ISYS2012 - Software Engineering 2

Semester 2013 B

Design Pattern Report

(Flyweight, Builder, Prototype, and Singleton Patterns)

NSST Game Development Team

Lecturer- Kevin Jackson

Phan Thanh San - s3342133

Bui Trong Nhan -s3275049

Hoang Ngoc Thanh - s3275145

Pham Dinh Son - s3260624

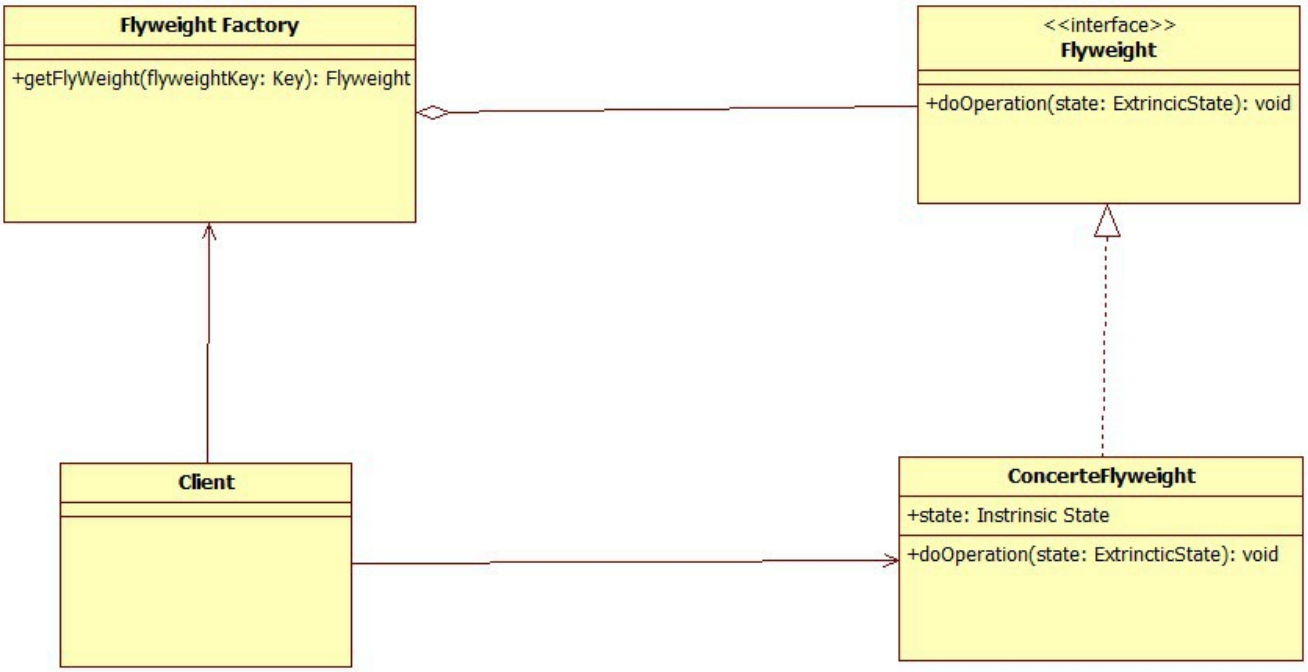
Flyweight Pattern

Motivation

- Flyweights Pattern is a design pattern which allows the reuse of memory space in an application when the program requires a large number of objects that have similarity to each others are created.
- This pattern help improve the performance of a program which perform low due to large number of similar heavy weight object being created during the process.

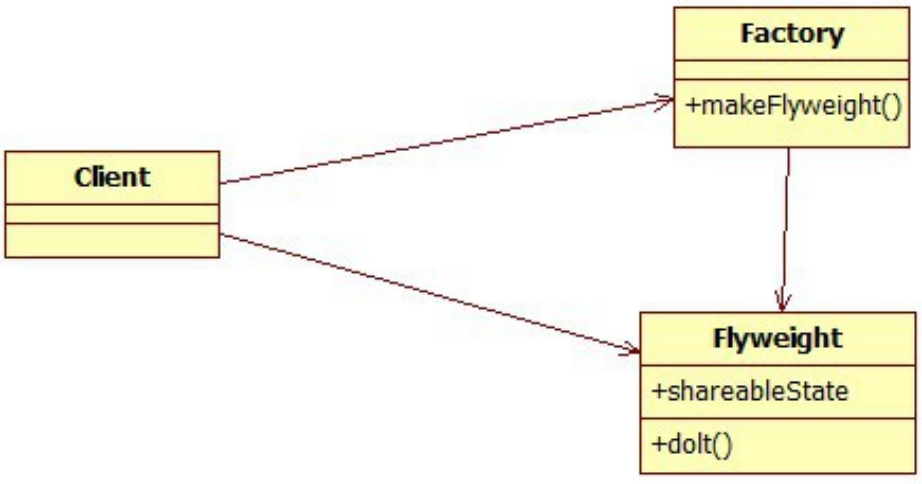
Structure

Below is the class diagram of a standard Flyweight Pattern



- In Flyweight pattern, the object internal state is divided into two main categories: intrinsic data and extrinsic data.
- The intrinsic data is the critical information that is required for a class to function properly. On the other hands, extrinsic data is the type of data which can be removed from class and stored it externally.

- This means by taking all the object which have the same intrinsic data and make it into a single shared object, the number of similar object will be reduced to unique intrinsic state.(1)
- A Factory is required to create the shared object. A new copy will only be created when an intrinsic state is different from what have been defined. The extrinsic state is stored using a manager object (1) When used they normally will be passed through argument.
- In the picture above, the flyweight is declared as an interface which helps receive and act on extrinsic state. The concrete Flyweight implements the flyweight interface and stored the intrinsic state. It is shareable and its object maintains the state that is intrinsic as well as able to manipulate the extrinsic state (2). The Flyweight Factory creates and manages the flyweight object as well as ensuring the sharing of these objects. The client maintains "references to flyweight in addition to computing and extrinsic state" (2).
- Flyweights cannot stands alone and are stored in a Factory's repository (5); the client cannot create a flyweight directly but rather request them from the Factory. Attribute that specified sharing must be given from client when a request is made (5)



Below is another diagram of flyweight pattern

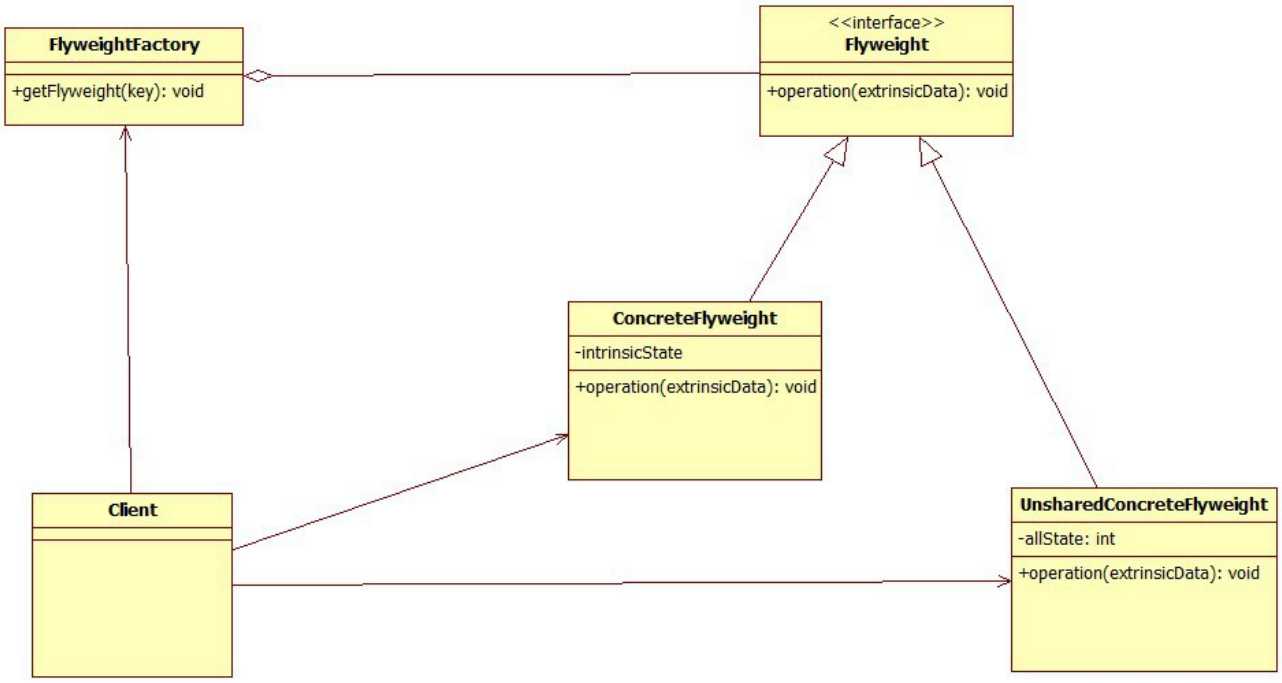


Chart from Java dzone

- As seen from the picture, the ConcreteFlyweight class is able to add capabilities for intrinsic. The object is shareable (6). The UnsharedConcreteFlyweight enable another way to use the Flyweight pattern without having the share concept as encourage in others object of Flyweight (6).

Implementation

Below is an example of Flyweight Pattern implementation. The program will be used to draw lines of different colors

Create a flyweight interface which the draw method will provide the extrinsic data where to draw line

```
//Flyweight
public interface LineFlyweight {
  public Color getColor();
  public void draw(Point location);
}
// Implementation
//ConcreteFlyweight
public class Line implements LineFlyweight {
  private Color color;
  public Line(Color c) {
     color = c;
  }
  public Color getColor() {
     return color;
  }
  public void draw(Point location) {
//draw the character on screen
  }
}

//The factory is used to manage the creation of line object.
//Flyweight factory
public class LineFlyweightFactory {
  private List<LineFlyweight> pool;
  public LineFlyweightFactory() {
     pool = new ArrayList<LineFlyweight>();
```

```
  }
  public LineFlyweight getLine(Color c) {
    //Check if we've already created a line with this color
     for (LineFlyweight line : pool) {
        if (line.getColor().equals(c)) {
           return line;
        }
     }

    //if not, create one and save it to the pool
     LineFlyweight line = new Line(c);
     pool.add(line);
     return line;
  }
}

//The client will use the factory if they want to create the line as below(6)
LineFlyweightFactory factory = new LineFlyweightFactory();
LineFlyweight line = factory.getLine(Color.RED);
LineFlyweight line2 = factory.getLine(Color.RED);
    ...
//can use the lines independently
line.draw(new Point(100,100));
line2.draw(new Point(200,100));
```

Pros and Cons of Flyweight Pattern

Benefit of using Flyweight Pattern:

- The Flyweight pattern can help reduce a large amount of page's resource loaded. (1)
- Doesn't require huge changes. Only change that will be making is to call the method of the manager object. (1)
- Only need slight alter if create Flyweight pattern to use as an API. (1)

- Efficient if make optimize once. (1)
- Improvement of speed (1)

Drawback of Using Flyweights Pattern:

- Flyweight only an optimize pattern, thus it only improve the performance and the efficiency of the code under a set of condition.(1)
- It will make the code less efficient if use wrong(1)
- Hard to debug because error now occurs at three place: factory, manager and flyweight.(1)
- Hard to maintain because it's optimize which leaves fragment with data store at 3 place.(1)
- Only use when optimization is needed, where system resource are almost utilized.(1)

When will it be Use

Flyweight pattern should be use when

- Many heavy weight objects is used (6)
- Storage cost is high (6)
- Majority of object can be make extrinsic (6)
- Few shared object can replace unshared (6)
- Identity of each other's does not matter (6)

Flyweight can be found applied in language such as C#, C++, Java or PHP (5)

Related Pattern:

- Factory and Singleton Pattern are usually used to create Flyweight Pattern so that each type or category of Flyweights is single instance returned. (2)
- State and Strategy Pattern are also included in Flyweight pattern as their objects are implemented as Flyweights. (2)

Alternative Pattern:

- Strategy Pattern can be considered as an alternative pattern to solve problems occurs for Flyweight pattern in situation where classes differ only from their own behavior. In a way it can be said that flyweight and strategy pattern both have a same ideal in optimizing the code by differ the differences and make a single similar class.
- The goal of Strategy Pattern is to define and encapsulate each one in a family of algorithm and make them interchangeable.

Below is a diagram example of strategy pattern:

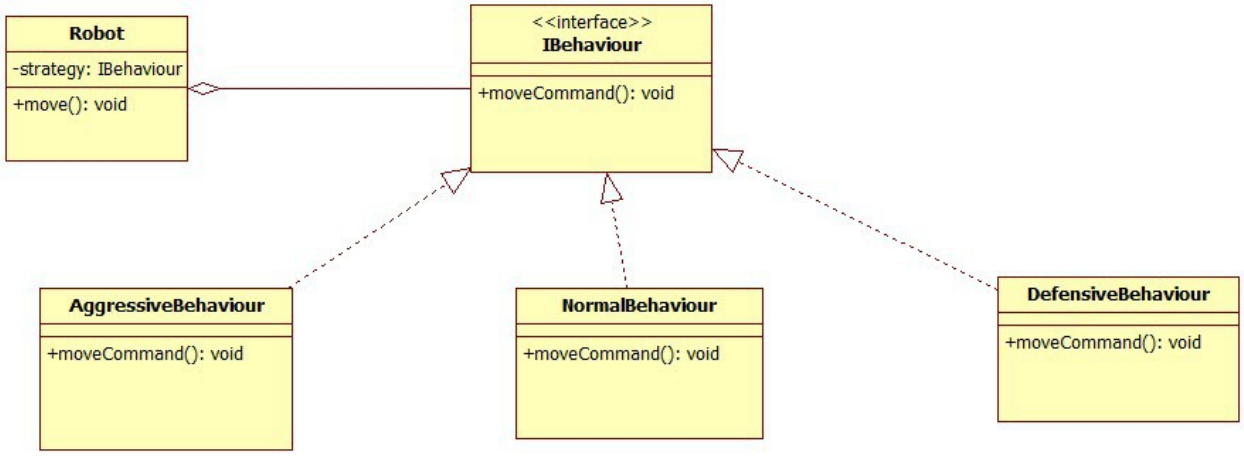


Chart from OOdesign

The strategy pattern splits the behavior of the class from the class itself. However this can be a disadvantage since client has implement issues if strategy pattern does not understand well. (2)

Implementation of Strategy pattern:

```
public interface IBehaviour {
        public int moveCommand();
}
public class AgressiveBehaviour implements IBehaviour {
        public int moveCommand() {
                System.out.println("AgressiveBehaviour: if find another robot attack it");
                return 1;
        }
}
public class DefensiveBehaviour implements IBehaviour {
        public int moveCommand() {
```

```
            System.out.println("Defensive Behaviour: if find another robot run from it");
            return -1;
        }
}
public class NormalBehaviour implements IBehaviour {
        public int moveCommand() {
            System.out.println("Normal Behaviour: if find another robot ignore it");
            return 0;
        }
}
public class Robot {
        IBehaviourbehaviour;
        String name;
        public Robot(String name) {
            this.name = name;
        }
        public void setBehaviour(IBehaviourbehaviour) {
            this.behaviour = behaviour;
        }
        Public IBehaviour getBehaviour() {
            Return behaviour;
        }
        public void move() {
            System.out.println(this.name + ": Based on current position" +
                        "thebehaviour object decide the next move:");
            int command = behaviour.moveCommand();
            // ... send the command to mechanisms
            System.out.println("The result returned by behaviour object " +
                        "is sent to the movement mechanisms " +
                        " for the robot '"  + this.name + "'");
        }
```

```
        public String getName() {return name;}
        public void setName(String name) {
                this.name = name;
        }
}
public class Main {
    public static void main(String[] args) {
        Robot r1 = new Robot("Big Robot");
         Robot r2 = new Robot("George v.2.1");
        Robot r3 = new Robot("R2");
        r1.setBehaviour(new AgressiveBehaviour());
        r2.setBehaviour(new DefensiveBehaviour());
        r3.setBehaviour(new NormalBehaviour());
        r1.move();
        r2.move();
        r3.move();
        System.out.println("\r\nNewbehaviours: " + "Big Robot gets really scared" +
           "'George v.2.1' becomes really mad because" +"it's always attacked by other robots" +
           "and R2 keeps its calm\r\n");
        r1.setBehaviour(new DefensiveBehaviour());
        r2.setBehaviour(new AgressiveBehaviour());
        r1.move();
        r2.move();
        r3.move();
   }
}
```

Using strategy can solve problems regards of application function if problems occur with memory issues. However once strategy pattern is applied into the code, it is hard to make changes in the future, especially with system with clients and regular update.

Builder design pattern (*An object creational pattern*)

Motivation

Hard to control the creation of complex objects that are made of parts from other complex objects. This could lead to serious coupling problems

Intent

- Encapsulate the construction of complex objects. Objects are independent from others.
- Construct objects through an interface

Consequences

- Builder design pattern is used to separate the creation of object from its representation and allow objects to be constructed with several components.
- It is also be used to create several different representation of objects with the same creation process.
- Easily controlled product creation process through director class.

Implementation

Practical demonstration: Building different types of car

CarSupervior: director

CarBuilder: abstract builder

Car: product

Concrete builder: SportCar, HydridCar, LuxuryCar, SmallCar, MediumCar, LargeCar

Director

```
public class CarSupervisor {
   private CarBuilder carBuilder;
   public void setCarBuilder(CarBuilder carBuilder) {
      this.carBuilder = carBuilder;
   }
   public Car getCar(){
      return carBuilder.getCar();
   }
   public void constructCar(){
      carBuilder.createNewCar();
      carBuilder.buildDoors();
      carBuilder.buildWindows();
      carBuilder.buildSeats();
   }
}
```

Abstract builder

```
public abstract class CarBuilder {
   protected Car car;
   protected String customer;
   protected String name;
   public void createNewCar(){
    car = new Car(name,customer);
   }
   public Car getCar() {return car;}
   public String getCustomer() { return customer;}
   public String getName() {return name;}
   public abstract void buildWindows();
   public abstract void buildDoors();
   public abstract void buildSeats();
}
```

Product

```
public class Car {
   private String name;
   private int seat;
   private int doors;
   private int windows;
   private String customer;

   public Car(String name, String customer) {
      this.name = name;
      this.customer = customer;
   }
   public String getName() {return name;}
   public void setName(String name) {
      this.name = name;
```

```
    }
    public int getSeat() { return seat;}
    public void setSeat(int seat) {
        this.seat = seat;
    }
    public int getDoors() {return doors;}
    public void setDoors(int doors) {
        this.doors = doors;
    }

    public int getWindows() {return windows;}
    public void setWindows(int windows) {
        this.windows = windows;
    }
    public String getCustomer() { return customer;}
    public void setCustomer(String customer) {
        this.customer = customer;
    }
}
```

Concrete builder #1

```
public class LargeCar extends CarBuilder{
    public LargeCar(String n,String cust) {
        super.name=n;
        super.customer=cust;
    }
    @Override
    public void buildWindows() {
        car.setWindows(8);
    }
    @Override
```

```
  public void buildDoors() {
     car.setDoors(3);
  }
  @Override
  public void buildSeats() {
     car.setSeat(30);
  }
}
```

Concrete builder #2

```
public class SmallCar extends CarBuilder{
  public SmallCar(String n,String cust) {
     super.name=n;
     super.customer=cust;
  }
  @Override
  public void buildWindows() {
     car.setWindows(4);
  }
  @Override
  public void buildDoors() {
     car.setDoors(4);
  }
  @Override
  public void buildSeats() {
     car.setSeat(4);
  }
}
```

Concrete builder #3

```
public class MediumCar extends CarBuilder{
   public MediumCar(String n,String cust) {
      super.name=n;
      super.customer=cust;
   }
   @Override
   public void buildWindows() {
      car.setWindows(4);
   }

   @Override
   public void buildDoors() {
      car.setDoors(4);
   }
   @Override
   public void buildSeats() {
      car.setSeat(4);
   }
}
```

Concrete builder #4

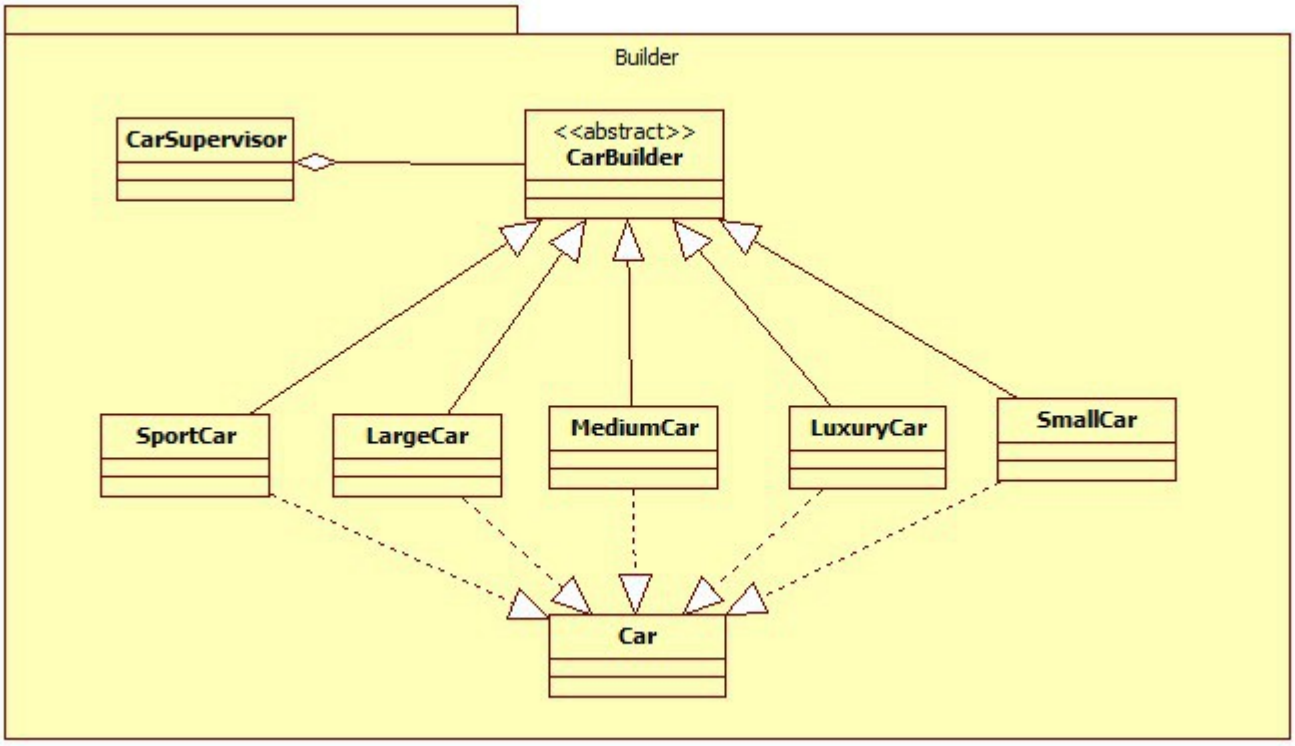
```
public class SportCar extends CarBuilder{
   public SportCar(String n,String cust) {
      super.name=n;
      super.customer=cust;
   }
   @Override
   public void buildWindows() {
     car.setWindows(2);
   }
```

```
  @Override
  public void buildDoors() {
   car.setDoors(2);
  }
  @Override
  public void buildSeats() {
    car.setSeat(2);
  }
}
```

Concrete builder #5

```
public class LuxuryCar extends CarBuilder{
  public LuxuryCar(String n,String cust) {
     super.name=n;
     super.customer=cust;
  }
  @Override
  public void buildWindows() {
     car.setWindows(4);
  }
  @Override
  public void buildDoors() {
     car.setDoors(4);
  }
  @Override
  public void buildSeats() {
  car.setDoors(5);
  }
}
```

Overall diagram



Alternative solution

- If considering the creating an complex object from other objects, the abstract factory can be an alternative for using builder pattern. Both builder pattern and abstract factory pattern can encapsulate an object creation into one method (or class).

- As mentioned above, the builder pattern focuses on building complexity object step by step, and there is a director for the creation, while the abstract factory pattern tends to create family of related objects. If using abstract factory pattern but considering the related objects are parts of one complex object, it can be an alternative for the builder pattern.

Here is the alternative of practical scenario above that using abstract factory pattern.

AbstractFactory

```
public abstract class CarFactory {
    public abstract Car createCar();
}
```

ConcreteFactory

```
public class LargeCarFactory extends CarFactory {
  @Override
  public Car createCar() {
     return new LargeCar(3,8,30);
  }

public class SmallCarFactory extends CarFactory {
  @Override
  public Car createCar() {
     return new SmallCar(4,4,4);
  }
}

public class SportCarFactory extends CarFactory {
  @Override
  public Car createCar() {
     return new SportCar(2,2,2);
  }
}
```

AbstractProduct

```
public abstract class Car {
   protected int seat;
   protected int door;
   protected int window;

   public int getSeat() { return seat;}
   public void setSeat(int seat) {
      this.seat = seat;
   }
   public int getDoor() { return door;}
   public void setDoor(int door) {
      this.door = door;
   }

   public int getWindow() { return window;}
   public void setWindow(int window) {
      this.window = window;
   }
   public String print() {
      return seat +":" + door +":" + window;
   }
}
```

Product

```
public class LargeCar extends Car {
   public LargeCar(int d,int w,int s) {
      door=d;
      window=w;
      seat=s;
   }
}
```

```
public class SmallCar extends Car {
  public SmallCar(int s,int w,int d) {
     seat =s;
     window = w;
     door = d;
  }
}
```

```
public class SportCar extends Car {
  public SportCar(int d,int w,int s) {
     door=d;
     window=w;
     seat=s;
  }
}
```

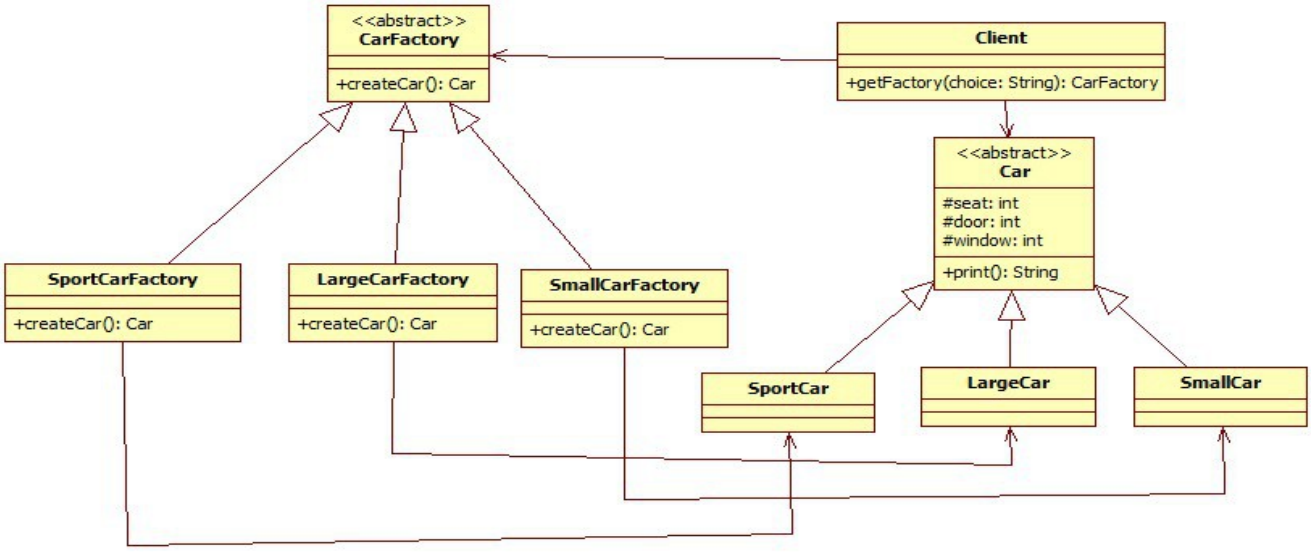
Client

```
public class Client {
  private static CarFactory carFactory=null;
  public static CarFactory getFactory(String choice){
    if (choice=="SportCar"){
      carFactory=new SportCarFactory();
    }  else if(choice=="SmallCar"){
       carFactory=new SmallCarFactory();
    } else if (choice=="LargeCar"){
       carFactory=new LargeCarFactory();
    }
     return carFactory;
  }
}
```

Main

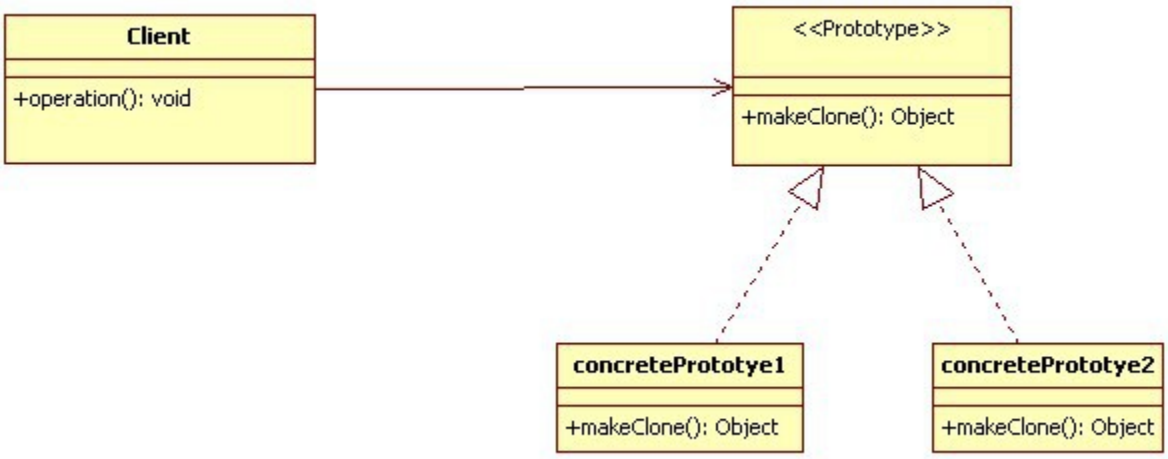
```
public class Main {
  public static void main(String[] args) {
        //Client client=new Client();
   CarFactory carFactorya=Client.getFactory("SportCar");
     Car car1=carFactory.createCar();
     String str=car1.print();
     System.out.println(str);



     carFactory=Client.getFactory("SmallCar");
     Car car2=carFactory.createCar();
     String str2=car2.print();
     System.out.println(str2);
  }
}
```

Overall diagram



Prototype Design Pattern

Class Diagram



Description

Name - Cloning an existing object

Problem

- When designing a project, improving performance is one of the most important aims the programmers need to consider. One part of designing project is handling with object creation. This because of the fact that, almost project requires new objects have to be created base on an object considered as a template or a model. If one or two objects will be created in future, it may not a problem. However, many objects have to be created; it is a big issue to the programmer.
- Adding and removing an object at run-time without initializing the object at starting of application.
- The problem here also is using “new” expression is harmful, and may reduce the performance of the project. Moreover, creating subclasses costs a lot of resources.

Solution

The better way of creating new object is cloning an object.

- Make a prototypical instance of object which has to be created.
- Make new object by cloning this prototype

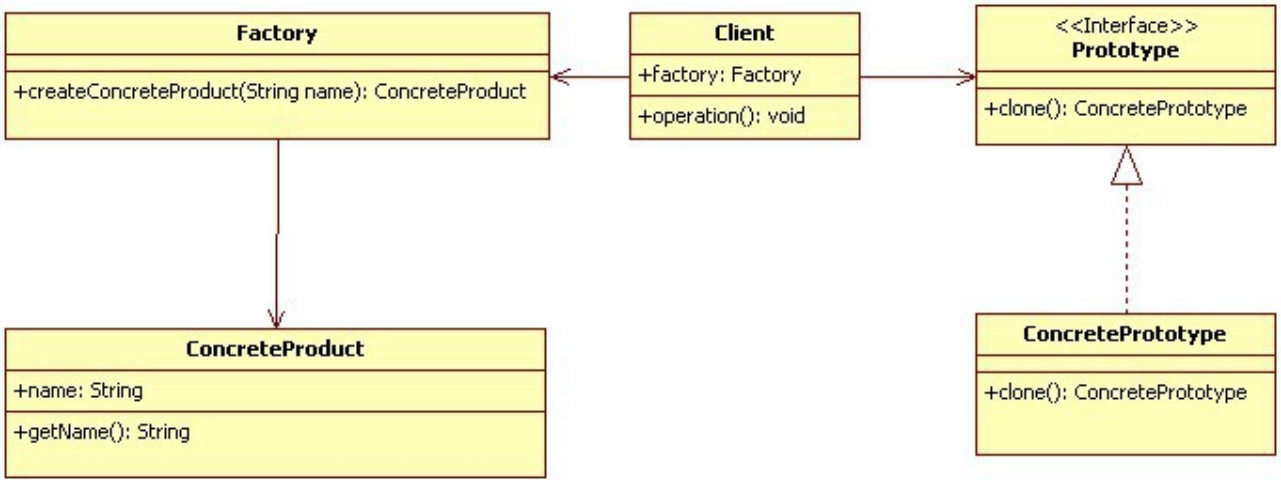
Sequences

- Avoiding “new” expression when creating new object because “new” expression is harmful
- Avoiding creating of many subclasses.
- Adding and removing products at run-time
- Having prototype manager to manage prototypes
- Implement the clone operation can be difficult
- Initializing clones

Alternative

- Based on the creation of objects point of view, factory pattern can be considered as an alternative for prototype pattern. With the factory pattern, the client can create any of derived objects without knowing the detail of the classes or how to create them. This is one of important strength points that prototype pattern handles.
- The difference between factory pattern and prototype pattern is the factory pattern concentrates on creating objects of a non existing object, while the prototype pattern creates an object based on an existing object. So, the factory pattern can alter for the prototype pattern if given specific attributes of the existing object need to be cloned, or simply instantiate the object instead of implementing method clone() from the interface

- Here is the alternative class diagram



Code example

```
public class ConcreteProduct {
        private String name;
        public ConcreteProduct(String name) {
                this.name = name;
        }
        public String getName() {return name;}
}
public class Factory {
        public ConcreteProduct createConcreteProduct(String name) {
                return new ConcreteProduct(name);
        }
}
```

```
public class Client {
	public static void main(String[] args) {
		Factory factory = new Factory();
		ConcreteProduct product1 = new ConcreteProduct("Product 1");
		//Instead of implementing product1.clone() through an interface
		ConcreteProduct product2 =
		factory.createConcreteProduct(product1.getName());
	}
}
```

Implementation

According to the diagram above, the prototype design pattern includes three participant classes

- Client: specifies prototypical instance of object, and create new object by asking prototype to clone itself
- Prototype: an interface for cloning
- ConcretePrototype: implement method clone() of Prototype interface.

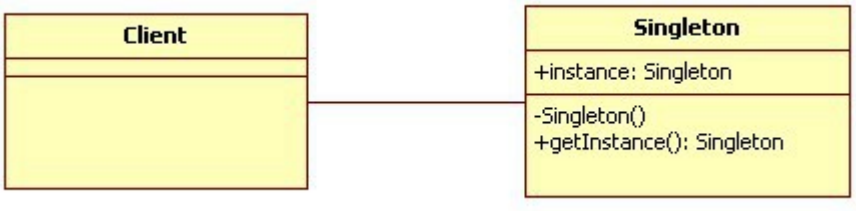
Here is the code of prototype pattern implementation in Java

```
public interface Prototype extends Cloneable {
  public Object makeClone();
}
public class ConcretePrototype1 implements Prototype {
  @Override
  public Object makeClone() {
	ConcretePrototype1 obj = null;
	try {
	obj = (ConcretePrototype1) super.clone();
	} catch (CloneNotSupportedException e) {
	e.printStackTrace();
	}
```

```
        return obj;
    }
}
public class ConcretePrototype2 implements Prototype {
    @Override
    public Object makeClone() {
        ConcretePrototype2 obj = null;
        try {
            obj = (ConcretePrototype2) super.clone();
        } catch (CloneNotSupportedException e) {e.printStackTrace();}
        return obj;
    }
}
public class Client {
    public static void main(String[] args) throws CloneNotSupportedException {
        ConcretePrototype1 obj1 = new ConcretePrototype1();
        ConcretePrototype1 obj2 = new ConcretePrototype2();
        ConcretePrototype1 obj1clone = (ConcretePrototype1)obj1.makeClone();
        ConcretePrototype1 obj2clone = (ConcretePrototype1)obj2.makeClone();
    }
}
```

Singleton Pattern

Class Diagram



Description

Name – Using unique instance for a class

Problem

- Application needs only one instance of objects. For example, thread pool, caches, dialog boxes, objects that handle preferences and registry setting, objects that use for logging, and objects that act as device drivers to device like printer and graphic cards. In these cases, if we have many instances, the application may take some problems such as overuse of resources, or inconsistent results, incorrect program behaviors.
- Lazy instantiation and global access are necessary. When an object is assigned as global variable, the object has to be created when starting application; however, this object is resource intensive and the application uses it many times

Solution

- With singleton pattern, a class can be ensured that only one instance is instantiated, and global point of access is provided.
- Define a private static attribute in "single instance" class
- Define all constructors to be private, or protected
- Define a public static accessor function to return the constructor instance in class.

Sequences

- Ensure having at most one instance of class in application
- Provide global access point to that instance
- Handle with multithreading
- Cannot subclass Singleton class because the constructor is private

3. Alternative

Singleton pattern in C++

```
class Singleton
{
    private:
```

```
        static Singleton *instance;
        //private constructor
        Singleton() {}


    public:
        static Singleton* getInstance();
};
Singleton* Singleton::instance = NULL;
Singleton* Singleton::getInstance() {
        if(! instance) {
            instance = new Singleton();
        }
        return instance;
}
```

Implementation

Here is the implementation of Singleton Design Pattern in Java

```
public class Singleton {
  private static Singleton instance;

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

This is basic implementation of singleton pattern. If the application has to deal with multithreading, the method getInstance() can be synchronized to force every thread to wait its turn before it can enter the method

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

However, using synchronizing may reduce the performance; we can move to an eagerly created instance rather than lazily created one

```
public class Singleton {
  private static Singleton instance = new Singleton();

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

Using double-checked locking to reduce the use of synchronization getInstance()

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

```
            }
            }
            }
            return instance;
    }
}
```

References:

(1)Yui, 'Flyweight Design pattern', pdf, YuiBlog, Viewed July 14th 2013,
URL<http://yuiblog.com/assets/projsdesignpatterns-ch9.pdf>

(2)OOdesign, "Flyweight Pattern, chart, OOdesign, Viewed July 13th 2013,
URL<http://www.oodesign.com/flyweight-pattern.html>

(3)DevLake, 'Flyweight Design Pattern', May 4th 2011, Viewed July 13th 2013,
URL<http://www.codeproject.com/Articles/186002/Flyweight-Design-Pattern>

(4)Dofactory, 'Flyweight Design Pattern, Viewed July 14th 2013,
'URL<http://www.dofactory.com/Patterns/PatternFlyweight.aspx>

(5)SourceMaking,'Flyweight design Pattern', chart, SourceMaking, Viewed July 13th 2013,
URL<http://sourcemaking.com/design_patterns/flyweight>

(6)James Sugure, 'Flyweight Design Patter', chart, Java dzone, Viewed July 14th 2013,
URL<http://java.dzone.com/articles/design-patterns-flyweight>

(7)Groovy, ‘flyweight pattern’, Groovy, viewed July 14th 2013,URL<http://groovy.codehaus.org/Flyweight+Pattern>

Car Classification:http://en.wikipedia.org/wiki/Car_classification

Car parts:http://en.wikipedia.org/wiki/List_of_auto_parts

Builder design pattern:http://www.oodesign.com/builder-pattern.html

Abstract Factory design pattern: http://www.oodesign.com/abstract-factory-pattern.html

Object Oriented Design, *Prototype Pattern,* OODesign.com, viewed June 26th 2013, <http://www.oodesign.com/prototype-pattern.html>

Object Oriented Design, *Factory Pattern,* OODesign.com, viewed July 20th 2013, <http://www.oodesign.com/factory-pattern.html>

Source Making, *Prototype Design Pattern,* sourcemaking.com, viewed June 26th 2013, <http://sourcemaking.com/design_patterns/prototype>

Object Oriented Design, *Singleton Pattern,* OODesign.com, viewed June 29th 2013, <http://www.oodesign.com/singleton-pattern.html>

Source Making, *Singleton Design Pattern,* sourcemaking.com, viewed June 29th 2013, <http://sourcemaking.com/design_patterns/singleton>

Eric F, Elisabeth F, Bert B, and Kathy S 2004, *Head fist design patterns*, Print Publication Date 25th October, Chapter 5, pg. 169-182.