**Team Name:** NSST Game Development

**Software Engineering 2**

**Design Pattern**

**Flyweight Pattern**

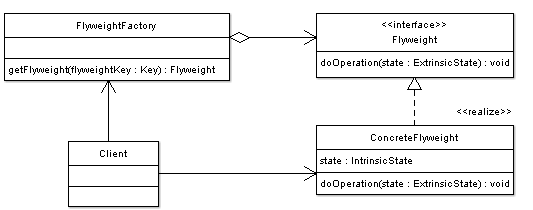
Phan Thanh San - s3342133

**Motivation**

Flyweight Pattern is a design pattern which allows the reuse of memory space in an application when the program require a large number of object 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

picture from oodesign.com

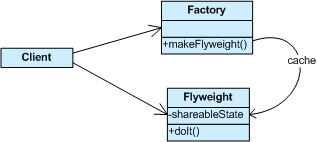
In Flyweight pattern, the object internal state is divided into two main categories: intrinsic data and extrinsic data.

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.

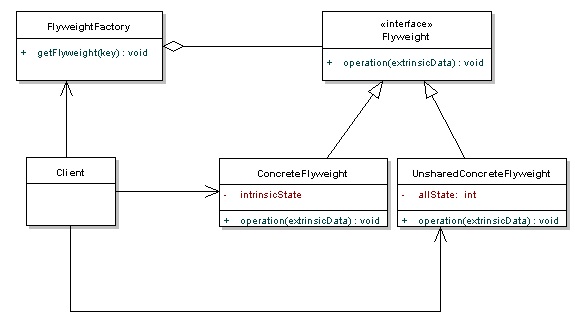
This means by taking all the object which have the same intrinsic data and make it into a single shared object, the number of similar object will be reduced to unique intrinsic state.(1)

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 are 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 help receive and act on extrinsic state. The concrete Flyweight implement the flyweight interface and stored the intrinsic state. It is shareable and its object maintain the state that is intrinsic as well as able to manipulate the extrinsic state(2). The Flyweight Factory create and manage the flyweight object as well as ensuring the sharing of these object. The client maintain “references to flyweight in addition to computing and extrinsic state”(2).

Flyweight 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). 

picture from sourcemaking

Below is another diagram of flyweight pattern

As seen from the picture, the ConcreteFlyweight class is able to add capabilities for intrinsic. The object is shable (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(6)

//Flyweight

2.public interface LineFlyweight

3.{

4.public Color getColor();

5.public void draw(Point location);

6.}

Implement the line(6)

01.//ConcreteFlyweight

02.public class Line implements LineFlyweight

03.{

04.private Color color;

05.

06.public Line(Color c)

07.{

08.color = c;

09.}

10.

11.public Color getColor()

12.{

13.return color;

14.}

15.

16.public void draw(Point location)

17.{

18.//draw the character on screen

19.}

20.

21.}

the factory is used to manage the creation of line object(6)

01.//Flyweight factory

02.public class LineFlyweightFactory

03.{

04.private List<LineFlyweight> pool;

05.

06.public LineFlyweightFactory()

07.{

08.pool = new ArrayList<LineFlyweight>();

09.}

10.

11.public LineFlyweight getLine(Color c)

12.{

13.//check if we've already created a line with this color

14.for(LineFlyweight line: pool)

15.{

16.if(line.getColor().equals(c))

17.{

18.return line;

19.}

20.}

21.//if not, create one and save it to the pool

22.LineFlyweight line = new Line(c);

23.pool.add(line);

24.return line;

25.}

26.

27.}

The client will use the factory if they want to create the line as below(6)

1.LineFlyweightFactory factory = new LineFlyweightFactory();

2.....

3.LineFlyweight line = factory.getLine(Color.RED);

4.LineFlyweight line2 = factory.getLine(Color.RED);

5.

6.//can use the lines independently

7.line.draw(new Point(100, 100));

8.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 make 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 FLyweight 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 object 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 others 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 are single instance returned. (2)

State and Strategy Pattern are also included in Flyweight pattern as their objects are implemented as Flyweights. (2)

**Reference**

(1)URL<<http://yuiblog.com/assets/projsdesignpatterns-ch9.pdf>>

(2)URL<<http://www.oodesign.com/flyweight-pattern.html>>

(3)URL<<http://www.codeproject.com/Articles/186002/Flyweight-Design-Pattern>>

(4)URL<<http://www.dofactory.com/Patterns/PatternFlyweight.aspx>>

(5)URL<<http://sourcemaking.com/design_patterns/flyweight>>

(6)URL<<http://java.dzone.com/articles/design-patterns-flyweight>>

(7)URL<<http://groovy.codehaus.org/Flyweight+Pattern>>

Builder design pattern

*An object creational pattern*

Student : Bui Trong Nhan -s3275049

Software Engineering 2

**1/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

**2/Intent**

-Encapsulate the construction of complex objects. Objects are independent from others.

-Construct objects through an interface

**3/Applicability**

**Use Builder design pattern when :**

**4/Implementation**

**Practical demonstration: Building different types of car**

**CarSupervior**: director

**CarBuilder**: abstract builder

**Car**: product

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

**Director**

**package com.builder;**

**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**

**package com.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**

**package com.builder;**

**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**

**package com.builder;**

**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**

**package com.builder;**

**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**

**package com.builder;**

**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**

**package com.builder;**

**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**

**package com.builder;**

**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);**

**}**

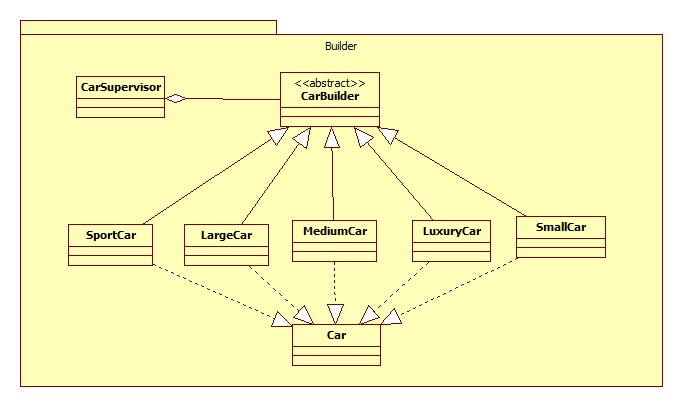
**@Override**

**public void buildSeats() {**

**car.setDoors(5);**

**}**

**}**

**Overall diagram**

**5/Advantages and Disadvantages**

**Advantages**

**Disadvantage**

**6/References**

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>

**Prototype Design Pattern**

(S3275145 – Hoang Ngoc Thanh)

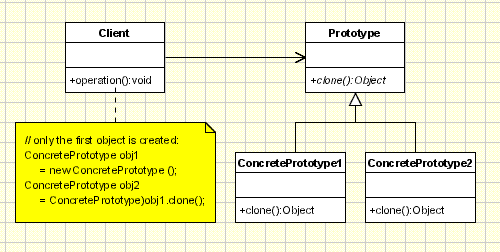
**1.**      **Class Diagram**

Image source: http://www.oodesign.com/prototype-pattern.html

**2.**      **Description**

a.    *Name -* Cloning an existing object

*b.*      *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.

-          The problem here is using “new” expression is harmful, and may reduce the performance of the project. Moreover, creating subclasses costs a lot of resources.

*c.*       *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

*d.*      *Sequences*

-          Avoiding “new” expression when creating new object because “new” expression is harmful

-          Avoiding creating of many subclasses.

**3.**      **Alternative**

**4.**      **Implementation**

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

o   Client: specifies prototypical instance of object, and create new object by asking prototype to clone itself

o   Prototype: an interface for cloning

o   ConcretePrototype: implement method clone() of Prototype interface.

Here is the code of prototype pattern implementation in Java

public interface Prototype extends Cloneable{

   public Object clone();

}

public class ConcretePrototype implements Prototype {

   @Override

   public Object clone() {

    ConcretePrototype obj = null;

    try {

        obj = (ConcretePrototype) super.clone();

    } catch (CloneNotSupportedException e) {

        e.printStackTrace();

    }

    return obj;

   }

}

public class Client {

   public static void main(String[] args) throws CloneNotSupportedException {

    ConcretePrototype obj1 = new ConcretePrototype();

    ConcretePrototype obj2 = (ConcretePrototype)obj1.clone();

   }

}

**References**

Object Oriented Design, *Prototype Pattern,* OODesign.com, viewed June 26th 2013,

<http://www.oodesign.com/prototype-pattern.html>

Source Making, *Prototype Design Pattern,* sourcemaking.com, viewed June 26th 2013,

<http://sourcemaking.com/design\_patterns/prototype>

**Prototype Design Pattern**

(S3260624 – Pham Dinh Son)

**1.**      **Class Diagram**

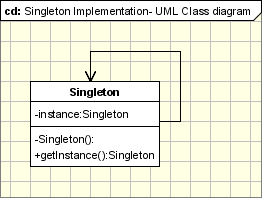


Image source: http://www.oodesign.com/singleton-pattern.html

**2.**      **Description**

*a.*      *Name – Using unique instance for a class*

*b.*      *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

*c.*       *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.

*d.*      *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**

**4.**      **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:**

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.