

Introduction to Object Oriented Design Pattern



THET KHINE · SATURDAY, JUNE 18, 2016

What is design pattern

Pattern ဆိုတာဖြစ်နေရာပုံစံလေးတွေကို ဆိုနိုင်တာပါ။ Design Pattern ဆိုတာကတော့ software construction Or programming မှာကြုံတွေ့နေရာ design problem လေးတွေအလိုက် ဘယ်လို solve လုပ်မယ်ဆိုရင်ဖြင့် အကောင်းဆုံးဖြစ်နိုင်မယ်ဆိုတာတွေကို အကြံပြုထားတဲ့ design တွေပါ။ ဥပမာ ဣန္ဒာတို. code မှာ database code တွေကို နေရာတစ်ခုထဲမှာပဲထားနိုင်တယ် ။ Database setting တွေကို ပြောင်းလဲလွယ်အောင် object တစ်ခုအနေနဲ့ပဲထားနိုင်တယ် ဒါဆိုရင် class တွေ object ကိုဘယ်လို design ခုမလဲ သင့်တော်တဲ့ design pattern တခုခုနဲ့ သုံးမယ်ဆိုရင် ဣန္ဒာတို.လိုနဲ့တဲ့ solution ကို ရမဲ့အပြင် ပိုကောင်းတဲ့ code, maintainable, reusable code, adaptable code ဖြစ်လာမှာပါ။ API library တွေကို programmer တွေက code reuse အတွက် သုံးသလိုပဲ design pattern ဆိုတာ design reuse ပါပဲ။ ကို.ထက် တော်တဲ့ expert programmer တွေရဲ့ design ကိုယူသုံးတဲ့အတွက် ကိုယ် ထုတ်တဲ့ design solution ထက်ပိုကောင်းတာဖြစ်နေနိုင်ပါတယ်။ နောက်ပြီး ကို. code တွေက design pattern တွေကိုသုံးထားတဲ့အခါ formalize လုပ်ပြီးသားဖြစ်တဲ့အတွက် တခြား team member တွေနဲ့ communicate လုပ်ရတာ လွယ်ကူစေပါတယ်။ ဥပမာ design pattern ကိုနားလည်တဲ့ team ထဲမှာ Singleton လို.ပြောလိုက်တာနဲ့ သူပြောချင်တာ object တစ်ခုတည်းပဲဆောက်မယ် ဒါကို နားလည်ပါလိမ့်မယ်။ အဲ့အတွက် communication ပိုကောင်းလာမယ် နောက်တခုက documentation ပိုကောင်းလာမယ်။ Pattern နဲ့ရေးထားတဲ့ code ကိုမြင်ရင် နောက်က programmer က အရင် programmer ဘာကိုလုပ်ချင် လဲဆိုတာ သိပါလိမ့်မယ်။ Pattern တစ်ခုခြင်းဆိုမှာလဲ နာမည်တွေ၊ ဘာလို.သုံးရလဲဆိုတဲ့ အကြောင်းတွေ ရှိ တဲ့အတွက် ဒါက programmer အခြင်းခြင်း ဆက်သွယ်ဖို့ ပိုလွယ်ကူစေပါတယ်။

Design pattern ရဲ့အစ က computer science ကမဟုတ်ပါဘူး။ Christopher Alexander ဆိုတဲ့ architect တယောက်က 1979 မှာ Timeless way of building ဆိုတဲ့စာအုပ်ထုတ်ပါတယ်။ အဲ့စာအုပ်မှာ ဘာတွေပါလဲဆိုတော့ Building တွေတည်ဆောက်တဲ့နေရာမှာ ဘာနဲ့.ဘာတွေဆောက်ရင် ခိုင်ခံ့မယ် ဘယ် လို design မှူးသုံးရင် ဘယ်လို problem မှူးကို solve လုပ်နိုင်မယ်ဆိုပြီးရေးခဲ့ပါတယ်။ နောက်သူ.စာအုပ် ကိုမှီပြီး software သမားတွေက design pattern တွေကိုသုံးကြပါတယ်။ 1995 မှာ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides ဆိုတဲ့ လေးယောက်က (Gang of Four GoF လို့.လူသိများပါတယ်) Design Patterns: Elements of Reusable Object-Oriented Software ဆိုတဲ့စာ အုပ်ကိုထုတ်ပါတယ်။ အဲ့စာအုပ်မှာ design pattern 23 မှူးကို catalog လုပ်ထားပါတယ်။ သူတို့.တခုချင်းစီ ရဲ့ နာမည်၊ သုံးရမဲ့နေရာ၊ ကောင်းကျိုးဆိုးကျိုး စတာတွေကို ရှင်းပြထားပါတယ်။ Langaugue အနေနဲ့.က တော C++ ကိုသုံးထားတာပါ။ Design Pattern ကိုလေ့လာတဲ့သူတွေအတွက် ဆိုရင် အဲ့စာအုပ်ကို မဖြစ်မ နေ reference လုပ်ရမှာပါ။ လေးယောက်စလုံးက OO software engineering မှာ ဆရာကြီးတွေပါပဲ။ Erich Gamma ဆိုရင် JUnit framework ရဲ့ contributor တယောက်ပါ။ Begineer တွေအတွက်ဖတ်ရတာ အ ဆင်ပြေမဲ့ design pattern စာအုပ်ကတော့ Head First Design Pattern ဆိုတဲ့စာအုပ်ပါပဲ။

Design pattern တွေနဲ့.ပတ်သတ်ပြီး Programmer တွေ ကြုံရတဲ့ ပြဿနာကတော့ရှိသမျှ code တွေကို design pattern နဲ့.စရေးတာပါပဲ။ အမှန်က သာမန် Object Oriented Analysis and Design ထဲက method တွေသုံးပြီး အရင်ရေးသင့်ပါတယ်။ နောက်မှ Design pattern အနေနဲ့. refactoring လုပ်သင့်ပါ

တယ်။Gardy Booth ရေးတဲ့ Object-Oriented Analysis and Design with Applications ဆိုတဲ့စာအုပ်က OOAD အတွက် အကောင်းဆုံး စာအုပ်လို့ပြောလို့ရပါတယ်။ OO Design ကနေ မှ Design Pattern ကို refactor လုပ်ဖို့အတွက်ကတော့ Joshua Kerievsky ရဲ့ Refactoring to Patterns စာအုပ်က အသုံးဝင်မှာပါ။ ပထမဆုံး Design Pattern ကိုမလေ့လာခင်မှာ proper Object Oriented Design ကိုနားလည်သင့်ပါတယ်။ ဒဲ့လိုမှမဟုတ်ရင် Design Pattern ကိုသုံးပေမဲ့လဲ OO Design တွေ Principle တွေကိုနားမလည်ရင် အဆင်မပြေပါဘူး။

Language Specific Pattern and UI Pattern

Language အပေါ်မူတည်ပြီး သုံးရတဲ့ implementation specific pattern တွေလဲရှိပါသေးတယ်။ ဥပမာပြောရရင် JavaScript မှာ OO Inheritance လုပ်ဖို့အတွက်သုံးတဲ့ pattern သုံးလေး ငါးမျိုးရှိပါတယ်။ GoF 23 မျိုးမှာမပါတဲ့ module, revealed module pattern ဆိုတာမျိုးတွေလဲရှိပါတယ်။ နောက်တခုက JavaScript မှာ MVC variation တွေဖြစ်တဲ့ MVVM, MVP,MV* ဒဲ့တာတွေကလဲ pattern တွေပါပဲ။ ဒါတွေကတော့ သက်ဆိုင်ရာ language အလိုက်ပဲ ရှိတဲ့ language specific pattern တွေပါပဲ။ နောက် Design Pattern တွေရဲ့ idea ကိုယူပြီး CSS, HTML မှာ သက်ဆိုင်ရာ UI element တွေ ဥပမာ tab ကိုဘယ်လိုဆောက်ရလဲဆိုတဲ့ UI Pattern တွေရှိပါတယ်။ Android တို့. Swift တို့.မှာလဲဆိုင်ရာ ဆိုင်ရာ UI implementation pattern တွေရှိပါတယ်။ Language တခုပေါ်မူတည်ပြီး ဘာလုပ်ခွင်ရင် ဘယ်လိုရေးဆိုတဲ့ (ဥပမာ Collection တခုကို iterate လုပ်ဖို့ အကောင်းဆုံးနည်းနဲ့.ဘယ်လိုရေးရမလဲဆိုတာမျိုး)Java အတွက်ထုတ်ထားတဲ့ စာအုပ်ကတော့ Kent Beck ရေးထားတဲ့ Implementation Pattern ဆိုတဲ့စာအုပ်ပါပဲ။ Kent Beck ကတော့ TDD ရဲ့. father နဲ့. JUnit ရဲ့. contributor ပဲဖြစ်ပါတယ်။

Enterprise pattern

Enterprise Application တွေ အပေါ်မူတည်ပြီး သုံးတဲ့ Design Pattern တွေလဲရှိပါတယ်။ ဥပမာ Domain Logic Pattern, Data Source Architecture Pattern, အစရှိတာတွေပါ။ ဒီအပိုင်းနဲ့.ပတ်သတ်ပြီး ကောင်းမွန်တဲ့စာအုပ်ကတော့ Martin Folwer ရဲ့. Patterns of Enterprise Application Architecture ဆိုတဲ့စာအုပ်ပါပဲ။ ဒဲ့ဒီမှာ Enterprise pattern တွေ အကြောင်းကို လေ့လာနိုင်မှာပါ။

Architecture Pattern

Software တခုကို architectre ပိုင်း တည်ဆောက်ရာမှာသုံးတဲ့ pattern တွေပါ။ ဥပမာ Distributed pattern, event driven pattern, layered pattern, MVC,REST အစရှိတာတွေပါ။ ခုနောက်ပိုင်း ခေတ်စားလာတဲ့ architectural pattern တခုက micro service architecture ပါ။ အရင်ကဆို application □ ကီးတွေဟာ service တွေအားလုံးကိုစုပြီး one process အနေနဲ့. run ကြပါတယ်။ ဒါက တခုခုကိုပြင်ခွင့်ရင် ဂျာန့်တဲ့အပိုင်းကိုသွားထိတာရယ် deployment လုပ်ရတာ အဆင်မပြေတာရယ် ကြောင့် service သေးသေးလေးတွေကို process လေးတွေခွဲပြီး သုံးကြတာပါ။ ဒါမျိုးကို micro service architecture pattern လို့.ခေါ်ပါတယ်။

Anti Pattern

Design Pattern တွေက ဘယ်နေရာမှာ ဘယ်လို design သုံးရမယ်ဆိုတာ suggest လုပ်တာပါ။ Anti pattern ကတော့ ဂျာန့်တော့တို.ဟာ ဘယ်လိုလုပ်နေရင်တော့ မှားနေပြီဆိုတာမျိုးပါ။ ဥပမာ ပြောရရင်

Programmer တယောက်က သူ့ code တခုမှာ မှားတယ်ဆိုပါစို့။ ဒါကို သေသေချာချာ trace မလိုက်ပဲနဲ့။
ဟို code လေးထဲကြည့်လိုက် မှန်လားကြည့်လိုက် ဒီ code လေး ထဲကြည့်လိုက် မမှန်ရင်ပြန်ထုတ်လိုက် အဲ့
လိုရေးနေတယ်ဆိုရင် ဒါ anti pattern ပါ။ Programming by permutation လို့ခေါ်ပါတယ်။
AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis ဆိုတဲ့စာအုပ်က anti
pattern အတွက် အသင့်တော်ဆုံးပါ။ ဒီနေရာမှာ code pattern မဟုတ်ပဲနဲ့ management pattern နေ
တွဲပါ ပါပါတယ်။ ဥပမာ team တစ်ခုကို ဘယ်လို manage လုပ်ရင် မှန်တယ် ဘယ်လိုလုပ်ရင် မှားတယ်ဆို
တာမျိုးပါ။

ဂျွန်တော်မသိသေးတဲ့ pattern များကုန်ခဲ့သေးရင် ထပ်ထဲပေးဖို့ မေတ္တာရပ်ခံပါရစေ

ဆက်ရန် 😊

Object Oriented Design Pattern Series

Part-1 Singleton Design Pattern



THET KHINE · THURSDAY, JUNE 23, 2016

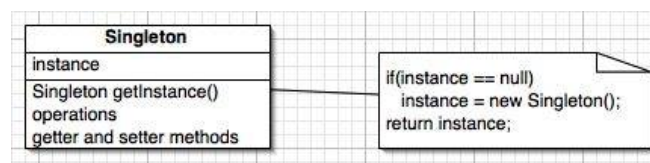
GoF Design Pattern တွေထဲမှာ အရိုးရှင်းဆုံးနဲ့ implement လုပ်ရအလွယ်ဆုံး ကိုပြောရရင် Singleton Design pattern ပါလို့ပဲပြောရမှာပါ။ Pattern တွေကို သေချာ မှန်မှန်ကန်ကန် အသုံးပြုနိုင်ဖို့ဆိုရင် ဒီ pattern ကဘယ်လို design problem ကို solve လုပ်ပေးသလဲ။ သူ့ကိုဘယ်အချိန်မှာသုံးသင့်သလဲ။ ကောင်းကိုး ဆိုးကိုး ဒါတွေကိုသိရမှာပါ။ Singleton ဆိုရင် နဂိုမူရင်း GoF pattern တွေထဲမှာပါခဲ့ပေမဲ့ နောက်ပိုင်းမှာ ဝေဖန်တာတွေရှိလာပါတယ်။ ဒါကိုသတိထားရမှာပါ။ မတူညီတဲ့ Developer တွေ အဆင်ပြေအောင် Java, JavaScript ဟိုးနဲ့ pattern တွေရဲ့ implementation ကို ရေးသွားမှာပါ။ အဓိက concept ကို နားလည်မိရင် language does not matter လို့ပဲဆိုရမှာပါ။ တခြား Language တွေထဲပေးချင်ပေမဲ့လဲ စာလေးသွားမှာစိုးတဲ့အတွက် နားလည်ပေးလိမ့်မယ်ထင်ပါတယ်ခင်ဗျာ။

Intent

Singleton pattern ရဲ့ ရည်ရွယ်ချက်က instance တခုတည်းရှိစေချင်တယ်။ Instance ဆိုတာ OO term အရ object ကိုရည်ညွှန်းတာပါ။ Entire application တခုလုံးမှာ object တခုပဲ ရှိစေချင်တယ်။ ဒီ Object ကိုထပ်ခါထပ်ခါ ဆောက်တာမျိုးမဖြစ်စေချင်ဘူး။ Application ရဲ့ ဘယ် code အပိုင်းကပဲ ခေါ်ခေါ် ဒီ object တခုတည်း ပဲရမယ်။ နောက် အဲဒီတခုတည်းသော Object ကို Global access ပေးနိုင်ရမယ်။ Application တခုရဲ့ ကြိုက်တဲ့နေရာကနေ ဒီ object ကို ယူသုံးလို့ရအောင် ပေးထားရမယ်။ ဒါက singleton ရဲ့ intent ပါ။

Why singleton

Application တခုမှာ class တခုဟာ single instance ပဲရှိရမယ်ဆိုတဲ့ requirement မျိုးမှာ singleton ကို သုံးပါတယ်။ ဥပမာ hardware resource ကို model လုပ်တဲ့ class မျိုးဆိုပါစို့။ CPU ကို model လုပ်ရင် CPU instance တခုပဲရှိရမယ်ဒါဆိုရင် singleton ကိုသုံးလို့ရပါတယ်။ နောက်တခုက Setting class ဆိုပါစို့။ object တခုတည်း ရှိသင့်ပါတယ်။ Setting ဆိုတာမျိုးကို application ရဲ့ဘယ်နေရာကပဲသုံးသုံး ဒီ properties တွေ data တွေပဲရသင့်ပါတယ်။ ဒီလိုနေရာမျိုးဆို singleton ကိုပဲသုံးရမှာပါ။ သတိထားရမှာက singleton object ပဲဖြစ်ရမယ် သူ့ကို application ရဲ့နေရာတိုင်းက နေခေါ်လို့ရ ရမယ်ဆိုရင် singleton ကိုသုံးပါ။ နောက်ထင်ရှားတဲ့ singleton example ကိုပြရရင်တော့ Logger ပါ။ Application ရဲ့ နေရာ အနှံ့မှာ logger ကိုလိုပါတယ် (Global Access point) ။ နောက်တခုက အဲဒီ logger က တခုတည်းပဲ ဖြစ်ရမှာပါ။ ဒီ global access point နဲ့ single instance ဟု ချက်နဲ့ကိုက်ညီရင် singleton ကိုသုံးလို့ရပါပြီ။



အပေါ်ကပုံမှာ Singleton ရဲ့ structure ကိုပြထားပါတယ်။ ဒီနေရာမှာ တစ်ခု သတိပေးချင်တာက Design Pattern တွေ OO Design တွေလေ့လာတော့မယ်ဆိုရင် UML ကလဲ မသိမဖြစ် လိုပါတယ်ဆိုတာပါ။ အပေါ်က ပုံမှာ Singleton class ကိုပြထားပါတယ် ။နောက် သူ့မှာ getInstance ဆိုတဲ့ method ရှိပါတယ် အဲ့ကောင်က global acces point ကိုပေးထားတာပါ။ public ပေါ့။ သူ့ထဲမှာတော့ single instance ပဲဖြစ်စေ အောင် code ကိုရေးထားရမှာပါ။ အဲ့ဒါကို note ကလေးနဲ့ပြထားပါတယ်။

Implementation Example

Java

```
public class Logger {
    private static Logger instance;
    private Logger() //prevent instantiation from other class
    {
    }

    public static Logger getLogger()
    {
        if(instance == null)//check object have already constructed
        {
            instance = new Logger(); //instantiate only once
        }
        return instance;
    }

    //Other public methods

    public void log(String text)
    {
        System.out.println("Log> "+ text);
    }
}

public class LoggerDemo {

    public static void main(String[]args)
    {
        Logger logger1 = Logger.getLogger();
        Logger logger2 = Logger.getLogger();

        System.out.println(logger1 == logger2);
    }
}
```

```
}  
}
```

အပေါ်က Java class ၂ခုမှာ Logger class က Singleton ပါ။ ပထမဆုံး Singleton ဆောက်မှာဖြစ်တဲ့အတွက် instance တခုတည်းသိမ်းဖို့လိုပါတယ်။ ဒါကြောင့် static Logger instance; ဆိုပြီးကြေငြာထားတာပါ။ ဒီနေရာမှာ static ဖြစ်တဲ့အတွက် class variable ဖြစ်သွားပြီဖြစ်တဲ့အတွက် class တခုအတွက် instance သည် one copy ပဲရမှာပါ။ ဘာမှ initialization လုပ်မထားတဲ့အတွက် default အနေနဲ့ referecne type ဖြစ်တဲ့အတွက် null ဝင်နေမှာပါ။ နောက်တခုက သူ့ကို အခြားသူတွေက ဝင် access လုပ်လို့မရအောင် private ပေးလိုက်ပါတယ်။ အပြည့်အစုံဆိုရင်တော့.

```
private static Logger instance;
```

ဖြစ်သွားမှာပါ။ ဒါက singleton object အတွက် instance တခုတည်းသိမ်းဖို့လုပ်ထားလိုက်တာပါ။ နောက်ပြီး constructor ကို private ထားပါတယ်။ ဒါမှသာ constructor ကိုတိုက်ရိုက်ယူသုံးပြီး တခြား object တွေ တည်ဆောက်မှုကို ကာကွယ်နိုင်မှာပါ။

နောက်ဆုံး point ကတော့ global access ပေးရမယ် single object ပဲထုတ်ပေးရမယ် ဒါကိုတော့ getLogger method မှာရေးထားတာပါ။ getLogger က static ဖြစ်တဲ့အတွက် class level ကနေခေါ်ရမှာပါ။ သူ့က Logger instance ကိုပဲ return ပြန်မှာပါ။ Singleton တွေမှာ constructor ကို private ပေးထားတဲ့အတွက် getLogger လို global access point ကနေပဲခေါ်သုံးလို့ရမှာပါ။ ဒါကြောင့်သူ့ကို public ပေးထားတာပါ။ သူ့ method ထဲမှာ instance ကို null နဲ့ညှိသလားလို့စစ်ပါတယ်။ ပထမဆုံး ခေါ်တဲ့တခေါက်မှာ instance က null နဲ့ညှိမှာပါ။ ဒါဆိုရင် private constructor ကိုသုံးပြီး object instance တခုဆောက်ပါတယ်။ နောက် instance ကို return ပြန်ပါတယ်။ နောက်တခါ getLogger ကိုခေါ်မယ်ဆိုရင် instance သည် null မဟုတ်တော့တဲ့အတွက် object ကိုထပ်ဆောက်စရာမလိုတော့ပါဘူး။ ဒါဆိုရင် ရှိပြီးသား instance object ကို return ပြန်လိုက်ရုံပါပဲ။ ဒီနည်းနဲ့ single instance တွေပဲရအောင် ထိန်းထားတာပါ။ တခြား singleton class မှာလိုအပ်တဲ့ public method တွေလဲ သင့်တော်သလိုထဲလို့ရပါတယ်။

သူ့ကိုသုံးနည်းကတော့ LoggerDemo class မှာပြထားပါတယ်။ Constructor သည် private အနေနဲ့ထားတဲ့အတွက် Logger ကို new နဲ့အပြင်ကနေဆောက်ဖို့ မဖြစ်နိုင်ပါဘူး။ ဒါကြောင့် getLogger ကိုပဲသုံးရမှာပါ။ Logger.getLogger() ကို ၂ခါခေါ်ပါတယ်။ singleton ဖြစ်အောင်ထိန်းထားတာကြောင့် ၂ခါလုံးရတဲ့ object ကတူညီမှာပါ။ ဒါကြောင့်အောက်က print statement မှာ true ဆိုပြီးထွက်လာမှာပါ။ ဒီလိုရေးနည်းကို lazy initialization နည်းလို့လဲသုံးပါတယ် ဘာလို့လဲဆိုတော့ singleton object ကို getLogger ကို မခေါ်မချင်း မဆောက်သေးလို့ပါ။

JavaScript

ဒါကတော့ JavaScript နဲ့ singleton implementation ပါ။

```
<script>  
var Logger = (function()  
{  
    var instance; //instance is undefined in javascript by default  
    function Singleton()  
    {  
        this.log = function(text)
```

```

    {
        console.log("Log> "+text);
    }
}

return {
    getLogger : function()
    {
        if( !instance )
        {
            instance = new Singleton();
        }
        return instance;
    }
};

})(); //immediately invoke
var logger1 = Logger.getLogger();
var logger2 = Logger.getLogger();
console.log( logger1 === logger2 );
</script>

```

အပေါ်က JS code ကိုနားလည်ဖို့ module pattern နဲ့အရင် ရင်းနှီးရပါမယ်။ ဒါ့မှာ Logger ထဲကို IIFE (Immediately invoking function expression)ကိုသုံးပြီး function ကိုတခါတည်း invoke လုပ်ပါတယ်။ Anonymous function ကိုတခါတည်း invoke လုပ်တာပါ။ ဒါ့တွော Logger သည် တခါတည်း run မှာပါ။ JS မှာ encapsulation ကို colusre ကိုသုံးပြီး ရေးရပါတယ်။ function ကို run နဲ့အခါမှာ နောက်ဆုံးမှာ

```

return {
    getLogger : function()
    {
        if( !instance )
        {
            instance = new Singleton();
        }
        return instance;
    }
};

```

ဒီကောင့်ကိုတွေ့မှာပါ ဒါက ခုနက run လိုက်တဲ့ anonymous function ကနေ return ပြန်ရမှာပါ။ ဘာကို return ပြန်သလဲဆိုတော့ object တခုကို object literal သုံးပြီး return ပြန်ပါတယ်။ return {} ဆိုတဲ့ပုံစံနဲ့ပါ။ ဒါ့ return ပြန်တဲ့ object ထဲမှာ getInstance ဆိုတဲ့ method ပါပါတယ်။ getLogger: function() {} ပုံစံနဲ့ရေးထားတာပါ။ ဒါ့ဒီ function ထဲမှာမှ if နဲ့ !instance ကိုစစ်ပါတယ် ဆိုရှင်တာက Java example မှာလိုပဲ instance သည် null သို့မဟုတ် zero သို့မဟုတ် undefined မဟုတ်ဘူးဆိုရင် true ဖြစ်မှာပါ။ anonymous function အစမှာ

var instance ; ဆိုပြီးကြေငြာထားတာတွေ့မှာပါ။ သူ့ကို ဘာမှ assign လုပ်မထားတဲ့အတွက် undefined အနေနဲ့ရမှာပါ။ ဒါဆိုပထမဆုံးခေါ်တဲ့အခေါက်မှာ if statement သည် true ဖြစ်ပြီး singleton object ကို

new Singleton နဲ့ဆောက်မှာပါ။ ဒီနေရာမှာ Singleton constructor (JavaScript မှာတော့ class က ECMA 6 မှရတာပါ) သည် Anonymous function ရဲ့ scope ထဲမှာပဲရှိတဲ့အတွက် အပြင်ကနေခေါ်လို့မရပါဘူး။ ထို့အတူ instance ကိုလဲခေါ်လို့မရပါဘူး။ function ထဲမှာ ရှိတဲ့ variable တွေကို JavaScript မှာ ခေါ်သုံးခွင့်ရင် တိုက်ရိုက် access လုပ်လို့မရပဲနဲ့ closure နဲ့ပဲခေါ်သုံးလို့ရမှာပါ။ Closure ဆိုတာကတော့ function တစ်ခုထဲမှာ function လေးတွေရေးပြီး အထဲက function ကနေ ပြင်ပက function ဆီက data ကိုယူသုံးတယ်။ နောက်ပြင်ပက function ကြီး ပြီးသွားတာတောင်မှ အထဲက function လေးသည် ပြင်ပ function က data ကို access လုပ်ဖို့ link ရှိနေသေးတယ်။ ဒီသဘောကို ဆိုလိုတာပါ။ Singleton constructor ထဲမှာ log ဆိုတဲ့ public method ကိုထည့်ပေးထားပါတယ်။

Java Example အတိုင်း logger1 နဲ့ logger2 တူက တူသလားစစ်လိုက်ရင် true ထွက်လာမှာပါ။

Consequences

Singleton ကိုသုံးခြင်းအားဖြင့် Global variable မျိုးကိုရှောင်နိုင်ပါတယ်။ နောက် resource intensive ဖြစ်တဲ့ object တွေမှာ singleton ကိုသုံးမယ်ဆိုရင် ဒါကလဲ resource ကိုခွေတာရာရောက်ပါတယ်။ Criticism အနေနဲ့ကတော့ Global access point ပေးထားတဲ့အတွက် application တစ်ခုလုံးမှာ singleton ဟာ class အရ ကို tight coupling ဖြစ်ပါတယ်။ အဲ့ဒီအတွက် testing ကိုခက်ခဲစေပါတယ်။ ဒီနေရာမှာ ရှိန်စော်ကကောင်းကျိုးဆိုးပြစ်နှစ်ခုလုံးကိုပြောရတာဟာ တာဝန်အရပါ။ Singleton နဲ့ပတ်သတ်တဲ့ criticism တွေကိုလဲ လေ့လာဖို့အကြံပြုပါရစေ။

မည်သူ့မဆိုလွတ်လပ်စွာကူးယူဖော်ပြနိုင်ပါတယ်။

Thet Khine

Object Oriented Design Pattern Series

Part-3 Abstract Factory Method Design Pattern

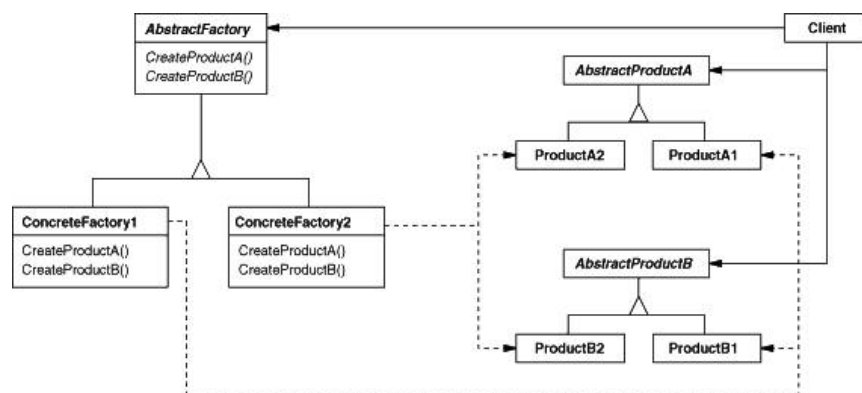
THET KHINE · MONDAY, JULY 25, 2016

Abstract Factory ဆိုတာကတော့ Creational pattern တွေထဲကတစ်ခုပါ။ သူ့နဲ့ဆက်စပ်နေတာကတော့ Factory Method pattern ပါ။ Factory Method ကတော့ Complex Object creation ကို hide လုပ်ခွင့်ရသုံးပါတယ်။ Factory method က class တစ်ခုရဲ့ object creation ကိုပဲ hide လုပ်တာပါ။ Abstract factory မှာတော့ family of product လို့ပြောရမှာပါ။ ဆိုခွင့်တာက related ဖြစ်နေတဲ့ class တွေရဲ့ object creation ကို ထိန်းချုပ်ရသုံးပါတယ်။

Intent

Abstract factory ကို Family of product object တွေ create လုပ်တဲ့အခါမှာသုံးပါတယ်။ Family of product ဆိုတာ အတူတကွသုံးရတဲ့ class တွေကိုဆိုခွင့်တာပါ။ ဥပမာ Car, Wheel , Engine ဆိုတာ Family of product ပါ။ ဒါပေမဲ့ အဲဒီ Car, Wheel ,Engine ဆိုတဲ့က စက်ရုံပေါ်မူတည်ပြီးကွာနိုင်ပါတယ်။ စက်ရုံကတော့ Factory ပါ။ ဒီ Family collection of product object တွေကိုသုံးတဲ့အခါမှာ client က နည်းနည်း ဘယ် Factory ကထုတ်လိုက်တယ်ဆိုတာကိုသိစရာမလိုပဲနဲ့ သုံးနိုင်အောင် ရည်ရွယ်တာပါ။

အောက်မှာ Abstract Factory Design pattern ရဲ့ class Diagram ကိုပြထားပါတယ်။



UML Diagram အရ ကျွန်တော်တို့က Family of product တွေကို (ProductA2,ProductA1) အစရှိတာတွေကို ရဲ့ creation ကို ထိန်းချုပ်တာပါ။ Client ကသုံးမှာက AbstractProductA, AbstractProductB ကိုပဲ သုံးမှာပါ။ ဒီနေရာမှာ object တွေက ဘယ် factory ကထုတ်တယ် ဆောက်ထားတယ်ဆိုတာကို Client က သိစရာမလိုပါဘူး ။ Implementation ကို hide လုပ်ထားတာပါ။

Why Factory Method Design Pattern?

Abstract factory နဲ့ ပတ်သတ်ပြီး ပေးလေ့ရှိတာကတော့ programming language တွေမှာသုံးတဲ့ GUI control တွေပါပဲ။ Control တွေမှာ Button, Label, အစရှိသဖြင့် family of product တွေရှိပါတယ်။ ဒါပေမဲ့ platform အပေါ်မူတည်ပြီး window Button လား Mac Button လား ကွာနိုင်ပါတယ်။ ဒါပေမဲ့ Client ကတော့ Button ဆိုရင် click လို့ ရရ မယ်ဆိုတာပဲ သိမှာပါ။ window Button လား Mac Button လား သိနေစရာမလိုပါဘူး ဒီနေရာမှာသူသိဖို့ လိုတာက Button ဆိုတာပါပဲ။ အပေါ်က Class Diagram ကိုနမူနာပြုပရရင် Button, Label သည် Client ကနေသုံးမဲ့ AbstractProductA, AbstractProductB နဲ့ တူမှာပါ။ WindowButton, MacButton ဆိုတာတွေကတော့ ProductA1, ProductB1 အစရှိတာတွေနဲ့ role ခွင်းတူမှာပါ။ WindowGUIBuilder နဲ့ MacGUIBuilder တွေကတော့ Factory တွေဖြစ်တဲ့ ConcreteFactory1 နဲ့ ConcreteFactory2 နဲ့ သွားတူမှာပါ။

ဥပမာ ဆိုကြပါစို့။ ကွန်ဇောတို.က ReportGenerator တခုကိုရေးရမယ်ဆိုပါစို့။ Report တခုမှာ ReportHeader ရှိမယ် ReportBody ရှိမယ်ပေါ့ဗျာ။ Report ကလဲ အမျိုးအစားကို HTML နဲ့ ဝေရာ PDF နဲ့ ဝေရာထုတ်ခွင့်တယ်ဆိုပါစို့။ ဒါဆို HTML နဲ့ ထုတ်မယ်ဆိုရင် HTMLReportHeader ရယ် HTMLReportBody ရယ်ထုတ်ရမယ် ။ XML နဲ့ ဆိုလဲထိုနည်းလည်းကောင်းပေါ့။ ဒါပေမဲ့နောင်တချိန်မှာ Client က Excel နဲ့ ထုတ်ခွင့်လာတယ်ဆိုရင် ဘယ်လိုလုပ်မလဲ ဒါမျိုးကို extensible ဖြစ်အောင် လုပ်ခွင့်ရင် Client သည် သူသုံးနေတာ HTMLReportHeader လား XMLReportHeader လား စတာတွေကိုသိနေရရင် implementation ကိုတိုက်ရိုက်သုံးနေမယ်ဆိုရင် extend လုပ်လို့ မလွယ်ပါဘူး။ ဒါဆို ခုနက HTMLReportHeader လား XMLReportHeader လားမခွဲပဲ ReportHeader ဆိုပြီး abstract class ပေးလိုက်ရင် ဒွဲဒီ ReportHeader ကို extends လုပ်သမျှနောက် implemenation တွေအကုန် သုံးလို့ ရမှာပါ။ Extensible ဖြစ်သွားပါပြီ။ ပြုပသနာက ReportHeader တင်မကပဲနဲ့ ReportBody ပါ report type ပေါ်မူတည်ပြီးထုတ်ရမှာဖြစ်တဲ့အတွက် Family of product ကို create လုပ်ရမှာပါ။ဒါဆို Abstract Factory နဲ့ ကိုက်နေပါပြီ။

Java Example

ပထမဆုံး family of product ဖြစ်တဲ့ abstract class တွေ ဖန်တီးရပါမယ်။

```
abstract class ReportHeader
{
    abstract void generateHeader();
}
abstract class ReportBody
{
    abstract void generateBody();
}
```

အပေါ်က class 2 ခုက client ကသုံးမဲ့ class တွေပါပဲ။ Client က ဒွဲဒီ abstract class တွေကိုပဲသုံးတော့ different implementation ကိုနောက်ပိုင်းထပ်ထဲရတာပိုလွယ်မှာပါ။ ခု သူတို့နဲ့ ဆိုင်တဲ့ concrete implementation တွေကို အောက်ပါအတိုင်းရေးပါတယ်။

```
class HTMLReportHeader extends ReportHeader
{
    @Override void generateHeader() {
        System.out.println("HTML report header");
    }
}
class HTMLReportBody extends ReportBody
{

```

```

@Override
void generateBody() {
    System.out.println("HTML report body");
}
}

```

အပေါ်ကကောင်ကတော့ HTML Report အတွက်ပါ။ နောက်ထပ် PDF အတွက် ထပ်ရေးပါမယ်။

```

class PDFReportHeader extends ReportHeader
{
    @Override
    void generateHeader() {
        System.out.println("PDF report header");
    }
}
class PDFReportBody extends ReportBody
{
    @Override
    void generateBody() {
        System.out.println("PDF report body");
    }
}

```

HTMLReportHeader,HTMLReportBody, PDFReportHeader,PDFReportBody တွေက family of product တွေအတွက် different implementation တွေပါ။ သူတို့အတွက် Factory method တွေဆောင်ပါမယ်။ဒါဆို Abstract factory method design pattern ရမှာပါ။

```

abstract class ReportFactory
{
    abstract ReportHeader createHeader();
    abstract ReportBody createBody();
}

class HTMLReportFactory extends ReportFactory
{
    @Override
    ReportHeader createHeader() {
        return new HTMLReportHeader();
    }
    @Override
    ReportBody createBody() {
        return new HTMLReportBody();
    }
}

class PDFReportFactory extends ReportFactory

```

```

{
    @Override
    ReportHeader createHeader() {
        return new PDFReportHeader();
    }

    @Override
    ReportBody createBody() {
        return new PDFReportBody();
    }
}

```

အပေါ်က class 3 ခုက abstract factory အတွက်အသက်ပါ။ ReportFactory သည် abstract factory class ပါသူ။ ထဲမှာပါတဲ့ createHeader ရယ် createBody ရယ်က abstract class တွေဖြစ်တဲ့ ReportHeader နဲ့ ReportBody ကိုပဲ return ပြန်ပါတယ်။ ဒီ Abstract Factory ကိုသုံးတဲ့ Client က concrete implementation ကိုသိစရာမလိုပဲနဲ့ ReportHeader နဲ့ ReportBody ကိုသိရင်ရပါပြီ။ ဒီနေရာမှာ ReportFactory ကို extends လုပ်တဲ့ Concrete Factory class တွေဖြစ်တဲ့ HTMLReportFactory နဲ့ PDFReportFactory က concrete class တွေကို သူတို့ ရဲ့ createHeader ,createBody မှာ return ပြန်ပါတယ်။ ဒါပေမဲ့ client က ဘယ် different implementation ကိုသိစရာမလိုပဲ ReportHeader, ReportBody အနေနဲ့ပဲသုံးရမှာပါ။ အောက် က code က Client code ပါ။

```

public class AbstractFactoryDemo {
    public static void main(String[] args) {
        ReportFactory fac = new HTMLReportFactory();// here create factory
        ReportHeader header = fac.createHeader();
        ReportBody body = fac.createBody();

        header.generateHeader();
        body.generateBody();
    }
}

```

Client က သုံးလိုင်းမြောက်မှာ HTMLReportFactory ကိုဆောက်ပြီး header တွေ body တွေ ဆောက်ယူပါတယ် ဣန္ဒြေတောတို့က ReportFactory fac ကိုသုံးပြီးဆောက်ပါတယ်။ HTMLReportFactory ကိုသုံးတဲ့အတွက် ဣန္ဒြေတောတို့ရလာမှာက HTMLReportHeader,HTMLReportBody object တွေပါ။ Client ဘက်ကနေကြည့်ရင် အဲဒီ implementation ကိုသိစရာမလိုပဲနဲ့ base abstract class ReportHeader,ReportBody ကိုသိဖို့ပဲလိုပါတယ်။ ဒါဆို ဘာကောင်းသလဲဆိုတော့ နောက်ထပ် ReportGenerator ဆိုပါစို့ Excel နဲ့လာမယ်ဆိုရင် သက်ဆိုင်ရာ Family Class တွေဆောက် Concrete Factory ကိုဆောက်လိုက်ရင်ရပါပြီ။ Client code ကိုပြင်စရာမလိုပါဘူး။ Family of product class တွေအများကြီးကို နောက်ထပ် လိုအပ်တဲ့အချိန်မှာ အေးအေးဆေးဆေးထပ်ထပ်လိုရပါပြီ။ဒါကတော့ Abstract Factory ရဲ့ advantage ပေါ့ဗျာ။ လိုအပ်ရင်လဲ ဘယ် ReportGenerator ပဲသုံးမယ်ဆိုပြီး client code မှာ တကြောင်းတည်း ပြင်လိုက်တာနဲ့ အဆင်ပြေပါပြီ။

```

ReportFactory fac = new HTMLReportFactory();// here create factory

```

အပေါ်က အကြောင်းလေးကိုပြင်လိုက်တာနဲ့ရပါပြီ ။

Object Oriented Design Pattern Series

Part-4 Builder Pattern

 THET KHINE · WEDNESDAY, AUGUST 17, 2016

Builder pattern ကတော့ Creational pattern တွေထဲကတစ်ခုပါပဲ။ သူ့ကို ဘယ်နေရာမှာသုံးလဲဆိုတော့ Complex Object တွေဆောက်ဖို့။ Object creation အတွက်လိုအပ်တဲ့ parameter တွေမှားနေမယ်ဆိုရင် Builder pattern ကိုသုံးပါတယ်။ ဥပမာ ဣန်တောတိုဆောက်ဖို့ Object သည် paramter ၅ ခု ၆ ခုလောက်လိုမယ်ဆိုပါတော့ ။ ဒါပေမဲ့တချို့တည်းမှာလဲ parameter အားလုံးသုံးခုလဲသုံးမယ် မသုံးပဲ ၃ ခုလောက်ပေးပြီးတော့ပဲဆောက်ခွင့်လဲဆောက်မယ်။ ဒါဆိုရင် Constructor နဲ့ရေးဖို့ကတော်တော်ခက်သွားပါပြီ။ ဥပမာ Text ဆိုတဲ့ class တခုပေါ့။ သူ့မှာ အောက်က properties တွေပါမယ်ပေါ့။

```
String displayValue;
String font;
String color;
String decoration;
```

UI application တခုမှာ Text Label ကို ကိုယ်စားပြုမဲ့ Class ပေါ့။ သူ့ထဲမှာ ပါတဲ့ properties တွေက အပေါ်က ကောင်တွေအတိုင်းပဲ အဲ့ထက်လဲ အမှန်ဆိုမှားနိုင်တယ် real world မှာ။ လိုချင်တာက problem က Text object ကိုဆောက်မယ်ဆိုရင် displayValue ပဲပေးလိုက်တာလဲဖြစ်နိုင်သလို ဣန်တာတွေကို လိုမှလဲထွဲမယ်။ ဆိုချင်တာက Object ဆောက်ရင် properties အကုန် ပေးချင်မှပေးမယ်။ ပေးရင်လဲပေးမယ်။ Object ရဲ့ properties တွေကတော့ object မဆောက်ခင်ပေးရမယ်။ ဒါဆိုသူ့ကို သာမန် Constructor နဲ့ရေးပြီး solve လုပ်မယ်ဆိုရင် Constructor တွေကို overload လုပ်ပြီးသုံးရမှာပါ။ အောက်ကကောင်လိုပါ။

```
public class Text {
    String displayValue;
    String font;
    String color;
    String decoration;

    Text(String displayValue) {
        this.displayValue = displayValue;
    }
    public Text(String displayValue,String font)
    {
        this.displayValue = displayValue;
        this.font= font;
    }
    public Text(String displayValue,String font,String color)
    {
        this.displayValue = displayValue;
```

```

    this.font = font;
    this.color = color;
}

}

```

အပေါ်က code မှာ Constructor တွေကို overload လုပ်ပြီးတော့ problem ကို solve လုပ်ဖို့ကြိုးစားပါတယ်။ ဒါပေမဲ့အဆင်မပြေပါဘူး။ ဘာလို့လဲဆိုတော့ code ကိုသေချာကြည့်ပါ။ code မှာ ဂျွန်တော်တို့က

```
Text(String displayValue,String font)
```

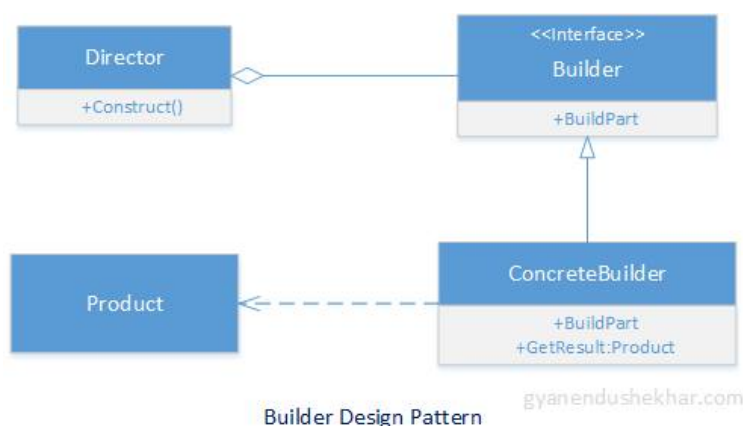
ဒီ Constructor ဆိုရင် displayValue, နဲ့. font ကိုလက်ခံပါတယ်။ တကယ်လို့မူား user ကနေ displayValue နဲ့. color ပေးရင် ရော code က အလုပ်လုပ်အောင် ရေးနိုင်ပမလားဆိုတော့မရပါဘူး။ ဘာလို့လဲဆိုတော့ Java လို static language တွေမှာ method overlaoding ဟာ parameter တွေရဲ့ type, number of parameter, order ဆိုတဲ့ သုံးခုပေါ်မူတည်ပြီး overload လုပ်ရပါတယ်။ ဒါကြောင့်နောက်ထပ်

```
Text(String displayValue,String color)
```

ဒီလို displayValue နဲ့. color ပဲလက်ခံတဲ့ Constructor ထပ်ဆောက်လို့မရပါဘူး ။ဘာလို့လဲဆိုတော့ အပေါ်က ဆောက်ထားတဲ့ Text(String displayValue,String font) နဲ့. Text(String displayValue,String color) သည် method signature ဖြစ်တဲ့ (String,String) တူနေတဲ့အတွက် Compiler ကနေပြဿနာရှာမှာပါ။ method signature တူနေတဲ့ method တွေကို overload လုပ်လို့မရပါဘူး။ ဒါဆိုဒီလို ပြဿနာမျိုးကို Builder design pattern နဲ့ဖြေရှင်းရင်ရပါတယ်။

Intent

သူ့ရဲ့ရည်ရွယ်ချက်က Complex Object တွေကိုတည်ဆောက်တဲ့အခါမှာ Object က parameter အများကြီးကို object creation stage မှာလက်ခံတယ်။ parameter အားလုံးကိုလဲ သုံးခွင်မှသုံးမယ်။ အဲ့လိုနေရာမှာသုံးပါတယ်။ အဓိက ကတော့ complex set of parameter in object creation ပါပဲ။ Object creation မှာ parameter တွေများလာရင် builder pattern နဲ့. solve လုပ်ရပါလိမ့်မယ်။ အောက်က ပေးထားတာ Builder Design pattern ရဲ့. class diagram ပါ။



အပေါ်က class diagram အရဆိုရင် ကွန်တောတို.က Object တခုအတွက် တိုက်ရိုက် parameter တွေ ကိုလက်ခံမဲ့အစား သူ့အတွက် Builder လို့ခေါ်တဲ့ Object မှာ parameter တွေကို ခန့်သိမ်းထားမယ်။ နောက်မှ build ဆိုတဲ့ method ကိုခေါ်မှ သူ့ရဲ့နုက သိမ်းထားတဲ့ parameter တွေကနေ တကယ်လိုတဲ့ Object ကိုဆောက်ပေးမယ်ပေါ့ဗျာ။ Builder pattern Code ကိုကြည့်ရအောင်။

```
public class Text {
    String displayValue;
    String font;
    String color;
    String decoration;

    private Text(Builder builder) {
        this.displayValue = builder.displayValue;
        this.font = builder.font;
        this.color = builder.color;
        this.decoration = builder.decoration;
    }
}
```

```
static class Builder
{
    String displayValue;
    String font;
    String color;

    String decoration;
    Builder displayValue(String dValue)
    {
        this.displayValue = dValue;
        return this;
    }

    Builder font(String fontName)
    {
        this.font = fontName;
        return this;
    }
    Builder color(String color)
    {
        this.color = color;
        return this;
    }
    Builder decoration(String decor)
    {
        this.decoration = decor;
        return this;
    }
}
```



```

Text build()
{
    Text text = new Text(this);
    return text;
}

@Override
public String toString() {
    return "Text{" + "displayValue=" + displayValue + ", font=" + font
        + ", color=" + color + ", decoration=" + decoration + '}';
}

public static void main(String[] args) {
    Text text = new Text.Builder()
        .color("green")
        .displayValue("Hello")
        .decoration("bold")
        .build();
    System.out.println("Text "+text);
}
}

```

အပေါ်က code မှာ Text class ကတွင် Text object အတွက်ပါ။ သူ့ properties တွေကတွင်အရင်အတိုင်းပဲ ဒီမှာ builder pattern နဲ့ဆောက်တဲ့နေရာမှာပထမ user ကနေပေးလိုက်တဲ့ parameter တွေကို Builder object တခုထဲမှာသိမ်းထားမယ် နောက်မှ builder object ကနေ Text class ကို ပေးလိုက်မယ်။ ဒီလိုနည်းနဲ့သွားတာပါ။ ဒါကြောင့် Text class ရဲ့ constructor မှာပထမ Constructor သုံးပြီး solve လုပ်တုန်းကလိုသူ့ရဲ့ properties တွေကိုလက်မခံပါဘူး။ Builder object ကိုလက်ခံတာပါ။

```

private Text(Builder builder) {
    this.displayValue = builder.displayValue;
    this.font = builder.font;
    this.color = builder.color;
    this.decoration = builder.decoration;
}

```

အပေါ်က Constructor က Text class ရဲ့ constructor ပါ။ Text ရဲ့ properties တွေကို builder object ကနေပဲယူလိုက်တာပါ။ ဂျာနတာကတွင်ဘာမှထူးထူးဆန်းဆန်းမပါပါဘူး။ အဓိက အသက်ကတွင်အောက်က builder class ပါ။ Builder class ကို Text class ရဲ့ static inner class အနေနဲ့ဆောက်ပါတယ်။ အပြင်မှာဆောက်ခွင့်လဲရပါတယ်။ ဒါပေမဲ့ encapsulation အတွက် အပြင်မထုတ်ပဲ အထဲမှာပဲရေးလိုက်တာပါ။ Builder class မှာရှိတဲ့ properties တွေသည် Text class မှာရှိတဲ့ properties တွေနဲ့အကုန်တူရမှာပါ။ ဘာလို့လဲဆိုတော့ Builder သည် user ကပေးတဲ့ Text class ရဲ့ parameter တွေကို ခဏ သိမ်းထားပေးမဲ့ကောင်ဖြစ်လို့ပါ။ နောက် property တခုခွင်းဆီအတွက်ဒီလို method တွေရေးပါတယ်။

```

Builder displayValue(String dValue) {
    this.displayValue = dValue;
    return this;
}

```

```

Builder font(String fontName)    {
    this.font = fontName;
    return this;
}

```

အပေါ်က method ၂ခုသည် displayValue နဲ့ font ကို လက်ခံပါတယ်။ Text class မှာသိမ်းမဲ့ property တခုခွင်းအတွက်ကို Builder မှာ method တခုစီရေးရမှာပါ။ လွယ်လွယ်လေးပါ ဝင်လာတဲ့ parameter ကို သိမ်းတယ် နောက် သူ့ object this ကို return ပြန်တယ်။ ဘာကြောင့် this ကို return ပြန်လဲဆိုတော့ method chaining pattern ကိုသုံးလို့ရအောင်ပါ။ jQuery မှာလိုပေါ့။ အဲ့တာကြောင့်ဒီလို method call တွေရေးနိုင်တာပါ။

```

new Text.Builder()
    .color("green")
    .displayValue("Hello")

```

အပေါ်က code မှာ new Text.Builder(). သည် Text class ထဲက Builder object ကို new သုံးပြီး constructor ဆောက်တာပါ။ နောက် new သည် object return ပြန်တဲ့အတွက် builder object ရဲ့ method တွေဖြစ်တဲ့ color ကိုခေါ်လို့ရပါမယ်။ color method သည် return this လို့ပြန်ထားတဲ့အတွက် builder object ကို return ပြန်မယ်။ ဒါကြောင့်နောက်ထပ် method တွေကို dot ခေါက်ပြီးဆက်တိုက် ခေါ်လို့ရမယ်။ ဒါကို method chaining pattern လို့ခေါ်ကြပါတယ်။ နောက် Builder pattern ရဲ့ Text object construction ကို ဒီလိုရေးပါတယ်

```

Text build() {
    Text text = new Text(this);
    return text;
}

```

User ကနေ builder object ကို parameter တွေပေးပြီးလို့စိတ်ကြိုက်ပြုဆိုရင် build ဆိုတဲ့ method ကိုခေါ်လိုက်ရမှာပါ။ ဒါဆို builder object ကို Text constructor ထဲကိုပေးလိုက်မှာပါ။ Text Constructor ကနေ Builder Object ရဲ့ properties တွေကို ကူးပြီး သူ့ Text object ဆောက်မှာပါ။ ဒီတော့ Text object ဆောက်ခွင့်ရင်ဒီလိုသုံးလို့ရပါပြီ

```

Text text = new Text.Builder()
    .color("green")
    .displayValue("Hello")
    .decoration("bold")
    .build();
System.out.println("Text "+text);

```

အဲ့တော့ ဘာကောင်းလဲဆိုတော့ ကိုကြိုက်တဲ့ parameter တွေကို ဥပမာ color အရင်ထဲမလား displayValue အရင်ထဲမလား ကြိုက်တာလုပ်လို့ရတယ် မထဲလဲရတယ် ။ method တခုခွင်းကိုခေါ်ရင် properties တွေကို builder object ရဲ့ properties အနေနဲ့ခန့်သိမ်းမယ် ။ ပီးတော့မှ build ကိုခေါ်လိုက်မယ် ဒါဆို build method ကနေ Builder object ကို Text constructor ကိုပို့မယ် Text constructor မှာ builder ကပို့တဲ့ properties တွေကို သူ့ Text Object ရအောင် ဆောက်မယ် ဒီနည်းနဲ့ရှင်းသွားတာပါ။ ဂျာနယ်က language တွေနဲ့လဲ ဒီသဘောတရားပါပဲ။

Object Oriented Design Pattern Series

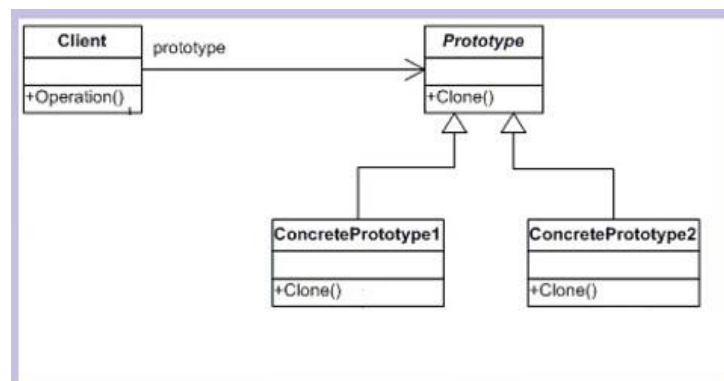
Part-5 Prototype Design Pattern

THET KHINE · FRIDAY, AUGUST 19, 2016

Prototype Pattern ကလဲနားလည်ရလွယ်တဲ့ pattern မှီးပါပဲ။ Creational Pattern အမျိုးအစားပါ။ သူကတော့ ရှိပြီးသား Object တခုကနေ နောက် Object တခုဆောက်တဲ့အခါမှာ new မသုံးတော့ပဲနဲ့ နဂိုရှိပြီးသား Object ကနေ property တွေကို copy ကူးပြီး တည်ဆောက်တာပါ။

Intent

Prototype pattern ကို Heavy Object တွေကိုတည်ဆောက်တဲ့အခါမှာ Object creation သည် computation time အရ ဒါမှမဟုတ် memory အရ အကုန်အကျ မှားမယ် costly ဖြစ်မယ်ဆိုရင် နောက်ထပ် Object တွေဆောက်တဲ့အခါမှာ ခုနက computational time , memory အစရှိတာတွေကိုသက်သာအောင် ရှိပြီးသား Object တွေရဲ့ မူရင်း property တွေကိုပဲ copy ကူးပြီး Object အသစ်ဆောက်လိုက်တာပေါ့။ ဒါကို prototype pattern လို့သုံးပါတယ်။ ဒီနေရာမှာ Heavy Object ဆိုတာဘာလဲဆိုတာကိုရှင်းဖို့လိုပါတယ်။ ဥပမာ Object တခုကိုတည်ဆောက်တဲ့အခါမှာ property တွေလိုတယ်ပေါ့ဗျာ။ အဲ့ဒီ property တွေကို DB ကနေယူရမယ် ဆိုပါစို့။ဒါဆိုရင် DB ကို query လုပ်ရတာကြာတဲ့ cost ကုန်မယ် နောက်ပြီး အဲ့ဒီ property ကို ဒီအတိုင်းမရနိုင်ဘူး။ ဥပမာ Game သို့မဟုတ် math နဲ့ဆိုင်တဲ့ application တခုခုမှာ ဒီ property ကိုတခြား အရာတခုခုက နေတွက်ထုတ်ရမယ်ဆိုရင် (heavy matrix calculation လို့မှီးပေါ့)computational time အရ အချိန်ကြာနိုင်ပါတယ်။ နောက်ထပ် memory အရ costly ဖြစ်တာကတော့ String လို့ object မှီးတွေပေါ့ ။ အရမ်းကြီးတဲ့ String တခုကို Property အနေနဲ့သိမ်းထားရတယ် အဲ့ဒီ String ကို DB ဒါမှမဟုတ် File ကနေဖတ်ရမယ်ဆိုရင် memory အားဖြင့်လဲ costly ဖြစ်ပါတယ်။ ဒါဆိုရင် နောက်ဆောက်မဲ့ Object တွေမှာ ခုနက costly ဖြစ်တဲ့ property တွေကိုပြန်သုံးရမယ် တခြား property တွေကတော့ နည်းနည်းလေး ပြောင်းလိုက်ရင် ရပြီ ။ ဒီလိုအခြေအနေမှာ ခုနက heavy property တွေကိုပြန်တွက်နေတာ DB ကနေပြန်ယူနေတာလုပ်မဲ့အစား ရှိပြီးသားတခုကနေပြီးတော့ Copy ကူးလိုက်ရင် အချိန်အားဖြင့် လည်းကောင်း memory အားဖြင့်လည်းကောင်းသက်သာနိုင်ပါတယ်။ အဲ့လို အခြေအနေမှီးမှာ prototype pattern ကိုသုံးရပါတယ်။ အောက်မှာ prototype pattern ရဲ့ class diagram ကိုပြထားပါတယ်။



Prototype ဆိုတဲ့ interface သည် clone ဆိုတာကို ပေးထားရပါမယ်။ Clone ဆိုတာဒီနေရာမှာ object မဆောက်ပဲ property တွေကိုပဲ copy ကူးပြီး တည်ဆောက်မယ်ဆိုတဲ့သဘောပါ။ Java မှာတော့ java.lang.Object class မှာ clone method ပါလာပါတယ်။ ခုနကပြောတဲ့ property တွေကို copy ကူးပြီး object ဆောက်ပေးပါတယ်။ ဒါ့ဒီတော့ ခုနက java.lang.Object က clone method ကိုယူသုံးရင် အဆင်ပြေပါတယ်။ Code ကိုကြည့်ရအောင်။

```
public interface Prototype extends Cloneable{  
    Prototype reproducce();  
}
```

ဒါကတော့ Prototype interface ပါ။ Java က Cloneable interface သည် marker interface ပါ။ ဘာmethod မှမပါဘူး ဒါပေမဲ့ clone လုပ်လို့ရတယ်ဆိုတာကိုမှတ်ထား ဆိုတဲ့သဘောနဲ့ သုံးတဲ့အတွက် marker interface လို့သုံးပါတယ်။ Serializable လို့ interface မှီးပါ။ သူ့မှာ ကွန်တိန်းတာတို့က reproducce ဆိုတဲ့ method ထဲပေးထားပါတယ် အမှန်က reproducce ကလုပ်မှာသည် clone လုပ်တဲ့ copy ပွားတဲ့ operation ပါပဲ။

```
public class HeavyObject implements Prototype{  
    String propertyGetFromDB;  
    String computationalHungryProperty;  
  
    static String getPropertyFromDB() {  
        return "PropertyFromDb"; //Here assume call to db to simulate costly memory  
    }  
  
    static String getComputationalHungryProperty() {  
        return "ComputationHungryProperty"; //Here assume call to costly computation time  
    }  
  
    public HeavyObject() {  
        this.propertyGetFromDB = HeavyObject.getPropertyFromDB();  
        this.computationalHungryProperty = HeavyObject.getComputationalHungryProperty();  
    }  
  
    @Override  
    public HeavyObject reproducce() {  
        try {  
  
            Prototype cop = (Prototype)super.clone(); //Here call lang.lang.Object.clone  
            HeavyObject newObject = (HeavyObject)cop;  
  
            return newObject;  
  
        } catch (CloneNotSupportedException ex) {  
            ex.printStackTrace();  
        }  
        return null;  
    }  
}
```

@Override

```
public String toString() {  
    return "HeavyObject{" + "propertyGetFromDB=" + propertyGetFromDB + ",  
    computationalHungryProperty=" + computationalHungryProperty + '}';  
  
}  
  
}
```

အပေါ်က code က HeavyObject ကိုပြထားတာပါ။ သူ့မှာ property ၂ခုသည် heavy computation+ heavy memory ကို ကိုယ်စားပြုထားပါတယ်။ နောက် Constructor မှာ အဲ့ကောင်တွေကို static method တွေဆီကနေယူပါတယ်။ အဲ့ဒီ static method တွေသည် တကယ်တမ်း DB call တို့၊ computation တို့ပါရမှာပါ။ ဒီမှာ code ရှည်မှာဆိုးတဲ့အတွက် ချန်ခဲ့တာပါ။ HeavyObject သည် Prototype interface ကို implement လုပ်ပါတယ်။ ဒါကြောင့် reproduce ကို override လုပ်ရမှာပါ။ reproduce မှာ ဣန္ဒိဏီတို့ cloneကိုခေါ်ပါတယ်။ Java မှာ super.clone() ဆိုပြီး java.lang.Object ထဲက clone method ကိုသုံးပြီး property တွေကို အသစ် copy ပွားပြီး newObject ဆောက်ပါတယ်။ သတိထားရမှာက ဒီနေရာမှာ new operator ကိုမသုံးဘူးဆိုတာပါ။ ဒီဟာက အသစ်ပါ။ clone() method ကိုပဲသုံးသွားတာပါ။ new operator ကိုသုံးပြီး object ပြန်ဆောက်ရင် constructor ကိုခေါ်ရမယ် ဒါဆို အချိန်ပိုကြာသွားမယ် လေးသွားမယ် ဒါကိုရှောင်ရှောင်လို့သာ prototype pattern သဘောအရ clone method ကိုသုံးတာပါ။ Prototype interface သည် Cloneable ကို extends လုပ်ထားတဲ့အတွက် Java runtime က clone operation ကိုခွင့်ပေးမှာပါ။ မဟုတ်ရင် CloneNotSupportedException တက်ပါလိမ့်မယ်။ အောက်မှာပေးထားတာကတော့ Client code ပါ။

```
public class PrototypeDemo {  
    public static void main(String[] args) {  
        HeavyObject firstObj = new HeavyObject();  
        HeavyObject another = firstObj.reproduce();  
  
        System.out.println(another);  
    }  
}
```

Client ကသုံးတဲ့နေရာမှာ firstObject တခုကိုပဲ new operator သုံးပြီး constructor နဲ့ခေါ်ပါတယ် ပထမဆုံးအကြိမ်မှာတော့ heavy computation နဲ့ costly operation ဖြစ်မှာပါ။ နောက် object တွေဖြစ်တဲ့ another object ကိုတော့ reproduce ကိုသုံးပြီး clone လုပ်ပြီး Object ဆောက်ပါတယ်။ ဒါကြောင့် ခုနက constructor ကိုမဖြစ်ရတော့ပါဘူး ။ ဒါကြောင့် heavy computation+memory cost ကိုရှောင်နိုင်ပါပြီ။

Prototype pattern ရဲ့အားသာချက်ကတော့ Heavy ဖြစ်တဲ့ Object တွေဆောက်တဲ့နေရာမှာ ခန့်ခန့် ထပ်ဆောက်မနေပဲ ဘုံတူတဲ့ property တွေကို copy ကူးပြီးသုံးတဲ့ အတွက် computational time+memory သက်သာတာပါပဲ။ ဒါကတော့သူ့ ကောင်းတဲ့အချက်ပါပဲ။

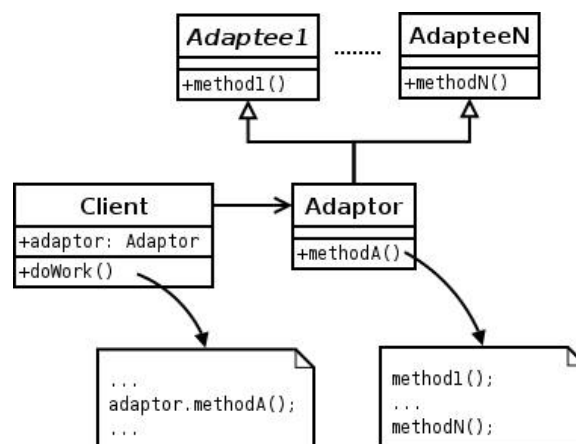
Object Oriented Design Pattern Series

Part-6 Adapter Design Pattern

THET KHINE · SATURDAY, AUGUST 20, 2016

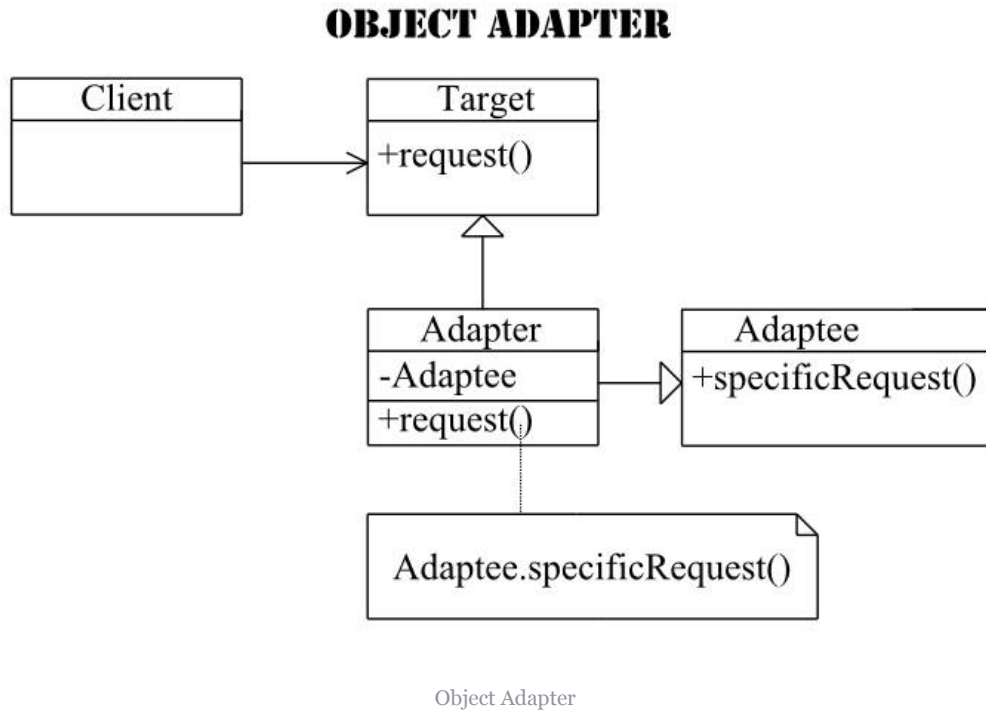
Adapter pattern ကတော့ Structural pattern တွေထဲကတစ်ခုပါ။ သူကဘာလုပ်ပေးလဲဆိုတော့ interface မတူတဲ့ module ဒါမှမဟုတ် code ဂရုကြားမှာ အလုပ်လုပ်လို့ရအောင် glue code ဒါမှမဟုတ် bridge လုပ်ပေးတာပါ။ ဒီနေရာမှာ interface သည် module တခု သို့မဟုတ် class တခုရဲ့ method name, parameter type, parameter order, အစရှိတာတွေကိုပြောတာပါ။ ရိုးရိုးရှင်းရှင်းပြောရရင် method ရဲ့ နာမည်တွေ parameter ပေးတဲ့ပုံစံတွေကိုဆိုနိုင်တာပါ။ ဥပမာ module A က လိုချင်တာ add ဆို method ဆိုပါစို့။ ဒါပေမဲ့ module B က ပေးထားတာကြောင့် addItem ဖြစ်နေတယ်ဆိုပါစို့။ ဒါဆို add လို့ module A က လိုချင်ပေမဲ့ addItem ဖြစ်နေတဲ့အတွက် မတူတဲ့အတွက် အဆင်မပြေပါဘူး။ ဒါဆိုရန်က module A ကလဲ add လို့ပဲခေါ်မယ် module B ကလဲ addItem ကိုမပြောင်းပဲထားမယ်ဆိုရင် သူတို့ကြားမှာ အဆင်ပြေဖို့အတွက် Glue code တခုလိုပါပြီ။ Real world က example ဆိုရင်တော့ IDE တွေ CMS လို complex software တွေမှာ plugin တွေရေးကြပါတယ်။ အဲလိုရေးကြတဲ့အခါ Plugin architecture ဆိုတာကိုသုံးကြပါတယ်။ ဘယ်လိုအလုပ်လုပ်သလဲဆိုတော့ Framework တွေ IDE တွေကနေ plugin တွေဟာ ဘယ် method တွေကို ရေးထားရမယ်ဆိုပြီး သတ်မှတ်ပါတယ်။ အဲဒါနဲ့ Framework ကလိုချင်တဲ့ method signature နဲ့ ကိုသုံးမဲ့ plugin အတွက် code (ဒီနေရာမှာ method signature ကိုပြင်လို့မလွယ်ဘူး ဒါမှမဟုတ် binary form နဲ့ပဲရလို့ ပြင်လို့အဆင်မပြေဘူး)သည် method signature မတူရင် Adapter Design pattern ကိုသုံးပါတယ်။

အပြင်လောက software မဟုတ်တဲ့ နယ်ပယ်မှာ ထင်ရှားတဲ့ Adapter တွေကတော့ ပါဝါကြိုးတွေမှာသုံးတဲ့ adapter ပါပဲ။ မြင်အောင်ပြောရမယ်ဆိုရင် ကွန်ပျူတာကြိုးတွေက 3 pin ပေါ့။ ဒါဆို plug ပေါက်က 2 ပင်ပဲရှိတယ်ဆိုရင် အဆင်မပြေပါဘူး။ ကြားထဲမှာ 3pin ကနေ 2 pin ကိုပြောင်းပေးနိုင်တဲ့ adapter ခေါင်းတခုလိုပါတယ်။ Adapter pattern ကလဲ အဲလိုပဲ ဟိုဘက်က code နဲ့ဒီဘက်က code interface မတူညီကြရင် ကြားထဲကလုပ်ပေးတဲ့ Glue code ပါပဲ။



Class Adapter

Adapter မှာ ၂မျိုးရှိပါတယ် Class Adapter နဲ့ Object Adapter ဆိုပြီးတော့။ အပေါ်ကပြထားတဲ့ UML diagram ကတော့ Class Adapter ဖြစ်ပြီးတော့ အောက်ကကောင်ကတော့ Object Adapter ပဲ။



Intent

Module တွေ Class တွေကြားထဲမှာ သုံးရမဲ့ code တွေသည် interface အားမြင့်မတူဘူး ဒါပေမဲ့ ခေါ်တွဲကောင်တွေကလဲ မပြင်နိုင်ဘူးဆိုရင် Adapter pattern ကိုသုံးရမှာပါ။

ဣန္ဒိဏီတို့မှာ Framework ဆိုတဲ့ interface တခုရှိတယ် သူ့ကို အားလုံးကလိုက်နာရတယ်ဆိုပါစို့။ ။ သူ့မှာ add(Integer item) ဆိုတဲ့ method တခုပါတယ်ဆိုပါစို့။ ဒါပေသိ Framework နဲ့တွဲသုံးခွင့်တဲ့ legacy class တခုကတော့ addItem(Integer item) ဖြစ်နေရော ဒါဆိုသူတို့ ၂ခုတွဲလုပ်လို့မရဘူးပေါ့။ အဲ့ဒါဆိုရင် Class Adapter သုံးပြီးဖြေရှင်းလို့ရပါတယ်။

```
public interface Framework {
    public void add(Integer item);
}
```

ဒါကတော့ interface ပါ သူ့ကို အားလုံးကလိုက်နာရမယ် အောက်ကကောင်ကတော့ Old legacy code ပါသူ့ကိုလဲ တခြားကောင်တွေကသုံးနေတော့ ပြင်ခွင့်မရှိဘူးဆိုပါစို့ဗျာ။

```
public class OldAPI {
    public void addItem(Integer item )
    {
        System.out.println("Old API addItem "+item);
    }
}
```


အပေါ်က interface Framework နဲ့ OldAPI မှာ add method နဲ့ addItem method က အဓိပ္ပာယ်သဘောတရားရှင်းတူပေမဲ့ interface method signature ကွာတဲ့အတွက် တွဲသုံးလို့အဆင်မပြေပါဘူး။ ဒါဆို ဣန်တီဂရေးရှင်တို့က ClassAdapter လို့ဆောက်ပြီး OldAPI ကို extend လုပ်မယ် Framework ကို implement လုပ်မယ် ပြီးတော့ framework method ကို override လုပ်မယ် အဲ့ကနေ OldAPI method addItem ကိုခေါ်ပေးလိုက်မယ်ဆိုရင် Class Adapter code ရပါပြီ သူတို့ ၂ ခုအလုပ်လုပ်နိုင်ပါပြီ။

```
public class ClassAdapter extends OldAPI implements Framework{
    @Override
    public void add(Integer item) {
        this.addItem(item); // Here call to old API method
    }
}
```

အပေါ်က code သည် Class Adapter ပါ သူ့အလုပ်က OldAPI ကို extend လုပ်တယ် ဒါဆို inheritance နည်းအရ OldAPI ကပေးထားတဲ့ api တွေ method တွေကိုသူသုံးလို့ရသွားတယ်ပေါ့။ နောက် Framework ကို implement လုပ်တယ် အဲ့တော့ Framework မှာ define လုပ်ထားတဲ့ add ဆိုတဲ့ method ကို သူ override လုပ်ပေးရမယ် ဒါဆို framework အတိုင်းသူ အလုပ်လုပ်နိုင်သွားပါလိမ့်မယ်။ add method မှာ သူကဘာမှမလုပ်ပါဘူး။ OldAPI ရဲ့ method အဟောင်းဖြစ်တဲ့ add ကိုလှမ်းခေါ်လိုက်တာပါ။ အဲ့ဒါသည် adapter pattern ရဲ့အသက်ပါပဲ ဒါဆို ClassAdapter သည် Framework ကလိုချင်နေတဲ့ interface နဲ့လဲကိုက်ပြီ OldAPI ကိုလဲ လှမ်းခေါ်နိုင်ပြီ ဖြစ်တဲ့အတွက် adapter ဖြစ်ပါပြီ။ Client ကတော့ဒီလိုသုံးရမှာပါ။

```
public ClientDemo {
    public static void main(String[] args) {
        Framework framework = new ClassAdapter();
        framework.add(30);
    }
}
```

အပေါ်က ClientDemo example မှာ client သည် OldAPI ကို တိုက်ရိုက်မသုံးဘူးဆိုတာပါပဲ။ ဘာလို့လဲဆိုတော့ ရှည်ရှယ်ချက်ကိုက OldAPI ကိုသုံးခွင့်တယ် ဒါပေမဲ့ method အဟောင်းအနေနဲ့မဟုတ်ပဲ Framework ကနေပေးထားတဲ့ method အတိုင်းသုံးခွင့်တာ။ ဒါကြောင့် ClassAdapter ကနေတဆင့် လှမ်းခေါ်လိုက်ရင် ဒီပြဿနာက အဆင်ပြေပါပြီ။

Object Adapter

Object Adapter ကတော့ ဘာကွာမလဲဆိုတော့ class adapter လို inheritance ကိုမသုံးပဲ object composition ကိုသုံးပြီး အလုပ်လုပ်တာလေးပါပဲ။ အောက်က code ကိုကြည့်ပါ။

```
public class ObjectAdapter implements Framework{

    OldAPI oldAPI = new OldAPI(); // Here is object composition

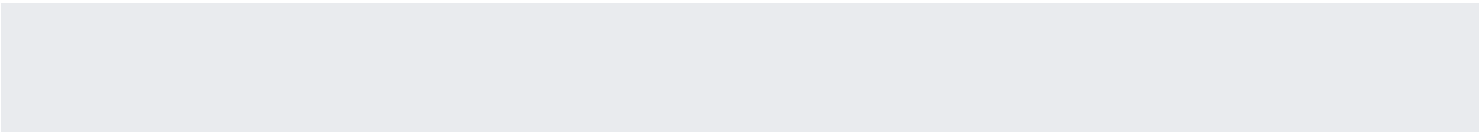
    @Override
    public void add(Integer item) {
```

```
        oldAPI.addItem(item); // Here call to old API method
    }

}
```

Object Adapter က inheritance မသုံးပဲ Object composition သုံးပြီး oldAPI object ကိုဆက်သွယ်ပေးပြီးတော့ ဒီကနေတစ်ဆင့် OldAPI ရဲ့ method addItem ကိုခေါ်ပါတယ်။ Client ကနေသုံးရင် ClassAdapter အစား ObjectAdapter လေးပြောင်းသုံးရပါမယ်။

Adapter ကဘာကောင်းလဲဆိုရင်တော့ Legacy code တွေမှာ interface နဲ့မကိုက်လို့ အဆင်မပြေဘူး ခေါ်လို့ မရဘူးဆိုရင် ကြားခံ Glue Code အနေနဲ့ သုံးလို့ ရအောင် ပြောင်းလိုက်လို့ ရပါတယ်။ ဒါပေမဲ့ Design အရ မှားနေတဲ့ class ဆိုရင်တော့ ဆိုရှင်တာက OldAPI class ကို ထိလို့ ရမယ်ဆိုရင် Adapter မသုံးပဲဖြေရှင်းတာပိုကောင်းပါတယ်။



Object Oriented Design Pattern Series

Part-7 Bridge Design Pattern

 THET KHINE · THURSDAY, SEPTEMBER 1, 2016

Bridge Design Pattern ကတော့ Structural Patternထဲမှာပါတဲ့ pattern ပါ။ သူကတော့နည်းနည်း နားလည်ရခက်ပါတယ်။

Intent

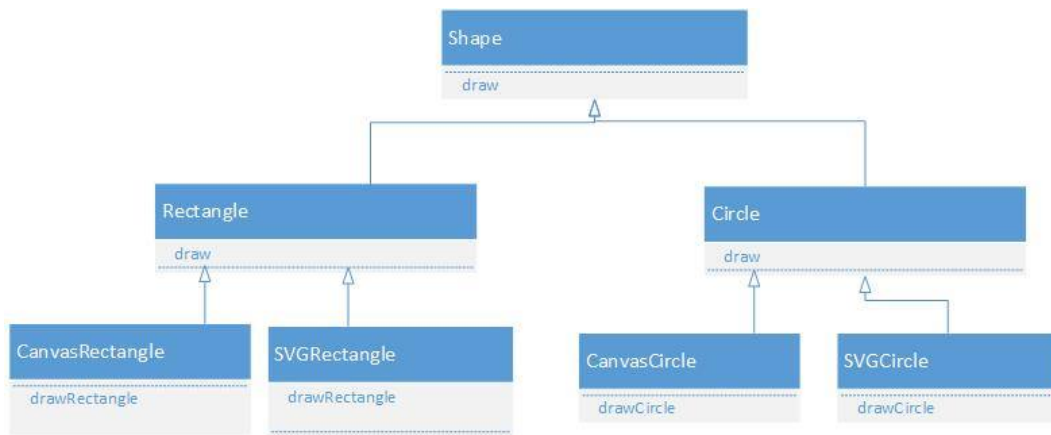
သူ့ intent ကိုတော့ GoF Book မှာဒီလိုရေးထားပါတယ်။

Decouple an abstraction from its implementation so that the two can vary independently.

ဒီနေရာမှာ Decouple လုပ်တယ်ဆိုတာ တခုနဲ့တခု ကိုအသေခွိတ်ထားတာ (class တခုမှာ နောက် class တခုကိုပဲသုံးထားတယ် abstract,interface ကိုမသုံးဘူး ဒါဆိုနောင်တချိန်မှာ အစားထိုးသုံးလို့မရဘူး။ extend လုပ်ဖို့ခက်မယ်, abstract,interface ကိုသာသုံးထားရင် သူတို့က ဆင်းလာတဲ့ကောင်တွေနဲ့သုံးလို့ရလို့ extend ရတာလွယ်မယ်) ကိုဆိုခွင်တာပါ။ Abstraction ဆိုတာတော့ implementation detail ကို hide လုပ်ထားတာကို ပြောခွင်တာပါ။ Implementation detail ဆိုတာကတော့ ဒီ code ကို implment လုပ်ဖို့အတွက် ဘယ် class ဘယ် method ဆိုတာကို ခေါ်တဲ့သူက သိနေရမယ် ဒီလိုဆိုခွင်တာပါ။ တခြားကောင်းတည်းကိုရှင်းတာ တော်တော်ရှုပ်နေပါပြီ။နောက်အဲ့တကြောင်းထဲကလဲ General ဂျာနယ်အတွက် ဘာကိုဆိုလိုတယ်ဆိုတာ ရုတ်တရုတ်သဘောပေါက်မှာမဟုတ်ပါဘူး။

Motivation

ဣန္ဒြေတော့တို့က Drawing application တခုဆောက်မယ်ဆိုပါစို့ဗျာ။ အဲ့မှာ Shape တွေအများကြီးရှိမယ်ပေါ့။ ဥပမာ Circle, Rectangle အစရှိသဖြင့်ပေါ့ဗျာ။နောက် ခုနက Shape တွေဖြစ်တဲ့ Circle, Rectangle တွေကို ဣန္ဒြေတော့တို့က drawing method ဖြိုးသုံးပြီး ဆောက်ခွင်တာက ဆိုပါစို့ဗျာ SVG နဲ့ရယ် Canvas နဲ့ရယ်ပေါ့။ အဲ့တော့ ဣန္ဒြေတော့တို့က ခုနကလိုတာတွေကို class diagram ဆောက်လိုက်မယ်ဆိုရင် ဒီလိုရမယ်ဆိုပါစို့။

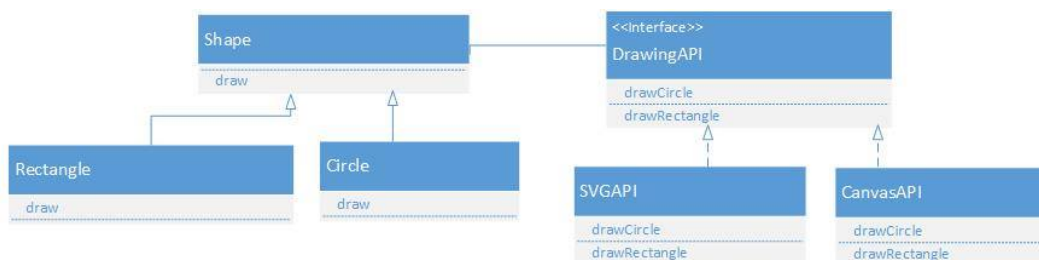


အပေါ်ကပုံမှာ Shape သည် abstract class ဖြစ်မယ် Diagram မှာတော့ထဲမထားဘူး။ Shape မှာ Rectangle, Circle သည် child shape ကနေ inherit လုပ်လာတာ။ နောက် ခုနက DrawingAPI Canvas or SVG ပေါ်မူတည်ပြီး CanvasRectangle,SVGRectangle အဲလိုခွဲထားတယ်။ ဒီ class diagram မှာ ဘာ အားနည်းချက်ရှိလဲကြည့်ရအောင်။ ဒီ diagram အရဆိုရင် Rectangle,Circle သည် abstractionလို.ပြောရမယ် ဘာလို့.လဲဆိုတော့ client ကသုံးမှာ SVG နဲ့.ဆွဲထားတာလား canvas နဲ့.ဆွဲထားတာလား သိစရာမလိုဘူး သူလိုချင်တာ conceptual ဖြစ်တဲ့ rectangle လား circle လားပဲ။ အဲတော့ conceptual level အကြည့်ရင် Rectangle, Circle,Shape သည် abstraction ပဲ။ တကယ့် တကယ်အလုပ်လုပ်တာကြတော့ CanvasRectangle ,SVGRectangle သူတို့.ကတော့ implementation လို.ဆိုရမယ် ဘာလို့.လဲဆိုတော့ တကယ့် Canvas drawing,SVG drawing ကိုလုပ်လို့.။ သူတို့.မှာ ဘာအားနည်းချက်ရှိလဲဆိုတော့ client ကသုံးမယ်ဆိုရင် ဒီလိုတွေသုံးရမယ်

```

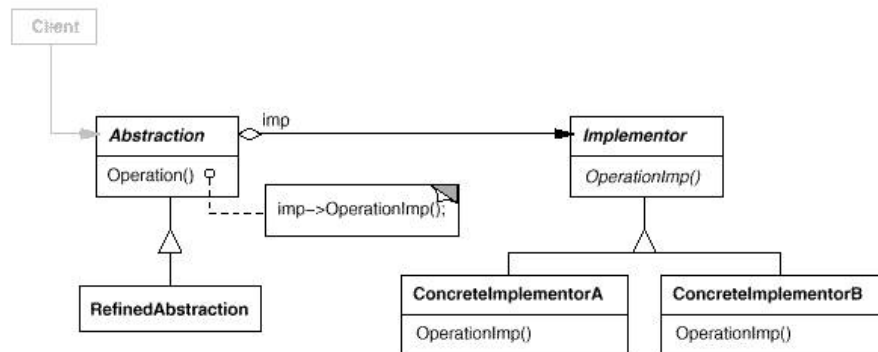
Shape s = new CanvasRectangle();
s = new SVGRectangle();
  
```

အဲတော့ ဘာအားနည်းချက်ဖြစ်လာမလဲဆိုတော့ client က သူသုံးရမဲ့ implemnetation (Canvas လား SVG လားပြောပြီး concrete class) နဲ့. create လုပ်နေရတယ်။ OO Programming မှာ program to interface not to implementation လို.ဆိုကြတယ်။ ဆိုတာက ဒီ implementation class တွေကို တိုက်ရိုက်ခေါ်နေရတော့ နောင်တခွန် 3D Drawing API လိုတခုဆောက်မယ်ဆိုရင် client က သပ်သပ်သီနေရဦးမယ် ဘယ် Object ဆောက်ဆိုတာကို။ client က implementation ကိုမသိပေးဆောက်ခွင့်ရင် Bridge Pattern ကိုသုံးလို့.ရတယ်။ ဥပမာ ဒီနေရာမှာ implementation ကို interface တခုအနေနဲ့.ထားပြီး class Diagram ကိုဒီလိုပြန်ဆောက်လိုက်မယ်ဆိုရင် implementation ကို runtime မှာပါခွဲနဲ့.လို့.ရမှာပါ။ Client ကတော့ Circle, Rectangle ဆိုတဲ့ abstraction တွေနဲ့.အလုပ်လုပ်သွားနိုင်မှာပါ။ ဘယ် specific implementation ဆိုတာကိုသိစရာမလိုတော့ပါဘူး။



အပေါ်ပုံဆိုရင် Bridge pattern နဲ့ Design ခုထားတာပါ။ ခုနက Drawing API ကို interface သက်သက်ထားပြီးတော့ သူ့ကို shape ကနေတဆင့်ပဲယူသုံးခိုင်းတာပါ။ ဒါဆို client ကကြိုက်တဲ့ API ကို ပြောင်းပြီးသုံးလို့ရပါတယ်။ နောက်ပြီး drawing code တွေကို ဆိုင်ရာ class တခုထဲအောက်မှာပဲထားတဲ့အတွက် (ဥပမာ အရင်ပုံဆိုရင် SVG drawing အတွက်ဆိုရင် ရှိသမျှ class တွေလိုက်ကြည့်ရမယ်) ခုကတော့ SVGAPI အောက်မှာကြည့်လိုက်တာနဲ့အကုန်ရပါပြီ။ ဒါကို SRP (Single Responsibility Principle) လို့ခေါ်ပါတယ်။ ဒါ့တော့ ခုပုံစံအရဆိုရင် Abstraction ဖြစ်တဲ့ Rectangle ,Circle နဲ့ implementation ဖြစ်တဲ့ SVGAPI နဲ့ CanvasAPI ကိုခွဲထုတ်လို့ရပါပြီ။ Coupling မဖြစ်တော့ဘူး ဒါ့တော့ နောက်ထပ် API တခုထပ်ထဲမယ်ဆိုရင် class hierarchy တွေမမှားပဲ class တခုပဲထဲလို့အဆင်ပြေပါပြီ။

အောက်ကတော့ GoF မှာပေးထားတဲ့ Bridge ရဲ့ class diagram ပါ။



Code ကတော့ရှင်းပါတယ်။

```
public interface DrawingAPI {
```

```
    void drawCircle();
    void drawRectangle();
```

```
}
```

ဒါကတော့ DrawingAPI interface ပါ။ သူ့မှာ drawing method တွေအကုန်နဲ့ထဲထားပါတယ်။ Different Implementation တွေကသူ့ကို implement လုပ်ယူပဲ။ ဒီမှာဆို SVGApi နဲ့ CanvasApi ပါ။

```
public class SVGApi implements DrawingAPI{
```

```
    @Override
```

```
    public void drawCircle() {
        System.out.println("Draw SVG Circle");
    }
```

```
    @Override
```

```
    public void drawRectangle() {
        System.out.println("Draw SVG Rectangle");
    }
```

```

}
public class CanvasAPI implements DrawingAPI{
    @Override

    public void drawCircle() {
        System.out.println("Draw Canvas Circle");
    }
    @Override
    public void drawRectangle() {
        System.out.println("Draw Canvas Rectangle");
    }
}

```

SVGApi နဲ့ CanvasAPI ၂ခုဟာ DrawingAPI ရဲ့ different implementation တွေပါ။ ဒါကတော့ implementation class တွေပါ။ Abstraction class တွေကတော့ဒီလိုပါ။

```

abstract class Shape {
    DrawingAPI api; //Here just use interface class

    Shape(DrawingAPI api)
    {
        this.api = api;
    }

    abstract void draw();
}

```

Shape class က DrawingAPI ကိုသုံးပါမယ် ဒါပေမဲ့ ဘယ် implementation ဆိုတာကိုမပြောပဲ DrawingAPI interface ကိုပဲ api ဆိုပြီးသွဲထားတဲ့အတွက် ကြိုက်တဲ့ implementation ကိုသုံးလို့ရမှာပါ။ ဒါကနောက်ပိုင်း DrawingAPI ကနေ တခြား different implementation တွေ ဥပမာ ThreeDAPI ထပ်ထဲ့ရင်လဲ အဆင်ပြေပါတယ်။ Shape constructor မှာ ဘယ် api နဲ့ဆွဲမလဲဆိုတာကိုလက်ခံပါတယ်။

အောက်က Circle နဲ့ Rectangle ပါ။

```

public class Circle extends Shape{
    public Circle(DrawingAPI api) {
        super(api); // call parent constructor
    }
    @Override
    void draw() {
        api.drawCircle();
    }
}

```

```

public class Rectangle extends Shape{
    public Rectangle(DrawingAPI api) {
        super(api);
    }

    @Override
    void draw() {
        api.drawRectangle();
    }

}

```

Circle ရဲ့ draw ရဲ့ Rectangle ရဲ့ draw မှာသက်ဆိုင်ရာ API ရဲ့ drawCircle ,drawRectangle ကိုခေါ်ပါ
တယ်ဒါပေမဲ့ ဒီမှာသတိထားရမှာက ဘယ် API ဆိုတာမပါပါဘူး ဒါသည် loose coupling ဖြစ်အောင်လုပ်
ထားတာလို့ဆိုရမှာပါ။ ကြိုက်တဲ့ API နဲ့တွဲသုံးလို့ရတယ်ပေါ့ဗျာ။ အောက်ကတော့ Client code ပါ။

```

public class BridgeDemo {
    public static void main(String[] args) {
        DrawingAPI api = new SVGAPI();

        Shape s = new Rectangle(api);
        s.draw();

        api = new CanvasAPI();
        s = new Circle(api);
        s.draw();

    }
}

```

Client demo အရဆိုရင် ဂျွန်တော်တို့သုံးနေတာသည် ဟိုးပထမဆုံးပြထားတဲ့ကောင်မှာလို SVG circle
လား Canvas Circle လားသိစရာမလိုပါဘူး ဒါသည် abstractionပါ။ နောက် ကြိုက်တဲ့ drawing API ကို
ပြောင်းလို့ရတဲ့အတွက် Decouple an abstraction from its implementation ဆိုတာကို လုပ်လို့ရနေ
တာပါပဲ။ ကောင်းတဲ့အခွက်ကတော့ class hierarchy ဖောင်းပွမှုကို ကာပေးနိုင်တယ်။ Runtime မှာ
different implementation ကို ပြောင်းလို့ရမယ်၊ specific implementation အပေါ် client code က မမှီ
ခိုတဲ့အတွက် နောက် implementation class API တွေထပ်ထဲရင်လွယ်မယ်ပေါ့ဗျာ။

Object Oriented Design Pattern Series

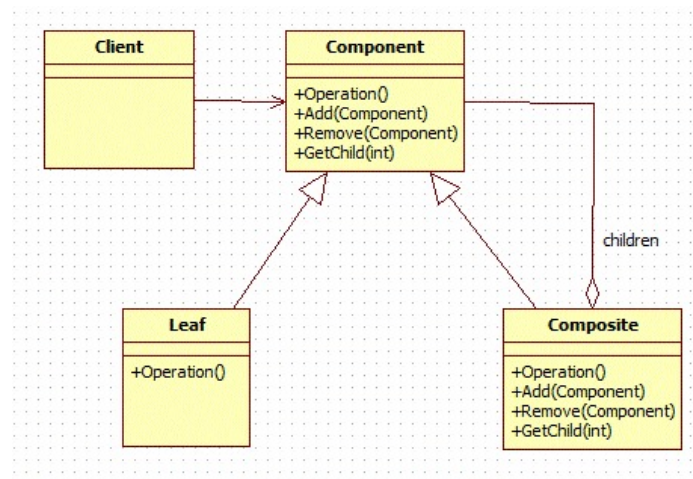
Part-8 Composite Design Pattern

THET KHINE · FRIDAY, SEPTEMBER 2, 2016

Composite Design pattern က Structural pattern တခုပါ။ သူ့ကိုသုံးနေရာ မြင်နေရပါ။ သူက Tree ပုံစံ Hierarchy object တွေကို တည်ဆောက်ပြီး သူတို့ကို same interface နဲ့ ထိန်းချုပ်ရင်းသုံးပါတယ်။ ဥပမာ Java Swing မှာ Panel ထဲမှာ panel တွေ button တွေ input groupတွေ ထဲလို ရပါတယ်။ နောက် Android မှာဆိုလဲ ViewGroup ထဲမှာ view တွေ ViewGroup တွေကို tree ပုံစံနဲ့ ထဲလို ရပါတယ်။ နောက် runtime system ကနေ သူတို့ကို screen ပေါ်ပြန်တဲ့အခါ paint method ကိုခေါ်ရပါတယ်။ ဒီအခါမှာ ခုနက tree structure ဆောက်ထားတဲ့ object တွေကို traverse လုပ်ဖို့ လိုပါပြီ။ ဒီလိုအခြေအနေမျိုးမှာ Composite pattern ကိုသုံးပါတယ်။ အဓိက ကတော့ object တွေဟာ hierarchical structure နဲ့ အထဲမှာ တဆင့်ခြင်းရှိနေမယ်ဆိုရင် ဒီ pattern နဲ့ တည်ဆောက်ယူလို့ ရပါတယ်။

Intent

"Compose objects into tree structure to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly". GoF မှာတော့သူ့ intent ကိုဒီလိုပြထားပါတယ်။ Object တွေကို tree structure အနေနဲ့ တည်ဆောက်ပြီး ဆောက်ထားတဲ့ object တွေကို uniformly treat လုပ်နိုင်တာပါ။ ဆိုနိုင်တာက individual ရယ် Composite ရယ်မခွဲပဲ သုံးလို ရနိုင်တယ် သုံးနိုင်တယ်ဆိုတဲ့ အဓိပ္ပာယ်ပါ။ ဥပမာ Android Runtime system ကနေ ViewGroup တခုကို paint ခေါ်လိုက်ရင် သူ့အထဲမှာရှိတဲ့ View တွေ ViewGroup တွေကိုပါ တခါတည်း paint လုပ်သွားစေနိုင်တယ်။ View လား ViewGroup လားခွဲခြားပြီး သိစရာမလိုနိုင်ဘူး အဲဒါဆိုဒီ Composite pattern နဲ့ ရေးလို့ ရပါပြီ။ သူ့ရဲ့ class diagram ကတော့ အောက်ကလိုပါ။



Component ကတော့ အားလုံးသုံးမဲ့ (Leaf, Composite) ကနေသုံးမဲ့ operation တွေစုထားတာပါ။ ရှု န်တော့တို. example အရ ဆို paint method ပေါ့။ နောက် leaf ကတော့ ထပ်ခွဲလို့မရတဲ့ nested child တွေ မပါတော့တဲ့ object တွေကို leaf လို့ သတ်မှတ်ရမှာပါ။ Leaf ကတော့ View ဖြစ်မှာပါ။ Composite

ကတော့ ViewGroup ပါပဲ။ သူ့မှာ child element တွေပါနိုင်ပါတယ်။ ViewGroup ထဲမှာ View တွေရော ViewGroup တွေပါ ကြိုက်သလောက်ပါနိုင်ပါတယ်။ ဒီနေရာမှာ Composite မှာ add ရယ် remove ရယ် getChild ဆိုတာရယ်ရှိပါတယ်။ သူတို့ကတော့ Composite ထဲကို child element တွေ ထဲခွဲထဲအခါ ရယ် traverse လုပ်ဖို့ရယ်သုံးတာပါ။ Code ကိုကြည့်ရအောင်။ Uniform interface ရဖို့အတွက်လှည့်ကျတော့ တို့အောက်က UIWidget class ကိုဆောက်လိုက်ပါတယ်။

```
public abstract class UIWidget {  
    String id;  
    public UIWidget(String id) {  
        this.id = id;  
    }  
  
    abstract void paint();  
}
```

သူ့မှာ id ပါပါတယ် ဒါကတော့ ပြုလွယ်အောင်ပါ။ Variable name ပေးထားသလိုပေါ့ဗျာ။ နောက် paint ပါပါတယ်။ ဒါကတော့ UIWidget တိုင်းက paint လုပ်ရမယ် ဒါကြောင့် သူ့ကို abstract ထားပေးထားတာ ပါ။ သူ့ကို extends လုပ်တဲ့ Child တွေက override လုပ်စေချင်လို့ပါ။ ဒါမှ child ပေါ်မူတည်ပြီး different implmenation ပေးလို့ရမှာကိုး။ နောက် Leaf အမျိုးအစားဖြစ်တဲ့ View class ကိုဒီလိုရေးပါတယ်။

```
public class View extends UIWidget{  
  
    public View(String id) {  
        super(id);  
    }  
  
    @Override void paint() {  
        System.out.println("Paint "+this.id);  
    }  
  
}
```

သူ့မှာ ထူးထူးဆန်းဆန်း ဘာမှမပါပါဘူး Constructor ဆောက်တယ် parent ကို Id ပေးလိုက်တယ်။ နောက် သူ့ paint မှာ ဘယ်သူ့ကို paint လုပ်တယ်မှန်းသိအောင် Paint id ဆိုပြီးထုတ်ပြတယ်။ ဒါပါပဲ။ နောက်က Composite class ဖြစ်တဲ့ ViewGroup ကိုဒီလိုရေးပါတယ်။

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class ViewGroup extends UIWidget{  
    List<UIWidget> children = new ArrayList<UIWidget>(); //Here store UIWidget list  
  
    public ViewGroup(String id) {  
        super(id);  
    }  
  
    public void add(UIWidget child)
```

```

{
    this.children.add(child);
}

@Override
void paint() {
    System.out.println("Paint "+this.id);
    for(UIWidget child : this.children)
    {
        child.paint();
    }
}
}

```

ViewGroupသည် composite ဖြစ်တဲ့အတွက်တခြား View, ViewGroup တွေပါလို့ရ ရမှာပါ။ ဒါကြောင့် အဲ့ကောင်တွေကိုသိမ်းဖို့.

```
List<UIWidget> children = new ArrayList<UIWidget>();
```

ဆိုပြီးရေးပါတယ်။ ဒီနေရာမှာ သတိထားရမှာက သိမ်းတာသည် UIWidget ပါ သူသည် parent ဖြစ်တဲ့အတွက် View ရော ViewGroup ပါသိမ်းလို့ရမှာပါ။ ဒါကအရေးကြီးပါတယ်။ နောက် View, ViewGroup တွေထဲဖို့အတွက် add method ကိုရေးပါတယ်။ ဒါကလဲ ဘာမှမဆန်းပါဘူး။ children list ထဲကိုထဲလိုက်တာပါ။ နောက် paint method မှာတူတော့ ဂျွန်တော်တို့က Composite ကိုယ်တိုင်ကို paint လုပ်ရပါမယ်။ နောက် သူ့ရဲ့ children တွေကိုလဲ for each loop ကိုသုံးပြီး paint လုပ်ခိုင်းပါတယ်။ Client Demo ကတော့ အောက်ပါအတိုင်းပါ။

```

public class CompositeDemo {
    public static void main(String[] args) {
        ViewGroup viewGroup = new ViewGroup("parentGroup");
        View view1 = new View("view1");
        View view2 = new View("view2");
        viewGroup.add(view1);
        viewGroup.add(view2);

        ViewGroup viewGroup2 = new ViewGroup("childGroup");
        viewGroup2.add(new View("level2child"));

        viewGroup.add(viewGroup2);

        viewGroup.paint();
    }
}

```

ViewGroup type viewGroup ဆိုပြီး parentGroup တခုဆောက်ပါတယ်။ သူ့ထဲကို View 2 ခုထဲပါတယ်။ နောက်ပြီး ViewGroup တခုဖြစ်တဲ့ childGroup ကိုထဲပါတယ်။ သူ့မှာ level2child ဆိုတဲ့ view တခုပါပါတယ်။ နောက် viewGroup.paint() ကိုခေါ်လိုက်ရင် ဒီလိုထွက်လာမှာပါ။

Paint parentGroup

Paint view1

Paint view2

Paint childGroup

Paint leve2child

viewGroup ကို paint လုပ်လိုက်တာနဲ့. Composite pattern ကြောင့် hierarchy တစ်ခုရောက် paint လုပ်သွားမှာပါ။ ဒါဆိုရင် ဒီ pattern ကိုဘယ်လိုအလုပ်လုပ်တယ် ဘယ်နေရာမှာသုံးတယ် ဆိုတာ သိလောက်ပါပြီ။ လက်တွေ့ကတော့ Android UI, Swing UI, အစရှိတဲ့ UI တွေမှာသုံးပါတယ်။ နောက်ထပ်သုံးလို့ရတဲ့နေရာကတော့ tree datastructure ဆောက်တဲ့နေရာမှာ ဒီ Composite pattern ကိုသုံးလို့ရတယ်ဆိုတာပါပဲ။ သူ့အားသာချက်ကတော့ ခု paint နေတာသည် Leaf လား composite လားသိစရာမလိုပဲ interface တခုတည်းပဲအလုပ်လုပ်သွားမှာပါ။ ဒီနေရာမှာတော့ paint ပေါ်မှာ။

Object Oriented Design Pattern Series

Part-9 Decorator Design Pattern

THET KHINE · SATURDAY, SEPTEMBER 10, 2016

Decorator pattern ကတော့ Sturctural pattern ထဲကတခုပါ။ သူ့ကို အသုံးပြုတာကတော့ runtime မှာ object တွေအတွက် responsibility တွေကို dynamically ထဲ ချင်တဲ့အခါမှာသုံးပါတယ်။ ဒီနေရာမှာ responsibility ဆိုတာ a set of public method (some useful method) လို့ဆိုရမှာပါ။ OO programming မှာ class တခုကို behaviour တွေ ထဲခွဲခွဲအခါမှာ သုံးလေ့ရှိတဲ့နည်းက Inheritance ၊ Composition ကိုသုံးကြပါတယ်။ ဥပမာ ဣန္ဒာတောတို့က basic logger တခုရှိတယ်ဆိုပါစို့။ သူက console ကိုပဲ logging ပဲလုပ်နိုင်တယ်။ နောက် HTMLLogger , XMLLogger ထပ်ရှိမယ်ဆိုပါစို့။ သူတို့တွေကတော့ HTML XML တွေနဲ့ပါ log ရှိနိုင်တယ်ဆိုပါစို့။ ဒါဆို ခုနက Logger တွေမှာ functionality အသစ်ထပ်ထဲ မယ် ဥပမာ ဖွာ ရိုးရိုး log message လေးတင်မကပဲ အရှေ့မှာ Time ထပ်မယ် ပေါ့ အဲဒီကောင်ကို HTML Logger မှာရော XML logger မှာရောသုံးစေချင်တယ်။ ဒါဆိုရင် ခုနက Inheritance အရလုပ်မယ်ဆိုရင် TimeAddedLogger ဆိုတာက HTMLLogger XMLLogger ဆိုတာတွေကို extend လုပ်ပစ်ရမှာပါ ဒါဆို ဣန္ဒာတောတို့က class တွေ ဖောင်းပွကုန်ပါလိမ့်မယ်။ နောက်ပြီး တကယ်လို့ JSON ထုတ်တဲ့ logger ဆိုပါ စို့။ အဲလိုနောက် Logger class တခုထပ်ထဲမယ်ဆိုရင် ခုနက Time ပါပါတဲ့ Logger ကို ထပ် extend လုပ် ရဦးမယ် အဲဒါတော့ အဆင်မပြေဘူး။ ဖြစ်စေချင်တာက ခုနက TimeAddedLogger ဆိုတာကို decorator class တခုအနေနဲ့ထားလိုက်မယ် ပြီးတော့မှ အဲဒီကောင်ကို တခြား Logger တွေနဲ့ tight coupling မဖြစ်စေပဲနဲ့။ Decorator pattern သုံးပြီး သုံးလိုက်မယ်ဆိုရင် Logger တွေဘယ်လောက်များများ အဆင်ပြေနိုင် ပါတယ်။ နောက်ပြီး Decorator ကလဲ time တခြား functionality တွေပါ ထဲဖို့ ထပ်လုပ်နိုင်ပါတယ်။ အဲဒါ တော့ရှိပြီးသား Logger class တွေကို မထိခိုက်စေပဲနဲ့။ သူတို့ functionality ကို runtime မှာထပ်ခွဲလို့ရမ ယ်။ Tightly coupled လဲမဖြစ်တော့ဘူးပေါ့။ ဒါက Decorator ရဲ့ intent ပါ။ Decorator က ဘာလဲဆိုတာ ကို သေချာ define လုပ်ရင်တော့ object တွေအတွက် additional behaviour ကို runtime မှာ ပေါင်း ထဲပေးနိုင်တဲ့ pattern လို့ပဲပြောရမှာပါ။

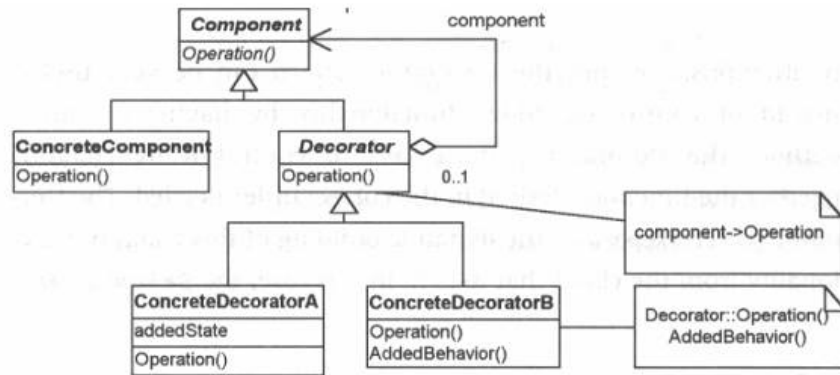
Intent

သူ့ Intent ကိုတော့ GoF မှာ ဒီလိုပြထားပါတယ်။

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Java IO library မှာဆိုရင် Decorator တွေကိုတော့တော့သုံးထားပါတယ်။ ဥပမာ `BufferedInputStream` ဆိုတာ Decorator class ပါ။ သူက ဘာကို decorate လုပ်သလဲဆိုရင် `InputStream` ကို decorate လုပ် ပါတယ်။ နဂိုမူရင်း `InputStream` မှာ buffer မသုံးပါဘူး။ ဒါကြောင့် သူ့ကိုသုံးရင် နေပါတယ်။ `BufferedInputStream` ကတော့ buffer သုံးပြီး `InputStream` ကို decorate လုပ်လိုက်တဲ့အတွက် ပိုမြန် ပါတယ်။ ဒါပေမဲ့ `BufferedInputStream` သည် `InputStream` ကို inherits လုပ်ပြီး သုံးတာမဟုတ်ပဲနဲ့။ Decorator pattern အနေနဲ့သုံးတာပါ။ ဒါကြောင့် `BufferedInputStream` ကို `FileInputStream` ကို decorate လုပ်ဖို့သုံးမယ်ဆိုလဲရသွားပါတယ်။ ဘာလို့လဲဆိုတော့ `FileInputStream` သည် `InputStream` ရဲ့ subclass ဖြစ်နေလို့ပါပဲ။ ဒီလောက်ဆို Decorator က class hierarchy ကိုမဖောင်းပွစေပဲ tight coupling မဖြစ်စေပဲ ဘယ်လို runtime behaviours တွေကို add လုပ်နိုင်တယ်ဆိုတာကို သဘောပေါက်မှာ

ပါ။ နောက်ပြီး Decorator တွေကို တဆင့်ပြီးတဆင့်သုံးလို့လဲရပါတယ်။ သူ့ class diagram ကိုတော့ အောက်မှာပြထားပါတယ်။



Code ကိုကြည့်ရအောင် အောက်ကတော့ Logger interface ပါ။

```

public interface Logger {
    String log(String msg);
}

```

ဘာမှ ထူးထူးထွေထွေမပါဘူး log() ဆိုတဲ့ method လေးပဲထွေးထားတာပါ။ ဣန်တာနက်တို. basic logger class တခုဆောက်ပါမယ်။ အောက်ကလိုပေါ့။

```

public class BasicLogger implements Logger{
    @Override
    public String log(String msg) {
        return msg;
    }
}

```

Basic Logger ကလဲထွေထွေထူးထူးမရှိပါဘူး Logger ကို implement လုပ်ပြီးသူ့ကို ပို.လာတဲ့ log message ကို return ပြန်ရုံပါပဲ။ ဣန်တာနက်တို.က HTML နဲ့. log ထုတ်ခွင့်တယ် Decorator ကိုသုံးမယ် ဒါဆို ဒီလိုရေးလိုက်ပါမယ်။

```

public class HTMLDecorator implements Logger{
    Logger logger;
    public HTMLDecorator(Logger logger) {
        this.logger = logger;
    }
}

```

```

@Override
public String log(String msg) {

```

```

    return "<html>" + logger.log(msg) + "</html>";
}
}

```

Decorator မှာ inheritance ကိုမသုံးပဲ Composition ကိုသုံးပါတယ်။

Logger logger ဆိုတာ composition ကိုသုံးထားတာပါ။ သူ့ထဲမှာ constructor တခုပါပါတယ်။ အဲ့ဒီ constructor က သူ decorate လုပ်မဲ့ class family ဖြစ်ရမှာပါ ဒီနေရာမှာ Logger interface ပါ။ ဒါဆိုရင် ဒီ HTMLDecorator ဟာ Logger ကို implement လုပ်ထားတဲ့ ဘယ် Logger ကိုမဆို decorate လုပ်နိုင်မှာပါ။ သူ့ထဲမှာ log method ကိုသေချာကြည့်ပါ။ Log method မှာ သူက ဒီလိုရေးထားပါတယ်။

```

return "<html>" + logger.log(msg) + "</html>";

```

သူက html start tag ,end tag တွေထဲပြီး decorate လုပ်ပါတယ် အလည်မှာတော့ logger.log ဆိုပြီး သူ့ကို parameter ပေးထားတဲ့ object method ကိုသုံးပါတယ်။ ဒါသည် Decorator ရဲ့အသက်ပါ။ နဂိုမူလက ရှိပြီးသား behaviour ဖြစ်တဲ့ log ကို ဂျွန်စော်တို့က html tag တွေထဲပြီး decorate လုပ်လိုက်တာပါ။ နောက် TimeDecorator ကိုဆောက်ပါမယ်။

```

import java.util.Date;

```

```

public class TimeDecorator implements Logger{
    Logger logger;
    public TimeDecorator(Logger logger) {
        this.logger = logger;
    }
}

```

```

@Override
public String log(String msg) {
    Date date =new Date();
    return date.toString() + " " + logger.log(msg);
}

}

```

ခု TimeDecorator သည်လဲ ခုနက HTMLDecorator နဲ့သဘောတရားတူတူပါပဲ။ Constructor မှာ Decorate အလုပ်ခံရမဲ့ object family ကိုလက်ခံတယ် composition သုံးပြီး သိမ်းမယ်။ နောက် log ဆိုတဲ့ method မှာ date.toSting() ဆိုပြီး နဂို logger.log ကို ထပ် decorate လုပ်တယ်။ Client Demo ကတော့ အောက်မှာပါ

```

public class Demo {
    public static void main(String[] args) {
        Logger lg = new HTMLDecorator(new TimeDecorator(new BasicLogger()));
        String msg = lg.log("LogString");
        System.out.println(msg);
    }
}

```


အပေါ်က ကောင်ကို run လိုက်ရင် ဒီလိုရမှာပါ။

```
<html>Sat Sep 10 17:24:38 MMT 2016 LogString</html>
```

Time ကတော့နည်းနည်းပြောင်းမှာပေါ့လေ။ အောက်က logger ကို create လုပ်တဲ့နေရာမှာ ဣန်တော တို့က decorator တွေကို chain လုပ်ပြီးသုံးသွားတာပါ။ တကယ်လို့ HTMLDecorator ကိုမလိုဘူးဆိုရင် [ဖုတ်ပစ်လိုက်လို့လဲရပါတယ်။ ဒါဆိုရင် Decorator သည် ဘာကြောင့် အသုံးတူတယ် ဆိုတာ သဘောပေါက်မယ်ထင်ပါတယ်။

```
Logger lg = new HTMLDecorator(new TimeDecorator(new BasicLogger()));
```

နောက်လာမဲ့ JavaScript ES7 မှာ decorator တွေကို propose လုပ်ထားပါတယ်။ Typescript မှာလဲ decorator တွေကို language construct အနေနဲ့ ထည့်ပေးထားပါတယ်။ ဒဲ့တော့ Angular2 မှာ Decorator တွေကို Component တွေကို decorate လုပ်ဖို့ သုံးကြပါတယ်။ ဆိုရှင်တာက language feature အနေနဲ့ pattern တွေကို ပေးထားလိုက်ရင် တော်တော်အသုံးတူမယ် ဒီလို pattern တွေပေါ်မူတည်ပြီး code တွေ သီးသန့် လိုက်ရေးစရာမလိုတော့ဘူးပေါ့ဗျာ။

Object Oriented Design Pattern Series

Part-10 Observer Design Pattern



THET KHINE · SATURDAY, SEPTEMBER 24, 2016

Observer pattern က behaviour pattern ထဲက တခုပါ။ သူ့က GoF 23 မျိုးထဲမှာ တော်တော်သုံးတဲ့ pattern လို့ဆိုရပါလိမ့်မယ်။ MVC (Model View Controller) design pattern ထဲမှာ Observer ကိုသုံးသုံးထားရပါတယ်။ နောက် Reactive Programming မှာ Reactive stream တွေကို Observer တွေကိုသုံးပြီး implement လုပ်ထားတဲ့အတွက် Reactive Programming ကိုနားလည်ခွင်ရင် Observer pattern ကိုနားလည်ထားတာ အကောင်းဆုံးပါ။

Intent

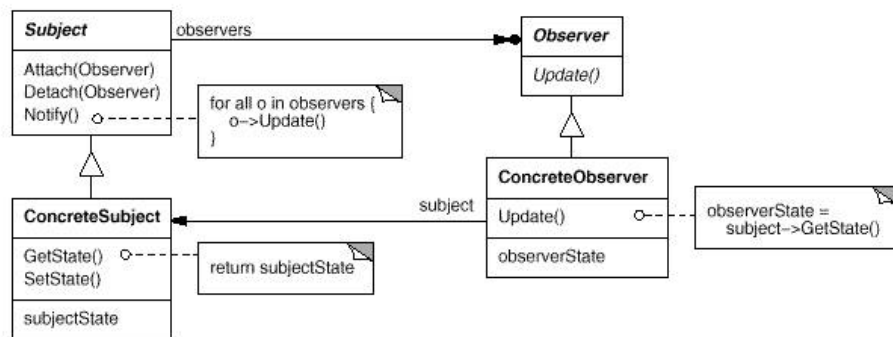
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

အပေါ်ကတော့ Observer pattern ရဲ့ intent ကို GoF မှာရေးထားတဲ့အတိုင်းပြထားတာပါ။ ဆိုခွင်တာက one to many dependency ရှိတဲ့ Object တွေကြားက behaviour ကိုထိန်းတဲ့ pattern ပါ။ Object တခုက သူ့ရဲ့ state ကိုပြောင်းလိုက်တာနဲ့ သူ့ကိုမှီခိုနေရတဲ့ dependency ရှိနေတဲ့ကောင်တွေကို ငါရဲ့ state တွေပြောင်းသွားပြီဆိုပြီး notify (အသိပေးတဲ့) pattern ပါ။ စာကိုစာအတိုင်းရှင်းတော့ရှုပ်ပါတယ်။ ဥပမာပြောရအောင်ပါ။

ဥပမာ ဂျွန်တော့တို့မှာ model တခုရှိတယ်ဆိုပါစို့။ Model ဆိုတာ အမှန်တော့ဘာမှမဟုတ်ဘူး business data ကိုသိမ်းထားတာ။ ဆိုပါစို့။ ဖွဲ့သောတဲ့ဥပမာ ပေါ့ stock price ဆိုတဲ့ model ပေါ့။ ဒါ့ဒီ Stock price တွေကိုပြဖို့ကြတော့ View တွေလိုတယ်။ View ကလဲကြည့်တဲ့သူအပေါ်မူတည်ပြီး Web ကကြည့်တာရှိမယ်။ နောက် desktop ကကြည့်တာရှိမယ်။ (ဒီနေရာမှာ client server တွေထဲမစဉ်းစားပဲခန့်မေ့ထားလိုက်ပါ။) ဒါ့တော့ StockPrice ဆိုတဲ့ model ရဲ့ data ပေါ်မူတည်ပြီး View1 View2 ပြရမယ်ဆိုပါစို့။ View1 က StockPrice ကိုပုံစံတမျိုးနဲ့ပြမယ် View2 က StockPrice ကိုနောက်ပုံစံတမျိုး(တူလဲရတာ ကိစ္စမရှိဘူး)ပြမယ်ပေါ့။ ဒါဆိုရင် ခုနက StockPrice ဆိုတဲ့ View1, View2 Object ၂ ခုသည် StockPrice ဆိုတဲ့ model object ကိုမှီခိုနေရတယ်။ Dependence ဖြစ်နေတယ် ဘာလို့လဲဆိုတော့ StockPrice က data ကိုသုံးပြီးတော့ သူတို့ View တွေကို render လုပ်ရလို့။ StockPrice က တခုတည်း View1, View2 က ၂ခု multi ပေါ့။ ဒါသည် one-to-many relation လို့ဆိုရမှာပေါ့။ နောက်ထပ် View တွေကို dynamically ထဲ့ပြီး 2 ခုထက်မကလဲဖြစ်နိုင်ရင်ဖြစ်နိုင်သေးတယ်။ View တွေလွှောသွားတာလဲဖြစ်ရင်ဖြစ်မယ်။ ဒါ့တော့ StockPrice ရဲ့ data ဟာ အကြောင်းတခုကြောင့် ပြောင်းသွားတယ်ဆိုရင် ခုနက သူ့ကိုမှီခိုနေရတဲ့ View1 နဲ့ View2 သည် လိုက်ပြောင်းပေးရမှာပါ။ ဒါဆိုရင် View1 နဲ့ View2 သည် StockPrice ပြောင်းလား မပြောင်းလားဆိုတာ ကို တချိန်လုံးထိုင်ကြည့်နေရမယ်ဆိုပါစို့။ ဒါဆိုရင် သိပ်အဆင်မပြေပါဘူး။ ဒါ့လိုမဟုတ်ပဲ Hollywood principle အရ Don't call us , we will call you သုံးရင်ရော။ ဆိုခွင်တာကမှာ View1 ရော View 2ရောက StockPrice ကိုကြည့်မနေပဲနဲ့ pulling မလုပ်ပဲနဲ့။ StockPrice ကနေ data change တော့မှ View1 ကိုရော View2 ကိုရော အကြောင်းကြားရင်ရော (notify) လုပ်တယ်ဆိုပါစို့။ ဒါဆိုပိုအဆင်ပြေပါတယ်။ ဘာလို့လဲဆိုတော့ တချိန်လုံး pulling လို ပြောင်းသွားပြီလား ပြောင်းသွားပြီလား ကြည့်စရာမလိုပဲနဲ့ တကယ်ပြောင်းတော့မှ notify (push) လုပ်ပေးတော့ပိုအဆင်ပြေတာပေါ့။ ဒီလို situation မျိုးဆို Observer pattern ကိုအသုံးပြုရပါလိမ့်မယ်။

အတိုချုပ်ပြောရရင် Object တွေမှာ 1-to-many dependency ရှိမယ်။ Object ခုကပြောင်းလိုက်တာနဲ့ နောက် Object တွေကို effect ဖြစ်မယ် changes လုပ်ရမယ်ဆိုရင် Observer pattern ကိုသုံးရပါလိမ့်မယ်။ MVC မှာ model တွေပြောင်းသွားတာနဲ့ View ကို update လုပ်လိုက်တာဟာ Observer ကိုသုံးပြီး လုပ်သွားတာပါ။

Class Diagram ကိုအောက်မှာပေးထားပါတယ်။



အပေါ်က Class Diagram ထဲက role တွေကိုနည်းနည်းရှင်းပါရစေ။ Subject ဆိုတာ စိတ်ဝင်စားတဲ့ entity တခုခု ခုနက ဥပမာအရဆို Model ပေါ့ဗျာ။ Observer ကတော့ ခုနက Subject ကိုစောင့်ကြည့်နေတဲ့ Subject အပေါ်မူတည်နေရတဲ့ Object တွေကို Observer လို့ဆိုခွင့်တာပါ။ ခုနက Example အရဆိုရင်တော့သူကတော့ View ပေါ့ဗျာ။ Subject တခုမှာ Observer တွေအများကြီးပါနိုင်ပါတယ်။ Observer တွေကိုလဲ dynamically add ,remove လုပ်လို့ရရပါလိမ့်မယ်။ Subject က တခုခုပြောင်းသွားတာနဲ့ Subject ကို dependence ဖြစ်နေတဲ့ Observer တွေကို notify လုပ်မှာပါ(Observer object တွေရဲ့ callback function တခုခုကို လှမ်းခေါ်လိုက်တာပါ) အဲ့တော့မှ Observer တွေက Subject change လုပ်တာကိုသိပြီး သူတို့လိုအပ်တဲ့ function တွေ (ဥပမာ View ကို model subject အပေါ်မူတည်ပြီး လိုက်ပြောင်းတာမျိုး) လုပ်မှာပါ။ နောက်တခုက Observer pattern ကို publish subscribe pattern ရယ်လို့လဲခေါ်ကြပါသေးတယ်။ Subject ကတော့ publish လုပ်တဲ့အတွက် publisher ပေါ့ဗျာ။ Observer ကတော့ Subscriber ပေါ့။ FB ဥပမာနဲ့ဆိုရင် Author နဲ့ Follower ပေါ့။ Author ကတော့ Subject ဖြစ်ပြီး Follower တွေကတော့ Subscriber or Observer ပေါ့။ Author ကတခုခု publish လုပ်တာနဲ့ Follower တွေဆီမှာလာပေါ်နေမှာပါ။ notify လုပ်တယ်ပေါ့ဗျာ။

Java API မှာ Observer,Observable ဆိုပြီးပါလာပြီးသားရှိပါတယ်။ ဂျွန်ဇော်ကတော့သူတို့ကိုမသုံးပဲ အစအဆုံး pattern ကို implement လုပ်ပါမယ်။

```

public interface Publisher {
    public void attach(Subscriber subscriber);
    public void change(String message);
}
  
```

ဒါကတော့ Subject သို့မဟုတ် Publisher interface ပါ။ Subject ရဲ့တာဝန်က Subscriber သို့မဟုတ် Observer တွေကို လက်ခံရပါမယ်။ ဒါကြောင့် attach(Subscriber) ဆိုတာကိုရေးထားတာပါ။ နောက် Publisher or Subject သည် data or state change လုပ်နိုင်ဖို့ပါရပါမယ်။ ဒါကို change အနေနဲ့ရေး

ထားပါတယ်။ နောက် class ကတော့ Subscriber ပါ။ ဒီနေရာမှာ publisher က detach လဲလှယ်နိုင်ရမှာ ပါ။ ဒီမှာတော့နေထားခဲ့ပါမယ်။

```
public interface Subscriber {  
  
    public void update(String message);  
  
}
```

Subject or Publisher က တခုခု change ပြုဆိုတာနဲ့ Subscriber ရဲ့ callback method တခုခုကိုခေါ်ရမှာပါ။ ဒါကြောင့်ဒီနေရာမှာ update method ဆိုပြီး callback ထားလိုက်ပါတယ်။ဒီ update ကို Publisher ကခေါ်မှာပါ။ဒါကြောင့် callback method (ကိုယ်ကိုယ်တိုင်သုံးတာမဟုတ်ပဲ သူများက ကိုယ့်ကိုပြန်ခေါ်ဖို့ ပေးထားတဲ့ method) လို့ဆိုလိုတာပါ။ အောက်က class ကတော့ DataSource ပါ အပေါ်က ပြခဲ့တဲ့ ဥပမာ အရဆိုရင် StockPrice model နဲ့ role တူတဲ့ class ပေါ့ဗျာ။

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class DataSource implements Publisher {
```

```
    List<Subscriber> subscriberList = new ArrayList<Subscriber>();  
    List<Subscriber>  
    subscriberList = new ArrayList<Subscriber>();
```

```
    @Override  
    public void attach(Subscriber subscriber) {  
  
        subscriberList.add(subscriber);  
    }
```

```
    @Override  
  
    public void change(String message) {  
        System.out.println("Publisher Change "+message);  
        for(Subscriber sub : subscriberList)  
        {  
            sub.update(message);  
        }  
    }  
  
}
```

DataSource သည် Publisher ကို implement လုပ်ပါတယ်။ သူသည် Subject ဖြစ်တဲ့အတွက် Observer or Subscriber တွေကလာ connect လုပ်မှာပါ။ ဒါကြောင့် သူ့ထဲမှာ Subscriber တွေကိုသိမ်းဖို့ဒီလိုရေး

ထားပါတယ်။

```
List<Subscriber> subscriberList = new ArrayList<Subscriber>();
```

နောက် attach method မှာ ခုနက subscriberList ထဲကို ဒီလိုထဲပေးလိုက်ရုံပါပဲ။

```
subscriberList.add(subscriber);
```

အဲ့တော့ Subscriber ကသူ့ကို connect လုပ်ခွင့်ရင် attach method ကို လှမ်းခေါ်ပြီး connect လုပ်လိုက်ရမှာပါ။ အများကြီး one-to-many လုပ်ခွင့်တဲ့အတွက် List နဲ့ သိမ်းထားတာပါ။ နောက်အရေးကြီးတဲ့ method က change ပါ။ ဆိုပါစို့. Publisher ,Subject ကပြောင်းပြီဆိုရင် ကျွန်တော်တို့က သူ့ကိုမှီခိုနေတဲ့ Observer or Subscriber တွေကို လှမ်း notify လုပ်ရပါမယ်။ ဒါကလွယ်ပါတယ်။ change method မှာ ဒီလိုရေးထားလိုက်ပါတယ်။

```
for(Subscriber sub : subscriberList)
{
    sub.update(message);
}
```

သဘောကတော့ သူ့ကို subscribe or connect လုပ်ထားတဲ့ subscriber တွေကို သူ့ changes တခုခုဖြစ်သွားပြီဆိုတာကို လှမ်းအကြောင်းကြားတာပါ။ ဒါဆို Observer , Subscriber တွေက သူတို့ data ကိုသူတို့ပြန် ပြောင်းအမှမဟုတ် တခုခုလုပ်ရမှာပါ။ Subscriber or Observer ကို ဒီလိုရေးထားပါတယ်။

```
public class View implements Subscriber{
```

```
    String viewName;
```

```
    public View(String viewName) {
        this.viewName = viewName;
    }
```

```
    @Override
```

```
    public void update(String message) {
        System.out.println("View "+viewName+" Update to "+message);
    }
```

```
}
```

View ကတော့ Subscriber ကို implement လုပ်ထားတဲ့အတွက် update method ကို override လုပ်ပေးရမှာပါ။တကယ်လို့. Subject or Publisher ကပြောင်းတာနဲ့. Subject or Publisher က View (Observer or Subscriber)ရဲ့ update ကိုလှမ်းခေါ်မှာပါ။ အဲ့မှာ View ကသူ့ဟာသူပြန် render လုပ်တာမျိုးလို့လုပ်ပေါ့။ခုကတော့ output ယူထုတ်ပြထားတာပါ။ Demo ကတော့အောက်မှာပါ။

```
public class Demo {
```

```
    public static void main(String[] args) {
        Publisher dataSource = new DataSource();
```

```

View view1 = new View("View1");
View view2 = new View("View 2");
View view3 = new View("View 3");

dataSource.attach(view1);
dataSource.attach(view2);
dataSource.attach(view3);

dataSource.change("Change1");
dataSource.change("Chage 2");

}

}

```

ပထမဆုံး datasource object တခုဆောက်ပါတယ်သူကတော့ ဥပမာအရဆို model ပေါ်မှာ။ နောက် View Object 3 ခုဆောက်ပါတယ်။ နောက် model နဲ့ View တွေကို attach လုပ်ပါတယ်။ အောက်က code သုံးပြီးတော့ပေါ့

```

dataSource.attach(view1);
dataSource.attach(view2);
dataSource.attach(view3);

```

နောက် datasource ရဲ့ change ကိုခေါ်ပါတယ်။ ဒါဆို ရင် datasourceရဲ့ change ကနေတဆင့် Subscriber တွေရဲ့ update ကိုလှမ်းခေါ်မှာပါ။ ဒါဆို Observer ရဲ့လိုရင်းဖြစ်တဲ့ Object တခု state change တာနဲ့ သူ့ကိုမှီခိုနေတဲ့ကောင်တွေကို notify လုပ်တာကိုရပါပြီ။ output ကဒီလိုထွက်လာမှာပါ။

```

Publisher Change Change1
View View1 Update to Change1
View View 2 Update to Change1
View View 3 Update to Change1

```

```

Publisher Change Chage 2
View View1 Update to Chage 2
View View 2 Update to Chage 2
View View 3 Update to Chage 2

```

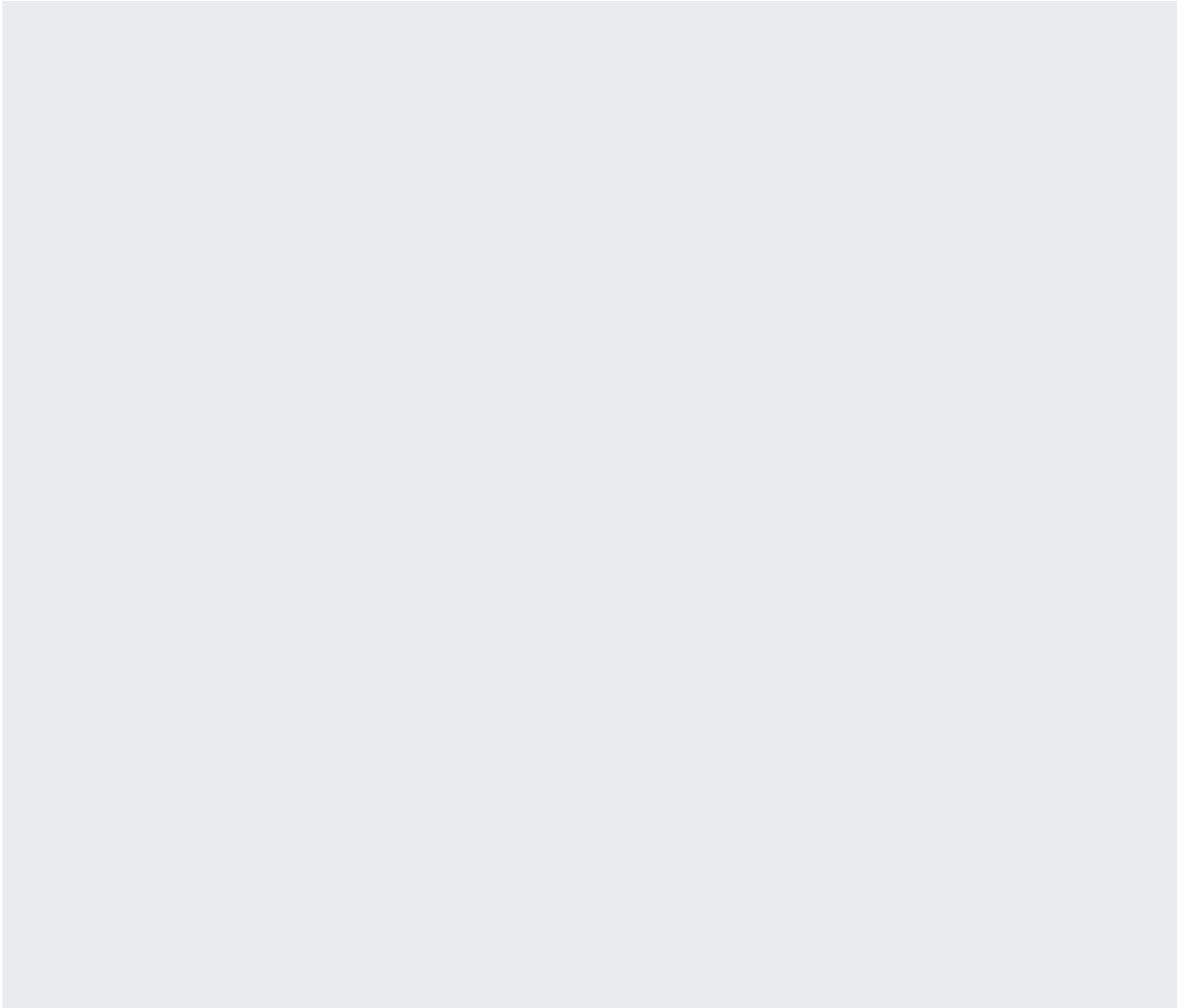
Output မှာမြင်တဲ့အတိုင်း datasource တခါ change တာနဲ့ View (Subscriber,Observer) တွေလိုက်ပြောတယ်ဆိုတာကိုတွေ့မှာပါ။

Observer pattern ရဲ့ကောင်းတဲ့အချက်တွေကတော့ Loosely Couple code ဖြစ်တယ် interface နဲ့ပဲ သွားတဲ့အတွက် Subject သည်သူ့ကိုဘယ်သူတွေလာ Subscribe လုပ်ထားလဲ ဘယ် Object တွေလဲသိစရာမလိုဘူး interface သိရင်ရပြီ ဒါက decoupled ဖြစ်စေတဲ့အကြောင်းရင်း ။ Statically attached ဖြစ်မနေဘူး။ runtime မှာ attach deattached လုပ်လို့ရတယ်။ one-to-many relationship မှာ View တွေသည် dyanmically add or remove လုပ်လို့ရတယ်။ Data Change တာနဲ့ push ကိုသုံးတဲ့အတွက် ပြောတယ်လားဆိုတာ လိုက်ကြည့်စရာမလိုဘူး ဒါကြောင့် efficient ဖြစ်တယ်။

မကောင်းတာတွေကတော့ သတိထားပြီးမသုံးရင် complexity ဖြစ်နိုင်တယ်။ နောက်တစ်ခုက deattach မလုပ်မိရင် memory leak ဖြစ်နိုင်တယ်။ ဥပမာ မရှိတော့တဲ့ View ကို deattach မလုပ်မိတာကြောင့် Subject ကနေ call back ကိုလှမ်းခေါ်နေတာမျိုး။

If you need source code, Here is the list of source code for design pattern in my github.

<https://github.com/mrthetkhine/designpattern>

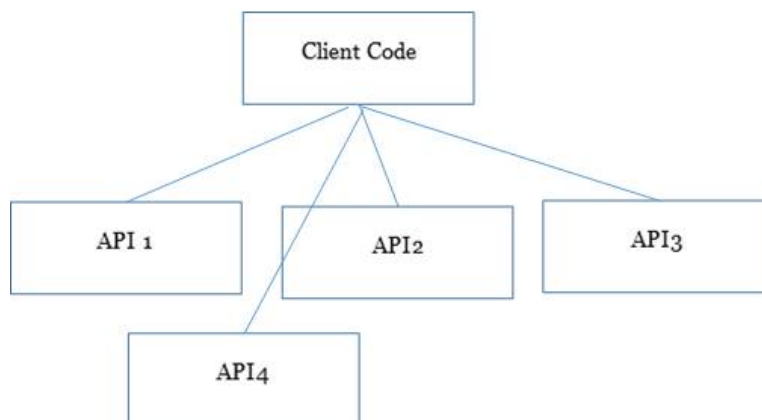


Object Oriented Design Pattern Series

Part-11 Facade Design Pattern

THET KHINE · SATURDAY, OCTOBER 15, 2016

Facade က Structural design pattern တွေထဲက တခုပါ။ ဂျွန်တော်တို့ code တွေရေးတဲ့အခါမှာ ဆိုပါစို့ module တခုသို့မဟုတ် package တခုထဲကနေ Class တွေအများကြီးဆီကနေ functionality ကိုတခါတလေ access လုပ်ဖို့လိုပါတယ်။ ဘာလို့လဲဆိုတော့ အဲဒီ functionality သည် class တခု API တခုတည်းကနေ ယူသုံးလို့ရတာမဟုတ်ပဲနဲ့ တခုထက်ပိုတဲ့ class တွေ API တွေ ကို ချိတ်ခေါ်မှသာရမဲ့ functionality မျိုးဖြစ်နေလို့ပါ။ အဲဒီကုန်ရင်ဂျွန်တော်တို့သုံးနေတဲ့ client code သည် အောက်ကပုံစံလိုဖြစ်နေမှာပါ။



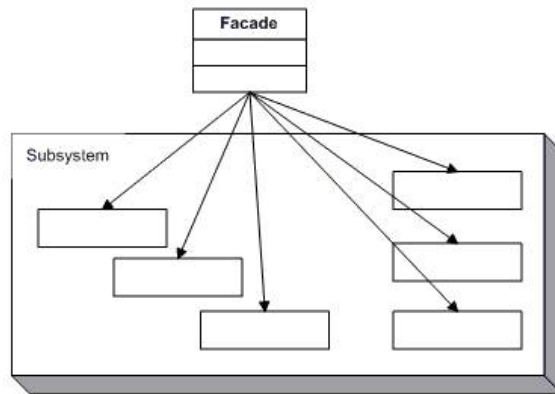
အပေါ်က code ပုံစံမျိုးမှာ ဘာပြသ နာရီလဲဆိုတော့ Client သည် API1, 2,3,4 စတဲ့ class တွေကိုတိုက်ရိုက်သုံးနေရတယ်။ ဆိုရှင်တာက functionality တခုခု ခေါ်ဖို့ဆိုရင် ခုနက class 4 ခုလုံးထဲက method တွေကို လှမ်းခေါ်နေရတယ် ဒါဆိုရင် code သည် တော်တော်ရှုပ်သွားပါတယ်။ Client ဘက်ကို လိုရှင်တဲ့ functionality လေးကို ပေးနိုင်တဲ့ class အသစ်ဆောက်မယ် အဲဒီ class ကနေခုနက API 1,2,3,4 အစရှိတာတွေကို ထပ်ဆင့်ခေါ်သွားမယ်ဆိုရင် client သည် API 1,2,3,4 ကိုတိုက်ရိုက်ထိစကမလိုတော့ပါဘူး။ Code တွေလဲရှုပ်ပွစရာမလိုတော့ပါဘူး။ ဒီသဘောတရားကို Facade design pattern လို့ခေါ်ပါတယ်။

Intent

Facade ရဲ့ intent ကို GoF မှာဒီလိုပြထားပါတယ်။

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use

သူ့ရယ်ရွယ်ချက်က ရှုပ်ထွေးနေတဲ့ Class တွေအများကြီးပါတဲ့ method တွေ functionality တွေကို client ဘက်ကနေသုံးလို့အဆင်ပြေအောင် higher level interface တခုထားပြီး ခုနက complex code တွေကို hide လုပ်လိုက်တဲ့သဘောပါပဲ။ အဲလိုအခြေအနေရောက်လာရင် Facade ကိုသုံးလို့ရပါတယ်။ Complex class, object interaction တွေကို method တခုသို့မဟုတ် API class တခုအောက်မှာပဲ hide လုပ်ပြီး abstraction ကိုပိုကောင်းအောင်လုပ်လိုက်တဲ့သဘောပါ။ သူ့ရဲ့ class Diagram ကအောက်မှာပြထား သလိုပါ။ ခုနက အပေါ်က ကောင်နဲ့ဘာကွာသွားသလဲဆိုတော့ client သည် API class တွေကို တိုက်ရိုက် မသုံးတော့ပဲ Facade class ကနေတဆင့်သုံးမှာပါ။ အဲတော့ပိုရှင်းသွားသလို decouple လဲပို့ဖြစ်သွားစေပါ တယ်။



Code ရေးရှာရအောင်။ ဆိုပါစို့. ကွန်ပတ်ာတို. Computer တခု ကို code နဲ့. example ရေးမယ်ဆိုပါစို့.ဗျာ။ Computer မှာဆို CPU, Memory, HardDisk ရှိမယ်ဆိုပါစို့။ Computer ကိုစဖွင့်ရင် Computer က boot loader ကို run ပြီး OS ကိုခေါ်တင်ရပါတယ်။ ပထမဆုံး HardDisk က boot sector ကိုဖတ်မယ် အဲကနေ boot loading code ကို memory ပေါ်တင်မယ်။ Memory ပေါ်က ကောင်တွေကိုမှ CPU က execute လုပ်မယ်ပေါ့ဗျာ။ ဒါဆို CPU, Memory,HardDisk ဆိုတဲ့ class တွေရှိမယ်။ ခုနက bootloading process သည် CPU , HardDisk, Memory အားလုံးကို access လုပ်ရမယ်။ ဒါကြောင့်သူသည် complex code, အဲတာကို Facade class တခုထားပြီးတော့ hide လိုက်မယ်။ ဘာမှအခက်ကြီးမဟုတ်ပါဘူး။ ဒီလို code ရပါ လိမ့်မယ်။

```
class CPU
{
    public void freeze()
    {
        System.out.println("CPU Freeze");
    }
    public void jump()
    {
        System.out.println("Jump to instruction");
    }
    public void execute()
    {
        System.out.println("Execute");
    }
}
```

```

    }
}

class Memory
{
    public void load()
    {
        System.out.println("Load Ram");
    }
}

class HardDisk
{
    public void readBootSector()
    {
        System.out.println("Read Bootsector");
    }
}

```

အပေါ်က CPU, Memory, HardDisk class တွေကတော့ရှင်းပါတယ် ဖတ်ကြည့်ရင်နားလည်မှာပါ။ ခု bootloading process ကို Facade class အနေနဲ့ အောက်ကလိုရေးလိုက်ပါမယ်။

```

class Facade
{
    CPU cpu = new CPU();
    Memory memory = new Memory();
    HardDisk hardDisk = new HardDisk();

    public void start()
    {
        hardDisk.readBootSector();
        memory.load();
        cpu.jump();
        cpu.execute();
    }
}

```

အပေါ်က Facade class မှာ start သည် facade method ပါ။ အဲ့ဒါမှာ bootloading process ရဲ့ complexity တွေကို user အဆင်ပြေအောင် start ဆိုတဲ့ method တခုတည်းနဲ့ hide လိုက်တာပါ။ အဲ့ဒါမှာ user က bootloading ကိုသုံးခွင့်ရင် CPU, HardDisk, Memory class တွေဆောက် ဆိုင်ရာ method တွေခေါ်အဲ့လိုလုပ်စရာမလိုပဲ။ Facade object ဆောက်ပြီး start ဆိုတာလေးကိုသုံးလိုက်ရင်ရပါပြီ။ အောက်ကလိုပါ။

```

public class FacadeDemo {

```

```

public static void main(String[] args) {
    Facade demo = new Facade();
    demo.start();
}
}

```

Facade pattern ကဘာကောင်းလဲဆိုတော့ Complex interface တွေကို simple interface ဖြစ်အောင်လုပ်နိုင်တယ်။ အဲ့တော့ maintainable ပိုဖြစ်မယ်။ နောက် Client code သည် CPU, Memory, HardDisk တို့ကို တိုက်ရိုက်ယူသုံးတာမဟုတ်ဘဲအတွက် loose coupling ဖြစ်မယ်။အဲ့တော့ easy to change,modify ဖြစ်မယ်ပေါ့ဗျာ။

ရှိနိုင်တဲ့ပြဿနာကတော့ ခုနက CPU, Memory, Harddisk အစရှိတဲ့ subsystem တွေဟာ Facade နဲ့ tightly copuled ဖြစ်နေတဲ့အတွက် သူတို့ကိုပြင်ရင် Facade ကိုလာထိနိုင်တယ်။ ဒါပေမဲ့ user ဘက်ကနေကြည့်ရင်တော့ Facade interface မပြောင်းမချင်း ဟိုဘက်က subsystem ပြောင်းလဲသိစရာမလိုဘူးပေါ့ဗျာ။

Code တွေကတော့ github မှာသွားယူပေါ့ဗျာ။

<https://github.com/mrthetkhine/designpattern>

Object Oriented Design Pattern Series

Part-12 Flyweight Design Pattern

 THET KHINE · WEDNESDAY, NOVEMBER 9, 2016

Flyweight pattern က structural design pattern တခုပါ။ သူကဘယ်နေရာမှာအသုံးဝင်လဲဆိုတော့ Object တွေဟာ create လုပ်ဖို့ memory cost ကြီးမယ် (ဆိုခွင်တာက object attribute တွေသိမ်းဖို့၊ ဖြစ်စေ Object တည်ဆောက်ဖို့အတွက် memory ကြီးတာဖြစ်စေပေါ့) ပြီးတော့ အဲဒီ Object တွေကိုလဲအများကြီးသုံးရမယ်။ ဒါဆိုရင် ခုနက Object create လုပ်ဖို့ costly ဖြစ်တဲ့ကောင်တွေကို cache သဘောမျိုးမှာထားပြီး reuse လုပ်မယ်ဆိုရင် ခန့်ခန့် Object ပြန်ဆောက်စရာမလိုဘူး။ နောက်ပြီး တခါတလေ မှာ share လုပ်ပြီးသုံးခွင့်လဲသုံးမယ်။ အဲလိုအခြေအနေတွေမှာဆို flyweight pattern ကိုသုံးပါတယ်။ မြင်သာတဲ့ realworld က Java မှာသုံးထားတဲ့ Flyweight pattern ကိုပြပါဆိုရင် String ပါပဲ။ String literal တွေ content တူရင် JVM က နောက်ထပ် String အသစ်တခုမဆောက်ပါဘူး။ ရှိပြီးသား String ကိုပဲပြန်ပေးပါတယ် (ဒါသည် costly ဖြစ်တဲ့ Object ကို share လုပ်ပြီးသုံးတာ)။ အဲတော့ ဘာကောင်းသလဲဆိုတော့ memory အစားသက်သာမယ်။ Heap fragmentation ကိုသက်သာမယ်။ ဘာလို့။ Heap Fragmentation သက်သာလဲဆိုတော့ Object တွေ ခန့်ခန့် ဆောက်တာသည် Heap memory မှာ allocation လုပ်ရတယ်။ အဲကုန်ရင် fragmentation ဖြစ်တယ် (heap memory သည် အပိုင်းပိုင်း အစိတ်စိတ်လေးတွေဖြစ်သွားတယ်။ Large collection ပြင်ညီမရတော့ဘူး အဲတော့ Heap ကိုပြန်ပြီး အစိတ်စိတ်ကလေးတွေကနေ တခုတည်းကြီးကြီးပြန်ရအောင် Garbage Collection Algorithm တွေသုံးပြီး compact ဖြစ်အောင်ပြန်လုပ်ရတယ်)။ Java မှာရှိတဲ့ String သည် flyweight pattern ကိုသုံးထားတယ်။ သူတင်မကသေးဘူး Wrapper class တွေဖြစ်တဲ့ Integer, Float စတာတွေမှာလဲ Flyweight ကိုသုံးထားတယ်။ ဒါဆိုရင် flyweight ရဲ့သဘောသည် costly ဖြစ်မဲ့ Object တွေ သူတို့တွေမှာ content တူမယ် ဒါဆိုရင် ခန့်ခန့် မဆောက်ပဲနဲ့ ဆောက်ပြီးသား ရှိပြီးသား Object ကို reuse လုပ်တာ ဆိုတာကို သဘောပေါက်မယ်ထင်ပါတယ်။

နောက်ထပ်ဥပမာတခုပေါ့၊ ပြွန်တောတို၊ Plant and Zombi ဆော့ဖူးမှာပါ အဲထဲမှာ zombi တွေပါတယ် plant တွေပါတယ်။ Plant တွေလဲသေတယ်။ zombi တွေလဲသေတယ်။ ဒါဆိုရင် သေသွားတဲ့ Plant Object တွေ zombi Object တွေကို GC ဖြစ်အောင် မထားပဲနဲ့ cache ထဲမှာသိမ်းထားမယ် နောက်မှ ပြန်ပြီး reuse လုပ်မယ်ဆိုရင် performance အားဖြင့်သော်လည်းကောင်း speed အားဖြင့်သော်လည်းကောင်း မြန်လာနိုင်ပါတယ်။ နောက် example တခုကတော့ အဲဒီ Game ထဲမှာပဲ တချို့ Object တွေရဲ့ state သည် share လုပ်လို့ရတယ် ဥပမာ နေကြာပန်းရဲ့ geomerty location သည် နေကြာပန်းအားလုံးမှာတူတူပါပဲ။ ဒါဆိုရင် ကွာတာက ဘယ်နေရာဆိုတာလေးပဲကွာတာ အဲတာဆိုရင် Flywieght သုံးပြီး မတူတဲ့နေရာလေးပဲ ထပ်သုံးမယ်ဆိုရင် large object collection တွေဆောက်စရာမလိုတော့ပါဘူး။

Intention

GoF မှာရေးထားတဲ့ သူ့ intention ကဒီလိုပါ။

Use sharing to support large numbers of fine-grained objects efficiently

ဆိုခွင်တာကတော့ Object တွေအများကြီးမဆောက်ပဲ တူတဲ့ Object တွေဆို sharing လုပ်ပြီးသုံးမယ် ဥပမာ String ဆိုရင် "Hello World" ဆိုတဲ့ String သည် ဘယ်နေရာမှာပါပါ အတူတူပဲ ဒါကို Object အသ

စ်ဆောက်မနေတော့ပဲ Object တခုတည်းဆောက်ပြီးမှသုံးမယ်ဆိုရင် application သည် performance ကို ပိုကောင်းလာပါလိမ့်မယ်။

Flyweight ကို အဓိပ္ပာယ်ဖွင့်ရရင် shared object လို့ဆိုရမှာပါ။သူ့မှာ state 2 မျိုးရှိနိုင်ပါတယ်။ intrinsic နဲ့ extrinsic ဆိုပြီးတော့ပါ။ Intrinsic ဆိုတာကတော့ Flyweight ထဲမှာပဲသိမ်းတဲ့ state မျိုးပါ ။Share လုပ် လို့ရပါတယ်။ ဥပမာ ခုနက zombie ထဲက နေကြာပန်းရဲ့ ပုံစံကာလာဆိုတာမျိုးက ဝိမ်းထဲမှာရှိတဲ့နေကြာပ န်း Object တွေအကုန်မှာအတူတူပါပဲ။ ဘာလေးပဲကွာသွားမလဲဆိုရင် နေကြာပန်းက ဘယ်နေရာမှာရှိသလဲ ဆိုတော့ physical location on grid လေးပဲကွာမှာပါ(တခြား extrinsic stateတွေရှိနိုင်ပါသေးတယ်)ဒါမျိုး ကိုတော့extrinsic ဆိုပြီးခေါ်ပါတယ် ။ သူကတော့ share လုပ်လို့မရပါဘူး။

Flyweight ကိုဘယ်နေရာမှာသုံးလို့ရသလဲ။

အောက်က အခြေအနေတွေအားလုံး ဖြစ်နေပြီဆိုရင် Flyweight ကိုသုံးလို့ရပါတယ်။

Application မှာ object တွေအများကြီးသုံးနေရတယ် (တူညီတဲ့ Object အမျိုးအစားတွေကိုဆိုလိုတာပါ ဒီ နေရာမှာ မတူညီတဲ့ object type တွေဆို များလဲ flyweight နဲ့သုံးလို့အဆင်မပြေပါဘူး တူတယ်ဆိုတာ type တူတယ်ကိုပြောတာပါ)

Object တွေများလာတာနဲ့အမျှ memory မှာ storage costတွေပိုများလာမယ်(ဥပမာ android device လို phone လို limit နဲ့လုပ်နေရတာမျိုးဆိုပိုသိသာမယ်)

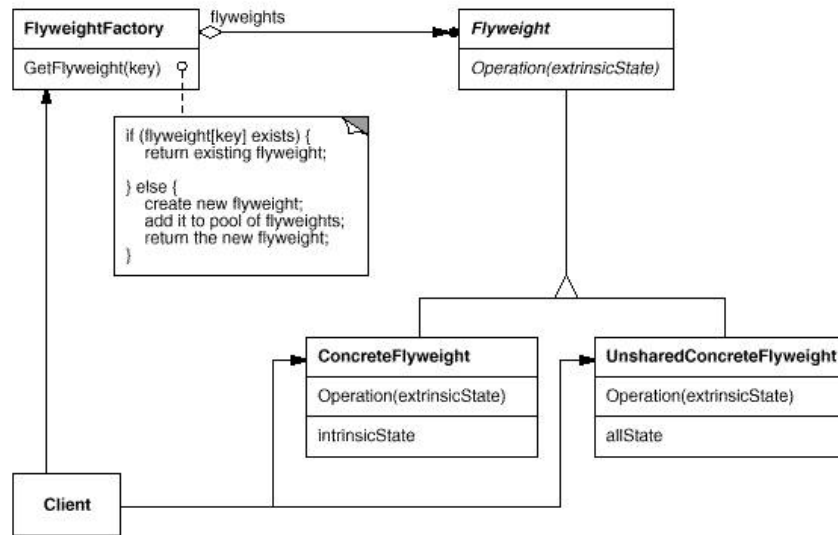
Object ရဲ့ state အများစုကို အပြင်ကိုခွဲထုတ်လို့ရမယ်။(extrinsic လုပ်လို့ရမယ် တူတာတွေကို share လုပ်ပြီး မတူတာလေးတွေကိုပဲ အသစ်ယူပြီးသုံးမယ်)

Extrinsic state ကိုဖယ်လိုက်လို့ Object အများစုကို Object အနည်းငယ်နဲ့ အစားထိုးလို့ရမယ်ဆိုရင်။

Application သည် Object identity အပေါ်မှာ မမူတည်ဘူး ဆိုခွင့်တာက Object နှစ်ခုသည် တူလားမတူ လား စစ်ရတဲ့ operation မျိုးမသုံးဘူး (ဒီနေရာမှာ ဒီအခွက်ကြောင့် Java String တွေကို == (object equality operator) နဲ့မစစ်ရပဲ equals method နဲ့စစ်ရတာ) ဆိုရင်

အပေါ်ကအခွက်အားလုံးဖြစ်နေရင် Flyweight ကိုသုံးလို့ရပါပြီ။

Flyweight ရဲ့ class diagram ကတော့အောက်မှာပြထားသလိုပါ။



အပေါ်က class diagram မှာအဓိက ဂျာတာသည် FlyweightFactory ပါ။ သူသည် Factory pattern ကို သုံးပါတယ်။ Object တွေကို cached လုပ်ပြီးသိမ်းထားတယ် နောက်အဲ့လို object မျိုး request လုပ်ရင် အသစ်မဆောက်တော့ပဲနဲ့. cache ကနေပြန်ပေးလိုက်တယ်။ ဒါသည် Flyweight ရဲ့အသက်ပါ။ ဂျာန်တဲ့ class တွေကတော့ မလိုက်နာလဲ သဘောပါ။ design pattern ဆိုတာ concept သဘောသာဖြစ်တဲ့အတွက် တချို့ pattern တွေမှာ တိုက်ရိုက်ကူးရမယ်ဆိုတဲ့ သဘောမရှိပါဘူး။

Example code ရေးကြရအောင်။

ဂျာန်တောတို့က Java Platform ရယ် .NET platform ရယ်အပေါ်မှာ code တွေ run ကြမယ်ဆိုပါစို့။ ခုနက Java Platform နော်လည်းကောင်း .NET platform နော်လည်းကောင်းကို code run ဖို့လုပ်တိုင်း မှာ Object အသစ်တွေပြန်မဆောက်ပဲ Factory ထဲကနေ cache ကနေပြန်ယူသုံးမယ်ပေါ့။ (အမှန်ကဒီနေရာမှာ platform 2 ခုသည် large collection of object ဖြစ်ရမှာ ဒါပေမဲ့ example ကရှုပ်သွားမှာစိုးလို့ အဲ့ဒါမျိုးမပေးတော့တာ။)

အောက်ကတော့ Code class ပေါ့

```

public class Code {
    String code;
    public Code(String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }
    public void setCode(String code) {
        this.code = code;
    }
}
    
```

အပေါ်က code ကဘာမှထူးထူးခြားခြားမပါပါဘူး bean သက်သက်ပါပဲ။ Platform interface ကိုအောက် လိုဆောက်မယ်။

```
public interface Platform {  
    public void execute(Code code);  
}
```

Platform ဂျာ run မှာဆိုတော့ ဘုံထုတ်ပြီး interface ဆောက်လိုက်တာပါ။ Interface ဆောက်လိုက်တော့ နောက်ထပ် Platform ထပ်ထဲခွင်ရင်လဲပိုလွယ်တာပေါ့။
နောက် Platform တခုခွင်းဆီအတွက်ရေးမယ်။ Java နဲ့ .NET တခုစီပေါ့။

```
public class JavaPlatform implements Platform{  
    public JavaPlatform() {  
        System.out.println("Create Java Platform");  
    }  
    @Override  
    public void execute(Code code) {  
        System.out.println("Execute " + code.getCode() + " On Java");  
    }  
}
```

ဘာမှထူးခြားပါဘူး interface ထဲက method ကို implement လုပ်ထားရုံပါပဲ။ နောက် .NET အတွက် ကလဲဒီလိုပါပဲ။

```
public class DotNetPlatform implements Platform{  
    public DotNetPlatform() {  
        System.out.println("Create .net platform");  
    }  
    @Override  
    public void execute(Code code) {  
        System.out.println("Execute Code " + code.getCode() + " on DotNet ");  
    }  
}
```

သူလဲ JavaPlatform လိုပါပဲ။။ တကယ့်အသက်ကအောက်က Platform Factory ပါသူသည် Flyweight ရဲ့အသက် ဖြစ်တဲ့ sharing ကို implement လုပ်သွားတာပါ။

```
import java.util.HashMap;  
import java.util.Map;
```

```
public class PlatformFactory {  
  
    private static Map<String,Platform> map = new HashMap<>();  
    public static Platform getInstance(String platformType)  
{
```

```

Platform p = map.get(platformType);
if(p == null)
{
    switch(platformType)
    {

        case ".NET":
            p = new DotNetPlatform();
            break;

        case "JAVA":
            p = new JavaPlatform();
            break;

    }
    map.put(platformType, p);

}

return p;

}
}

```

အပေါ်က factory မှာ sharing လုပ်ဖို့. HashMap ကိုသုံးပါတယ်။ Platform တခု request လုပ်ပြီဆိုရင် map ထဲမှာရှိလား အရင်ထုတ်ပါတယ် ရှိရင်ပြန်ပေးပါတယ် ဒီနည်းနဲ့. sharing ကိုလုပ်သွားတာပါ။ မရှိရင်တော့ Object ဆောက်ပြီး map ထဲကိုထည့်သိမ်းပါတယ်။ သူ့ကိုသုံးမဲ့ client code ကတော့ဒီလိုပါ။

```

public class FlywieghtDemo {

    public static void main(String[] args) {
        Code javaCode = new Code("Java program");
        Code dotNetCode = new Code("C#Program");

        Platform p = PlatformFactory.getInstance(".NET");
        p.execute(dotNetCode);

        Platform java = PlatformFactory.getInstance("JAVA");
        java.execute(javaCode);

        java = PlatformFactory.getInstance("JAVA");
        java.execute(javaCode);
    }
}

```



```

    p = PlatformFactory.getInstance(".NET");
    p.execute(dotNetCode);

}

}

```

အပေါ်က client code မှာ Platform တစ်ခုကို create လုပ်မယ်ဆိုရင် PlatformFactory ရဲ့ getInstance ကိုသုံးပါတယ်။ getInstance ကနေ object ဆောက်ပြီးသားဆိုရင် အသစ်ထပ်မဆောက်တော့ပဲရှိပြီးသား Platform ကိုပဲပြန်ပေးမှာပါ။ ဒါဆိုရင် platform တွေ ဥပမာ Java Platform ကိုဘယ်နှခါ request လုပ်လုပ် ပဲ Object အများကြီးသုံးစရာမလိုပဲနဲ့ တခုတည်းနဲ့ သုံးလိုရပါပြီ အဲဒီအတွက် application သည် performance အားဖြင့်လည်းကောင်း speed အားဖြင့်လည်းကောင်းပိုမြန်လာမှာဖြစ်ပါတယ်။

Code example တွေကိုဒီမှာလဲသွားယူလိုရပါတယ်။

<https://github.com/mrthetkhine/designpattern>

Object Oriented Design Pattern Series

Part-13 Proxy Design Pattern



THET KHINE · WEDNESDAY, NOVEMBER 23, 2016

Proxy design pattern က Java မှာရော တခြား language တွေမှာ အတော်အသုံးများတဲ့ pattern လို့ပြောရမှာပါ။ Proxy ရဲ့သဘောက Object တစ်ခုကို proxy နောက်မှာထားပြီး client သည် real object ကိုတိုက်ရိုက်မထိပဲ Proxy ကနေ ယူသုံးရတာမျိုးပါ။ ဘာကောင်းလဲဆိုတော့ Proxy သည် real object အတွက် access control လုပ်တာမျိုး နောက်လိုအပ်တဲ့ functionality တွေထပ်ထဲပေးတာမျိုးလုပ်ဖို့ရာအတွက်အသုံးဝင်ပါတယ်။ Spring လို့ framework တွေမှာ AOP (Aspect Oriented Programming) လို့ functionality မျိုးရဖို့အတွက် JDK dynamic proxy တွေကိုသုံးရပါတယ်။ Interface နဲ့ရေးထားတဲ့ component တွေဆိုရင်တော့ JDK Dynamic proxy နဲ့ လိုအပ်တဲ့ code ကို inject လုပ်ပါတယ်။ မဟုတ်ဘူးဆိုရင်တော့ byte code generation library တွေဖြစ်တဲ့ cglib လိုကောင်တွေသုံးပြီး runtime မှာ component တွေကို modified လုပ်ပါတယ်။ ဒါမှသာ runtime ရောက်တဲ့အခါကရင် controller တွေ component တွေမှာ AOP လိုကောင်တွေသုံးလို့ရမှာဖြစ်ပါတယ်။ ဥပမာ controller method တွေ မခေါ်ခင်နဲ့ ခေါ်ပြီးရင် ဘယ် method ကိုထုတ် run ပေးပါဆိုပြီးလုပ်လို့ရပါတယ်။ User ကတော့ရိုးရိုး POJO လောက်ရေးထားပေမဲ့ runtime မှာ POJO component တွေကို (Interface နဲ့ dependency inject လုပ်ရင် JDK Dynamic proxy သုံးတယ် မဟုတ်ရင် class base component တွေဆိုရင် cglib လိုကောင်မျိုးကို သုံးပြီး runtime မှာ bytecode ကို ပြင်တယ်။)ဒါကြောင့် Spring runtime ဒါမှမဟုတ် Hibernate runtime မှာရတဲ့ object တွေသည် ကိုရေးထားတဲ့ Object တွေမဟုတ်ပဲ Framework က inject လုပ်ထားတဲ့ Proxy တွေဖြစ်နေတတ်တယ် မှားသောအားဖြင့်ပေါ့။)

နောက် JavaScript မှာဆိုရင် Proxy တွေကို MVC framework တွေ မှာသုံးလေ့ရှိပါတယ်။ ဥပမာ လူနဲ့တောတို့က ရိုးရိုး JS object တစ်ခုကိုဆောက်လိုက်မယ်။ ဒဲ့ Model object ထဲကနေ Property တစ်ခုကို write လုပ်တာနဲ့ View ကိုပြောင်းပြန်ရမယ်။ ဒါဆို framework တွေရဲ့အထဲမှာဘယ်လို implement လုပ်ထားလဲဆိုရင် ခုနက User ရဲ့ model object ကို proxy နဲ့ wrap လုပ်လိုက်တယ်။ User က object.property = something ; ဒဲ့လိုတဲ့လိုက်တာနဲ့ proxy method ကို လှမ်း run မယ်။ Proxy ကနေ View ကို update လုပ်ဖို့ခိုင်းမယ်။ ဒီလိုနေရာတွေမှာလဲ Proxy ကိုသုံးကြတယ်။

အဓိက ကတော့ Proxy ကိုသုံးရခြင်းသည် မူရင်း Object ကို access သို့မဟုတ် functionality အရ ထိန်းချုပ်ဖို့ လုပ်ဆောင်တာပဲဖြစ်ပါတယ်။ GOF reference အရတော့ Proxy တွေကိုဒီလိုခွဲခြားလို့ရပါတယ်။

Remote Proxy

Remote location မှာရှိတဲ့ Object တွေကိုထိန်းဖို့သုံးတဲ့ Proxy မျိုးပါ။ ဥပမာ Java RMI မှာသုံးတဲ့ RMI stub, skelton code တွေသည် Remote proxy တွေပါ သူတို့ကဘာလုပ်ပေးလဲဆိုရင် တခြား စက်ရဲ့ JVM မှာရှိတဲ့ Object ရဲ့ method တွေကို invoke လုပ်ဖို့ လိုအပ်တဲ့ parameter တွေကို marshalling လုပ်ပေးတာမျိုး (searialization နဲ့သဘောတရားတူတူပဲ) နောက် network communication တွေလုပ်ပေးတာမျိုးကိုလုပ်ပေးတယ်။ အဓိက ကတော့ Remote proxy သည် remote location မှာရှိတဲ့ object ကို ထိန်းချုပ်တာမျိုးလုပ်တာပါ။

Virtual Proxy

Virtual Proxy ဆိုတာကတော့ real object သည် ရှိခွင်မှရှိနေမယ် ဘာလို့လဲဆိုတော့ သူ့ကို create လုပ်

ဖို့. memory အရ CPU အရ costly ဖြစ်နေမယ်ဆိုရင် လိုအပ်မှပဲ create လုပ်မယ် ဒါမျိုးကို virtual proxy လို့.ခေါ်ပါတယ်။

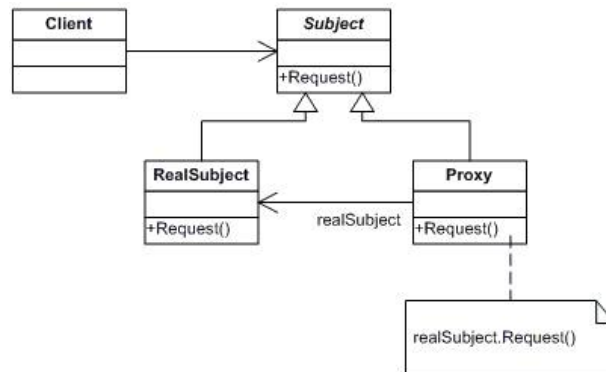
Protection Proxy

သူကတော့ Object တစ်ခုရဲ့. access control ကိုထိန်းတဲ့ကောင်မျိုးပေါ့။ ဥပမာ OS မှာ terminal မှာ command တွေ run တဲ့ အခါ user အပေါ်မူတည်ပြီး ဘယ်လို command မျိုးပေးသုံးမလဲပေါ့။ Access right ကိုထိန်းတဲ့ proxy မျိုးပေါ့။ ဥပမာ code ကို protection proxy နဲ့.ပြမှာပါ။

Smart Reference

ဒီကောင်ကတော့ အပေါ်က ဥပမာတွေမှာပြသွားတဲ့ Spring တို့. Hibernate တို့.မှာသုံးတဲ့ proxy တွေ လို့.ဆိုရမှာပါ။ သူတို့.က မူရင်း Object ရဲ့. method တစ်ခုခု ဒါမှမဟုတ် properties တစ်ခုခုကို access လုပ်ပြီ invoke လုပ်ပြီဆိုရင် မူရင်း Object ရဲ့. code အပြင် Proxy ကနေ ထပ်ပေါင်းထားတဲ့ code ပါ invoke လုပ်ပေးတဲ့ကောင်မျိုးပါ။ ဥပမာ Model object ရဲ့. proxy ဆိုရင် model object ရဲ့. state ပြောင်းသွားတာနဲ့. view ကို change လုပ်ဖို့. code ပါ invoke လုပ်မှာပါ။ ဒါမျိုးကို smart reference proxy လို့.ခေါ်ပါတယ်။

အောက်ကတော့ Proxy ရဲ့. class diagram ပုံပါ။



Subject သည် ကို model လုပ်မဲ့ Object နဲ့. Proxy ကို common interface အနေနဲ့.ထားဖို့.အတွက်ပါ။ RealSubject ကတော့ တကယ့် Proxy နောက်မှာထားမဲ့ Object ပေါ့။ Proxy ကတော့ RealSubject ကို control လုပ်မဲ့ကောင်ပေါ့။

Proxy ကိုဘယ်နေရာမှာသုံးသင့်သလဲဆိုရင်တော့ Object တစ်ခုခုကို runtime မှာ ထိန်းချုပ်တော့မယ် access အရသော်လည်းကောင်း creation အရသော်လည်းကောင်း functionality အရထပ်ဖြည့်တော့မယ် ဆိုရင် Proxy ကိုသုံးလို့.ရပါတယ်။

Example code အနေနဲ့. Protection proxy တစ်ခုရေးရအောင်။ အဓိက ကတော့ OS ရဲ့. execution command တွေကို protect လုပ်မှာပါ။ user နဲ့. admin ပေါ့။ command တွေကို proxy ကနေတဆင့် run ခိုင်းမယ်။ တကယ်လို့. user account နဲ့. admin command တွေ run ခိုင်းမယ်ဆို ပေးမ run ဘူးပေါ့။

RealSubject ရော Proxy ပါလိုက်နာရမဲ့ Subject လို interface တစ်ခုအောက်ကလိုဆောက်လိုက်ပါမယ်။

```
public interface CommandExecutor {
    public void runCommand(String command)throws Exception;
}
```

Object Oriented Design Pattern Series Part-14 Chain of Responsibilities Design Pattern

Structural design pattern တွေကပြီးသွားပြီ။ ဒီ Chain of Responsibilities Design pattern က behavioural design pattern တွေထဲက တခုလို့ ပြောရမယ်။ ဘာလို့။ behavioural design pattern လို့ ခေါ်ရသလဲဆိုရင် Object တွေရဲ့ runtime မှာ interaction နဲ့ ပတ်သက်ပြီး သက်ဆိုင်တဲ့ design pattern တွေကို behavioural design pattern လို့ ဆိုကြတယ်။

COR (Chain of responsibility) pattern က ဘယ်လိုကောင်မှူးလဲဆိုရင် command request တခုရှိမယ်။ နောက် ဒဲ့ဒီ command or request ကို process လုပ်ဖို့ action handler object တွေသည် chain ပုံစံနဲ့ ရှိမယ်။ ပထမဆုံး Command request သည် chain ထဲကထိပ်ဆုံးက Action Handler ကို run မယ်။ Action Handler က သူ handle လုပ်နိုင်တယ်ဆိုရင် သူနဲ့ ပဲပြီးမယ်။ နောက်ကိုမပို့။ တော့ဘူး။ ဒါမှမဟုတ်သူလဲ handle မလုပ်နိုင်ဘူး နောက်တခုက တခြား handler တွေဆက်ပြီး process လုပ်စေခွင့်သေးတယ်ဆိုရင် chain ထဲမှာရှိတဲ့နောက် handler တခုကိုပို့မယ်။ ဒီလိုနည်းနဲ့ Object interaction လုပ်ပုံကို COR pattern လို့ ခေါ်တယ်။ ဥပမာ Linux မှာဆို command တွေဟာ chain လုပ်လို့ ရတယ်။ vertical bar or pipe လေးသုံးပြီး command တခု run ပြီး ရင် နောက် command ကို run မယ်။ နောက် command သည်ရှေ့က command ရဲ့ output ကိုသုံးမယ်။ ဒါသည်လဲ COR pattern ပဲ။ နောက် Java မှာရှိတဲ့ logger , logger တွေသည် hierarchical ထားလို့ ရတယ်။ အဆင့်ဆင့်သွားလို့ ရတယ်။ နောက် Servlet မှာရှိတဲ့ filter တွေ ဥပမာ Security တွေကိုထိန်းတဲ့အခါ filter တွေနဲ့ ထိန်းတယ်။ တကယ့် operation ကိုမလုပ်ခင်မှာ security filter တွေ က ထဲ run တယ်။ ဒါမျိုးသည် COR ရဲ့ သဘာဝ အဆင့်ဆင့် process လုပ်တယ်ပေါ့။ နောက် Java, JavaScript မှာရှိတဲ့ object တခုရဲ့ method တွေခေါ်ပုံပေါ့။ Object တခုရဲ့ method ကိုခေါ်မယ်ဆိုရင် ဒဲ့ဒီ Object မှာ method ရှိရင် ဒဲ့ method ကိုသုံးတယ်။ မရှိဘူးဆိုရင် parent ကိုဆက်ရှာတယ်။ parent မှာမရှိရင် parent ရဲ့ parent ကိုဆက်ရှာတယ်။ ဒါသည် သဘာဝအားဖြင့် COR pattern လို့ ဆိုရမယ်။

ခုနက လို အဆင့်ဆင့် လုပ်သွားရမဲ့ လက်လွှဲသွားရမဲ့ object interaction လိုတဲ့အခါ COR ကိုသုံးလို့ ရတယ်။

Intent

သူ intent ကိုတော့ GoF မှာဒီလိုပြထားတယ်။

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command object ရဲ့ sender နဲ့ receiver ကို decouple လုပ်ထားတာ။ ဆိုခွင်တာက ဒီ request ကိုဘယ်သူက handle လုပ်ရမယ် ဆိုပြီးသတ်မှတ်ထားတာမဟုတ်ဘူး။ Request နဲ့ handler သည် tightly coupled ဖြစ်နေဘူး။ Handler chain ထဲက အဆင်ပြေတဲ့ handler ကနေ ကောက်ပြီးတော့ handle လုပ်မယ်။ သူလုပ်လို့ အဆင်မပြေရင် သူ့နောက်က Handler object ကိုပို့မယ်။ ဥပမာ Browser မှာ event တခုတက်ပြီဆိုပါစို့။ click event ပေါ့။ ဒါဆိုရင် လက်ရှိ target element မှာ client event bind ထားလားကြည့်တယ်။ ဆိုကြပါစို့။ body ထဲက div ထဲက span လေးကို နှိပ်လိုက်တယ်။ span မှာ click event bind လိုက်ရင် span ရဲ့ click event ကို execute လုပ်တယ်။ ဒါနဲ့ ရပ်သလားဆိုတော့မရဘူး။ သူ့အပေါ် div ကိုလှမ်းပို့တယ်။ div မှာ click event ရှိရင် run မယ်။ မရှိဘူးဆိုရင် body ဆီကိုပို့တယ်။ ဒါကိုတော့ browser တွေမှာ event bubbling လို့ ဆိုကြတယ်။ သဘောတရားကတော့ COR နဲ့ ခပ်ဆင်ဆင် တူတယ်။

ဒဲ့တော့ ဘယ်လိုအခြေအနေတွေမှာ COR ကိုသုံးရမလဲဆိုရင် အောက်ကအခြေအနေတွေမှာသုံးလို့ ရတယ်။

Request ကို handle လုပ်ဖို့ Object တခုထက်ပိုရှိနိုင်တယ်။ ဘယ်သူက handle လုပ်ရမယ်ဆိုတာလဲကြိုမသိနိုင်ဘူး။ Handler ကိုလဲ သူ့ဟာသူရွေးပေးရမယ်ဆိုပါစို့။

Request တခုကို ဘယ်သူဘယ်ဝါ handle လုပ်ပါမပြောပဲ Handler အမှူးကြီးကို ပို့ ခွင့်ရင်

Handler Object တွေသည် dynamically သက်မှတ်ရမယ်ဆိုရင် (ဆိုခွင်တာ runtime ရောက်မှသိနိုင်မယ်။ နောက်တခုက အရင်က သူက handle လုပ်ပေမဲ့နောက်သူဖြစ်နိုင်မှဖြစ်မယ်။)

ဒဲ့လိုတွေဆိုရင် COR ကိုသုံးလို့ ရပါတယ်။

အပေါ်က တော့ COR ရဲ့ class diagram ပုံပါ။ သူ့မှာ handler interface တခုရှိမယ်။ handleRequest ရှိမယ်။ နောက် handler တွေကို အဆင့်ဆင့်နှိတ်ထားမယ်။

Example တခုအနေနဲ့ sale နဲ့ ပတ်သက်တာတခုရေးကြည့်ရအောင်။ ဥပမာ ပိုက်ဆံနည်းနည်းဆို Manager က handle လုပ်လို့ ရမယ်။ Manager handle လုပ်လို့ မရဘဲ amount ဆိုရင် Director ဆီလွှဲမယ်။ Director ဆီမရရင် Vice President ဆီလွှဲမယ်ပေါ့။

Command class ကိုတော့ဒီလိုရေးပါတယ်။

```
public class Command {
    int amount;
    public Command(int amount) {
        this.amount = amount;
    }
}
```

အောက်ကကောင်ကတော့ Handler class မျိုးပေါ့။ PurchasePower လို့ နာမည်ပေးထားမယ်။

```

abstract class PurchasePower {
static final int BASE = 10;
PurchasePower successor;
abstract void handleRequest(Command command);
public PurchasePower getSuccessor() {
return successor;
}
public void setSuccessor(PurchasePower successor) {
this.successor = successor;
}
}

```

Handler တွေကသူတို့ handle လုပ်ဖို့ handleRequest လိုပါတယ် နောက်သူတို့ နောက်က ကောင်ကိုမှတ်ဖို့ PurchasePower successor ဆိုပြီး field တခုနဲ့ သိမ်းထားပါတယ်။ နောက် method ၂ခုကတော့ getter setter တွေပါ။ PurchasePower တိုင်းမှာ သူတို့ handle လုပ်နိုင်တဲ့ amount ကိုပေးထားပါတယ်။ ဒါဆို Manger ကို ဆောက်ရအောင်။

```

public class ManagerPower extends PurchasePower{
static final int ALLOW = BASE * 10;
@Override
void handleRequest(Command command) {
if(command.amount <= ALLOW )
{
System.out.println("Sale handled by Manager");
}
else
{
if(this.getSuccessor()!=null)
{
successor.handleRequest(command);
}
}
}
}

```

Manger သည် ဝင်လာတဲ့ amount သည် သူ handle လုပ်နိုင်တဲ့ပမာဏဆိုရင် သူ handle လုပ်တယ်။ မဟုတ်ဘူးဆိုရင် သူ နောက်မှာ ရှိသေးတယ်ဆိုရင် နောက်ကို request ကိုလွှဲပေးမယ် ဒါသည် COR ရဲ့အသက်ပေါ့။ နောက်တဆင့်က Director သူက manager မရရင် သူနဲ့ handle လုပ်မယ်။

```

public class DirectorPower extends PurchasePower{
static final int ALLOW = BASE * 20;
@Override
void handleRequest(Command command) {
if(command.amount <= ALLOW )
{
System.out.println("Sale handled by Director");
}
else
{
if(this.getSuccessor()!=null)
{
successor.handleRequest(command);
}
}
}
}

```

Manager နဲ့ director ကွာသွားတာက Allow ပမာဏပဲလျန်တာက သဘောတရားအတူတူပဲ နောက်ထက် VicePredient လဲဒီလိုပဲ။

ရးတယ်ပေါ့မှာ။ ဒါဆိုရင် client ကသုံးရင် ဒီလိုရေးရမှာပါ။

```
PurchasePower manager = new ManagerPower();
```

```
PurchasePower director = new DirectorPower();
```

```
PurchasePower vice = new VicePresident();
```

```
manager.setSuccessor(director);
```

```
director.setSuccessor(vice);
```

```
Command command = new Command(10);
```

```
manager.handleRequest(command);
```

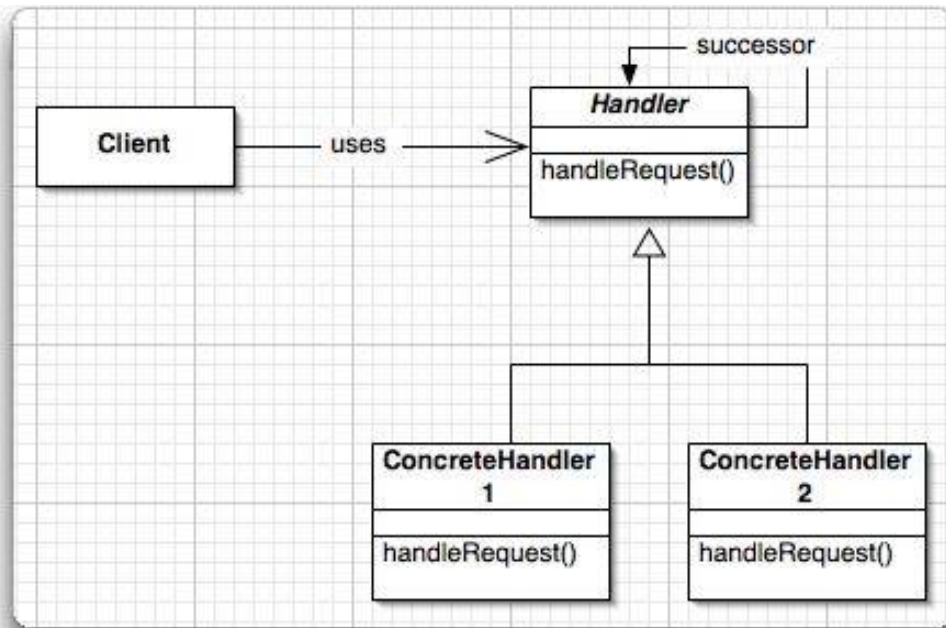
```
manager.handleRequest(new Command(1020));
```

```
manager.handleRequest(new Command(200));
```

```
manager.handleRequest(new Command(600));
```

Object တွေရှိတာသွားပုံကိုကြည့်ပါ manager ရဲ့ successor ထဲကို director ထဲတယ်နောက် director ရဲ့ successor အနေနဲ့ vice president ကိုထဲတယ်။ နောက် request ကိုပိုတာကတော့ manager ဆီအရင်ပိုတယ် Manager က handle လုပ်လို့ရတဲ့ amount ဆိုရင် သူလုပ်မယ် မဟုတ်ရင် သူ့နောက် က director , vice president အဲလိုအဆင့်ဆင့်လွှဲသွားမယ်။ ဒါသည် COR ပါပဲ။ source code ကတော့ ဒီမှာပါ။

<https://github.com/mrthetkhi.../designpattern/.../master/src/cor>



သူက command တစ်ခုကို execute လုပ်ဖို့အတွက်ပါ။ RealObject ရော Proxy ရော သူ့ကို implement လုပ်ရမှာပါ။ Access right မရှိပဲ command တွေ run ရင် exception throw လုပ်မှာပါ။ RealSubject ကို implement လုပ်ရအောင်ပါ။

```
public class CommandExecutorImpl implements CommandExecutor{
    @Override
    public void runCommand(String cmd) throws Exception {
        Runtime.getRuntime().exec(cmd);
        System.out.println(""" + cmd + "" command executed.");
    }
}
```

RealSubject သည် ပေးလာတဲ့ command ကို Runtime.getRuntime().exec သုံးပြီး run ပါတယ်။ C မှာ exec , VB.NET မှာ Shell, PHP မှာ exec, system အစရှိတာတွေနဲ့တူပါတယ်။ ဒီနေရာမှာ တစ်ခုသတိထားရမှာက RealSubject ဖြစ်တဲ့ CommandExecutorImpl သည်သူ့ဆီရောက်လာတဲ့ command ကို ဘာမှမစစ်ပဲ execute လုပ်တယ်ဆိုတာပါ။ စစ်တဲ့တာဝန်ကဘယ်သူကယူရမှာလဲဆိုတော့ Proxy ကပါ။ ဒါကြောင့် Proxy သည် မူလ Object ရဲ့ access right ကိုထိန်းဖို့၊ သော်လည်းကောင်း functionality ကို add လုပ်ဖို့၊ သော်လည်းကောင်း သုံးတယ်ဆိုတာ သဘောပေါက်ရမှာပါ။ Proxy code ကအောက်မှာပါ။ သူ့မှာတော့ user, admin ဂုဏ်သတ္တိပြီး access right ကိုပါထိန်းပါတယ်။

```
public class CommandExecutorProxy implements CommandExecutor{

    private boolean isAdmin;
    private CommandExecutor executor;

    public CommandExecutorProxy(String user, String pwd){
        if("admin".equals(user) && "admin".equals(pwd))
        {
            isAdmin=true;
        }
        executor = new CommandExecutorImpl();
    }

    @Override
    public void runCommand(String cmd) throws Exception {
        if(isAdmin)
        {
            executor.runCommand(cmd);
        }else
        {
            if(cmd.trim().startsWith("rm")){
                throw new Exception("rm command is not allowed for non-admin users.");
            }else{
                executor.runCommand(cmd);
            }
        }
    }
}
```

```

    }
}
}
}

```

အပေါ်က proxy မှာ constructor မှာ admin နဲ့ user ကိုခွဲပြီး admin ဆိုရင် isAdmin ထဲကို true ထဲပါတယ်။ နောက် RealSubject CommandExecutorImpl ကိုဆောင်ထားပါတယ်။ နောက် method ဖြစ်တဲ့ runCommand ကတော့စိတ်ဝင်စားစရာပါ။ Access right ကိုသူ့မှာစစ်ပါတယ်။ isAdmin ဆိုရင် command ကို subject ဆီပို့ပြီး run ပါတယ်။ မဟုတ်လို့ rm command ကိုတွေ့ရင် user ဆိုရင် ပေးမ run ပါဘူး။ Exception ကို throw လုပ်ပစ်ပါတယ်။ တခြား command ဆိုရင်ပေး run ပါတယ်။ ဒါကို if else နဲ့စစ်ထားတာပါ။ client code ကဒီလိုပါ။

```

public class ProxyDemo {

    public static void main(String[] args) throws Exception {
        CommandExecutor command = new CommandExecutorProxy("admin", "admin");
        command.runCommand("notepad.exe");

        command = new CommandExecutorProxy("user", "user");
        command.runCommand("rm");
    }
}

```

Client code မှာကွန်တောတို့က RealSubject ကိုမသုံးရပါဘူး။ Proxy ကိုပဲသုံးရပါတယ်။ Framework တွေမှာတော့ runtime မှာ real object နဲ့ proxy တွေကို change ပေးပါတယ်။ အဲဒါမှာ ပထမ command notepad.exe သည် admin အတွက်ဆိုရင် ပေး run မှာပါ။ နောက်ဒုတိယ command အတွက် user ဖြစ်နေတဲ့အတွက် rm command ကို run လိုက်ရင် Exception ကို throw ပါလိမ့်မယ်။

မေးစရာက ဘာလို့ Proxy ကိုသုံးလဲပေါ့။ Access control ကို RealSubject မှာပဲထိမ်းမယ် မရဘူးလားပေါ့။ ရပါတယ်။ ဒါဆိုရင် RealSubject သည် responsibilities 2 ခုဖြစ်သွားပါပြီ။ ဒါသည် OO principle အရ မကောင်းပါဘူး။ နောက်တခုက Access control ကိုပြောင်းခွင့်တိုင်းမှာ RealSubject code ကိုထိနေရတာပါ။ ဒါကမကောင်းပါဘူး။ Proxy နဲ့ဆိုထိစရာမလိုပါဘူး နောက်တချက်က လောလောဆယ် ခုရှိတာက Access Control proxy တခုပေါ့ တခြား proxy တွေ access control method တွေ ထပ်ထဲခွင့်ရင် Hard code လုပ်ထားတဲ့နည်းနဲ့ဆိုအဆင်မပြေဘူး။ AOP မှာ အဓိက key သည် မတူညီတဲ့ logic code တွေကို စုမထားပဲ တနေရာဆီခွဲထုတ်လိုက်တာပါ။ ဒါမျိုးကို Proxy pattern နဲ့မှအဆင်ပြေမှာပါ။

Proxy code ကတော့ဒီမှာ

<https://github.com/mrthetkhine/designpattern/tree/master/src/proxy>

JDK Dynamic Proxy စိတ်ဝင်စားရင်တော့ code ကဒီမှာ

<https://github.com/mrthetkhine/designpattern/tree/master/src/jdkdynamicproxy>

Object Oriented Design Pattern Series

Part-15 Command Design Pattern

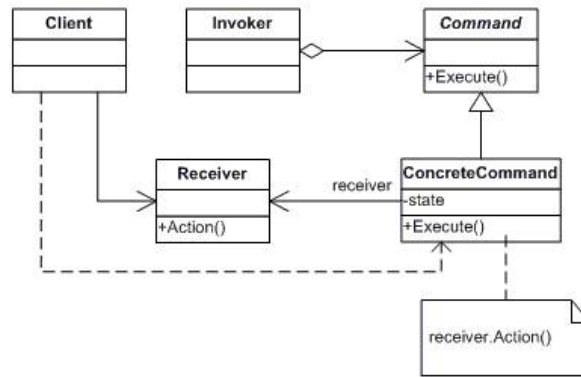
 THET KHINE · MONDAY, JANUARY 16, 2017

Command design pattern ဆိုတာ behavioural design pattern တွေထဲကတစ်ခုပါ။ သူ့ကိုအတွေ့ရများပါတယ်။ ဘယ်မှာအများဆုံး သုံးကြလဲဆိုရင် UI action တွေမှာ command pattern ကမပါမဖြစ်ပါ။ Java Swing Framework က ActionListener တို့၊ Android က View ရဲ့ OnClickListener ကို implement လုပ်တဲ့ကောင်တွေဟာ Command pattern ကိုသုံးထားတာတွေပါ။

Motivation

Command pattern ကိုဘယ်နေရာမှာသုံးလဲဆိုရင် request တစ်ခုခုကို (ဥပမာ menu ကိုနှိပ်ရင် လုပ်မဲ့ action တစ်ခုခုပေါ့ဗျာ) တယောက်ယောက်ကိုလုပ်ခိုင်းမယ် (button နှိပ်ရင်လဲဖြစ်ရင်ဖြစ်မယ် mouse click ရင်လဲ ဖြစ်ရင်ဖြစ်မယ်) ဒါပေမဲ့ ခုနက command ကိုတ ကယ် invoke လုပ်မဲ့သူက Command သည်ဘယ်ကလာတယ်ဆိုတာ သိစရာမလိုဘူး။ နောက်တခုက Command ထဲမှာ ဘာတွေပါလာတယ်ဘယ်လို context information တွေပါလာတယ်ဆိုတာ သိစရာမလိုဘူး။ သူ့အလုပ်သည် ပေးထားတဲ့ command ကို execute လုပ်ပေးလိုက်ယုံပဲ။ ဘယ်သူကပို့တာ အဲဒါဘာတွေပါတယ် (parameter or context information) အဲလိုအခြေအနေမှာ Command pattern ကိုသုံးမယ်။ မြင်သာအောင်ပြောရရင် button တစ်ခုခုကို click ရင် ဂျွန်တော်တို့က action တစ်ခုခုကို execute လုပ်စေချင်တယ်။ အဲဒါကို command object ထဲမှာထဲလိုက်မယ် button နှိပ်လိုက်ရင် ခုနက command ကို handle လုပ်ရမဲ့ object သည် command ကို execute လုပ်ယုံပဲ။ ဥပမာ အဲဒီ action command ကိုပို့တာသည် keyboard action ကနေလဲဖြစ်နိုင်သလို mouse ကနေလဲဖြစ်နိုင်တယ်။ Command ကိုဘယ်သူလွှတ်လဲဆိုတာသိစရာမလိုဘူး။ ဥပမာ Copy command ပေါ့ သူ့ကို keyboard ကနေလဲ လွှတ်နိုင်သလို mouse menu ကနေလဲလွှတ်နိုင်တယ်။ ဒါပေမယ့်အလုပ်က execute လုပ်ယုံပဲ။ ဘယ်သူဆိုတာအရေးမကြီးဘူး run ပေးယုံပဲ။ သူ့ထဲဘာပါလဲ ဆိုတာလဲ သိစရာမလိုဘူး။ အဲလိုအခြေအနေမှာ သူ့ကိုသုံးတယ်။ Java ရဲ့ ActionListener က actionPerformed သည် command execute လုပ်တဲ့ method ပါပဲ၊ ActionListener ကို implement လုပ်တာသည် command request object ပါပဲ။ အဲဒီ Command Object ကို form control ကိုပို့လို့ရသလို တခြား element တွေကိုလဲပို့လို့ရတယ်။ ဒါလောက်ဆို Command ကိုဘာကြောင့်သုံးရတယ်ဆိုတာ သဘောပေါက်ပါပြီ။

နောက်ဥပမာတစ်ခုအနေနဲ့ စစ်ဘက်ဆိုင်ရာ အဖွဲ့အစည်းတွေမှာ အမိန့်သည် အထက်က လာတယ် ဒါပေမဲ့ ဘယ်သူ့ဆီကလာတယ်ဆိုတာ သိခွင့်မှသိမယ်။ လုပ်ကွာဆို လုပ်လိုက်ရတာပဲ။ Command pattern သည်လဲ သည်သဘောပါပဲ။ နောက်တခုက Command pattern သည် undoable operation တွေကို လုပ်ဖို့အသုံးဝင်တယ်။ Series of command object တွေကိုမှတ်ထားလိုက်ရင်အဆင်ပြေတာကိုး။ နောက်တခုက မတူညီတဲ့ command တွေကို ဘုံထားထားတဲ့ interface သုံးပြီး execute လုပ်လို့ရမယ်။ ဥပမာ keyboard ကလာတဲ့ command ဖြစ်ဖြစ် mouse ကလာတဲ့ command ဖြစ်ဖြစ် အားလုံးကို same interface နဲ့ထားပြီး execute လုပ်နိုင်မယ်။ ဒါတွေသည် command pattern ရဲ့ ရည်ရွယ်ချက်နဲ့အသုံးဝင်ပုံပါပဲ။ အောက်ကပုံမှာ command pattern ရဲ့ class diagram ကိုပြထားပါတယ်။



အပေါ်ကပုံမှာ client သည် command pattern ကိုသုံးမဲ့ class ပါ။ နောက် Command ဆိုတာက Command request interface သူ့မှာ execute ဆိုတဲ့ကောင်ပါမယ်။ Command ကို handle လုပ်ရမဲ့ ကောင်တွေက execute ကို run ယုံပဲ။ Command သည် Interface ဖြစ်တဲ့အတွက် သူ့ကနေ Concrete Command တွေအများကြီးဆောက်လို့ရမယ်။ ဒါမှအမျိုးမတူတဲ့ command တွေ ဆောက်လို့အဆင်ပြေမှာ ကိုး။ Invoker ကတော့ Command တွေကို တကယ် handle လုပ်ပေးတဲ့သူ execute လုပ်မဲ့သူပေါ့။ သူက Command တွေကို execute လုပ်ယုံတင်မကပဲ history ကိုပါမှတ်ပေးထားပါတယ်။ ဒါမှ undo လုပ်လို့ရ မှာကိုး။ Code တွေကတော့ အောက်မှာပါ။

```

public interface Command {
    public void execute();
}

```

ဒါကတော့ Command request interface ပါ ဘာမှမပါပါဘူး Command လုပ်ခွင်တဲ့ class တွေသည် သူ့ကို implement လုပ် execute ကို override လုပ်ယုံပါပဲ။ ConcreteCommand class ကိုဒီလိုဆောက် ပါမယ်။

```

public class CopyCommand implements Command{

    @Override    public void execute() {
        System.out.println("Copy Executed");
    }
}

```

CopyCommandသည် Command interface ကို implement လုပ်ပြီး execute မှာသူလုပ်ခွင်တာလေး ကိုရေးယုံပါပဲ။ အောက်က EditCommand သည်လဲဒီလိုပါပဲ။

```

public class EditCommand implements Command{

    @Override
    public void execute() {
        System.out.println("Edit execute");
    }
}

```

နောက် Command တွေကို execute လုပ်ဖို့ Invoker ကိုရေးပါမယ်။ Invoker သည် history ကိုမှတ်ထား ပါမယ်။ ဘာလို့လဲဆိုတော့ undo operation တွေလုပ်ခွင်ရင်လုပ်လို့ရအောင်ပါ။

```

public class Invoker {

```

```
ArrayList<Command> history = new ArrayList<Command>();
```

```
public void invoke(Command command)
{
    history.add(command);
    command.execute();
}

public void undo()
{
    int len = this.history.size()-1;
    Command command = history.get( len );
    history.remove(len);
    System.out.println("Undo ");
    command.execute();
}
}
```

Invoker မှာ comand history ကိုသိမ်းဖို့. ArrayList ကိုသုံးပါတယ်။ သူ့ရဲ့ invoke မှာ command တခုကို invoke မလုပ်ခင် history ထဲထွပ်ပါတယ်။ နောက် command ကို execute လုပ်ပါတယ်။ ဒါဆိုရင် ဒီ invoker ထဲကို မတူညီတဲ့ command တွေကို series အလိုက် ထဲပြီး execute လုပ်နိုင်နိုင်ပါပြီ။ နောက် undo မှာတော့ history ထဲကနေ ထုတ်တယ် နောက် command ကို execute လုပ်ပါတယ်။ ဒါဆိုရင် undo operation အဆင်ပြေပါပြီ။ Invoker သည် list of command ကိုသိမ်းထားတဲ့အတွက် လာသမျှ command တွေ ဘယ်သူက ပို့ပို့. ဘယ်လို မတူညီတဲ့ command တွေဖြစ်ပါစေ execute လုပ်နိုင်ပါတယ် ဘာလို့လဲဆိုတော့ command pattern ကြောင့်ပါ။ ဒါသည် command ကို issue လုပ်တဲ့သူကို သိစရာမလိုပဲ command ကို execute လုပ်နိုင်ပါတယ်။ Loose coupling ဖြစ်တယ်ကောင်းတယ်လို့ဆိုရမှာပါ။ Client code ကတော့ဒီလိုပါ။

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

        Invoker invoker = new Invoker();
        Command copy = new CopyCommand();
        invoker.invoke(copy);

        Command paste = new Paste();
        invoker.invoke(paste);

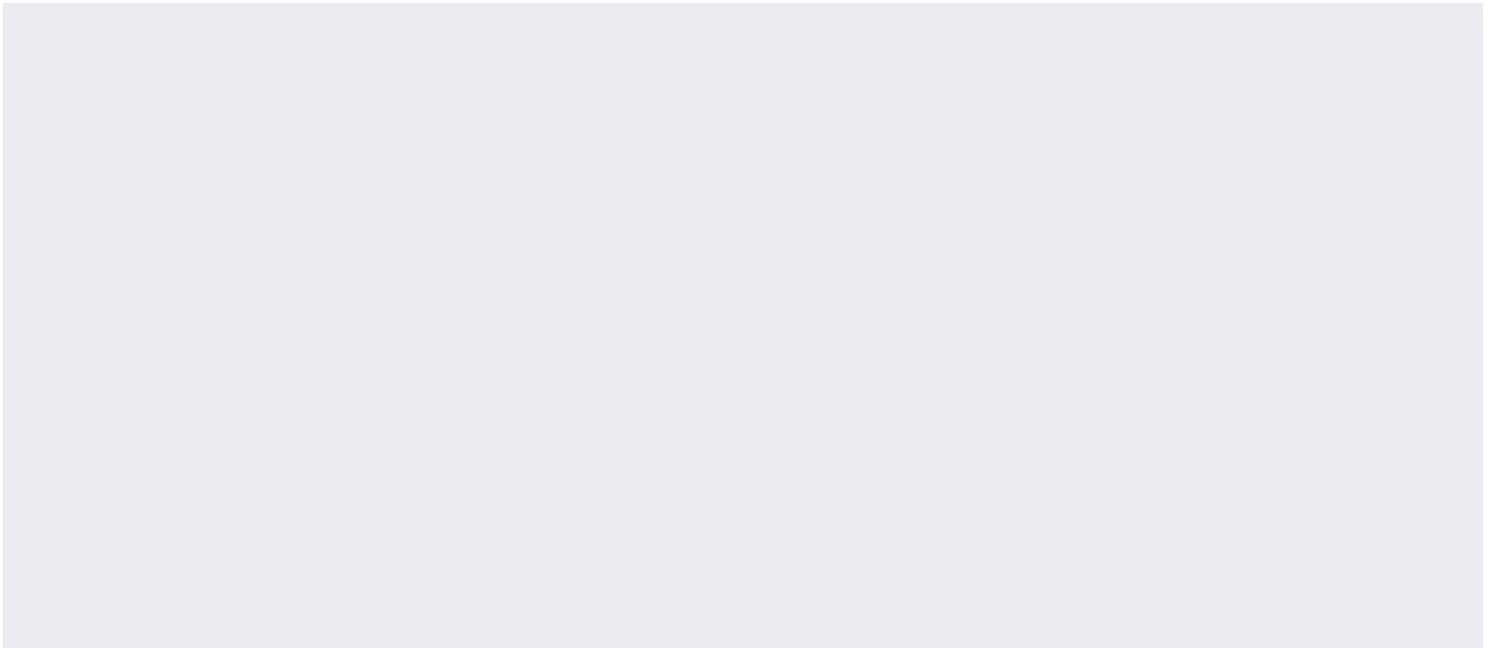
        invoker.undo();

        invoker.undo();
    }
}
```

}

Client code မှာ command တွေ create လုပ်တယ် နောက် Invoker ကနေ တဆင့် invoke လုပ်ခိုင်းတယ်။ နောက်သူတို့ကို undo ပြန်လုပ်တယ်။ ဒါပါပဲ။ Command pattern ရဲ့အသုံးအများဆုံးကတော့ GUI application တွေမှာပါပဲ။ source code ကတော့ အောက်မှာပါ။

<https://github.com/mrthetkhine/designpattern/tree/master/src/command>



Object Oriented Design Pattern Series

Part-16 Strategy Design Pattern

 THET KHINE · SUNDAY, MAY 7, 2017

Strategy design pattern က behaviour design pattern တွေထဲက တခုပါ။ သူ့ရဲ့ Intent ကတော့ GOF အရ ဒီလိုရေးထားပါတယ်။

Define a family of algorithms, encapsulate each one, and make them interchangeable.
Strategy lets the algorithm vary independently from clients that use it.

ဆိုခွင်တာက program မှာ interface အားဖြင့်တူညီပေမဲ့ မတူညီတဲ့ Algorithm တွေကိုပြောင်းပြီး သုံးရတဲ့ အချိန်တုရင် strategy ကိုသုံးလို့ရတယ်။ ဥပမာ Sorting စီမယ်။ ဒါပေမဲ့ sorting algorithm က တခုတည်းမဟုတ်ဘူး data အပေါ်မူတည်ပြီး ဘယ် algorithm နဲ့ စီမလဲဆိုတာမျိုး Algorithm ကို dynamically ပြောင်းခွင့်တယ် ဒါဆိုရင် strategy ကိုသုံးလို့ရတယ်။ နောက် ဥပမာ တခုပေါ် Payment Gateway တွေ အများကြီးရှိနေတယ် user ရဲ့ payment အပေါ်မူတည်ပြီး ဘယ် gateway နဲ့သုံးမယ်ဆိုပြီးရေးရမယ် ဒီအခါ if နဲ့ hard code မလုပ်ပဲ Strategy pattern ကိုသုံးပြီးရေးလို့ရတယ်။ အဲဒီတော့ algorithm ကိုပြောင်းရ လွယ်တယ်။ Strategy pattern ရဲ့ ရည်ရွယ်ချက်က family of algorithm တွေကို ပြောင်းပြီး သုံးလို့ရအောင် ရေးထားတာ အဲသဘောပဲ။ သူ့ကိုသုံးနေတဲ့ client code သည် နောက်ကွယ်က algorithm ပြောင်းလား မပြောင်းလား သိစရာမလိုပဲနဲ့ သုံးလို့ရအောင် ရေးခွင့်ရင် strategy ကိုသုံးလို့ရတယ်။

အောက်က အခြေအနေတွေမှာ strategy pattern ကိုသုံးလို့ရတယ်။

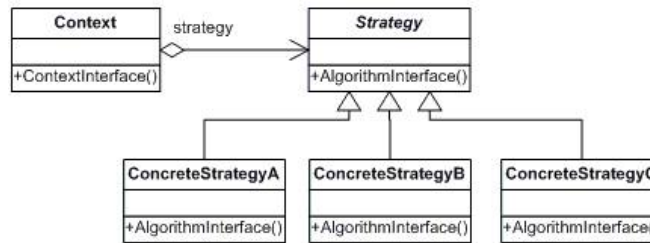
ဆင်တူဖြစ်နေတဲ့ classes တွေသည် behaviours ပိုင်းအရ ပဲကွဲတယ်။ ဥပမာ selection sort, merge sort သည် sorting လုပ်ထုံးလုပ်နည်းသာ ကွာမယ်။ တကယ့်အလုပ်က sorting စီတာပဲ ဒါမျိုးကိုဆိုခွင်တာ။ အဲလို behaviour မတူတဲ့ class တွေကို တစုတည်း သုံးဖို့အတွက် Strategy ကိုသုံးလို့ရတယ်။

Algorithm တွေရဲ့ variation ကိုလိုမယ်။ ဥပမာ အခြေအနေအပေါ်မူတည်ပြီးတော့ algorithm ပြောင်းရမယ်။ ဒါဆိုရင် လွယ်လွယ်ကူကူပြောင်းရအောင် Strategy ကိုသုံးရမယ်။ ဥပမာ Master, Visa ကို payment gateway 1 ကနေ handle လုပ်နိုင်တယ်။ တခြား card တွေကိုတော့ တခြား gateway သုံးမယ်။ ဒါဆိုရင် User ရွေးလိုက်တဲ့ card အပေါ်မူတည်ပြီး ဘယ် gateway သုံးမယ်ဆိုတာကို strategy pattern သုံးပြီး encapsulate လုပ်ထားလို့ရတယ်။

Client code က မသိသင့်တဲ့ data တွေ complex operation တွေကို encapsulate လုပ်ဖို့အတွက်လဲ သုံးလို့ရတယ်။

တခါတလေ မှာ multiple if တွေနဲ့အများကြီး hard code စစ်နေရမဲ့အချိန်မျိုးမှာလဲ သုံးလို့ရတယ်။ မေးစရာရှိတယ် if သုံးလို့ရနေသားနဲ့ဘာလို့ strategy သုံးခိုင်းလဲပေါ့။ Maintenance ကပိုကောင်းသွားတယ်။ If သည် hard code လုပ်ထားတာ အသေ ဖြစ်သွားတယ်။ Strategy ဂျာနာ extend လုပ်လို့ရသွားတယ်။ ထပ်တဲ့လို့အဆင်ပြေတယ်။ ဒါကို Open Close principle လို့ခေါ်တယ်။ open for extension , close for implementation ကိုပြောတာ။

အောက်က သူ့ရဲ့ pattern structure ပုံ။



Context သည် client code သူသည် strategy ကိုထိန်းထားတယ်။ Client ကနေ Context ကိုပဲသုံးမယ်။ strategy ကိုတိုက်ရိုက်မသုံးဘူး။ ဒီနည်းနဲ့ encapsulation ကိုထိန်းထားတယ်။ Strategy က interface သူ့ကနေ ConcreteStrategy တွေ အများကြီး ဆောက်ထားလို့ရတယ်။ Interface ကိုသုံးထားပြီး extend လုပ်လို့ရတဲ့အတွက် Open for extension ကိုပေးပြီးသားဖြစ်သွားမယ်။ ဆိုရှင်တာက Algorithm အသစ် တခုထပ်ထဲ့ရှင်ရင် strategy ကို implement လုပ်ယုံပဲ။ Context ကနေ Strategy ကို composition အနေ နဲ့သုံးတယ်။ ဘာလို့ composition ကိုသုံးလဲဆိုရင် ပြောင်းရ ပြုရလွယ်တယ်။ Favor composition over inheritance ဆိုတဲ့ principle အရ ကိုသုံးတာ။

ကဲ Code ရေးရအောင် sorting algorithm တွေလိုမယ်ပေါ့။ Data အပေါ်မူတည်ပြီး ဘယ် Algorithm ကို သုံးမလဲဆိုတာ ဆုံးဖြတ်ရမယ်။ Algorithm တွေကို runtime မှာ ပြောင်းရှင်တယ်ပေါ့။ ဒါကို strategy နဲ့ ရေးရင် ဒီလိုရမယ်။ ပထမဆုံးကတော့ Strategy interface ဆောက်လိုက်မယ်။

```
public interface SortStrategy
{
    public void sort();
}
```

သူ့မှာ sort ဆိုတာပဲ ပါမယ်။ interface နဲ့ထားတဲ့အတွက်ကြောင့် SortStrategy ကို implement လုပ်တဲ့ class တိုင်းသည် SortStrategy မှာ အစားထိုးသုံလို့ရမယ်။ ဒီနည်းနဲ့ family of algorithm ကို encapsulate လုပ်နိုင်မယ်။ နောက်တခုက Context class.

```
public class Context
{
    SortStrategy strategy;
    public SortStrategy getStrategy()
    {
        return strategy;
    }

    public void setStrategy(SortStrategy strategy)
    {
        this.strategy = strategy;
    }
}
```

```

    public void sort()
    {
        this.strategy.sort();
    }
}

```

Context သည် SortStrategy ကို composition အနေနဲ့ ယူသုံးထားတယ်။ Concrete class ကို ယူမသုံးပဲနဲ့ interface ဖြစ်တဲ့ SortStrategy ကိုပဲ composition အနေနဲ့ သုံးထားတာ။ ဒုတိယအတွက် SortStrategy ကနေဆင်းလာတဲ့ class အားလုံးကို exchange လုပ်ပြီး သုံးလိုရမယ်။ ဒါသည် Open for extension ပဲ။

နောက် Sort Algorithm တွေကို ဒီလို implement လုပ်မယ်။

```

public class MergeSort implements SortStrategy
{
    @Override
    public void sort()
    {
        System.out.println("Sorting with Merge Sort");
    }
}

public class SelectionSort implements SortStrategy
{
    @Override
    public void sort()
    {
        System.out.println("Sorting with selection sort");
    }
}

```

ဘာမှ ထူးထူးဆန်းဆန်းမရှိဘူး varied behaviour ကို class တခုချင်းဆီမှာ လိုက်လံလုပ်တာပဲ။ ဒါပေမဲ့ အားလုံးက SortStrategy ကို implement လုပ်မယ့်။ ဒါမှ context ကေန သုံးလိုရမှာကိုး။ Client code ကေတုဒီမှာ။

```

public class StrategyDemo
{
    public static void main(String[] args)
    {
        Context context = new Context();

        context.setStrategy(new SelectionSort());
        context.sort();

        context.setStrategy(new MergeSort());
        context.sort();
    }
}

```

```
}  
}
```

Context ရဲ့. setStrategy ကိုသုံးပြီး strategy တွေကို runtime မှာပြောင်းပြီးသုံးတယ်။ client code သည် ဘာမှပြောင်းစရာမလိုဘူး။ Algorithm ကို လွယ်လွယ်ကူကူ change နိုင်တယ်။ Java API မှာဆို Sorting မှာ Comparator တွေကိုသုံးတာသည် Strategy design pattern ကိုသုံးတာပဲဖြစ်တယ်။

ဘာကောင်းလဲဆိုရင် if နဲ့. logic ကို hard code လုပ်စရာမလိုဘူး။ Open for extension လုပ်ထားတဲ့အတွက် နောက်ပိုင်း Algorithm တွေထပ်ထဲ့ရတာလွယ်မယ်။ ပြင်ရင်လဲ client code လေးလောက်ပဲထိရမယ်။ ဒဲ့တော့ maintenance ကောင်းမယ်။ Code ကတော့အိမှာ

<https://github.com/mrthetkhine/designpattern/tree/master/src/strategy>