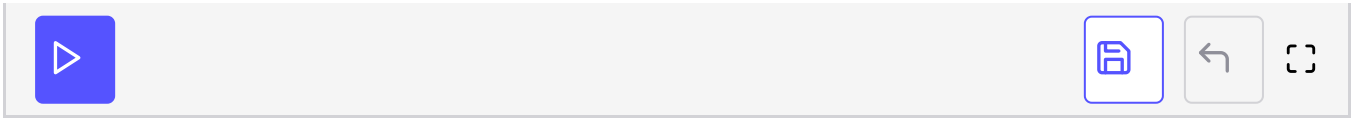
Each object structure will have an associated visitor class. This visitor interface will need to declare a visitConcreteElement operation for each class of concreteElement defining the object structure. Each visit method on the visitor interface will need to declare its argument to be a particular concreteElement, allowing the visitor to access the interface of the concreteElement directly.

## Double Dispatch

Take a look at the following code snippet and run it. Even though we save the reference for the `BetterF16` object in a variable of the super class `F16`, the outputs are printed for each of the object types correctly. This is an example of dynamic dispatch where Java determines at runtime what class an instance belongs to and chooses the appropriate, possibly overridden, method.

```
class Demonstration {
    public static void main( String args[] ) {
        F16 f16 = new F16();
        f16.whoAmI();

        System.out.println();
```

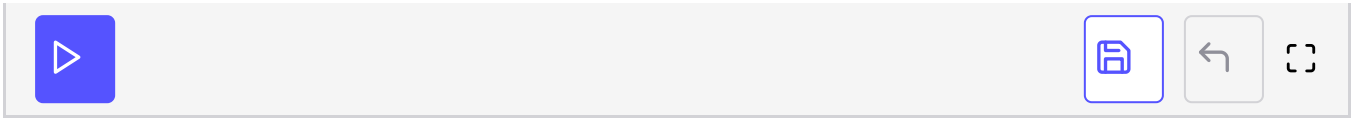
```
        // Reference for the derived object
        // is held in the superclass type
        F16 betterF16 = new BetterF16();
        betterF16.whoAmI();
    }
}

class F16 {

    public void whoAmI(){
        System.out.print("I am the mighty F-16.");
    }
}

class BetterF16 extends F16 {

    public void whoAmI(){
        System.out.print("I am the better than the mighty F-16.");
    }
}
```



Now consider the below code snippet. We add a method `fire()` which takes in an object of type `Missile`. We overload the `fire` method with an object of a derived class `BetterMissile`.

```
class Demonstration {
    public static void main( String args[] ) {
        F16 f16 = new F16();
        F16 betterF16 = new BetterF16();
        Missile missile = new Missile();
        Missile betterMissile = new BetterMissile();

        System.out.println("Expected output");
        f16.fireMissile(missile);
        betterF16.fireMissile(missile);
        System.out.println();

        System.out.println("Failed double dispatch attempt");
        f16.fireMissile(betterMissile);
        betterF16.fireMissile(betterMissile);
        System.out.println();

        System.out.println("Expected output");
        BetterMissile reallyBetterMissile = new BetterMissile();
        f16.fireMissile(reallyBetterMissile);
        betterF16.fireMissile(reallyBetterMissile);
        System.out.println();
    }
}
```

```
class BetterMissile extends Missile {

    @Override
    public String explode() {
        return " very very big baaam";
    }
}

class Missile {

    public String explode() {
        return " baaaam";
    }
}

class BetterF16 extends F16 {

    public String whoAmI() {
        return "Better F16";
    }
}

class F16 {

    public String whoAmI() {
        return "F16";
    }

    public void fireMissile(Missile missile) {
        System.out.println(whoAmI() + " fired ordinary missile: " + missile.explode());
    }

    public void fireMissile(BetterMissile missile) {
        System.out.println(whoAmI() + " fired better missile: " + missile.explode());
    }
}
```



If you run the above code the **lines 14-15** call the `fire` with a reference of type `Missile` pointing to an object of type `BetterMissile`. The JVM doesn't check the type of the object at runtime and invokes `fire( Missile )` instead of `fire( BetterMissile )`. Languages which support double dispatch or multiple dispatch, would have invoked the right intended method `fire( BetterMissile )`.

In the visitor pattern the `accept()` method simulates the double dispatch effect.

```
    public void accept(IAircraftVisitor visitor) {
```

```
        visitor.visitF16(this);
    }
```

The first dispatch is when an airplane object calls the accept method. If the airplane object is of type `F16` or `Boeing747`, the corresponding accept method on those classes is called. The second dispatch happens when the visitor interface reference passed into the accept method correctly calls the corresponding visit method on the concrete visitor object the reference points to.

## Other Examples

- `java.nio.file.FileVisitor` interface has an implementation class of `SimpleFileVisitor` which is an example of a visitor. The interface is defined as a visitor of files. An implementation of this interface is provided to the `Files.walkFileTree` methods to visit each file in a file tree.
- `javax.lang.model.element.Element` interface represents a program element such as a package, class, or method. To implement operations based on the class of an Element object not known at compile an implementation of the `javax.lang.model.element.ElementVisitor` interface is required.

## Caveats

- The visitor pattern cautions that if one expects the object structure classes to change often then it might be a good idea to just keep the new functionality within the visited classes instead of using the visitor pattern. The key consideration in applying the Visitor pattern is if the algorithm applied over an object structure is more likely to change or the classes of objects that make up the structure. The

visitor class hierarchy can be difficult to maintain when new concrete element classes are added frequently. In such cases, it's probably easier just to define operations on the classes that make up the structure. If the Element class hierarchy is stable, but you are continually adding operations or changing algorithms, then the visitor pattern will help you manage the changes.

- Adding new concrete classes will require modifying all the visitor classes, which makes it hard to add new types to the object structure.
- Iteration over the object structure can happen via an iterator, inside the object structure or by the visitor.
- Note that an iterator requires that a composite be made up of elements that all conform to the same base class or interface, whereas a visitor can visit all the elements of a composite even if they are unrelated.
- Languages that support double or multiple dispatch lessen the need for the visitor pattern.

# Summary

Summary

| Pattern | Purpose |
| --- | --- |
| Builder Pattern | The builder pattern is used to create objects. It seperates out how the object is represented and how it is created. Additionally, it breaks down the creation into multiple steps. For instance in Java the `java.lang.StringBuilder` is an example of the builder pattern. |
| Singleton Pattern | The singleton pattern is applied to restrict instantiation of a class to only one instance. For instance in the Java language the class `java.lang.Runtime` is a singleton. |
| Prototype Pattern | Prototype pattern involves creating new objects by copying existing objects. The object whose copies are made is called |

| | |
|---|---|
| | the **prototype**. In Java the `clone()` method of `java.lang.Object` is an example of this pattern. |
| Factory Method Pattern | The factory method is defined as providing an interface for object creation but delegating the actual instantiation of objects to subclasses. For instance the method `getInstance()` of the class `java.util.Calendar` is an example of a factory method pattern. |
| Abstract Factory | The abstract factory pattern is defined as defining an interface to create families of related or dependent objects without specifying their concrete classes.The abstract factory is particularly useful for frameworks and toolkits that work on different operating systems. For instance, if your library provides fancy widgets for the UI, then you may need a family of products that work on MacOS and a similar family of products that work on Windows. |
| Adapter Pattern | The Adapter pattern allows two incompatible classes to |

| | |
|---|---|
| | interoperate that otherwise can't work with eachother. Consider the method `asList()` offered by `java.util.Arrays` as an exampe of the adapter pattern. It takes an array and returns a list. |
| Bridge Pattern | The bridge pattern describes how to pull apart two software layers fused together in a single class hierarchy and change them into parallel class hierarchies connected by a bridge |
| Composite Pattern | The pattern allows you to treat the whole and the individual parts as one. The closest analogy you can imagine is a tree. The tree is a recursive data-structure where each part itself is a sub-tree except for the leaf nodes. |
| Decorator Pattern | The decorator pattern can be thought of as a wrapper or more formally a way to enhance or extend the behavior of an object dynamically. The pattern provides an alternative to subclassing when new functionality is desired. A prominent example of this pattern is the `java.io` package, which includes several |

| | |
|---|---|
| | decorators. For example the `BufferedInputStream` wraps the `FileInputStream` to provide buffering capabilities. |
| Facade Pattern | The facade pattern is defined as a single uber interface to one or more subsystems or interfaces intending to make use of the subsystems easier |
| Flyweight Pattern | The pattern advocates reusing state among a large number of fine grained object. Methods `java.lang.Boolean.valueOf()` and `java.lang.Integer.valueOf()` both return flyweight objects. |
| Proxy Pattern | In a proxy pattern setup, a proxy is responsible for representing another object called the subject in front of clients. The real subject is shielded from interacting directly with the clients. The `java.rmi.*` package contains classes for creating proxies. RMI is Remote Method Invocation. It is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine. |

| | |
|---|---|
| Chain of Responsibility Pattern | In a chain of responsibility pattern implementation, the sender's request is passed down a series of handler objects till one of those objects, handles the request or it remains unhandled and falls off the chain. Multiple objects are given a chance to handle the request. This allows us to decouple the sender and the receiver of a request. The `log()` method of the `java.util.logging.Logger` class is an example of this pattern. |
| Observer Pattern (Publisher/Subscriber) | The pattern is formally defined as a one to many dependency between objects so that when one object changes state all the dependents are notified. All types implementing the interface `java.util.EventListener` are examples of this pattern. |
| Interpreter Pattern | The interpreter pattern converts a language's sentences into its grammar and interprets them. |
| Command Pattern | The pattern is defined as representing an action or a request as an object that can then be passed to other objects as |

| | |
|---|---|
| | parameters, allowing parameterization of clients with requests or actions. The requests can be queued for later execution or logged. Logging requests enables undo operations. Types implementing the interface `java.lang.Runnable` are examples of this pattern. |
| Iterator Pattern | An iterator is formally defined as a pattern that allows traversing the elements of an aggregate or a collection sequentially without exposing the underlying implementation. All types implementing the `java.util.Iterator` interface are examples of this pattern. |
| Mediator Pattern | The pattern is applied to encapsulate or centralize the interactions amongst a number of objects. Object orientated design may result in behavior being distributed among several classes and lead to too many connections among objects. The encapsulation keeps the objects from referring to each other directly and the objects don't hold references to each other anymore. The `java.util.Timer` class represents this pattern |

| | |
|---|---|
| | where tasks may be scheduled for one-time execution, or for repeated execution at regular intervals in a background thread. |
| Memento Pattern | The memento pattern let's us capture the internal state of an object without exposing its internal structure so that the object can be restored to this state later. Classes implementing `java.io.Serializable` interface are examples of the memento pattern. |
| State Pattern | The state pattern encapsulates the various states a machine can be in. The machine or the context, as it is called in pattern-speak, can have actions taken on it that propel it into different states. Without the use of the pattern, the code becomes inflexible and littered with if-else conditionals. |
| Template Method | The template method pattern defines the skeleton or steps of an algorithm but leaves opportunities for subclasses to override some of the steps with their own implementations. Non-abstract methods of |

| | |
|---|---|
| | `java.util.AbstractList`,<br><br>`java.util.AbstractSet` and `java.util.AbstractMap` are examples of this pattern. |
| Strategy Pattern | The pattern allows grouping related algorithms under an abstraction, which the client codes against. The abstraction allows switching out one algorithm or policy for another without modifying the client. `java.util.Comparator` has the method `compare()` which allows the user to define the algorithm or strategy to compare two objects of the same type. |
| Visitor Pattern | The visitor pattern allows us to define an operation for a class or a class hierarchy without changing the classes of the elements on which the operation is performed. The pattern is suitable in scenarios, where the object structure class or the classes that make up its elements don't change often but new operations over the object structure are desired. `java.nio.file.FileVisitor` interface has an implementation class of `SimpleFileVisitor` which is an example of a visitor. The |

| | |
|---|---|
| | interface is defined as a visitor of files. An implementation of this interface is provided to the `Files.walkFileTree()` methods to visit each file in a file tree. |

# Epilogue

Closing remarks

As we reach the end of the course, I'd like to share my experience working with patterns as a veteran of the Silicon Valley. First, you'll not always find patterns implemented as the text-book versions you just read. They'll usually deviate a little here or there, but knowing the text-book version would let you immediately recognize the pattern at hand and improve your understanding of the code and your ability to navigate and make changes to it.

The frequency with which you see patterns applied will depend on what company or product you work for. If you are working in an off-shore small scale wordpress-website-building sweat shop, chances are slim you'll work on any of the patterns. The cost benefits are just not there. If you are working on a website of ten pages and your customer will pay you $2000 then neither you nor your boss would be inclined to spend the effort and energy in going the extra mile and using nifty patterns in code. However, for a project of $200,000 you better use design patterns where applicable or your customer will not be happy when change requirements start to cost him a fortune because you need to cut and rip apart the code and start all over again.

If you work for a company that produces a framework or library for consumption by other developers then chances are you'll be working with and applying patterns all over the place. The further up in the food chain you work and the more uncertainty and abstraction you deal with, the more likely you are to use design patterns. However, at the very least, you'll find yourself working with creational design patterns at any company as any decent sized project will have you `new()` up objects.

The foremost benefit you should have from this course is to learn the

design pattern vocabulary. Being cognizant of what a *builder pattern* or a

*factory method pattern* refers to in a dev meeting exudes maturity and depth in a software engineer. Furthermore, it makes communication of complex designs amongst developers easy, leveraging the shared vocabulary.

Last but not the least, I would like to caution to not get overboard with design patterns and apply them for the sake of applying design patterns. They are a means to an end, not the end in themselves. The end being maintainable, extensible and flexible code. Keep it simple and apply patterns when and where it makes sense.

Wishing you all the best in your tech future.

**C. H. Afzal.**
**21st Sept, 2018**

***taught with passion and crafted with ♡ in Santa Clara, U.S.A.***

## Credits

Every great product is a result of team-effort and so is this course. Our team members include:

- Ahsan Khalil (Illustrations and graphic design)
- Educative's Proofreading Ninjas