Object Oriented Programming

And

Design Pattern Series

Object Oriented Programming and Design Pattern Series

Object Orient Design Pattern Part 1

ခုလောလောဆယ် Industry မှာ အသုံးအများဆုံး programming language paradigm အရဆိုရင်တော့ Object Oriented Programming က popular အဖြစ်ဆုံးလို.ပြောရမှာပါ။ အသုံးများတဲ့ OO Language တွေကတော့ Java, C#, JavaScript, PHP, Ruby , Python တို.ပါပဲ၊ Language တခုကိုလေ့လာတဲ့အခါမှာ element ၃ ခုကိုလေ့လာရပါတယ်။ Syntax, Semantic နဲ. Pragmatic တို.ဖြစ်ပါတယ်။ Syntax ကတော့ရှင်းပါတယ် language basic component တွေကို ဘယ်လိုရေးရတာလဲဆိုတဲ့ Grammar ပါ။ ဥပမာ for loop ပတ်ရင်ဘယ်လိုရေးရသလဲပေါ့။ Semantic ကတော့ for loop ရေးထားရင် for loop ကဘယ်လိုအလုပ်လုပ်မလဲဆိုတဲ့ တိကျတဲ့အဓိပ္ပာယ်ကို ဆိုချင်တာပါ။ ဥပမာ for initialization ရှိရင် run မယ်။နောက် for conditional ရှိရင်စစ်မယ် result က true ဆိုရင် for body ကို execute မယ်။ ပြီးရင် for increment ရှိခဲ့ရင် သူ.ကို run မယ် နောက် loop အစကိုပြန်တက်မယ် condition ပြန်စစ်မယ် ပေါ့။ ဒါတွေကို semantic လို.ခေါ်ရမှာပါ။ Programming သင်တယ်လို.ပြောလိုက်ရင် syntax ကိုပဲသင်ကြတာကို တွေ.ရပါတယ်။ language 2 ခုက syntax အရ ဆင်နိုင်ပါတယ် ဒါပေမဲ့ semantic အရ မတူနိုင်ပါဘူး ဥပမာ for loop ဆိုပါစို့. C,C++,Java,C#,JavaScript တို.မှာ syntax အရသိပ်မကွာပါဘူး ဒါပေမဲ့ semantic အရ မတူနိုင်ပါဘူး။ ဥပမာ C/C++/JavaScript မှာ conditinal part ဟာ boolean မဟုတ်တဲ့ integer တခုခု (JS မှာဆို Object type ပါဖြစ်နိုင်ပါတယ်) ဖြစ်ခွင့်ရှိပေမဲ့ Java,C# မှာဆိုရင် boolean type ကိုဖြစ်ရမှာပါ။ နောက်တခုက for initialization မှာ variable ကြေငြာ ခဲ့ရင် C,C++,Java,C# မှာဆိုရင် for loop အတွင်းမှာပဲ variable scope

ကရှိပေမဲ့ JavaScript မှာဆို the whole function တခုလုံးလို.ပြောရမှာပါ။ ဒါဟာ semantic အရ မတူကြောင်းကိုပြောတာပါ။

နောက်တခုက Pragmatic ပါ။ သူကတော့ language တခုရဲ. construct တွေကို လက်တွေ့.ကျကျဘယ်လိုသုံးရမလဲဆိုတာပါ။ Programming language တခုကပေးထားတဲ့ feature တွေ language construct တွေကို သင့်တော်မှန်ကန်စွာ အသုံးချနိုင်မှု.ပါ။ ဘာကိုဆိုချင်သလဲဆိုတော့ OO paradigm ကို support ပေးတဲ့ language မှာ OO program တွေကို ဘယ်လိုရေးရသလဲ ။ ဥပမာ ဘယ်အချိန်မှာ inheritance ကိုသုံးလဲ ရိုးရိုး class ကိုသုံးရမှာလား abstract class ကိုသုံးရမှာလား abstract class နဲ. interface usage ဘယ်လိုကွာလဲ ဘယ်နေရာဘယ်သူ.ကိုသုံးရမှာ correct usage က ဘာတွေလဲ ဒါတွေကို ဆိုချင်တာပါ။ Pragmatic ကိုအဆင့်ထပ်မြှင့်ရင်တော့ design pattern တွေ ,principle တွေ ပါ ပါလာမှာပါ။ Programmer တယောက်အနေနဲ. Language construct တွေ paradigm နဲ.ပတ်သတ်တဲ့ concept တွေကိုသိဖို.လိုပါတယ်။ Java နဲ.ရေးနေပေမဲ့ C မှာရေးသလို function တွေနဲ.ပတ်လည်ရေးနေတယ်ဆိုရင် မဟုတ်တော့ပါဘူး။

Evolution to OOP

Modern high level Programming language တွေမပေါ်ခင်က machine language သို.မဟုတ် assembly language ကိုသုံးရပါတယ်။ Assembly ကိုသုံးရတာဟာ machine language ကို symbolic code အစားထိုးသုံးရတာနဲ.ဘာမှမကွာပါဘူး။ သိပ်မဟန်ပါဘူး ။ဒါကြောင့် high level language တွေထုတ်ပါတယ်။ အစောပိုင်းက data structure နဲ. algorithm တွေရေးလို.အဆင်ပြေအောင် ထုတ်ထားတဲ့ Procedural Language(Fortan, Algo, C) တွေပေါ်လာပါတယ်။ Procedural language တွေရဲ.အဓိက concept ကတော့ fuction

တွေ့ခဲ့ရေးမယ် ။ function တွေကို reusable ဖြစ်အောင်လုပ်မယ်။ ဒီလိုနည်းနဲ့.သွားပါတယ်။ နောက်ကျတော့ data abstraction ကိုထဲ့လာပါတယ်။

Data Abstraction

Data Abstraction ဆိုတာ custom data type တွေ create လုပ်ခွင့်ပေးထားတာကိုပြောတာပါ။ ဥပမာ Stack, LinkedList ဆိုတာမျိုးကို data type တခုအနေနဲ့.ကိုယ်ပိုင် ဖန်တီးလို့.ရမယ်။ ပြန်သုံးလို့.ရမယ် ဒါကို data abstraction လို့.ပြောလို့.ရပါတယ်။ ဥပမာ C မှာ ဆို struct, enum, class ဆိုတာ data abstraction တွေပေးထားတာပါ။ Primitive type တွေကနေ custom data type ဖန်တီးလို့.ရမယ် အဲ့ data type ကို maniuplate လုပ်ဖို့. function တွေ create လုပ်လို့.ရမယ်ဆိုရင် အဲ့ဒီ language ကို data abstraction ပေးတယ်လို့.ပြောရမှာပါ။

Beginning of OOP

Data abstraction ကြောင့် custom data type တွေ တော့ဖန်တီးလို့.ရပါပြီ။ ဒါပေမဲ့ ပြဿ နာ တခုက သူ့.ကိုပြင်မယ်ဆိုရင် ဆိုပါစို့. existing ADT(Abstract Data Type) တခုကိုပြင်မယ်ဆိုရင် သူ့. source code ကိုပြင်ရမှာပါ။ ဒါကို destructive modification လို့.ခေါ်ပါတယ်။ အဲ့ဒါက အန္တရာယ်များပါတယ် ဘာလို့.လဲဆိုတော့ project တခုမှာ ADT တခုကို reference လုပ်ပြီးသုံးထားတဲ့ code တွေ အများကြီးရှိနိုင်လို့.ပါ။ ADT ရဲ့. source code ကိုမထိပဲပြင်နိုင်တဲ့ နည်းကတော့ OOP သုံးပြီး inheritance နဲ့. functionality အသစ်ကို ထပ်ထဲ့တာ existing method ကို modify လုပ်တာပါပဲ။ ဒါဆို ရှိပြီးသား code ကို မထိပဲနဲ့. ပြင်လို့.ရပါပြီ။ Software ဆိုတာ ရေးပြီးတာနဲ့.ပြီးတာမဟုတ်ပဲ အမြဲ ပြောင်းလဲနေနိုင်ပါတယ် OOP ရဲ့. feature တွေက ဒါတွေကို handle လုပ်နိုင်ပါလိမ့်မယ်။

Data Abstraction versus OOP

Data abstraction က custom data type တွေ တည်ဆောက်လို.ရမယ်။ OOP ဖြစ်ဖို.ကတော့ အောက်ပါ သုံးချက်ကိုမဖြစ်မနေပေးရပါတယ်။ အဲ့ဒါတွေကတော့

- Encapsulation
- Inheritance
- Polymorphism

တို.ပါပဲ။ Object တော့ပေးဆောက်တယ် feature သုံးခုလုံးမပါဘူဆိုရင် Object based language လို.ပဲခေါ်မှာပါ ဥပမာ Visual Basic(VB.NET ကတော့ OOP Feature သုံးခုလုံးပေးထားပါတယ်). OO language ရဲ.အားသာချက်ထဲက တခုက တော့ conceptual modelling ပါပဲ၊ အရင်က software development ကို function တွေ algorithm တွေနဲ.စဉ်းစားမဲ့အစား real world မှာရှိတဲ့အတိုင်း object တွေအနေနဲ. စဉ်းစားရတဲ့ ပိုလွယ်ကူပါတယ်။ real world မှာရှိတဲ့ အတိုင်း classification (by mean of class), taxonomy (by mean of inheritance), specialization (by mean of polymorphism) အတိုင်း model လုပ်လို.ရသွားပါတယ်။

Pure and Impure OO Language

Programming language တခုဟာ Object ကလွဲပြီး တခြား construct တွေပေးမထားဘူးဆိုရင် သု.ကို pure Object Oriented Language လို.သုံးပါတယ်။ ဥပမာ Smalltalk မှာဆို loop ဆိုတာမျိုး မရှိပဲ loop ဆိုတာကို method တစ်ခုအနေနဲ.ယူဆပါတယ်။ integer လိုကောင်မျိုးတွေကအစ object တွေပါ ဒါကြောင့် pure OO language လို.သုံးပါတယ်။ ဘာကောင်းလဲဆိုတော့ program တစ်ခုလုံးကို OO နည်းနဲ.ပဲစဉ်းစားရအောင် လုပ်ထားတာပါ။ C++, Java,C#,JS တို.ကတော့ impure object oriented language လို.ပြောရမှာပါ။ သူတို.မှာ primitive data type တွေပါပြီး control structure (for/switch etc) တွေပါလို.ပါ။

Encapsulation

Procedural language တွေမှာ data ကိုဘုံထားပြီး function တွေကနေဝိုင်းသုံးကြပါတယ်။ Project size ကြီးလာတာနဲ့.အမျှ အဲ့ဒီ ဘုံသုံးထားတဲ့ variable (global variable) တွေကို ဘယ် module, ဘယ် function ကသုံးထားတယ်ဆိုတာ လိုက်ကြည့်ဖု.ိတော်တော်ခက်သွားပါပြီ။ PHP မှာဆို global variable သုံးပြီး page အများကြီး ဝိုင်းသုံးတာမျိုးပါ။ နောက်ပြီး ကိစ္စရှိလို. အဲ့ဒီ global variable ကိုပြင်မယ်ဆိုရင် သူ.ကို သုံးထားတဲ့ module တွေ function တွေကို ပါလိုက်ပြင်ရမှာပါ။ တခုခုကိုပြင်ရင် တခြား အပိုင်းတွေပါ ထိခိုက်ကုန်ပါတယ်။ ဒါကြောင့် OO language မှာ အဲ့တာကိုမဖြစ်အောင် ကာကြပါတယ်။ Program တွေကို modular ဖြစ်အောင်ရေးဖို.ကြိုးဆကြပါတယ်။ Modular ဖြစ်တယ်ဆိုတာ program module, function, contruct တခုဟာ သူ.ဟာသူနဲ.ရပ်တည်နိုင်ရပါမယ်။ သူများ function တွေကို အများကြီးသုံးနေရတာ dependency များနေတာ မဖြစ်ရပါဘူး။ Modular ဖြစ်မှ dependency နည်းမှပဲ ပြင်ရတာ လွယ်ကူမှာဖြစ်ပါတယ်။ ဒါကြောင့် global variable အစား ကို. data ကိုပဲသုံးမယ် သူများက လာသုံးမရအောင် access ကို restrict လုပ်ထားမယ် သူ. data ကို သူ. method တွေကပဲသုံးမယ် တခြားသော module တွေက ဒီ module ကို သုံးချင်ရင် module ကပေးထားတဲ့ public interface(publicly accessible method) တွေ ကနေ သုံး။ ဒါဆိုရင် တခြား module တွေဟာ သူများရဲ. data ကိုသွားထိလို.မရတဲ့အတွက် သီးသန့်.ဆန်သွားတဲ့အတွက် modularity ပိုကောင်းပါတယ်။ module တခုကိုပြင်မယ်ဆိုရင် public interface(public interface ကပဲသူများ module နဲ့. dependence ဖြစ်ပါတယ်) ကိုမပြင်မချင်း လွတ်လွတ်လပ်လပ်ပြင်နိုင်မယ်။ ဒါကို encapsulation လို.သုံးပါတယ်။ Encapsulation ရဲ.အနှစ်သာရက ကို. data ကိုသူများက accidentally destroy မဖြစ်အောင် dependency မရှိအောင် ထားရတာပါ။ Encapsulation ကို data hidiing နဲ့.တွဲသုံးပါတယ်။ Encapsulation နဲ့.

data hiding ဟာ တူသလိုလိုနဲ့. ကွဲပါတယ် data hiding က ကို.data ကိုပြင်ပက လာသုံးလို.မရအောင်ကာထားတာမျိုးပါ။ Encapsulation ကတော့ data ကို ကို. function တွေနဲ့. modularity ရအောင် ချိတ်သုံးတာကို ဆိုချင်တာပါ။ Java, C#, C++ မှာဆို private accessifier တွေသုံးပြီး encapsulation ရအောင်လုပ်ကြပါတယ်။ Member variable ကို Private ထားပြီး getter ,setter function တွေနဲ့.သုံးကြပါတယ်။ အဲ့လိုသုံးတိုင်း encapsulation မရနိုင်ပါဘူး ။ အောက်က Java Program ကိုကြည့်ပါ။

```
class WrongEncapsulation
{
private CreditCard card;
public void setCard(CreditCard c){...}
public CreditCard getCard()
{
return card;
}
}
```

ဒီ Program မှာ card က private ပါ reference type ပါ။ သူ့.ကို တိုက်ရိုက်ယူသုံးလို.မရပေမဲ့ getCard ကနေ ယူသုံးလို.ရပါတယ်။ဒါဆို တနည်းအားများ card ကို တိုက်ရိုက်ယူသုံးနိုင်တာနဲ့.အတူတူပါပဲ။ Java က reference model ကိုသုံးတဲ့အတွက် အပြင်ကနေ card ကို ပြင်လို.ရမှာပါ။ဒါဆို encapsulated ဖြစ်တယ်လို.မပြောနိုင်တော့ပါဘူး။ JavaScript လို language မျိုးကျတော့ access modifier မပါပါဘူး ဒါပေမဲ့ full encapsulation ကိုလုပ်လို.ရပါတယ်။ Closure သုံးပြီး encapsulate လုပ်နိုင်ပါတယ်။

```
<script>
function getObject() {
```

```
var data;

function dataGetter() {
return data;
}

function dataSetter(d)
{
data = d;
}
return {
setData : dataSetter,
getData : dataGetter
};
}
var obj = getObject();
obj.setData(100);
console.log(obj.getData());
</script>
```

အပေါ်မှာပြထားတဲ့ JavaScript code မှာ encapsulation ကိုပြထားပါတယ်။ getObject function ထဲမှာရှိတဲ့ data ဟာ encapsulated လုပ်ခံထားရတာပါ။ သူ.ကိုအပြင်ကနေ တိုက်ရိုက်ယူသုံးလို.မရပါဘူး။ ဘာလို.လဲဆိုတော့ local varaible မို.လို.ပါ။ getObject function အောက်ဆုံးမှာ object တခု return ပြန်ထားပါတယ် အဲ့မှာ setData, getData ကို getObject function ထဲက dataSetter,dataGetter ကိုပေးထားပါတယ်။ dataSetter,နဲ. dataGetter ဟာ inner function တွေဖြစ်တဲ့အတွက် data ကို enclosing scope က data ကိုယူသုံးလို.ရပါတယ် ။ သူတို.ကိုတော့အပြင်ကနေ တိုက်ရိုက်သုံးလို.မရပဲ setData နဲ. getData ကတဆင့်သုံးရမှာပါ။ JS

မှာဒီနည်းနဲ့. Encapsulation ထိန်းလို့.ရပါတယ်။ Encapsulation ဆိုတာ class level ထိန်းမှမဟုတ်ပါဘူး package, module တွေဟာလဲ ပြင်ပ ကကောင်တွေတိုက်ရိုက်သုံးလို့.မရအောင် ထိန်းပေးထားတဲ့ encapsulation construct တွေပါပဲ။ Encapsulation နဲ့. ပတ်သတ်တဲ့ OO Principle တခုရှိပါတယ် အဲ့ဒါကတော့ Program to interface, not to implementation ပါ။ Program ရေးတဲ့အခါမှာ module တွေ class တွေဟာ သူများရဲ့. implementation detail ကိုမသိသင့် မထိသင့်သလို. ကို.ဟာကိုလဲ expose မလုပ်သင့်ပါဘူး။ အပြင်ကသုံးမဲ့ ကိုနဲ့.တကယ် interact ပေးလုပ်မဲ့ကောင်တွေကိုပဲ public interface အနေနဲ့.ထားပေးရမယ်ဆိုလိုတာပါ။ ဒါမှ maintenance ကောင်းမှာပါ။Encapsulation နဲ့.ပတ်သတ်တဲ့ တခြား OO principle တခုကတော့ Open Close Principle ပါပဲ။ A module should be open for extension but closed for modification. Module တွေ classes တွေဟာ သူတို့.ကို accidentally modified မလုပ်နိုင်အောင်ထားသင့်ပြီးတော့ extend လုပ်နိုင်အောင်တော့ ထားပေးရမယ်ဆိုတာပါ။ Encapsulation ရဲ့. Central theme ကိုက close for modification ပါပဲ။

Object Orient Design Pattern Part 2

Inheritance

OO paradigm မှာမပါမဖြစ်တဲ့နောက် feature တစ်ခုက Inheritance ပါ။ Inheritance ကဘာလုပ်ဆောင်မှု.တွေကိုခွင့်ပေးသလဲဆိုတော့ existing class တခုကနေ code reuse သို.မဟုတ် နဂိုမူလရှိတဲ့ base class ကို extend လုပ်ပြီး functionality ကို modify လုပ်တာဖြည့်စွက်တာတွေကို ခွင့်ပေးပါတယ်။ Inheirtance ကိုသုံးပြီး နဂိုမူရင်း base class ရဲ. source code ကိုမထိပဲ ပြင်လို.ရနိုင်ပါတယ် (extend လုပ်ပြီးပေါ့)။အဲ့လို non-destructive modification တွေဟာ modern software development မှာမဖြစ်မနေကိုလိုအပ်ပါတယ်။ Modified class ကို base class နေရာမှာသုံးလို.ရတာကြောင့် software maintenance ကိုပိုမိုလွယ်ကူစေပါတယ်။ Inheritance ရဲ. major usage က ၂ခုရှိပါတယ် design မှာသုံးလို.ရသလို implementation အတွက်လဲသုံးပါတယ်။ အဲ့ဒါတွေကတော့

Inheirtance as conceptual modelling

Inheritance as incremental program modification

OOP ရဲ.တခြား paradigm တွေထက် popular ဖြစ်ရတဲ့အကြောင်းရင်းက Object Orientation ဟာ implementation သက်သက်သာ မဟုတ်ပဲနဲ. modelling or design အတွက်ပါသုံးလို.ရလို.ဖြစ်ပါတယ်။ Real world problem တွေကို OO thinking နဲ. model ချတဲ့အခါမှာပိုမိုလွယ်ကူပါတယ်။ လူတွေဟာ အရာဝတ္ထုတွေ အကြောင်းအရာတွေကို ခွဲခြားစိတ်ဖြာတယ် (classification ဥပမာ လူဆိုတာ သတ္တဝါ တိရစ္ဆာန်ကလဲ သတ္တဝါ) ၊ generalization (ဥပမာ car engine ကလဲစက် motor engine ဆိုတာလဲ စက်)၊ grouping,composition အစရှိတာတွေနဲ.တွေးခေါ်ကြပါတယ်။ ခုနကပြောတဲ့ classification၊

generalization၊ specialization အစရှိတာတွေကို OO paradigm မှာ easiliy model လုပ်လို့.ရပါတယ်။ classification ကို class contruct နဲ့.လုပ်ကြပါတယ်။ Generalization နဲ့. Specialization ကိုတော့ inheritance နဲ့. model လုပ်ကြပါတယ်။ Grouping နဲ့. composition ကိုတော့ class တွေမှာ တခြား classes တွေရဲ့. reference variable တွေထဲ့သွင်းခြင်းအားဖြင့် သုံးကြပါတယ်။

Inheritance as conceptual modelling

Inheritance ကို Generalization, Specialization အတွက်သုံးပြီဆိုရင် ဒါဟာ inheritance as coneptual modelling ပါ။ Base class or parent class ကို generalized classes လို့.ခေါ် ပြီးတော့ child class ကိုတော့ specialized class လို့.ခေါ်ပါတယ်။ Conceptual modelling လုပ်မယ်ဆိုရင် base class နဲ့. drived class ဟာ taxonomy အရတူရမှာပါ။ ဥပမာ Teacher, Doctor နဲ့. Human ဆိုတာဟာ taxonomy အရတူပါတယ်။ Hierarchical relationship ရှိတာကိုပြောချင်တာပါ။ Teacher, Doctor သည် is a kind of Human ပါပဲ။ Taxonomy အရ မတူဘူးဆိုရင် Generalization Specialization မလုပ်သင့်ပါဘူး ဥပမာ Bird နဲ့. Aeroplane သည် ပျံတာချင်းတူပေမဲ့ သူတို့.ကို Inheritance နဲ့.ချိတ်လို့.မရပါဘူး။ သူတို့.က taxonomy အရမတူပါဘူး။ Inheritance လို conceptualization လုပ်ပေးနိုင်တဲ့နောက်တခုက တော့ Subtyping ပါ။Subtyping ဆိုတာ programming lanaugag ကနေပေးထားတဲ့ feature တွေကိုသုံးပြီး type တခု သည် အခြား type တခုရဲ့. subtype ဖြစ်ပါတယ်ဆိုပြီးသုံးတာပါ။ Base type နေရာမှာ based type ကိုအခြေခံထားတဲ့ တခြား type ကိုအစားထိုးပြီးသုံးလို့.ရပါတယ်။ Subtyping ကို type polymorphism လို့.လဲသုံးကြပါသေးတယ် ။ Java, C# တို့.မှာဆိုရင် interface construct ကိုထဲ့ပေးထားပါတယ်။ သူ.ကိုထဲ့ပေးထားရတဲ့အကြောင်းရင်းက conceptual modelling အရ

taxonomy မတူတဲ့ကောင်တွေကို polymorphic လုပ်ချင်ရင်သုံးဖို့.ပါ။ Java programmer တော်တော်များများ က abstract class နဲ့. interface ဘာကွာလဲမေးလိုက်ရင် ရေရေရာရာကွဲပြားတဲ့အဖြေကို မဖြေနိုင်ကြပါဘူး။ အဓိက ကွာခြားချက်က taxonomy တူမယ်ဆိုရင် abstract class ကိုသုံးပြီး inheritance နဲ့.ဖြေရှင်းမယ် မတူဘူးဆိုရင် subtyping ကို သုံးပြီး interface နဲ့. design လုပ်ရမှာပါ။ ဥပမာ work ဆိုတဲ့ method ကို polymorphic ဖြစ်အောင်လုပ်ချင်တယ်ဆိုပါစို့. Teacher မှာရော Doctor မှာရော လုပ်ချင်တာဆိုရင် သူတို့ ၂ခုဟာ taxonomy အရတူတဲ့အတွက် abstract class ဒါမှမဟုတ် ရိုးရိုး class သုံးပြီး inheritance နဲ့. model လုပ်ရမှာပါ။ Fly ဆိုတဲ့ method ကို Bird နဲ့. Aeroplane အတွက် polymorphism လုပ်ချင်တယ်ဆိုရင် သူတို့ ၂ ခုဟာ taxonomy မတူတဲ့အတွက် Flyable ဆိုတဲ့ interface တခုထားပြီး model လုပ်ရမှာပါ။ Static type language တွေမှာ polymorphic operation တွေလုပ်ဖို့ subtype တွေဖြစ်မှ လုပ်လို့.ရပါတယ်။ Subtype မဟုတ်ရင် method ရှိနေလဲ ခေါ်လို့.မရပါဘူး။ Dynamic language တွေမှာ duck typing ကိုပေးထားတဲ့အတွက် အဲ့လိုခေါ်လို့.ရပါတယ်။ ဒါပေမဲ့ PHP လို dynamic language ဖြစ်တယ် duck typing ရတဲ့ language မှာ interface ကိုထဲ့ပေးထားတာဟာ conceptual modelling အရ subtyping ကိုသုံးစေချင်လို့.ဖြစ်ပါတယ်။ နောက်တခုက normal class နဲ့. abstract class ဘယ်လို usage ခွဲမလဲပေါ့ ။ Parent ကနေပဲ child က code တွေကို one direction ခေါ်ချင်ရင် hook အနေနဲ့.ထားချင်ရင် abstract class ကိုသုံးပါတယ် မဟုတ်ရင် both direction ခေါ်ချင်ရင်တော့ normal class ကိုသုံးပြီး model လုပ်လို့.ရပါတယ်။

Inheritance as incremental program modification

Inheritance as incremenatal program modification ဆိုတာ base class ရဲ. source code ကိုမပြင်ပဲနဲ. သူ.ကို implementation အရ ပြင်ချင်တဲ့အခါမှာ extend လုပ်ပြီးသုံးတာမျိုးပါ။ ဥပမာ Java မှာဆို window frame တခုဆောက်ချင်ရင် JFrame ကို extend လုပ်တာမျိုးပါပဲ။ ဒါကတော့ implementation အတွက်သုံးတာပါ။ တခါတလေမှာကိုပြင်ချင်တဲ့ source code ကိုမရနိုင်လို. binary class file ပဲရတဲ့အတွက် သူ. functionality ကို reuse လုပ်ချင်တာဖြစ်ဖြစ် modify လုပ်ချင်တာဖြစ်ဖြစ်သုံးတာမျိုးကိုလဲ ဒီနည်းထဲမှာအကျုံးဝင်ပါတယ်။ Modification ကိုပေးတဲ့နေရာမှာ language တခုနဲ.တခုမတူပါဘူး။ Inheritance breaks Encapsulation ဆိုတာလဲရှိပါတယ်။ ဘာကိုဆိုချင်တာလဲဆိုတော့ inheritance ကိုသုံးတာလွဲခဲ့ရင် သူဟာ parent classes ရဲ. function တွေကို တလွဲသုံးမိရင် encapsulation ဆိုတာကို ချိုးဖောက်တာဖြစ်နိုင်ပါတယ်။ သတိထားရမှာက parent classes ရဲ. encapsulation ကိုမထိအောင်သုံးရမှာပါ။ Favour inhertiance over composition ဆိုတာလဲရှိပါသေးတယ်။ တချို့က code reuse လုပ်ချင်ယုံသက်သက်နဲ. inheritance ကိုသုံးကြပါတယ် ။ taxonomy အရ မတူရင်သော်လည်းကောင်း modifcation or added functionality မထဲ့နိုင်ရင်သော်လည်းကောင်း အဲ့လိုသုံးတာမှားပါတယ်။ Code reuse လုပ်ချင်ရင် composition ကိုသုံးပါ။ ဘာလို.လဲဆိုတော့ inheritance hierarchicy များလာတာနဲ.အမျှ classes တွေဟာ dependency များလာပါတယ်။ Base class တခုကို ပြောင်းလိုက်တာနဲ. အောက်က child classes တွေမှာပါ effect ဖြစ်နိုင်ပါတယ်။ Composition ဆိုတာက ကိုသုံးလို.တဲ့ class ကို reference variable သုံးပြီး ယူသုံးတာပါပဲ။

Single Inheritance vs Multiple Inheritance

C++ မှာတော့ multiple inheritance ကို support လုပ်ပါတယ်။ Class တခုဟာ တခုထက်ပိုသော Base class တွေရှိနိုင်ပါတယ်။ Java,C# တို့.မှာတော့ single inheritance ပဲခွင့်ပေးထားပါတယ်။

Class Inheritance Versus Prototype Inheritance

Programming language တွေဟာ inheritance ကို support လုပ်တဲ့ပုံစံချင်းမတူပါဘူး။ အဓိက ကွဲပြားတဲ့ပုံစံ ၂ခုက classical inheritance နဲ့. prototypical inheritance ပဲဖြစ်ကြပါတယ်။

Classical Inheritance

Java,C++,C# တို့.လို language တွေမှာ support လုပ်တဲ့ inheritance ပုံစံကို classical inheritance လို့.သုံးပါတယ်။ ဥပမာ အောက်က Java program ဆိုပါစို့.

```
class Base {
 int baseData;
 }
class Child extends Base {
int childData;
 }
```

Child object တိုင်းမှာ base class ကရတဲ့ baseData နဲ့. child class ရဲ့. childData ဆိုပြီး property 2 ခုရမှာပါ။ Child object တခုဆောက်တိုင်း သီးခြား baseData တခုစီရနေမှာပါ။ Parent က property တွေကို child object အတွက် seperate copy ပေးထားတဲ့သဘောပါ။ Inheritance with copy semantic လို့.ပြောလို့.ရပါတယ်။ ဒါဟာ classical inheritance ပါ။

Prototypical Inheritance

JavaScript မှာပေးထားတဲ့ inheritance model က prorotypical inheritance ပါ။

```
function Base()
```

```
{
this.baseData = [];
}
function Child()
{
this.childData = "childdata";
}
Child.prototype = new Base();//set up inheritance

var c1 = new Child();
();
var c2 = new Child();
();
c1.baseData.push(100);
.baseData.push(100);
console.log(.log(c2.baseData);
```

အပေါ်က program မှာ Child.prototype = new Base(); ဆိုတဲ့ statement သုံးပြီး prototype chain ကို setup လုပ်ပါတယ် meaning ကတော့ child ရဲ. parent သည် new Base() (Base object) တခုဖြစ်ပါတယ် ဆိုတာပါပဲ။ Classical inheritance မှာ parent သည် class တခုဖြစ်ပေမဲ့ prototypcial inheritance မှာတော့ parent သည် object တခုဖြစ်ပါတယ်။ အဲ့ဒီအပြင် child object c1 နဲ. c2 ဟာ parent object တခုတည်းကို share လုပ်သုံးရပါတယ်။ အောက်ဆုံးအကြောင်းမှာထုတ်ထားတဲ့ c2.baseData ဆိုရင် [100] လို.တွေ.ရမှာပါ။ တကယ် change လိုက်တာက c1 ရဲ. baseData ကို ပြောင်းလိုက်တာပါ။ ဒါပေမဲ့ c1 ရော c2 ရောက same parent object တခုတည်းကို share သုံးရတဲ့အတွက် c2.baseData ဆိုရင်လဲ [100] လို.ထွက်ပါတယ်။ အောက်က statement လေးကို တချက်ကြည့်ပါ

```
c1.baseData.push(100);
```

ဒါက c1 ဟာ parent ရဲ. baseData ကို read access လုပ်တာပါ။ JavaScript ရဲ. inheritance model ကနည်းနည်းရှုပ်ပါတယ်။

တကယ်လို.များ

```
c1.baseData = [200];
```

လို.ရေးလိုက်ရင် c1 မှာ သူ.ရဲ.ကိုယ်ပိုင် baseData ဆိုတဲ့ attribute ကို JS ကထဲ့ပေးတော့မှာပါ။ Attribute တွေကို JS မှာရှာတဲ့ပုံစံက read ဆိုရင် current object ကနေ ရှာပါတယ်။ နောက်မတွေ့.ရင် သူ. prototype chain ကိုလိုက်ရှာပါတယ်။ တကယ်လို. write သာလုပ်မယ်ဆိုရင် parent မှာရှိနေလဲ သူ.current object မှာမရှိရင် attribute အသစ်အနေနဲ.ထဲ့ပေးမှာပါ။ ဒါကိုသတိထားစေချင်ပါတယ်။ Prototypical inheritance သည် share semantic ဖြစ်ပါတယ်။ Parent object ထဲကို attribute တွေ ထပ်ထဲ့တာ နှုတ်တာ အားလုံးသည် child အားလုံးမှာ effect ဖြစ်ပါတယ်။

Dynamic Inheritance

Prototypical inheritance ပေးတဲ့ language တွေမှာ parent object ကို dynamically ပြောင်းလို.ရပါတယ်။ ဒါကို dynamic inheritance လို.ခေါ်ပါတယ်။ state အပြောင်းအလဲပေါ်မူတည်ပြီး လုပ်ရတဲ့ code တွေမှာဆို Dynamic Inheritance ဟာ အသုံးဝင်ပါတယ်။

Object Orient Design Pattern Part 3

Polymorphism

Polymorphism ဆိုတာ Greek ဘာသာစကား Poly (များစွာသော) Morph(ပုံသဏ္ဌာန်ပြောင်းခြင်း) ဆိုတာကနေ ဆင်းသက်လာတာပါ။

Polymorphism is the ability to present the same interface for differing underlying forms (data types).

တူညီတဲ့ interface (public method or method) ပေါ်မှာ contextual object သို့.မဟုတ် data type အပေါ်မူတည်ပြီး ပြောင်းလဲနိုင်မှု.ကိုပြောတာပါ။ မြန်မာလို ရိုးရိုးရှင်းရှင်းပြောရမယ်ဆိုရင် Human လူဆိုရင် work ဆိုတဲ့ method ရှိပါတယ်။ Teacher, Doctor တွေဟာလဲ kind of Human ဖြစ်တဲ့အတွက် work ဆိုတဲ့ method ရှိပါတယ်။ ဒါပေမဲ့ Teacher က work ဆိုရင်တော့ စာသင်မှာဖြစ်ပြီးတော့ Doctor ရဲ. work ဆိုရင်တော့ ဆေးကုမှာပါ။ work ဆိုတဲ့ တူညီတဲ့ interface ပေါ်မှာ method call (message passing) လုပ်တာပါပဲ။ ဒါပေမဲ့ အခေါ် ခံရတဲ့ object အလိုက်မူတည်ပြီး work ရဲ. implementation ကွာမှာပါ။ အောက်က Java Code ကိုကြည့်ပါ။

```
class Human
{
public void work()
{}
}

class Teacher extends Human
{
public void work()
{
System.out.println("I teach tutorial");
}
```

```
}

class Doctor extends Human
{
public void work()
{
System.out.println("I give medical treatment");
}
}
class Test
{
public static void main(String args[])
{
Human h = new Teacher();
h.work();

h = new Doctor();
h.work();
}
}
```

Dynamic Polymorphism

အပေါ်က code မှာ Teacher ရော Doctor ၂ခုစလုံးက Human ကို extends လုပ်ပြီးတော့ work method ကို override လုပ်ထားပါတယ်။ အောက်က Test class ရဲ. main မှာ ပထမဆုံး Teacher ကိုဆောက်ပြီး Human အမျိုးအစားဖြစ်တဲ့ h ထဲကိုထဲ့ပါတယ်။ h သည် base class reference ဖြစ်တာကြောင့် child object (Teacher) ကိုထဲ့ခွင့်ရပါတယ်။ ဒါက sub typing သဘောအရခွင့်ပေးတာပါ။ Parent reference type ထဲကို child object တွေထဲ့ခွင့်ရှိပါတယ်။ ပြီးတော့ h.work() လို. method ကိုခေါ်ပါတယ်။ ဒါပေမဲ့ h.work() သည် Teacher object ရဲ. work

ကိုလှမ်းခေါ်မှာဖြစ်ပါတယ်။ ဘာလို.လဲဆိုတော့ java method တွေက static မကြေငြာထားရင် auto virtual ဖြစ်လို.ပါ။ Human ရဲ. work ကိုခေါ် မဲ့အစား h ထဲမှာတကယ်ရှိတဲ့ object Teacher ကိုခေါ်ပါလိမ့်မယ်။ဒါဟာ polymorphism ပါပဲ။ နောက်တခါ h ထဲကို Doctor ထဲ့ထားပြီး h.work လို.ခေါ်မယ်ဆိုရင်တော့ doctor class ရဲ. work ကိုခေါ်ပါလိမ့်မယ်။ ကျွန်တော်တို. ခေါ်တာ h.work ပါပဲ (same method) ပါ။ ဒါပေမဲ့ implementation (execute လုပ်မှာသည် Teacher ရဲ. work လား Doctor work လား)ကိုတော့ h ထဲမှာ ရောက်နေတဲ့ object ပေါ်မူတည်ပြီး ဆုံးဖြတ်မှာပါ။ ဒါကြောင့် Polymorphism သည် same method with different implementation လို.လဲဆိုကြပါတယ်။ တစ်ခုသတိထားရမှာက static language (Java,C#,C++)မှာ polymorphism သည် virtual method မှာပဲအလုပ်လုပ်ပါတယ်။ Java မှာတောဘာမှမရေးရပေမဲ့ C++ မှာ virtual လို.ရေးရပြီး pointer နဲ.သုံးရပါတယ်။ C# မှာတော့ parent class method မှာ virtual လို.ရေးရပြီး child class method မှာတော့ override ဆိုတဲ့ keyword ကိုထဲ့ပေးရပါတယ်။

ဘာကြောင့် polymorphism ကိုသုံးရတာလဲ သူ.ရဲ. ကောင်းကျိုးတွေက ဘာတွေလဲလို.မေးစရာရှိပါတယ်။ Polymorphism သုံးခြင်းအားဖြင့် hard code ရေးရတာ ဥပမာ object က Teacher ဆို teacher work ကိုခေါ်ပါဆိုပြီး လိုက်ရေးစရာမလိုပါဘူး။ ဒါကဘာကောင်းလဲဆိုတော့ နောင်တချိန်မှာ တခြား Engineer လို class တခု ထပ်ထည့်မယ်ဆိုရှိပြီးသား code တွေ ထိစရာမလိုပဲနဲ. ပြင်နိုင်ပါတယ်။ Extensibility ကောင်းတယ် လို.ပြောရမှာပါ။ ဒီပြင့်နေရာတွေလဲရှိပါသေးတယ်။ ဥပမာ ပြရရင် Procedural language နဲ.ယှဥ်ပေးရမှာပါ။ PHP စပေါ်ကာစက OOP ကို support မလုပ်သေးပါဘူး ။ အဲ့အချိန်မှာ database ကို connect လုပ်တဲ့ code တွေကို function တွေအနေနဲ.ရေးရပါတယ်။ ဥပမာ MySQL ကိုချိတ်မယ်ဆိုရင် mysql_connect ဆိုတဲ့ function ကိုသုံးရပါတယ်။ Oracle ကို

connect လုပ်ချင်ရင်တော့ oci_connect ဆိုတဲ့ function ကိုသုံးရပါတယ်။ တခြား sql statment တွေ execute လုပ်ချင်ရင်လဲ သက်ဆိုင်ရာ mysql oracle function တွေလိုက်သုံးရပါတယ်။ တခြား datsbase server တွေဆိုလဲ သူတို. နဲ့ဆိုင်တဲ့ method တွေလိုက်မှတ်ရပါတယ်။ ဒါဟာ programmer တယောက်အတွက် မကောင်းလှပါဘူး။ နောက်ပိုင်းမှာ PHP မှာ OOP ရလာပြီး PDO (PHP Data Object) ဆိုပြီး Polymorphism,Inheritance သုံးပြီး database function တွေကိုပြောင်းရေးလိုက်ပါတယ်။ တကယ်တော့ mysql connect နဲ. oracle connect သည် implementation ပဲကွာတာပါ ဒါကို OO thinking အရ connect ဆိုပြီး ဘုံထုတ် polymorphism အရ oracle PDO ဆိုရင် oracle connection code လှမ်းခေါ် MySQL PDO ဆိုရင် MySQL connection code ကိုလှမ်းခေါ်ပေးရုံပါပဲ။ ဒါဆို database တွေဟာ implementation တွေသာ ကွာချင်ကွာမယ် connection ချိတ်တာ sql statement တွေ run တာကို method တွေအနေနဲ့ဘုံထုတ်လိုက်တော့ Programmer က MySQL ချိတ်နေတာလား Oracle ချိတ်နေတာလား ပြောစရာမလိုပဲသုံးလို့.ရပါပြီ။ Abstraction ပိုင်းအရကြည့်ရင် Programmer က အများကြီး လိုက်မှတ်စရာမလိုတော့ပါဘူး ။ PDO မှာ database server အားလုံးအတွက် လိုမဲ့ public interface တွေထားပြီး database တခုခြင်းဆီအတွက် different implemnetation ကို extends လုပ်ပြီးပေးလိုက်ယုံပါပဲ။ တကယ်လို. Programmer က MySQL ကနေ Oracle ကိုပြောင်းမယ်ဆိုရင် PDO မှာလွယ်ပါတယ် connection string ပြောင်းယုံပါပဲ။ နဂို procedural programming style အရသာဆိုရင် database code အကုန်ကိုပြောင်းရမှာပါ။ နောက်တခါ database server အသစ်တခု ထပ်ထွက်တယ်ဆိုပါစို့.ဒါဆိုရင်လဲ ရှိတဲ့ PDO class ကို inherit သုံးပြီး different implementation ပေးလိုက်ယုံပါပဲ။ ဒါဆို polymorphism ရဲ့.အသုံးဝင်ပုံကိုသိလောက်ပါပြီ။ Java , C# တို့.မှာလဲ ဒီလိုသဘောတရားသုံးပြီး JDBC , ADO.NET API တွေကိုဆောက်ထားတာပါ။ ခုကျွန်တော်ပြောသွားတဲ့ Polymorhpism အမျိုးအစားကို

dynamic polymorphism လို.ခေါ်ပါတယ်။ ဘာလို.လဲဆိုတော့ ဘယ် method ရဲ. code (implementation) run မယ်ဆိုတာကို run time (dynamic) ရောက်မှ ဆုံးဖြတ်လို.ပါပဲ။

Subtyping vs Duck Typing

Dynamic polymorphism ကိုလုပ်မယ်ဆိုရင် static language (C++,Java,C#) တို.မှာ parent type (super type) reference ထဲကို child type(sub type) object တွေထဲ့ရပါတယ် ။ ပြီးတော့မှ parent type ရဲ. reference ကနေ method ကိုခေါ်ရပါတယ်။ Static language တွေမှာ dynamic polymorphism ကိုပုံဖော်ချင်ရင် အနည်းဆုံး subtyping ဖြစ်အောင်လုပ်ရပါတယ်။ Inheritance နဲ. interface inheritance(Java မှာတော့ interface ကို implements လို.သုံးတာပါ) နည်းနဲ. subtyping ကိုလုပ်လို.ရပါတယ်။ Dynamic language တွေမှာတော့ type တွေ သည် dynanmic (variable တခုရဲ. type သည် ပုံသေမဟုတ် run time မှာ ပြောင်းလဲနိုင်သည်) ဖြစ်တဲ့အတွက် subyting မလိုပါဘူး။ ဒါကို ကျတော့ duck typing လို.ခေါ်ပါတယ်။ ဥပမာ static language တွေမှာ object တခုမှာ work ဆိုတာရှိလဲ subtyping (type အရ assignable)ဖြစ်မှသာခေါ်လို.ရမှာပါ။ Dynamic language မှာတော့ method ရှိတာနဲ.တင်ခေါ်လို.ရပါပြီ။ ခေါ်တဲ့ object သည် ဘာ type ဖြစ်ရမယ် ဆိုတဲ့ ကန်.သတ်ချက်မရှိပါဘူး။ Duck typing လို.ဘာလို.ခေါ်သလဲဆိုတော့ ဘဲလို အော်တယ် ဘဲလိုသွားရင် ဘဲ ပဲပေါ့တဲ့။ Object တခုမှာ ကိုခေါ်တဲ့ method ရှိရင်ရပြီ ဘာ type rule မှရှိစရာမလိုဘူးလို.ဆိုချင်တာပါ။

Static Polymorphism

Static polymorphism ဆိုတောကတော့ method overloading ကိုပြောချင်တာပါ။ method တွေကို နာမည်တူရမယ် ၊ return type (sub type ဆိုလဲရ) တူရမယ် ဒါပေမဲ့ protocol မတူပဲ

ရေးရင် ဒါကို method overloading လို.ခေါ်ပါတယ်။ Method protocol ဆိုတာ ဒီနေရာမှာ method တခုရဲ. parameter အရေအတွက် ၊ parameter type ၊ parameter order အားလုံးကိုပြောတာပါ။ static polymorphism လို.ခေါ်ရခြင်းကတော့ method overloading မှာ ဘယ် metod ကိုခေါ်မယ်ဆိုတာကို compile time မှာ ဆုံးဖြတ်လို.ပါ။ Dynamic language တွေဖြစ်တဲ့ Ruby, Python, JavaScript နဲ. PHP တို.မှာ method overloadig မရှိပါဘူး။

Parametric Polymorphism

နောက်ဆုံးတခုကတော့ Parametric polymorphism ပါသူကတော့ C++ မှာဆို template, Java, C# မှာဆိုရင် generic လို.ခေါ်ပါတယ်။ Parametric polymorphism သည် dynamic language တွေမှာမရှိပါဘူး။ Static language တွေမှာပဲရှိတာပါ။ ဥပမာ Stack တို. LinkList တို.ဆိုတာ ဘုံသုံးပါ integer တွေထဲ့မဲ့ stack ရှိနိုင်သလို string တွေထဲ့မဲ့ stack လဲရှိနိုင်ပါတယ်။ Integer အတွက် stack တခု string အတွက် stack တခုရေးမယ်ဆိုရင် code တွေဖောင်းပွကုန်ပါတယ်။ ဒါကြောင့် template, generic code တွေရေးပြီး တကယ်သုံးတော့မှ data type ကို parameter အနေနဲ.ပို.လိုက်တာပါ။ အဲ့တော့ compiler, runtime system ကနေပြီး ဆိုင်ရာ stack (integer ပေးလိုက်ရင် integer stack ပေါ့ဗျာ) ထုတ်ပေးပါတယ်။ အဲ့တော့ code သည် reusable ဖြစ်တယ်။ type safe ဖြစ်တယ်ပေါ့ဗျာ။

Introduction to Object Oriented Principle

Introduction to Object Oriented Principle

Introduction

OO Program တွေရေးတဲ့အခါမှာ Object Orientation အကြောင်းကိုနားလည်ဖို့.လိုပါတယ်။ Good quality ရှိတဲ့ OO program တွေဖြစ်ဖို့.ဆိုရင် Object Oriented Analysis နဲ့. OO Design အကြောင်းကိုသိဖို့.လိုပါတယ်။ OO ပေးထားတဲ့ language ကိုသုံးနေပေမဲ့ ရှိသမျှ အားလုံး method တစ်ခုထဲမှာ စုပြုံရေး အဲ့ကနေ အကုန် လှမ်းခေါ်ဆိုရင်တော့ OO Language ကိုသုံးသာသုံးနေတယ် မှားနေတယ်လို.ပြောလို.ရပါတယ်။ အဲဒါမျိုးကို writing C program in Java လို.ခေါ်ကြပါတယ်။ OO ရတဲ့ Java လို language မျိုးမှာ C style procedural programming ရေးနေသလို ရေးတာလို.ပြောချင်တာပါ။ Good OO program တွေဖြစ်ဖို့.အတွက် basic OO term တွေ concept တွေကို သင့်တော်သလို အသုံးပြုတတ်ရပါလိမ့်မယ်။ ဥပမာ encapsulation ကဘာလို.အသုံးဝင်တာလဲ ဘယ်နေရာမှာသုံးရမှာလဲ ဘယ်လိုဆိုရင် encapsulation ကို ချိုးဖောက်သလဲ။ ဆိုချင်တာက Java နဲ့. runnable program တစ်ခု C# နဲ့. runnable program တစ်ခု ထွက်လာတာနဲ့. ဒါကို program ရေးနိုင်ပြီလို. သတ်မှတ်တာ တော်တော်လွဲတာပါ။ Program ရေးတတ်ပြီဆိုတာထက် bad program ရေးတတ်ပြီလို.ပဲပြောရမှာပါ။ OO programming မှာဆိုရင်တော့ နည်းနည်းလေး syntax လောက်ကို နားလည်သွားရင် Design Pattern လိုကောင်မျိုးကို လေ့လာလေ့ရှိပါတယ်။ ဒီနေရာမှာ ဘာပြ ဿ နာရှိလဲဆိုတော့ Design Pattern တွေကို သာသွားလေ့လာတယ် OO အခြေခံ သဘောတရားတွေ ကို နားမလည်ပဲ သွားလေ့လာရင် အူတူတူနဲ့. လိုရင်းမရောက်ပဲ သူများက သုံးဆိုလို.သာသုံးတယ် ဘာကောင်းမှန်းမသိ copy paste ပုံစံနဲ့.ကို လွတ်မှာမဟုတ်ပါဘူး။

Design Pattern တွေကို မလေ့လာခင်ဘာသိသင့်သလဲဆိုတော့ OO basic principle တွေကို သိသင့်ပါတယ်။ Design Principle ဆိုတာဘာလဲဆိုတော့ လိုက်နာသင့်တဲ့ ဆောင်ရန် ရှောင်ရန် နည်းလမ်းတွေ concept တွေလို.ပဲပြောရမှာပါပဲ။ ဒီအခြေခံ principle တွေ အပေါ်မူတည်ပြီးတော့ design pattern လို more mature design template တွေ ထုတ်ထားတာပါ။ တကယ်တမ်း basic OO design principle တွေကို နားလည်ရင် design pattern တွေ မသုံးလဲ ကောင်းမွန်တဲ့ OO program တခု ရေးနိုင်မှာပါ။

OO Principle ဆိုပေမဲ့ တချို. principle တွေက OO မဟုတ်တဲ့ အခြား procedural language တွေမှာ apply လုပ်မယ်ဆိုရင်လဲ အသုံးဝင်တဲ့ principle တွေလဲရှိပါတယ်။ ဥပမာ DRY (Don't Repeat Yourself) ဒါဆို OO Principle မဟုတ်ပါဘူး။ သူ.သဘောတရားက code တခုဟာ ရေးပြီးသား tested လုပ်ပြီးသားရှိရင် အချိန်ကုန်ခံပြီးရေး မနေနဲ့. reuse လုပ်လို.ပြောတာပါ။ အဲ့လို Principle မျိုးဆိုရင် OO တင်မကဘူး ကျန်တဲ့ programming paradigm တွေမှာပါ သုံးလို.ရပါတယ်။

Coupling and Cohesion

Principle တွေအကြောင်းမရှင်းခင်မှာ သူတို.ရဲ. basic term တွေကို သိအောင်အရင်ပြောရပါလိမ့်မယ်။ OO Program သို.မဟုတ် တခြား paradigm နဲ.ရေးထားတဲ့ code တိုင်းမှာ coupling နဲ. cohesion ဆိုတာရှိပါတယ်။ Coupling program code တခု သို.မဟုတ် class တခုဟာ သူ.ရဲ. responsibility လုပ်ဖို.အတွက် တခြား ဘယ် class ဘယ် code တွေကို မှီခိုရသလဲ။ မရှိမဖြစ်မှီခိုနေရသလဲဆိုတာကို ပြောတာပါ။ Class တခုရဲ. Responsibility ဆိုတာ အဲ့ဒီ class က လုပ်ဆောင်ပေးနိုင်တဲ့ public method (behaviour) ကိုပြောတာပါ။ ဆိုချင်တာက class တခုက သူ.အလုပ်ကို သူလုပ်နိုင်ဖို. အတွက် တခြား ဘယ် class တွေကို မှီခိုနေရသလဲ။

ဒါကို coupling လို.ခေါ်ပါတယ်။ အဲ့တော့ အမှီအခို နည်းလေ ကောင်းလေပါပဲ။ ဘာလို.လဲဆိုတော့ class တခုက သူ. responsibility အတွက် တခြား class 3 ခုလောက်ကို မှီခိုရပြီဆိုပါစို့ ။ သူ couple ဖြစ်နေတဲ့ class 3 ခုတခုခု public interface တွေသာ change လုပ်ရင် သူလဲ လိုက်ပြောင်းရပါလိမ့်မယ်။ ဒီတော့ coupling ဟာ နည်းလေ ကောင်းလေပါပဲ။ coupling ကို လုံးဝမရှိအောင်တော့ လျော့ဖို့.မလွယ်ပါဘူး။ ဘာလို.လဲဆိုတော့ software ဆိုတာ individual piece of code တွေ တခုနဲ့.တခု ဆက်နွယ်ပြီး လုပ်နေရလို.ပါ။ ဒါပေမဲ့တတ်နိုင်သမျှ နည်းအောင် လုပ်ရမှာပါ။ Coupling ကိုဆက်ခွဲရင် Strong Coupling နဲ့. loose coupling ဆိုပြီးတွေ.ရပါလိမ့်မယ်။

Strong Coupling vs loose Coupling

Strong Coupling ဆိုတာ class တခုသည် တခြား class တခုရဲ. public interface(public method) မဟုတ်ပဲ သူ.ရဲ. field တွေ. variable တွေကို တိုက်ရိုက်ယူသုံးမယ်ဆိုရင် ဒါ Strong coupling ပဲ။ OO သဘောတရားအရ encapsulation ကို ချိုးဖောက်တာပါ။ သူဟာ သူများ class ရဲ. private variable ဒါမှမဟုတ် non public variable တွေကို တိုက်ရိုက်သုံးနေပြီဆိုရင် တကယ်လို. သူသုံးတဲ့ class က အဲ့ variable ကိုပြောင်းမယ်ဆိုရင် မှီခိုနေတဲ့ class ပါလိုက်ပြောင်းရမှာပါ။ ဒါကို strong coupling လို.သုံးပါတယ်။

Loose coupling ဆိုတာက တော့ class တခုဟာ တခြား class တခုရဲ. public interface(public method)တွေကို မှီခိုနေရတယ် ယူသုံးနေရတယ်ဆိုရင် ဒါ loose coupling ပါ။ loosely couple ဖြစ်တဲ့ program တွေဟာ ပိုပြီးတော့ maintainable ဖြစ်ပါတယ်။ Android မှာဆိုရင် message passing လို facility သုံးပြီး broadcast receiver လိုကောင်တွေရှိပါတယ် ။ဥပမာ တခုခု broadcast လုပ်လိုက်ရင် ဆိုင်ရာ app က အဲ့ ကောင်ကို register လုပ်ထားရင် android os ကနေ

ဆိုင်ရာ registered လုပ်ထားတဲ့ activity ကိုခေါ်ပေးမှာပါ။ ဒါက loose coupling ကိုအသုံးချထားတဲ့ပုံစံတခုပါ။

```
class A
{
B b;
void method()
{
b.privateData = “”;
}
}

class B
{
String privateData;
public void doSomething()
{
//Do something on privateData
privateData = “Do something as example”;
}
}
```

အပေါ်က ပြထားတဲ့ code မှာ class A သည် class B ရဲ. privateData ဆိုတာကို method မှာခေါ်သုံးပါတယ် public interface ကနေသုံးတာမဟုတ်တဲ့အတွက် ဒါကို strong coupling လို.ခေါ်ပါတယ်။ ဘယ်လောက် couple ဖြစ်သလဲဆိုရင်တော့ 1 လို.ပြောရမှာပါ။ field တခုတည်းကိုသုံးလို.ပါ။. A သည် B အပေါ်မှာ couple ဖြစ်နေပါတယ် ။ တကယ်လို.များ အကြောင်းတခုခုကြောင့် class B ရဲ. privateData သည် type သို.မဟုတ် variable name ပြောင်းသွားရင် A မှာ လိုက်ပြောင်းရမှာပါ။ class တခုကို change လိုက်လို. နောက် class

တခုကိုပါလိုက်ပြောင်းနေရတယ်ဆိုရင် ဒါ coupling များလို. maintainable code မဟုတ်လို.ပါ။ JavaScript လို dynamic language မျိုးမှာဆို ဒီလို coupling ဟာ ဒုက္ခကောင်းကောင်းပေးပါတယ် ။ တယောက်ယောက်က များ B ရဲ. privateData ကို နာမည်ပြောင်းလိုက်မယ်ဆိုရင် JS ဆိုရင် A မှာယူသုံးထားတဲ့ privateData ကိုအသစ်နေအနဲ. အစားထိုးပေးမှာပါ။ ဒါဆို bug တွေဖြစ်လာပါပြီ။

Cohesion

Cohesion ဆိုတာ class တခုဟာ သူ. public interface (public method)တွေထဲမှာ သူ.ရဲ. data ကို ဘယ်လောက်ယူသုံးလဲ ။သူ.ဟာသူ ကျစ်ကျစ်လစ်လစ် (သူများ class တွေကို ယူ့မသုံးပဲနဲ.) သူ.အထဲမှာပဲ သူ.method တွေကို ဘယ်လောက် use လဲဆိုတာကို တိုင်းတာပါ။ ဆိုလိုချင်တာကတော့ သူများ class ကိုမသုံးပဲနဲ. သူ.ရဲ. responsibility တွေကို သူ.ကိုယ်ပိုင် method တွေသုံးပြီး ဖြေရှင်းထားသလဲ။ အပြန်အလှန်အသုံးပြုသလဲပါ။ သူ. method တခုဟာ သူ.class ရဲ. တခြား method တွေ private data တွေကိုခေါ်လေ ပြင်ပ ကကောင်ကိုမသုံးလေ cohesive ဖြစ်လေပါပဲ။ဘာလို.လဲဆိုတော့ highly cohesive ဖြစ်ပြီဆိုရင် သူများကို မမှီခိုတော့ဘူး အဲ့တော့ maintainable,reusable ဖြစ်ပါတယ်။ Cohesion က များလေကောင်းလေပါ။ တိုင်းမယ်ဆိုရင် Class တခုရဲ. method တခုခြင်းစီဟာ သူ.ရဲ. method body ထဲမှာ သူ.ကိုယ်ပိုင် method တွေ data တွေကို ခေါ်သုံးလေ cohesive ဖြစ်လေပါပဲ။ Class တခုရဲ. method ဟာ သူ.ရဲ. ကိုယ်ပိုင် method ,data တွေကိုမသုံးဘူးဆိုရင်ဒါဟာ cohesion နည်းတာကိုပြတာပါ။ အပေါ်က class မှာ class B သည် cohesive ဖြစ်ပါတယ် ဘာလို.လဲဆိုတော့ သူ. public interface (doSomethingလို.နာမည်ပေးထားတဲ့ method)မှာ သူ. privateData

ကိုပဲသုံးပြီး ဖြေရှင်းလို.ပါ သူများ class ကိုလုံးဝ မမှီခိုလို.ပါ။ ဒီလို class မျိုးဆို highly cohesive class ပါပဲ

OO Principle တွေထဲမှာ အထင်ရှားဆုံးကတော့ SOLID ပါ။ နောက် KISS, GRASP,အပြင်တခြားသော Principle တွေလဲရှိပါသေးတယ်။

Part-2 Don't Do Stupid

လိုက်နာရမဲ့ Principle တွေ မပြောခင်မှာ ရှောင်သင်တဲ့ principle တွေ အရင်ပြောပါမယ် ဒါကတော့ STUPID ပါ။ STUPID အရှည်ကောက်ကတော့ အောက်ပါအတိုင်းပါ

- Singleton
- Tight Coupling
- Untestability
- Premature Optimization
- Indescriptive Naming
- Duplication

Singleton

Singleton design pattern ဆိုတာ GoF(Gang of Four) major design pattern ၂၃ မျိုးထဲမှာ တစ်ခုအပါအဝင်ပါပဲ။ သူ.ကို ဘာအတွက်သုံးလဲဆိုရင်တော့ class တခုကို single instance ပဲလိုချင်ရင် သုံးပါတယ်။ ဆိုချင်တာက class တခုအတွက် object တစ်ခုပဲဆောက်ချင်တာပေါ့။ ဘယ်လိုနေရာမျိုးမှာ အသုံးဝင်လဲဆိုတော့ globally available ဖြစ်တဲ့ တစ်ခုထက်ပိုဆောက်ဖို.မလိုတဲ့ object မျိုးလိုရင် သူ.ကိုသုံးလို.ရပါတယ်။ ဥပမာ သာမာန် small application တွေမှာ DAO connector အတွက် singleton ကိုသုံးလို.ရပါတယ်။ Application တခုလုံးမှာ single object ပဲရှိနေစေချင်ရင် singleton ကိုသုံးပါတယ်။ Java မှာဆိုရင်တော့ java.lang.Runtime class နဲ. java.awt.Desktop တို.ဟာ singleton class တွေပါ။ Runtime class ဆိုတာ ခုလက်ရှိ run နေတဲ့ OS နဲ.ပတ်သတ်ပြီး process တွေ execute လုပ်ချင်ရင်သုံးလို.ရတဲ့ method တွေ တော်တော်များများပါ ပါတယ်။ Runtime object ကိုတစ်ခုထက်ပိုဆောက်ဖို.မလိုပါဘူး ဘာလို.လဲဆိုတော့ OS runtime က တစ်ခုထဲရှိလို.ပါ။ အဲ့လိုနေရာတွေမှာ Singleton ကိုသုံးပေမဲ့လဲ အလွန်အကျွံသုံးမယ်ဆိုရင် singleton ဟာ bad

pattern ဖြစ်လာပါတယ်။ Singleton pattern ရဲ့.မကောင်းတဲ့အချက်တွေက အောက်ပါအတိုင်းဖြစ်ပါတယ်။

Global vairable ပုံစံဖြင့် အသုံးပြုတဲ့အတွက် singleton classes ဟာ application တခုလုံးမှာ နေရာအနှံ့.ပါနေနိုင်ပါတယ်။ ဒါက ဘာကိုဖြစ်စေလဲဆိုတော့ tight coupling between classes ကိုဖြစ်စေနိုင်ပါတယ်။ Singleton object တွေဟာ application အစကနေ အဆုံးထိ lifetime ရှိနိုင်ပါတယ်။ ဒါက performance ကိုကျစေနိုင်ပါတယ်။ နောက်တခုက singleton object တွေဟာ သူတို့.ရဲ့. state (variable)တွေကို persistence state (အသေထားထားတဲ့အတွက်) unit testing code တွေရေးတဲ့အခါမှာ singleton object တွေဟာ ဒုက္ခပေးနိုင်ပါတယ်။

ဒါဆို singleton မသုံးပဲဘယ်လိုလုပ်မလဲ ပေါ့။ ရှင်းပါတယ် new နဲ့. object တစ်ခုအပြင်ကနေဆောက်ပြီးတော့ လိုတဲ့ module ဆီကို dependency injection သုံးပြီး ပို့.လိုက်ပါ။ Singleton ကိုအလွန်အကျွံမဟုတ်ပဲ တော်သင့်ရုံပဲသုံးမယ်ဆိုလဲ ပြ ဿ နာကို အထိုက်အလျောက်ပြေလည်နိုင်ပါလိမ့်မယ်။ Principle ဆိုတာ အတိအကျလိုက်နာရမယ်ဆိုတာ မဟုတ်ပါဘူး။ Rule မဟုတ်ပါဘူး ။

Tight Coupling

အရင်အပိုင်းမှာ Coupling နဲ့. tight coupling အကြောင်းကိုရှင်းထားပြီးပါပြီ။ Coupling ကိုပြန်ပြောရမယ်ဆိုရင် program class တခု module တခုဟာ သူ.ရဲ့. responsibility (behaviour or functionality, set of method) အတွက် တခြား module သို့.မဟုတ် class တွေကို တိုက်ရိုက်ယူသုံးရမယ်ဆိုရင် ဒါဟာ coupled ဖြစ်တာပါပဲ။ Tight coupled ဖြစ်နေတာကို ဥပမာ ပြရရင် ကျွန်တော်တို့. window မှာ run လို့.ရတဲ့ application ကိုသုံးတယ်ဆိုပါစို့. ဒါဆို ကျွန်တော်တို့.ရဲ့. application ဟာ window os အပေါ်မှာ coupled ဖြစ်သွားပါပြီ။ Coupled

ဖြစ်သွားတော့ဘာဖြစ်လဲပေါ့ တကယ်လို.ကျွန်တော်တို.က OS ကိုပြောင်းမယ် upgrade လုပ်မယ်ဆိုရင် အဆင်မသင့်ရင် ကျွန်တော်တို. application ဟာ သုံးလို.မရတာ ဒါမှမဟုတ် အဆင်မပြေတာမျိုးဖြစ်မှာပါ။ ဘာလို.လဲဆိုတော့ ကျွန်တော်တို. application ဟာ window အပေါ်ကို coupled ဖြစ်ထားလို.ပါ။ ဒီသဘောပါပဲ class တခု သို.မဟုတ် module တခုဟာ တခြားတခုကို မှီခိုနေရတယ် coupled ဖြစ်နေရတယ်ဆိုရင် သူမှီခိုရတဲ့ class or module ပြောင်းတိုင်းမှာသူက လိုက်ပြောင်းရမဲ့ဒုက္ခ ရှိပါတယ်။ ဒါက maintenance ကို မကောင်းစေပါဘူး။ နောက်တခုက testing လုပ်တော့မယ်ဆိုရင် အဆင်မပြေပါဘူး။ ဒီနေရာမှာ testing ဆိုတာ unit testing ကိုပြောတာပါ။ unit testing ဆိုတာ program တခုရဲ. smallest functionality ကို code ရေးပြီး test လုပ်တာကိုပြောတာပါ။ ဥပမာ class A သည် class B ကို မှီခိုတယ် coupled ဖြစ်တယ်ပေါ့ ဒါဆိုရင် class A ကို test code ရေးဖို.အတွက် class B ကို ကျွန်တော်တို.လိုပါပြီ ။ အဲ့ဒါကြောင့် testing မှာ အဆင်မပြေဘူးပြောတာပါ။ ဒါဆို coupling မရှိလို.ရလားဆိုတော့မရပါဘူး။ loosely coupled တော့ဖြစ်အောင်လုပ်လို.ရပါတယ်။ ဒါဆို tight coupling ဆိုတာ ဘာကိုလဲဆိုတာထပ်ရှင်းရပါမယ်။ class တခုဟာ တခြား class တခုကို highly dependence ဖြစ်နေတယ်ဆိုရင် ဒါက tight coupling ပါ။ class တခုဟာ တခြား class တခုကို အောက်ပါ အကြောင်းတွေကြောင့် coupled ဖြစ်နိုင်ပါတယ်။

တခြား class ၏ Data ,instance variable များကို တိုက်ရိုက်ယူသုံးခြင်း ၊ dot ခေါက်ပြီး တခြား class ကကောင်တွေကို တိုက်ရိုက်ယူသုံးတာပါ။ တခြား class ၏ public interface (public method or behaviour) ကိုတိုက်ရိုက်ယူသုံးခြင်း တခြား class ၏ object အား new သုံးခြင်းသို.မဟုတ် အခြားနည်းဖြင့် object ဆောက်ခြင်း

အထက်ပါ အချက်တွေကြောင့် coupling ဖြစ်နိုင်ပါတယ်။ ပထမဆုံး အချက်ဆိုရင် tightly coupled ဖြစ်နေပြီလို.ပြောလို.ရပါတယ်။ encapsulation ကိုချိုးဖောက်ရာလဲကျပါတယ်။ Tight coupling က maintenance ကိုရော testing ကိုရော ထိခိုက်နိုင်ပါတယ်။

```
class Message
{
String content;
//other behaviour of Message
public void send()
{
EmailSender sender = new EmailSender();
sender.send(this);
}

}
class EmailSender
{
public void send(Message msg)
{
//Mail sending code
}
}
```

အပေါ်က class 2 ခုကို ကြည့်ပါ။ အဲ့ ၂ခုမှာ class Message သည် EmailSender class ကို tightly coupled ဖြစ်နေပါတယ်။ ဆိုချင်တာက Message class ရဲ. send မှာ EmailSender ကိုယူသုံးထားပါတယ်။ Object ဆောက်တဲ့အပြင် send ဆိုတဲ့ method ကိုပါယူသုံးထားတာပါ။ အဲ့တော့ EmailSender သာ တခုခုပြောင်းရင် ဥပမာ EmailSender ရဲ. constructor ကို no argument မဟုတ်ပဲ တခြား constructor ပြောင်းခဲ့မယ်ဆိုရင် (ဒါက ဥပမာပါ EmailSender ဟာ

တခြားနည်းနဲ့.လဲပြောင်းနိုင်ပါတယ်) Message က ယူသုံးတဲ့ EmailSender ရဲ. constructor နဲ့. send method တွေ သာ တနည်းနည်းနဲ့.ပြောင်းခဲ့မယ်ဆိုရင် ကျွန်တော်တို.ရဲ. Message class ကို Effect ဖြစ်မှာပါ ဒါက Tight coupling ကြောင့်ဖြစ်တဲ့ပြ သာ နာပါ။ နောက်တခုက ကျွန်တော်တို.ရဲ. Message class ကို unit testing code ရေးခဲ့မယ် send method အတွက်ဆိုပါတော့ အဲ့ကောင်အတွက် unit testing code ရေးတော့မယ်ဆိုရင် အဆင်မပြေပါဘူး။ Unit testing ရဲ.သဘောတရားက တခုခုကိုစမ်းတဲ့အခါမှာ သူတို.ချည်း ပဲစမ်းလို.ရရင်အဆင်ပြေပါတယ် ဥပမာ class တခုကိုစမ်းဖို.နောက် class တခုကို လိုမယ်ဆိုရင် error တက်တဲ့အခါ ဘယ် class ကတက်တာလဲဆိုတာ ပြောရခက်ပါတယ်ဒါကြောင့်ပါ။ ဒါကြောင့် coupling ကိုလျော့ချင်ရင် EmailSender object ကို Message send ဆီကိုပို.လိုက်ရင် coupling နည်းသွားမှာပါ။ unit testing code ရေးမယ်ဆိုရင်လဲ ကျွန်တော်တို.က EmailSender mock object တခုပို.လိုက်ယုံပါပဲ။ အောက်က code ဟာ ပိုပြီးတော့ coupling နည်းသွားပါတယ်။

```
class Message
{
String content;
//other behaviour of Message

public void send(EmailSender sender)
{
sender.send(this);
}
}

class EmailSender
{
```

```
public void send(Message msg)
{

//Mail sending code
}
}
```

အပေါ်ကပေးထားတဲ့ example မှာ EmailSender ကို Message ရဲ. send ထဲမှာ new နဲ. object မဆောက်တော့ပဲ parameter အနေနဲ.အပြင်ကနေပေးလိုက်တဲ့အတွက် Dependency Injection သုံးပြီး ထဲ့လို.ရသွားပါပြီ ။ဒါဆို coupling နည်းသွားပါပြီ။ ဒါပေမဲ့ Message သည် concrete class ဖြစ်တဲ့ EmailSender အပေါ်မှာ coupled ဖြစ်နေတုံးပါ။ ဘာပြ သေ နာရှိလဲဆိုတော့ ကျွန်တော်တို.က နောက်ထပ် EmailSender(ဆိုပါစို.ဗျာ code က requirement ကြောင့် gmail နဲ.မပို.တော့ပဲ yahoo mail server သုံးမယ် ဒါမှမဟုတ် email sending API တခုခုပြောင်းသွားတယ်ဆိုရင် ကျွန်တော်တို. code က maintenance အတွက် အဆင်မပြေပါဘူး)ဒါကြောင့် concerte class ကို coupled လုပ်မဲ့အစား EmailSender interface ထုတ်ပြီးအပြင်ကနေ Dependency Injection နဲ.ထဲ့ပေးလိုက်ရင်ပိုအဆင်ပြေမှာပါ။ ဒါဆို Message သည် concrete class EmailSender အပေါ်မှာ မမှီခိုတော့ပဲနဲ. interface EmailSender မှာပဲ copuled ဖြစ်တဲ့အတွက် different implementation ပေးနိုင်မှာပါ အဲ့တော့ code ကပိုပြီး maintainable ဖြစ်မှာပါ။ နောက်ဆုံးပုံစံက ဒီလိုဖြစ်မှာပါ။

```
class Message
{
String content;
//other behaviour of Message
```

```
public void send(EmailSender sender)
{
sender.send(this);
}
}

interface EmailSender
{
public void send(Message msg);

}
class EmailSenderImplOne implements EmailSender
{
public void send(Message msg)
{
//Implementation Code
}
}
```

အပေါ်မှာရေးထားတဲ့ code အရ coupling ကိုလျော့လိုက်တဲ့အတွက် EmailSender ကို EmailSendImplOne အပြင်တခြား နောက်မှာလိုသေးရင် ထပ်ထဲ့လို.လွယ်ပါလိမ့်မယ် ဒါက loosely coupled ဖြစ်သွားလို.ပါ။

Untestability ဒါကတော့ ရှင်းပါတယ် unit test ရေးလို. အဆင်မပြေတဲ့ code ပါ ။နောက်ပိုင်းမှာပိုခေတ်စားလာတာက TDD ပါ(Test Driven Design) ပါ code တွေကို မရေးခင် unit test code တွေရေးပါတယ်။ နောက်မှ code ကိုရေးပါတယ်။ ပြီးရင် testing fail မဖြစ်အောင် code ကို refactored လုပ်ပါတယ်။ ဒီလိုနဲ. code ရဲ. quality ကပိုကောင်းလာပါတယ်။ unit test လုပ်ခြင်းအားဖြင့် ကျွန်တော်တို.ရဲ. code ကိုမှန်မမှန် စစ်ချင်ရင် unit testing code ကို run

လိုက်ရုံပါပဲ။ Major open source software တွေမှာ unit test code က ပါလာပြီးသားပါ။ နောက်တခုက dyanmic language တွေဟာ တကယ်တမ်း runမကြည့်ရင် compile language တွေလိုမဟုတ်တဲ့အတွက် error ကို statically မစစ်နိုင်ပါဘူး။ဒါကြောင့် dynamic language နဲ့.ရေးထားတဲ့ code တော်တော်များများကြီးလာပြီဆိုရင် unit test တွေလိုပါပြီ။ ကျွန်တော်တို့.က class တွေ အများကြီးရှိတဲ့အထဲကို class တခုထဲလိုက်မယ် အဲ့ class အတွက် unit test ကိုရေးမယ် ပြီးရင် class အားလုံးအတွက် unit test ကို run လိုက်လို. error ဖြစ်လာပြီဆိုရင် ကျွန်တော်တို. ရေးထားတဲ့ class က error ဖြစ်တယ်ဆိုတာ သိပါပြီ။ Testable Code ဖြစ်အောင်ရေးဖို့.တော့ TDD ကိုလေ့လာပါလို့.အကြံပြုပါရစေ။ Responsibility တခုထက်ပိုတဲ့ method တွေ long method တွေ tightly coupled သို့.မဟုတ် dependency များတဲ့ class တွေ concrete class ပေါ် မှီခိုတဲ့ကောင်တွေက test လုပ်ရတာခက်ပါတယ်။ အပေါ်က ဥပမာ မှာပြသွားသလိုပါပဲ။

Premature Optimization

Premature Optimization is evil လို. Don Knuth ကပြောသွားပါတယ်။ ဘာကိုဆိုလိုချင်တာလဲဆိုတော့ မလိုအပ်ပဲနဲ. Optimization မလုပ်ပါနဲ့.လို့.ပြောတာပါ။ Optimization ဆိုတာဘာလဲဆိုတော့ Program တခုကို runtime မြန်အောင် သို့.မဟုတ် memory အစားသက်သာအောင် Program ကို ပိုမိုကောင်းမွန်တဲ့ နည်းတွေသို့.မဟုတ် Algorithmတွေသုံးပြီး ပြုပြုင်တာဖြစ်ပါတယ်။ Optimization လုပ်လိုက်ရင် တနည်းနည်းနဲ. code တွေဟာ design ကိုထိခိုက်နိုင်ပါတယ် ဒါကြောင့် မလိုအပ်ပဲနဲ. Optimization ကိုအရင်စလိုမလုပ်သင့်ပါဘူးလို့.ဆိုချင်တာပါ။ 90/10 Rule ဆိုတာရှိပါတယ်။ Program တခုရဲ.

execution time 90 ရာခိုင်နှုန်းက 10 ရာခိုင်နှုန်းလောက်ပဲရှိတဲ့ code တွေ အပေါ်မူတည်တာပါတဲ့ခင်ဗျာ.

Indescriptive Naming

Programmer အများစုက naming ကိုဂရုမစိုက်ကျပါဘူး။ Indescriptive Naming ဆိုတာက ဘာကိုပြောချင်တာလဲဆိုတော့ program မှာ နားလည်ရမလွယ်ကူတဲ့ ဖတ်လိုက်တာနဲ့.မသိတဲ့ variable name တွေ class name တွေ function name တွေ ပေးတာကိုဆိုလိုတာပဲဖြစ်ပါတယ်။ Programmer တွေ အများစုက code ရေးရတဲ့အချိန်ထက် code ကို ဖတ်ရတဲ့အချိန်က ပိုကုန်ပါတယ်။ ဒါကို Project ကြီးကြီးတွေမှာ maintenance လုပ်ရရင်ပိုသိသာပါတယ်။ variable တွေကိုနာမည်ပေးတဲ့နေရာမှာ standard မကျတာတွေ အတိုကောက်တွေ ဘုံနာမည်တွေနဲ့.ဆို အတော်ကိုဖတ်ရခက်ပါတယ်။ ဥပမာ C, PHP library တွေမှာရှိတဲ့ strcmp လိုကောင်မျိုးပါ။ C ခေတ်ကတော့ library အတော်နည်းတဲ့အတွက် သိပ်ပြဿ နာမဟုတ်ပေမဲ့ ဒီဘက်ခေတ်မှာတော့အဆင်မပြေပါဘူး။ class တခုတည်းမှာ ဆင်တူရိုးမှား method နာမည်တွေ ဆယ်ဂဏန်းကျော်ကျော်ပေးထားတဲ့ programmer တယောက်ကို မြင်ဖူးပါတယ်။ သူ. code ကိုသူပဲ maintenance လုပ်ရတော့ ဘယ် method က ဘာဆိုတာ ထိုင်စဉ်းစားရင်းနဲ့. နှစ်ပေါက်အောင် အချိန်ကုန်နေတာကို တွေ့.ဖူးပါတယ်။ နောက်တခုက descriptive name တွေပေးရမဲ့အပြင် method တခုမှာ line ကို ဥပမာ static language တွေမှာဆို line 20 maximum dynamic language တွေမှာဆို 10 line လောက်ပဲရေးသင့်ပါတယ် မဟုတ်ရင် long code တွေဖြစ်လာရင် variable တွေများလာမယ် မှတ်ရတာများလာမယ် နောက်ပြီး control structure if တွေက အစက ဟိုး line 10 လောက်မှာ အပိတ်က line 200 ကျော်လောက်မှာ ဒါမျိုးကို လိုက်ကြည့်ရတာ ဖတ်ရတာ အတော် စိတ်ပျက်စရာကောင်းပါတယ်။ Quality code ဆိုတာ self

documented ဖြစ်ရပါမယ်။ ဥပမာ documentation သို့.မဟုတ် comment ကိုဖတ်စရာမလိုပဲနဲ့. ဒီ program ဟာ ဘာလုပ်လဲဆိုတာ သိလေကောင်းလေပါပဲ literate programming ဆိုတာရှိပါတယ်။ လေ့လာကြည့်စေလိုပါတယ်။ နောက်ပြီး open source software တွေရဲ့. code ရေးသားပုံကိုလည်း လေ့လာစေချင်ပါတယ်။

Duplication

Duplication ကတော့ အလွယ်တကူသိနိုင်မှာပါ။ Code တွေ class တွေကို ဟိုက ဟာကို ဒီကိုကူး ဒီက ဟာကို ဟိုကိုကူးပါ။ Copy paste ကိုသုံးနေရပြီဆိုရင် ဒါဟာ bad programmer ဖြစ်နေပြီဆိုတာကိုသိသင့်ပါတယ်။ Duplicate code တွေဟာ ထပ်နေတဲ့အတွက် တနေရာကို ပြင်မယ်ဆိုရင် နောက်ကူးထားတဲ့နေရာမှာလဲ လိုက်ပြင်ရမှာဖြစ်တဲ့အတွက် maintenance ကိုခက်စေပါတယ်။ နောက် error rate ကိုလဲပိုများစေပါတယ်။ ခုခေတ်က programming က internet သုံး Google ခေါက် stackoverflow ကနေ copy paste လုပ်တော့ပိုဆိုးပါတယ်။ ကိုယ်ပိုင် code မဟုတ်ပဲ သူများ code ကို copy သုံးတဲ့အတွက် side effect ဘာရှိမယ် ဘာကြောင့်သုံးရမယ်ဆိုတာ မသိတော့ပဲ ဒါလေးထဲ့လိုက်တော့ အလုပ်လုပ်တယ်လေ ဆိုပြီး သုံးကြတာမျိုးတွေပါ။ တကယ်တမ်း maintenenace လုပ်တော့ အဲ့လို code တွေက error ဖြစ်ကြပါတယ်။ Duplicated code ဖြစ်ပြီဆိုရင် ဘုံ method classes တွေ ခွဲထုတ်ပြီး method တွေ classes တွေ ယူသုံးတဲ့နည်းနဲ့. refactoring လုပ်ပြီး ရှင်းသင့်ပါတယ်။

Object Oriented Design Pattern Series

Object Oriented Design Pattern Series

Object-Oriented Design Pattern Series Part-1 Singleton Design Pattern

GoF Design Pattern တွေထဲမှာ အရိုးရှင်းဆုံးနဲ့. implement လုပ်ရအလွယ်ဆုံး ကိုပြောရရင် Singleton Design pattern ပါလို့.ပဲပြောရမှာပါ။ Pattern တွေကို သေချာ မှန်မှန်ကန်ကန် အသုံးပြုနိုင်ဖို့.ဆိုရင် ဒီ pattern ကဘယ်လို design problem ကို solve လုပ်ပေးသလဲ။ သူ.ကိုဘယ်အချိန်မှာသုံးသင့်သလဲ။ ကောင်းကျိုး ဆိုးကျိုး ဒါတွေကိုသိရမှာပါ။ Singleton ဆိုရင် နဂိုမူရင်း GoF pattern တွေထဲမှာပါခဲ့ပေမဲ့ နောက်ပိုင်းမှာ ဝေဖန်တာတွေရှိလာပါတယ်။ ဒါကိုသတိထားရမှာပါ။ မတူညီတဲ့ Developer တွေ အဆင်ပြေအောင် Java, JavaScript ၂မျိုးနဲ့. pattern တွေရဲ့. implementation ကို ရေးသွား:မှာပါ။ အဓိက concept ကို နားလည်မိရင် languge does not matter လို့.ပဲဆိုရမှာပါ။ တခြား Language တွေထဲ့ပေးချင်ပေမဲ့လဲ စာလေးသွားမှာစိုးတဲ့အတွက် နားလည်ပေးလိမ့်မယ်ထင်ပါတယ်ခင်ဗျာ။

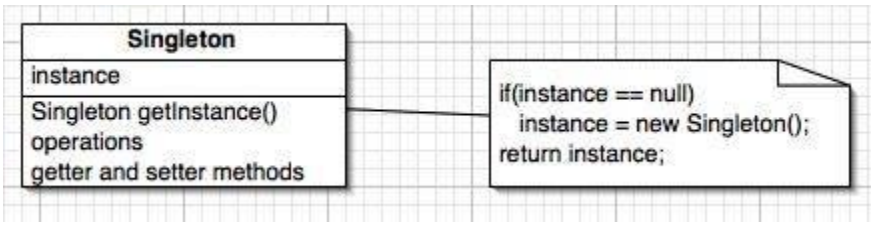
Intent

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 ပဲရရမယ်။ နောက် အဲ့ဒီတခုတည်းသော Object ကို Global access ပေးနိုင်ရမယ်။ Application တခုရဲ့. ကြိုက်တဲ့နေရာကနေ ဒီ object ကို ယူသုံးလို့.ရအောင် ပေးထားရမယ်။ ဒါက singleton ရဲ့. intent ပါ။

Why singleton

Application တခုမှာ class တခုဟာ single instance ပဲရှိရမယ်ဆိုတဲ့ requirement မျိုးမှာ singleton ကိုသုံးပါတယ်။ ဥပမာ hardware resource ကို model လုပ်တဲ့ class မျိုးဆိုပါစို့. CPU

ကို model လုပ်ရင် CPU instance တခုပဲရှိရမယ်ဒါဆိုရင် singleton ကိုသုံးလို.ရပါတယ်။ နောက်တခုက Setting class ဆိုပါစို. object တခုတည်း ရှိသင့်ပါတယ်။ Setting ဆိုဆာမျိုးကို application ရဲ.ဘယ်နေရာကပဲသုံးသုံး ဒီ propoerties တွေ data တွေပဲရသင့်ပါတယ်။ ဒီလိုနေရာမျိုးဆို singleton ကိုပဲသုံးရမှာပါ။ သတိထားရမှာက singleton object ပဲဖြစ်ရမယ် သူ.ကို application ရဲ.နေရာတိုင်းက နေခေါ်လို.ရ ရမယ်ဆိုရင် singleton ကိုသုံးပါ။ နောက်ထင်ရှားတဲ့ singleton example ကိုပြရရင်တော့ Logger ပါ။ Application ရဲ. နေရာ အနှံ.မှာ logger ကိုလိုပါတယ် (Global Access point) ။ နောက်တခုက အဲ့ဒီ logger က တခုတည်းဖြစ်ရမှာပါ။ ဒီ global access point နဲ. single instance ၂ ချက်နဲ.ကိုက်ညီရင် singleton ကိုသုံးလို.ရပါပြီ။



Singleton Structure

အပေါ်ကပုံမှာ Singleton ရဲ. structure ကိုပြထားပါတယ်။ ဒီနေရာမှာ တခု သတိပေးချင်တာက Design Pattern တွေ OO Design တွေလေ့လာတော့မယ်ဆိုရင် UML ကလဲ မသိမဖြစ် လိုပါတယ်ဆိုတာပါ။ အပေါ်က ပုံမှာ Singleton class ကိုပြထားပါတယ် ။နောက် သူ.မှာ getInstance ဆိုတဲ့ method ရှိပါတယ် အဲ့ကောင်က global acces point ကိုပေးထားတာပါ။ public ပေါ့။ သူ.ထဲမှာတော့ single instance ပဲဖြစ်အောင် code ကိုရေးထားရမှာပါ။ အဲ့ဒါကို note ကလေးနဲ.ပြထားပါတယ်။

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။ ပထဆုံး Sigleton ဆောက်မှာဖြစ်တဲ့အတွက် instance တခုတည်းသိမ်းဖို့.လို့.ပါတယ်။ ဒါကြောင့် static Loggger 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 သူ.ကို အခြားသူတွေက ဝင် access လုပ်လို့.မရအောင် private ပေးလိုက်ပါတယ်။ အပြည့်အစုံဆိုရင်တော့.

private static Logger instance;

ဖြစ်သွားမှာပါ။ ဒါက singleton object အတွက် instance တခုတည်းသိမ်းဖို့.လုပ်ထားလိုက်တာပါ။ နောက်ပြီး constructor ကို private ထားပါတယ်။ ဒါမှသာ contructor ကိုတိုက်ရိုက်ယူသုံးပြီး တခြား object တွေ တည်ဆောက်မှု.ကို ကာကွယ်နိုင်မှာပါ။

နောက်ဆုံး point ကတော့ global access ပေးရမယ် single object ပဲထုတ်ပေးရမယ် ဒါကိုတော့ getLogger method မှာရေးထားတာပါ။ getLogger က static ဖြစ်တဲ့အတွက် class level ကနေခေါ်ရမှာပါ။ သူ.က Logger instance ကိုပဲ return ပြန်မှာပါ။ Singleton တွေမှာ constructor

ကို private ပေးထားတဲ့အတွက် getLogger လို golobal access point ကနေပဲခေါ်သုံးလို.ရမှာပါ။ ဒါကြောင့်သူ.ကို public ပေးထားတာပါ။ သူ. method ထဲမှာ instance ကို null နဲ.ညီသလားလို.စစ်ပါတယ်။ ပထမဆုံး ခေါ်တဲ့တခေါက်မှာ instance က null နဲ.ညီမှာပါ။ 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။ ဒီနည်းနဲ. single instance တွေပဲရအောင် ထိန်းထားတာပါ။ တခြား singleton class မှာလိုအပ်တဲ့ public method တွေလဲ သင့်တော်သလိုထဲ့လို.ရပါတယ်။

သူ.ကိုသုံးနည်းကတော့ LoggerDemo class မှာပြထားပါတယ်။ Constructor သည် private အနေနဲ.ထားထားတဲ့အတွက် Logger ကို new နဲ.အပြင်ကနေဆောက်ဖို. မဖြစ်နိုင်ပါဘူး။ ဒါကြောင့် getLogger ကိုပဲသုံးရမှာပါ။ Logger.getLogger() ကို ၂ခါခေါ်ပါတယ်။ singleton ဖြစ်အောင်ထိန်းထားတာကြောင့် ၂ခါလုံးရတဲ့ object ကတူညီမှာပါ။ ဒါကြောင့်အောက်က print statement မှာ true ဆိုပြီးထွက်လာမှာပါ။ ဒီလိုရေးနည်းကို lazy initialization နည်းလို.လဲသုံးပါတယ် ဘာလို.လဲဆိုတော့ singleton object ကို getLogger ကိုမခေါ်မချင်း မဆောက်သေးလို.ပါ။

JavaScript

ဒါကတော့ JavaScript နဲ. 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;
}
};
})(); //inmediately invoke
instance;
}
};
})(); //inmediately invoke
var logger1 = = Logger.getLogger();
.getLogger();
var logger2 = = Logger.getLogger();
console.log( .log( logger1 === === logger2 );
</script>
```

အပေါ်က JS code ကိုနားလည်ဖို. module pattern နဲ့.အရင် ရင်းနှီးရပါမယ်။ အဲ့မှာ Looger ထဲကို IIFE (Immediately invoking function expression)ကိုသုံးပြီး function ကိုတခါတည်း invoke လုပ်ပါတယ်။Annomymous fuction ကိုတခါတည်း invoke လုပ်တာပါ။ အဲ့တော့ Logger သည် တခါတည်း run မှာပါ။ JS မှာ encapsulation ကို colusre ကိုသုံးပြီး ရေးရပါတာယ်။ function ကို run နဲ့.အခါမှာ နောက်ဆုံးမှာ

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

ဒီကောင့်ကိုတွေ့.မှာပါ ဒါက ခုနက run လိုက်တဲ့ annomymous function ကနေ return ပြန်ရမှာပါ။ ဘာကို return ပြန်သလဲဆိုတော့ object တခုကို object literal သုံးပြီး return ပြန်ပါတယ်။ return {} ဆိုတဲ့ပုံစံနဲ့.ပါ။ အဲ့ return ပြန်တဲ့ object ထဲမှာ getInstance ဆိုတဲ့ method ပါပါတယ်။ getLogger: function(){} ပုံစံနဲ့.ရေးထားတာပါ။ အဲ့ဒီ function ထဲမှာမှ if နဲ့. !instance ကိုစစ်ပါတယ် ဆိုချင်တာက Java example မှာလိုပဲ instance သည် null သို့.မဟုတ် zero သို့.မဟုတ် undefined မဟုတ်ဘူးဆိုရင် true ဖြစ်မှာပါ။ annomymous function အစမှာ

var instance ; ဆိုပြီးကြေငြာထားတာတွေ့.မှာပါ။ သူ့.ကို ဘာမှ assign လုပ်မထားတဲ့အတွက် undefined အနေနဲ့.ရမှာပါ။ ဒါဆိုပထမဆုံးခေါ်တဲ့အခေါက်မှာ if statement သည် true ဖြစ်ပြီး

singleton object ကို new Singleton နဲ့.ဆောက်မှာပါ။ ဒီနေရာမှာ Singleton constructor (JavaScript မှာတော့ class က ECMA 6 မှရတာပါ) သည် Annomymous function ရဲ့. scope ထဲမှာပဲရှိတဲ့အတွက် အပြင်ကနေခေါ်လို့.မရပါဘူး။ ထို့.အတူ instance ကိုလဲခေါ်လို့.မရပါဘူး။ function ထဲမှာ ရှိတဲ့ variable တွေကို JavaScript မှာ ခေါ်သုံးချင်ရင် တိုက်ရိုက် access လုပ်လို့.မရပဲနဲ့. closure နဲ့.ပဲခေါ်သုံးလို့.ရမှာပါ။ Closure ဆိုတာကတော့ function တခုထဲမှာ 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 ကိုထဲ့ပေးထားပါတယ်။

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 ကိုချွေတာရာရောက်ပါတယ်။ Criticism အနေနဲ့.ကတော့ Global access point ပေးထားတဲ့အတွက် application တခုလုံးမှာ singleton ဟာ class အရ ကို tight coupling ဖြစ်ပါတယ်။ အဲ့ဒီအတွက် testing ကိုခက်ခဲစေပါတယ်။ ဒီနေရာမှာ ကျွန်တော်က ကောင်းကျိုးဆိုးပြစ်နှစ်ခုလုံးကိုပြောရတာဟာ တာဝန်အရပါ။ Singleton နဲ့.ပတ်သတ်တဲ့ criticism တွေကိုလဲ လေ့လာဖို့.အကြံပြုပါရစေ။

Object-Oriented Design Pattern Series Part-2 Factroy Method Design Pattern

GoF Design Pattern တွေထဲမှာ creational pattern တွေထဲမှာ Object creation နဲ့.ဆိုင်တဲ့ Factory Method pattern အကြောင်း ပြောပါရစေ။ သူနဲ့. ဆက်စပ်နေတာကတော့ Simple Factory pattern , Abstract Factory Pattern တို့.ပါပဲ။ Factory Method design pattern ကတော့ Object တွေကို new နဲ့. ဆောက်ခိုင်းတာမလုပ်ပဲ Factory class တခုရဲ့. method ကိုသုံးပြီး Object creation ကိုထိန်းချုပ်ချင်ရင် သုံးပါတယ်။ ဒါဆို object ကို new နဲ့.ဆောက်တော့ ဘာဖြစ်မှာလဲဆိုပြီး မေးစရာရှိပါတယ်။ Object ကို new နဲ့.ဆောက်သုံးတော့ ကျွန်တော်တို့.က concrete class (abstract class or interface မဟုတ်သော class) ကို တိုက်ရိုက်ယူသုံးတာပါပဲ။ ဘာပြဿနာရှိလဲဆိုတော့ code က concrete class အပေါ်မှာတိုက်ရိုက် dependency ရှိသွားပါပြီ။ အဲဒါမျိုးကို ဘယ်လိုခေါ်လဲဆိုတော့ program to implementation လို့.ခေါ်ပါတယ်။ Concrete class ရဲ့. object တွေနဲ့. application module တွေကို couple လုပ်တာကြောင့်ပါ။ OO principle အရ program to interface, not to implementation ကိုသုံးရမှာပါ(အဲ့ဒီ principle ကို ဒီနေရာမှာ ဖတ်ပါ https://bit.ly/2L7qUao ။ဆိုချင်တာ ကျွန်တော်တို့.က concrete class သုံးမဲ့အစား abstract class တွေ interface တွေအနေနဲ့.သုံးပြီး program to interface ဆိုတဲ့ principle ကိုလိုက်နာပြီးသုံးမှသာ maintainable program တွေဖြစ်မှာပါ။ Loose copuling လဲဖြစ်တော့ အဆင်ပြေတာပေါ့ဗျာ။ (Coupling ကတော့ ဒီမှာရှင်းထားပါတယ် https://bit.ly/39LBcaF .Code example ကို Java, JavaScript, PHP သုံးပြီး ပြပါမယ်။

Intent

ဒါကတော့ Factory Method design pattern ရဲ့. intent ပါ။

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Object ကို ဆောက်ဖို့.အတွက် interface လုပ်ပေးရမယ် (ဒီနေရာမှာ public method or API လို့.ဆိုချင်တာပါ)။ ဘယ် class ရဲ့. object ကိုဆောက်ရမလဲဆိုတာကိုတော့ subclass တွေက ဆုံးဖြတ်မယ်။ သူ.ရဲ့.လိုရင်းက ဘယ် class ရဲ့. Object ဆောက်ရမလဲဆိုတာကို subclasses တွေကို တာဝန်ပေးလိုက်တာပါပဲ။ Object creation ကို လုပ်တဲ့အတွက်သူဟာ Creational Pattern ထဲမှာပါပါတယ်။ Factory Method desing pattern ကို virtual constructor လို့.လဲခေါ်ပါသေးတယ်။ Factory Methodက new နဲ့. object ဆောက်ရင်ဖြစ်လာမဲ့ hard coupling ကို loose copuling ဖြစ်အောင် ပြောင်းရေးတာပဲဖြစ်ပါတယ်။

Why Factory Method Design Pattern?

ဥပမာ တခုပြပါမယ်။ Application တွေ develop လုပ်တဲ့အခါမှာ database ကို access လုပ်ရပါတယ်။ အဲ့ဒီအခါကျရင် database connection တွေ ဆောက်ပြီး database operation တွေကို အဲ့ဒီ connection object ကနေတဆင့်လုပ်ရပါတယ်။ Database server တွေက MySQL, Oracle, MSSQL Server အစရှိသဖြင့် အများကြီးရှိပါတယ်။ ကျွန်တော်တို့.က Database connection object တခုကိုပဲလိုတာပါ။ ဒါပေမဲ့ ကျွန်တော်တို့.လိုတဲ့ Connection object သည် MySQL connection လား Oracle connection လားဆိုတာကို ကျွန်တော်တို့. ကိုယ်တိုင် သိနေရမယ် တိုက်ရိုက်ဆောက်နေရမယ်ဆိုရင် အဆင်မပြေပါဘူး။ ကျွန်တော်တို့.က connection string လေးပဲ ပေးလိုက်မယ် MySQL လဲဖြစ်နိုင်တယ်။ Oracle လဲဖြစ်ရင်ဖြစ်မယ် ဒါပေမဲ့ ဘယ် specific implementation ကိုဆောက်ရမယ်လို့.မပြောဘူး။ အဲ့ဒီ specific object ဆောက်ဖို့.တာဝန်ကိုတော့ Factory Method design pattern သုံးထားတဲ့ class ကိုပေးလိုက်မယ်ဆိုပါဆို. ဒါဆိုကျွန်တော်တို့.က ကျွန်တော်တို့.သုံးနေတာသည် Oracle connection လား MySQL connection လားသိစရာမလိုပဲနဲ့. သုံးလို့.ရပါပြီ။ MySQLConnection, OracleConnection (concrete implmentation)ဆိုတာထက် Connection (abstract

implemenation) ကိုသုံးတာက ပိုပြီး loose couple ဖြစ်ပါတယ်။ ဘာကောင်းလဲဆိုတော့ နောက်ပိုင်း ကျွန်တော်တို့က MySQL သုံးနေရင်းနဲ့. တခြားအကြောင်းကြောင့် Oracle ပြောင်းသုံးတယ်ဆိုပါစို့. ဒါဆို Factory ကိုပို့တဲ့ connection string လေး ပြောင်းပေးလိုက်တာနဲ့. Fatctory Method design pattern က OracleConnection ကို ဆောက်ပေးမှာပါ။ ဒါဆို ကျွန်တော်တို့. code က ပို maintenance ကောင်းသွားတာပေါ့ဗျာ။ မလိုအပ်တဲ့ specific implementation (ဥပမာ Oracle Connection Object ဆောက်တာနဲ့. MySQL Connection ဆောက်တာသည် step အရ တူချင်မှ တူမယ်) ကိုသိစရာမလိုတဲ့အတွက် abstraction အရလဲပိုကောင်းတယ် နောက်တခုက encapsulation လဲပိုကောင်းပါတယ်။

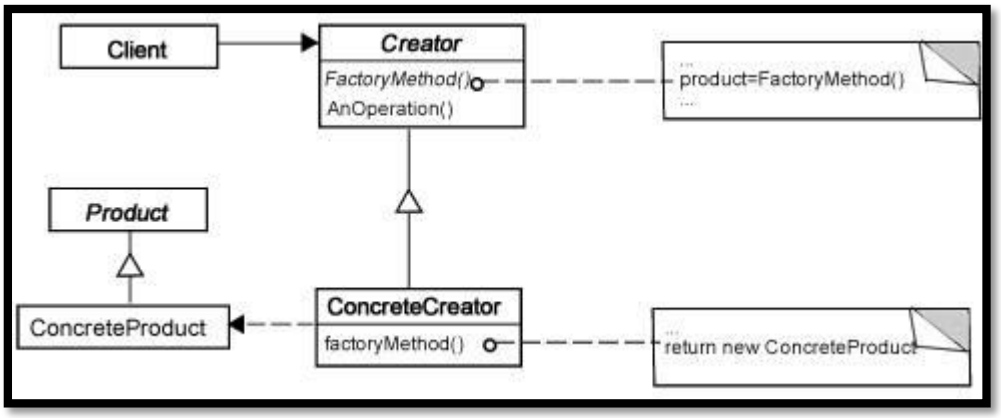
နောက်ဥပမာ တခုပေါ့ ကျွန်တော်တို့.က Logger တွေ သုံးတယ်ပေါ့ဗျာ။ တခါတလေမှာ console မှာရိုက်ပြီး နောက်ကျ text file မှာ သွား log ရိုက်တာလဲ ဖြစ်ချင်ဖြစ်မယ်။ XML file မှာ log သွားရိုက်တာလဲဖြစ်မယ် ဒါဆို Looger ကို inteface or abstract class တခုအနေနဲ့.ထားပြီး ဘယ် Logger ဆောက်ရမယ်ဆိုတာကို (XMLLogger, ConsoleLogger) ကို FactoryMethod design pattern ကိုသုံးလိုက်ရင် client တွေက (Logger ကိုသုံးမဲ့ code တွေက) ဘယ် logger implementation ကိုသုံးနေလဲသိစရာမလိုပဲနဲ့. ကို အလုပ်လုပ်နိုင်ပါတယ်။ ကျွန်တော်တို့. လိုချင်တဲ့ Logger အမျိုးအစားကိုသာ Factory Method design pattern ကိုပြောလိုက်ယုံပါပဲ။

Factory Method Design Pattern ရဲ့. ရည်ရွယ်ချက်ကိုက Object ကိုဘယ်လိုဆောက်ရမယ်ဆိုတာသိစရာမလိုပဲ (how to do)မလိုပဲ ဘာလိုချင်တယ်ဆိုတာ ပြောရုံနဲ့. (what to do) သိရုံနဲ့. Factory Method Design Pattern က လုပ်ပေးမှာပါ။

အောက်ကပုံမှာ Factory Method Pattern ရဲ့. class structure ကိုပြထားပါတယ်။ Client codeက Creator ဆိုတဲ့ Factory ကိုပဲသုံးရမှာပါ။ Creator ကသူကိုယ်တိုင် Object တွေ create မလုပ်ပဲ

သူ. subclass factory တွေကသာ ဘယ် object ကိုဆောက်မယ်ဆိုတာ ဆုံးဖြတ်တာပါ။ ဒီနေရမှာတော့ ConcreteProduct ပေါ့ ဒါပေမဲ့ return ပြန်ရင်တော့ specific instance (ConcreteProduct) ကိုမပြန်ပဲ abstrac class or interface ဖြစ်တဲ့ Product ကိုပဲပြန်ရမှာပါ။ ဒါက Factory Method design pattern ရဲ့ ပုံစံပါပဲ။ Client သည် Creator ကနေ object ကိုတောင်းရုံပါပဲ။

Class Diagram for Factory Method Design Pattern



Implementation Example

Java

```
interface Logger {
   void log(String message);
}
class ConsoleLogger implements Logger {
   public void log(String message) {
      System.out.println("ConsoleLog > " + message);
   }
}
class XMLLogger implements Logger {
   public void log(String message) {
      System.out.println("XMLLog > " + message);
```

```
    }
}
//Here start Factory classes
abstract class LoggerFactory {
    abstract Logger getLogger();
}
class ConsoleFactory extends LoggerFactory {
    @Override
     Logger getLogger() {
        return new ConsoleLogger();
    }
}
class XMLFactory extends LoggerFactory {
    @Override
    Logger getLogger() {
        return new XMLLogger();
    }
}
ublic class FactoryDemo {
    public static void main(String[] args) {
        LoggerFactory factory = new ConsoleFactory();
        factory.getLogger().log("Message 1");
        factory = new XMLFactory();
        factory.getLogger().log("Message 2");
    }
}
```

အပေါ်က code မှာ Logger သည် client ကသုံးမဲ့ Logger ပါ။ ဒီနေရာမှာ ဘယ် Logger ဆိုတဲ့ specific implementation(XMLLogger,ConsoleLogger) ဆိုတာကိုမသုံးပဲ interface Logger

ကနေပဲ operation တွေလုပ်မှာပါ။ အဲ့ဒီတော့ ကြိုက်တဲ့ Logger ကိုအချိန်မရွေးပြောင်းသုံးနိုင်ပါတယ်။ Client code တွေပြင်စရာမလိုပါဘူး။ နောက် LoggerFactory ဆိုတာကတော့ ဒီ Factory Method design pattern ရဲ.အသက်ပါပဲ။ သူ.မှာ abstract Logger getLogger(); ဆိုတဲ့ public interface လေးပေးထားပါတယ်။ သူ.ရဲ. return type က Logger ပါ။ ဘယ် Logger ဆိုတာမပါပါဘူး။ ဘယ် Logger ဆောက်ရမလဲဆိုတဲ့ တာဝန်ကတော့ ConsoleFactory နဲ. XMLFactory တွေရဲ. တာဝန်ပါ။ ဒါက Subclass တွေကိုဘယ် object ဆောက်ရမလဲဆိုတဲ့တာဝန်ကို လွှဲပေးထားတဲ့ Factory Method design pattern ရဲ. main theme ပါပဲ။ နောက် FactoryDemo သည် client code ပါ အဲ့မှာ Factory object တွေဆောက်တယ် Factory object ကနေမှ getLogger ဆိုတာကိုခေါ်သုံးတယ် ။ Factory class ပေါ်မူတည်ပြီး ရလာမဲ့ Logger implementation ကွာသွားမယ်။ ဒါပေမဲ့ client က အဲ့တာကိုသိစရာမလိုဘူး။ ဆိုချင်တာက FactoryDemo class မှာ ConsoleLogger, XMLLogger ဆိုတာကို သုံးကိုမသုံးဘူး အဲ့တော့ loose copuling ဖြစ်တယ်။ Logger ပြောင်းချင်ရင် Factory class ရဲ. object ဆောက်တဲ့နေရာလေး ပြောင်းသုံးရုံပဲ ။ Client code က logging လုပ်တဲ့ code တွေထိစရာမလိုဘူး။

Factory Method design pattern နဲ.ခပ်ဆင်ဆင်တူတဲ့ pattern တခုကတော့ simple Factory pattern ပါသူ.မှာဆိုရင်တော့ Factory အတွက် subclass တွေဆောက်မနေတော့ပဲ။ လိုအပ်တဲ့ object တွေကို method တခုကနေ return ပြန်ပေးလိုက်တာမျိုးပါပဲ။

JavaScript

```
<script>
  function ConsoleLogger() {
    this.log = function (msg) {
```

```
        console.log('Console Log' + msg);
    }
  }
  function XMLLogger() {
    this.log = function (msg) {
        console.log("XMLLog " + msg);
    }
  }
  function LoggerFactory() {
    var loggers = {};
    loggers['console'] = ConsoleLogger;
    loggers['xml'] = XMLLogger;
    this.getLogger = function (type) {
        return new loggers[type]();
    }
  }
  var loggerFactory = new LoggerFactory();
  var logger = loggerFactory.getLogger('console');
  logger.log("Log for console");
  logger = loggerFactory.getLogger("xml");
  console.log("Log for xml");
</script>
```

အပေါ်က JavaScript example က Factory method design pattern ဆိုတာထက် simple factory လို့ပဲသုံးရင် ပိုသင့်တော်မှာပါ။ JavaScript မှာ abstract class တွေ interaface တွေမရှိပါဘူး ဒါကြောင့် class based language တွေဖြစ်တဲ့ Java,C++,C#,PHP တို့နဲ့. implmentation ကကွာသွားပါတယ်။ အဓိက LoggerFactory မှာ object literal တခုထဲကို ConsoleLogger ရယ် XMLLogger ရယ် ထဲ့သိမ်းထားပါတယ်။ ဒီနေရာမှာသာဆို တခြား langauge တွေမှာဆို if

နဲ့.စစ်နေရမှာပါ ။အဲ့ကျ hard coding ဖြစ်နေတဲ့အတွက် JavaScript အနေနဲ့.ရေးမယ်ဆိုရင် object literal ထဲထဲ့ပြီးသုံးတာ pragmatic အရပိုကောင်းပါတယ် JavaScript way ပေါ့ဗျာ။ LoggerFactory မှာ getterLogger မှာ prameter အနေနဲ့. type ကိုလက်ခံတယ် type အပေါ်မူတည်ပြီး object literal loggers ရဲ့. construction function ကို new နဲ့.ဆောက်ပေးလိုက်တယ်။ အဲ့တော့

```
ogger = loggerFactory.getLogger('console');
```

ဒီ code မှာဆို ConsoleLogger instance တခုရမှာဖြစ်ပြီး နောက်

```
logger = loggerFactory.getLogger("xml");
```

သူ.အတွက်ဆိုရင်တော့ XMLLogger object တခုရမှာဖြစ်ပါတယ်။

```
PHP
<?php
interface Logger
{
   function log($message);
}
class ConsoleLogger implements Logger
{
   public function log($message)
   {
      echo("ConsoleLog > " . $message . "<br/>");
   }
}
class XMLLogger implements Logger
{
```

```
    public function log($message)
    {
        echo("XMLLog > " . $message . "<br/>");
    }
}
abstract class LoggerFactory
{
    abstract function getLogger();
}
class ConsoleFactory extends LoggerFactory
{
    public function getLogger()
    {
        return new ConsoleLogger();
    }
}
class XMLFactory extends LoggerFactory
{
    public function getLogger()
    {
        return new XMLLogger();
    }
}
//Client code
$factory = new ConsoleFactory();
$factory->getLogger()->log("Message 1");
$factory = new XMLFactory();
$factory->getLogger()->log("Message 2");
?>
```

PHP ရဲ့ OOP concept တွေက Java ကနေ borrow လုပ်ထားတာဖြစ်တဲ့အတွက် အပေါ်က PHP code ကို Java Example ကိုနားလည်ရင် သူ့.ကိုလဲနားလည်နိုင်မယ်ဖြစ်တဲ့အတွက် မရှင်းတော့ပါဘူးခင်ဗျာ ။

အနှစ်ချုပ်အနေနဲ့. Factory method design pattern ကို

1 Client သည် ဘယ်လို object type တွေ ဆောက်မလဲဆိုတာ ကြိုမသိနိုင်တဲ့အခါ ဒါမှမဟုတ် ပြောင်းရလွယ်ချင်တဲ့အခါ

2 Object creation ဟာ complex ဖြစ်လို့.(ဥပမာ parameter setting တွေအပေါ်မူတည်ပြီး ဘယ် object ဆောက်ရမလဲ ဆိုတာမျိုး) Object creation encapsulate လုပ်ချင်တဲ့အခါ

3 Class တခုနဲ့.တခုကြားက Tight copuling ကိုလျော့ချင်တဲ့အခါ

သုံးလို့.ရပါတယ်။

ကောင်းတဲ့အချက်တွေကတော့ tight coupling ကို လျော့နိုင်မယ် Client က specific implementation တွေထပ်ထဲ့ရင် အဆင်ပြေတယ်။ Code က extensible ဖြစ်မယ် maintenance လုပ်လို့.ကောင်းမယ်ပေါ့။ မကောင်းတာကတော့ Object creation ကို factory တွေက ထိန်းထားတာဖြစ်တဲ့အတွက် unit testing code ရေးတဲ့အခါ mock object ဆောက်ဖို့.အဆင်မပြေတာဖြစ်နိုင်တယ်။

လိုတာလေးတွေထပ်ဖြည့်ပေးဖို့. မေတ္တာရပ်ခံပါရစေ ☺

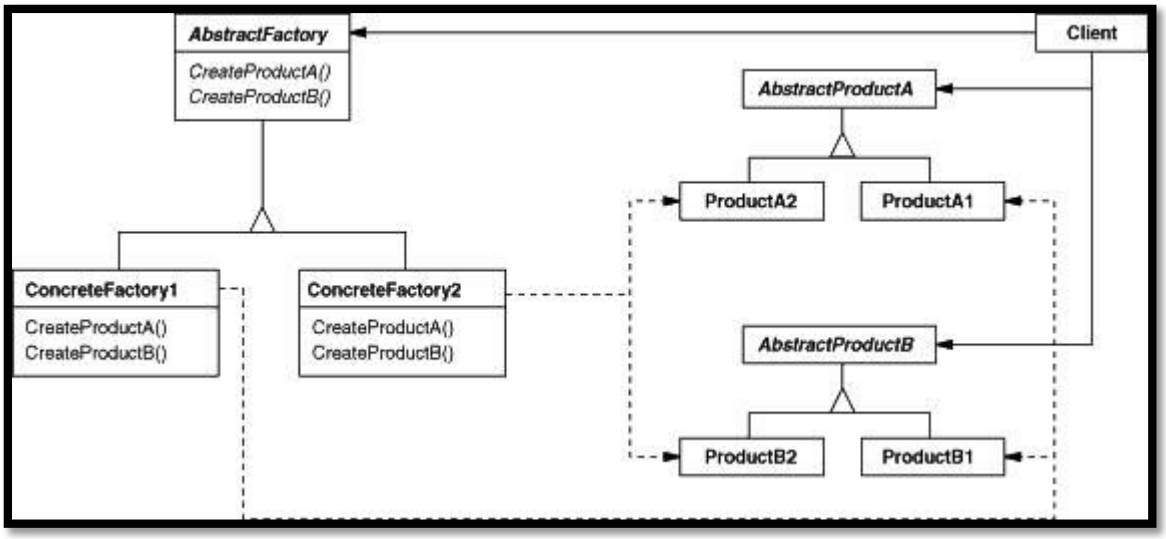
Object-Oriented Design Pattern Series Part-3 Abstract Factory Method Design Pattern

Abstract Factory ဆိုတာကတော့ Creational pattern တွေထဲကတခုပါပဲ။ သူ.နဲ.ဆက်စပ်နေတာကတော့ Factory Method pattern ပါ။ Factory Method ကတော့ Complex Object creation ကို hide လုပ်ချင်ရင်သုံးပါတယ်။ Factory method က class တခုရဲ. object creation ကိုပဲ hide လုပ်တာပါ။ Abstract factory မှာကျတော့ family of product လို.ပြောရမှာပါ။ ဆိုချင်တာက related ဖြစ်နေတဲ့ class တွေရဲ. object creation ကို ထိန်းချင်ရင်သုံးပါတယ်။

Intent

Abstract factory ကို Family of product object တွေ create လုပ်တဲ့အခါမှာသုံးပါတယ်။ Family of product ဆိုတာ အတူတကွသုံးရတဲ့ class တွေကိုဆိုချင်တာပါ။ ဥပမာ Car, Wheel , Engine ဆိုတာ Family of prodct ပါပဲ။ ဒါပေမဲ့ အဲ့ဒီ Car, Wheel ,Engine ဆိုတွေက စက်ရုံပေါ်မူတည်ပြီးကွာနိုင်ပါတယ် စက်ရုံကတော့ Factory ပါပဲ။ ဒီ Family collection of product object တွေကိုသုံးတဲ့အခါမှာ client ကနေပြီး ဘယ် Factory ကထုတ်လိုက်တယ်ဆိုတာကိုသိစရာမလိုပဲနဲ. သုံးနိုင်အောင် ရည်ရွယ်တာပါ။

အောက်မှာ Abstract Factory Design pattern ရဲ. class Diagram ကိုပြထားပါတယ်။



UML Diagram အရ ကျွန်တော်တို့.က Family of product တွေကို (ProductA2,ProductA1) အစရှိတာတွေကို ရဲ. creation ကို ထိန်းချုပ်တာပါ။ Client ကသုံးမှာက AbstractProductA, AbstractProductB ကိုပဲသုံးမှာပါ။ ဒီနေရာမှာ object တွေက ဘယ် factory ကထုတ်တယ် ဆောက်ထားတယ်ဆိုတာကို Client ကသိစရာမလိုပါဘူး ။ Implementation ကို hide လုပ်ထားတာပါ။

Why Factory Method Design Pattern?

Abstract factory နဲ့.ပတ်သတ်ပြီးပေးလေ့ရှိတာကတော့ programming langauge တွေမှာသုံးတဲ့ GUI control တွေပါပဲ။ Control တွေမှာ Button, Label, အစရှိသဖြင့် family of product တွေရှိပါတယ်။ ဒါပေမဲ့ paltform အပေါ်မူတည်ပြီး window Button လား Mac Button လားကွာနိုင်ပါတယ်။ ဒါပေမဲ့ Client ကတော့ Button ဆိုရင် click လို.ရရ မယ်ဆိုတာပဲ သိမှာပါ။ window Button လား Mac Button လား သိနေစရာမလိုပါဘူး ဒီနေရာမှာသူသိဖို.လိုတာက Button ဆိုတာပါပဲ။ အပေါ်က Class Diagram ကိုနမူနာပြရရင် Button, Label သည် Client ကနေသုံးမဲ့ AbstractProductA, AbstractProductB နဲ့.တူမှာပါ။ WinodwButton, MacButton ဆိုတာတွေကတော့ ProductA1,ProductB1 အစရှိတာတွေနဲ့. role ချင်းတူမှာပါ။

WindowGUIBuilder နဲ့. MacGUIBuidler တွေကတော့ Factory တွေဖြစ်တဲ့ ConcreteFactory1 နဲ့. ConcreteFactory2 နဲ့.သွားတူမှာပါ။

ဥပမာ ဆိုကြပါစို့. ကျွန်တော်တို့.က ReportGenerator တခုကိုရေးရမယ်ဆိုပါစို့. Report တခုမှာ ReportHeader ရှိမယ် ReportBody ရှိမယ်ပေါ့ဗျာ။ Report ကလဲ အမျိုးအစားကို HTML နဲ့.ရော PDF နဲ့.ရောထုတ်ချင်တယ်ဆိုပါစို့. ဒါဆို HTML နဲ့. ထုတ်မယ်ဆိုရင် HTMLReportHeader ရယ် HTMLReportBody ရယ်ထုတ်ရမယ် ။ XML နဲ့.ဆိုလဲထိုနည်းလည်းကောင်းပေါ့။ ဒါပေမဲ့နောင်တချိန်မှာ Client က Excel နဲ့.ထုတ်ချင်လာတယ်ဆိုရင် ဘယ်လိုလုပ်မလဲ ဒါမျိုးကို extensible ဖြစ်အောင် လုပ်ချင်ရင် Client သည် သူသုံးနေတာ HTMLReportHeader လား XMLReporHeader လား စတာတွေကိုသိနေရရင် implementation ကိုတိုက်ရိုက်သုံးနေမယ်ဆိုရင် extend လုပ်လို.မလွယ်ပါဘူး။ ဒါဆို ခုနက HTMLReportHeader လား XMLReportHeader လားမခွဲပဲ ReportHeader ဆိုပြီး abstract class ပေးလိုက်ရင် အဲ့ဒီ ReportHeader ကို extends လုပ်သမျှနောက် implemenation တွေအကုန် သုံးလို.ရမှာပါ။ Extensible ဖြစ်သွားပါပြီ။ ပြဿနာက ReportHeader တင်မကပဲနဲ့. ReportBody ပါ report type ပေါ်မူတည်ပြီးထုတ်ရမှာဖြစ်တဲ့အတွက် Fmaily of product ကို create လုပ်ရမှာပါ။ဒါဆို Abstract Factory နဲ့.ကိုက်နေပါပြီ။

Java Example

ပထမဆုံး fmaily of product ဖြစ်တဲ့ abstract class တွေ ဖန်တီးရပါမယ်။

```
abstract class ReportHeader
{
   abstract void genereateHeader();
}
```

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

အပေါ်က class 2 ခုက client ကသုံးမဲ့ class တွေပါပဲ။ Client က အဲ့ဒီ abstract class တွေကိုပဲသုံးတော့ different implementation ကိုနောက်ပိုင်းထပ်ထဲ့ရတာပိုလွယ်မှာပါ။ ခု သူတို့နဲ့ဆိုင်တဲ့ concrete implementation တွေကို အောက်ပါအတိုင်းရေးပါတယ်။

```
class HTMLReportHeader extends ReportHeader
{
   @Override    void genereateHeader() {
       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 genereateHeader() {
      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။ သူတို့.အတွက် Factory method တွေဆောက်ပါမယ်။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 ပါသူ့.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 concreate class တွေကို သူ့တို့. ရဲ့. createHeader ,createBody မှာ return ပြန်ပါတယ်။ ဒါပေမဲ့ client က ဘယ် different implementation ကိုသိစရာမလိုပဲ ReportHeader, ReportBody အနေနဲ့.ပဲသုံးရမှာပါ။ အောက်က 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.genereateHeader();
        body.generateBody();
    }
}
```

Client က သုံးလိုင်းမြောက်မှာ HTMLReportFactory ကိုဆောက်ပြီး header တွေ body တွေ ဆောက်ယူပါတယ် ကျွန်တော်တို့.က ReportFactory fac ကိုသုံးပြီးဆောက်ပါတယ်။ HTMLReportFactory ကိုသုံးတဲ့အတွက် ကျွန်တော်တို့.ရလာမှာက HTMLReportHeader,HTMLReportBody object တွေပါ။ Client ဘက်ကနေကြည့်ရင် အဲ့ဒီ implementation ကိုသိစရာမလိုပဲနဲ့. base abstract class ReportHeader,ReportBody ကိုသိဖို့.ပဲလိုပါတယ်။ ဒါဆို ဘာကောင်းသလဲဆိုတော့ နောက်ထပ် ReportGenerator ဆိုပါစို့. Excel နဲ့.လာမယ်ဆိုရင် သက်ဆိုင်ရာ Family Class တွေဆောက် Concrete Factory ကိုဆောက်လိုက်ရင်ရပါပြီ။ Client code ကိုပြင်စရာမလိုပါဘူး။ Family of product class တွေအများကြီးကိုနောက်ထပ်လိုအပ်တဲ့အချိန်မှာအေးအေးဆေးဆေးထပ်ထဲ့လို.
ရပါပြီ။ဒါကတော့ Abstract Factory ရဲ. advantage ပေါ့ဗျာ။ လိုအပ်ရင်လဲ ဘယ် ReportGenertor ပဲသုံးမယ်ဆိုပြီး client code မှာတကြောင်းတည်း ပြင်လိုက်တာနဲ့. အဆင်ပြေပါပြီ။

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

အပေါ်က အကြောင်းလေးကိုပြင်လိုက်တာနဲ့.ရပါပြီ ။

Object Oriented Design Pattern Series Part-4 Builder Pattern

Builder pattern ကတော့ Creational pattern တွေထဲကတခုပါပဲ။ သူ.ကို ဘယ်နေရာမှာသုံးလဲဆိုတော့ Complex Object တွေဆောက်ဖို. Object creation အတွက်လိုအပ်တဲ့ parameter တွေများနေမယ်ဆိုရင် Builder pattern ကိုသုံးပါတယ်။ ဥပမာ ကျွန်တော်တို.ဆောက်မဲ့ Object သည် paramter ၅ ခု ၆ ခုလောက်လိုမယ်ဆိုပါတော့ ။ ဒါပေမဲ့တချိန်တည်းမှာလဲ parameter အားလုံးသုံးချင်မှလဲသုံးမယ် မသုံးပဲ ၃ ခုလောက်ပေးပြီးတော့ပဲဆောက်ချင်လဲဆောက်မယ်။ ဒါဆိုရင် Constructor နဲ.ရေးဖို.ကတော်တော်ခက်သွားပါပြီ။ ဥပမာ Text ဆိုတဲ့ class တခုပေါ့။ သူ.မှာ အောက်က properties တွေပါမယ်ပေါ့။

String displayValue;

String font;

String color;

String decoration;

UI application တခုမှာ Text Label ကို ကိုယ်စားပြုမဲ့ Class ပေါ့။ သူ.ထဲမှာ ပါတဲ့ properties တွေက အပေါ်က ကောင်တွေအတိုင်းပဲ အဲ့ထက်လဲ အမှန်ဆိုများနိုင်တယ် real world မှာ။ လိုချင်တာက problem က Text object ကိုဆောက်မယ်ဆိုရင် displayValue ပဲပေးလိုက်တာလဲဖြစ်နိုင်သလို ကျန်တာတွေကို လိုမှလဲထဲ့မယ်။ ဆိုချင်တာက Object ဆောက်ရင် properties အကုန် ပေးချင်မှပေးမယ်။ ပေးရင်လဲပေးမယ်။ Object ရဲ. properties တွေကတော့ object မဆောက်ခင်ပေးရမယ်။ ဒါဆိုသူ.ကို သာမာန် Constructor နဲ.ရေးပြီး 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 လုပ်ဖို့.ကြိုးစားပါတယ်။ ဒါပေမဲ့အဆင်မပြေပါဘူး။ ဘာလို့.လဲဆိုတော့ code ကိုသေချာကြည့်ပါ။ code မှာ ကျွန်တော်တို့.က

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

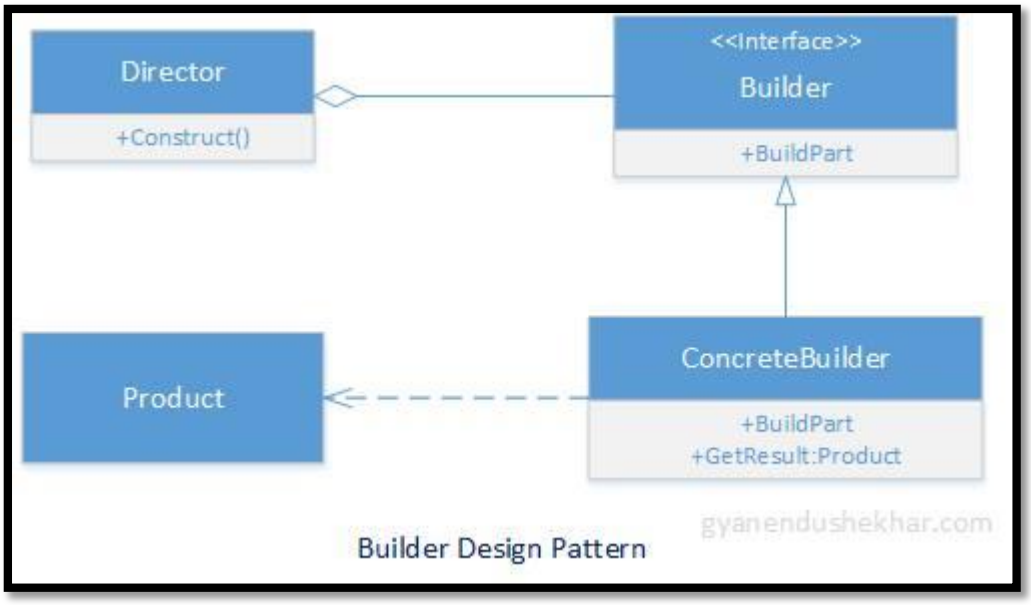
ဒီ Constructor ဆိုရင် displayValue, နဲ့. font ကိုလက်ခံပါတယ်။ တကယ်လို့.များ user ကနေ displayValue နဲ့. color ပေးရင် ရော့ code က အလုပ်လုပ်အောင် ရေးနိုင်ပမလားဆိုတော့မရပါဘူး။ ဘာလို့.လဲဆိုတော့ Java လို static language တွေမှာ method overlaoding ဟာ parameter တွေရဲ့. type, number of parameter, order ဆိုတဲ့ သုံးခုပေါ်မူတည်ပြီး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

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

ဒီလို displayValue နဲ့. color ပဲလက်ခံတဲ့ Constructor ထပ်ဆောက်လို့.မရပါဘူး ။ဘာလို့.လဲဆိုတော့ 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 လုပ်လို့.မရပါဘူး။ ဒါဆိုဒီလို ပြဿနာမျိုးကို Builder design pattern နဲ့.ဖြေရှင်းရင်ရပါတယ်၊

Intent

သူ.ရဲ့.ရည်ရွယ်ချက်က Complex Object တွေကိုတည်ဆောက်တဲ့အခါမှာ Object က parameter အများကြီးကို object creation stage မှာလက်ခံတယ်။ parameter အားလုံးကိုလဲ သုံးချင်မှသုံးမယ်။ 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 ကျွန်တော်တို့.က Object တခုအတွက် တိုက်ရိုက် parameter တွေကိုလက်ခံမဲ့အစား သူ့.အတွက် Builder လို့.ခေါ်တဲ့ Object မှာ parameter တွေကို ခနသိမ်းထားမယ်။ နောက်မှ build ဆိုတဲ့ method ကိုခေါ်မှ သူခုနက 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 ကတော့ Text object အတွက်ပါပဲ။ သူ့. properties တွေကတော့အရင်အတိုင်းပဲ ဒီ မှာ builder pattern နဲ့.ဆောက်တဲ့နေရာမှာ ပထမ user ကနေပေးလိုက်တဲ့ parameter တွေကို Builder object တခုထဲမှာသိမ်းထားမယ် နောက်မှ builder object ကနေ Text class ကို ပေးလိုက်မယ်။ဒီလိုနည်းနဲ့.သွားတာပါ ။ ဒါကြောင့် Text class ရဲ့. constructor မှာ ပထမ Constructor သုံးပြီး solve လုပ်တုံးကလို သူ့.ရဲ့. 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 ရဲ. constructor ပါ။ Text ရဲ. properties တွေကို builder object ကနေပဲယူလိုက်တာပါ။ ကျန်တာကတော့ဘာမှထူးထူးဆန်းဆန်းမပါပါဘူး။ အဓိက အသက်ကတော့အောက်က builder class ပါ။ Builder class ကို Text class ရဲ. static inner class အနေနဲ.ဆောက်ပါတယ်။ အပြင်မှာဆောက်ချင်လဲရပါတယ်။ ဒါပေမဲ့ encapsulation အတွက် အပြင်မထုတ်ပဲ အထဲမှာပဲရေးလိုက်တာပါ။ Builder class မှာရှိတဲ့ properties တွေသည် Text class မှာရှိတဲ့ properties တွေနဲ့.အကုန်တူရမှာပါ။ ဘာလို.လဲဆိုတော့ Builder သည် user ကပေးတဲ့ Text class ရဲ. parameter တွေကို ခဏ သိမ်းထားပေးမဲ့ကောင်ဖြစ်လို.ပါ။ နောက် property တခုချင်းဆီအတွက်ဒီလို method တွေရေးပါတယ်။

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

အပေါ်က method ၂ခုသည် displayValue နဲ. font ကို လက်ခံပါတယ်။ Text class မှာသိမ်းမဲ့ property တခုချင်းအတွက်ကို Builder မှာ method တခုစီရေးရမှာပါ။ လွယ်လွယ်လေးပါ ဝင်လာတဲ့ parameter ကိုသိမ်းတယ် နောက် သူ. object this ကို return ပြန်တယ်။ ဘာကြောင့်

this ကို return ပြန်လဲဆိုတော့ method chaining pattern ကိုသုံးလို.ရအောင်ပါ။ 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 ရဲ. method တွေဖြစ်တဲ့ color ကိုခေါ်လို.ရပါမယ်။ color method သည် return this လို.ပြန်ထားတဲ့အတွက် builder object ကို return ပြန်မယ်။ ဒါကြောင့်နောက်ထပ် method တွေကို dot ခေါက်ပြီးဆက်တိုက် ခေါ်လို.ရမယ်။ ဒါကို method chaining pattern လို.ခေါ်ကြပါတယ်။ နောက် Builder pattern ရဲ. Text object construction ကို ဒီလိုရေးပါတယ်

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

User ကနေ builder object ကို parameter တွေပေးပြီးလို.စိတ်ကြိုက်ပြီဆိုရင် build ဆိုတဲ့ method ကိုခေါ်လိုက်ရမှာပါ။ ဒါဆို builder object ကို Text contructor ထဲကိုပေးလိုက်မှာပါ။ Text Constructor ကနေ Builder Object ရဲ. properties တွေကို ကူးပြီး သူ. Text object ဆောက်မှာပါ။ ဒီတော့ Text object ဆောက်ချင်ရင်ဒီလိုသုံးလို.ရပါပြီ

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

အဲ့တော့ ဘာကောင်းလဲဆိုတော့ ကိုကြိုက်တဲ့ parameter တွေကို ဥပမာ color အရင်ထဲ့မလား displayValue အရင်ထဲ့မလား ကြိုက်တာလုပ်လို.ရတယ် မထဲ့လဲရတယ် ။ method တခုချင်းကိုခေါ်ရင် properties တွေကို builder object ရဲ. properties အနေနဲ.ခနသိမ်းမယ် ။ ပီးတော့မှ build ကိုခေါ်လိုက်မယ် ဒါဆို build method ကနေ Builder object ကို Text constructor ကိုပို.မယ် Text contructor မှာ builder ကပို.တဲ့ properties တွေကို သူ. Text Object ရအောင် ဆောက်မယ် ဒီနည်းနဲ.ရှင်းသွားတာပါ။ ကျန်တဲ့ language တွေနဲ.လဲ ဒီသဘောတရားပါပဲ။

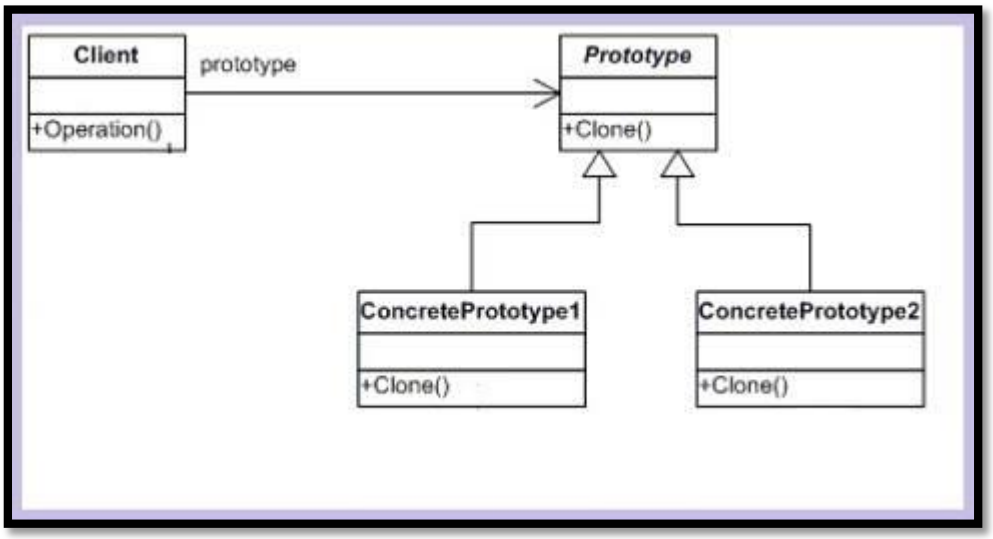
Object Oriented Design Pattern Series Part-5 Prototype Design Pattern

Prototype Pattern ကလဲနားလည်ရလွယ်တဲ့ pattern မျိုးပါပဲ။ 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 များမယ် costly ဖြစ်မယ်ဆိုရင် နောက်ထပ် Object တွေဆောက်တဲ့အခါမှာ ခုနက computational time , memory အစရှိတာတွေကိုသက်သာအောင် ရှိပြီးသား Object တွေရဲ. မူရင်း property တွေကိုပဲ copy ကူးပြီး Object အသစ်ဆောက်လိုက်တာပေါ့။ ဒါကို prototype pattern လို.သုံးပါတယ်။ ဒီနေရာမှာ Heavy Object ဆိုတာဘာလဲဆိုတာကိုရှင်းဖို.လိုပါတယ်။ ဥပမာ Object တခုကိုတည်ဆောက်တဲ့အခါမှာ property တွေလိုတယ်ပေါ့ဗျာ။ အဲ့ဒီ property တွေကို DB ကနေယူရမယ် ဆိုပါစို.ဒါဆိုရင် DB ကို query လုပ်ရတာကြာတဲ့ cost ကုန်မယ် နောက်ပြီး အဲ့ဒီ property ကို ဒီအတိုင်းမရနိုင်ဘူး။ ဥပမာ Game သို.မဟုတ် math နဲ.ဆိုင်တဲ့ application တခုခုမှာ ဒီ property ကိုတခြား အရာတခုခုက နေတွက်ထုတ်ရမယ်ဆိုရင် (heavy matrix calculation လိုမျိုးပေါ့)computational time အရ အချိန်ကြာနိုင်ပါတယ်။ နောက်ထပ် memory အရ costly ဖြစ်တာကတော့ String လို object မျိုးတွေပေါ့ ။ 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 အခြေအနေမျိုးမှာ prototype pattern ကိုသုံးရပါတယ်။ အောက်မှာ prototype pattern ရဲ. 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။ အဲ့ဒီတော့ 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 လုပ်လို့.ရတယ်ဆိုတာကိုမှတ်ထား ဆိုတဲ့သဘောနဲ့. သုံးတဲ့အတွက် marker interface လို့.သုံးပါတယ်။ Serializable လို interface မျိုးပါ။ သူ.မှာ ကျွန်တော်တို့.က reprdouce ဆိုတဲ့ method ထဲ့ပေးထားပါတယ် အမှန်က reprdouce 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 getCompuationalHungryProperty() {
return "ComputationHungryProperty";//Here assume call to costly computation time
}
public HeavyObject() {
this.propertyGetFromDB = HeavyObject.getPropertyFromDB();
this.computationalHungryProperty = HeavyObject.getCompuationalHungryProperty();
}

@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 HeayObject ကိုပြထားတာပါ။ သူ့မှာ property ၂ခုသည် heay computation+ heay memory ကို ကိုယ်စားပြုထားပါတယ်။ နောက် Contructor မှာ အဲ့ကောင်တွေကို static method တွေဆီကနေယူပါတယ်။ အဲ့ဒီ static method တွေသည် တကယ်တမ်း DB call တို့ computation တို့ပါရမှာပါ။ ဒီမှာ code ရှည်မှာဆိုးတဲ့အတွက် ချန်ခဲ့တာပါ။ HeayObject သည် Prototype interface ကို implement လုပ်ပါတယ်။ ဒါကြောင့် reproduce ကို override လုပ်ရမှာပါ။ reproduce မှာ ကျွန်တော်တို့ 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 အသက်ပါ။ clone() method ကိုပဲသုံးသွားတာပါ။ new operator ကိုသုံးပြီး object ပြန်ဆောက်ရင် constructor ကိုခေါ်ရမယ် ဒါဆို အချိန်ပိုကြာသွားမယ် လေးသွားမယ် ဒါကိုရှောင်ချင်လို.သာ 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.reproducce();

System.out.println(another);
}
}
```

Client ကသုံးတဲ့နေရာမှာ firstObject တခုကိုပဲ new operator သုံးပြီး constructor နဲ့.ခေါ်ပါတယ် ပထဆုံးအကြိမ်မှာတော့ heay computation နဲ့.costly operation ဖြစ်မှာပါ။ နောက် object တွေဖြစ်တဲ့ another object ကိုကျတော့ reproduce ကိုသုံးပြီး clone လုပ်ပြီး Object ဆောက်ပါတယ်။ ဒါကြောင့်ခုနက constructor ကိုမဖြတ်ရတော့ပါဘူး ။ ဒါကြောင့် heavy computation+memory cost ကိုကျော်နိုင်ပါပြီ။

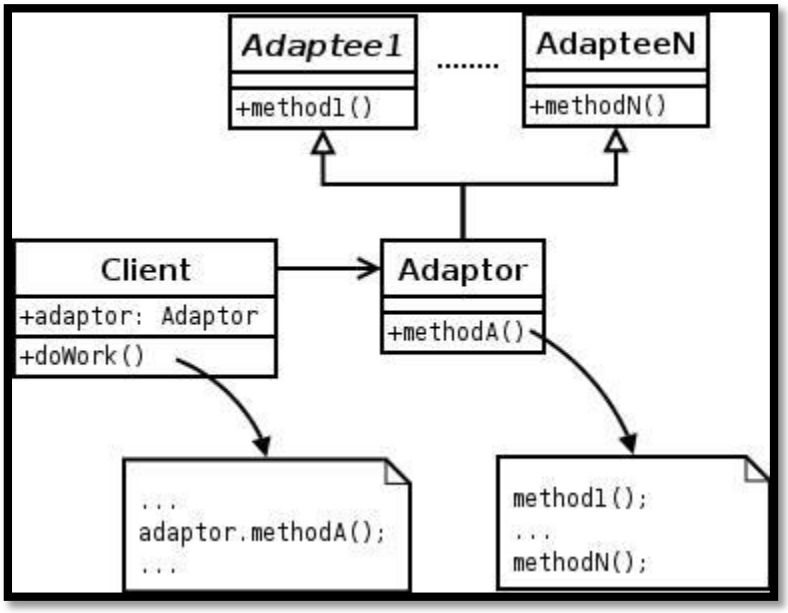
Prototype pattern ရဲ့အားသာချက်ကတော့ Heavy ဖြစ်တဲ့ Object တွေဆောက်တဲ့နေရာမှာ ခနခန ထပ်ဆောက်မနေပဲ ဘုံတူတဲ့ property တွေကို copy ကူးပြီးသုံးတဲ့ အတွက် computational time+memory သက်သာတာပါပဲ။ ဒါကတော့သူ. ကောင်းတဲ့အချက်ပါပဲ။

Object Oriented Design Pattern Series Part-6 Adapter Design Pattern

Adapter pattern ကတော့ Structural pattern တွေထဲကတခုပါ။ သူကဘာလုပ်ပေးလဲဆိုတော့ interface မတူတဲ့ module ဒါမှမဟုတ် code ၂ခုကြားမှာ အလုပ်လုပ်လို.ရအောင် glue code ဒါမှမဟုတ် bridge လုပ်ပေးတာပါ။ ဒီနေရာမှာ interface သည် module တခု သို.မဟုတ် class တခုရဲ. method name, parameter type, parameter order, အစရှိတာတွေကိုပြောတာပါ။ ရိုးရိုးရှင်းရှင်းပြောရရင် method ရဲ. နာမည်တွေ parameter ပေးတဲ့ပုံစံတွေကိုဆိုချင်တာပါ။ ဥပမာ moule A က လိုချင်တာ add ဆို method ဆိုပါစို. ဒါပေမဲ့ module B က ပေးထားတာကြတော့ addItem ဖြစ်နေတယ်ဆိုပါစို. ဒါဆို add လို. module A က လိုချင်ပေမဲ့ addItem ဖြစ်နေတဲ့အတွက် မတူတဲ့အတွက် အဆင်မပြေပါဘူး ။ဒါဆိုခုနက modula A ကလဲ add လို.ပဲခေါ်မယ် module B ကလဲ addItem ကိုမပြောင်းပဲထားမယ်ဆိုရင် သူတို.ကြားမှာ အဆင်ပြေဖို.အတွက် Glue code တခုလိုပါပြီ။ Real world က example ဆိုရင်တော့ IDE တွေ CMS လို complex software တွေမှာ plugin တွေရေးကြပါတယ်။ အဲ့လိုရေးကြတဲ့အခါ Plugin architecture ဆိုတာကိုသုံးကြပါတယ်။ ဘယ်လိုအလုပ်လုပ်သလဲဆိုတော့ Framework တွေ IDE တွေကနေ plugin တွေဟာ ဘယ် method တွေကို ရေးထားရမယ်ဆိုပြီး သတ်မှတ်ပါတယ်။ အဲ့အချိန် Framework ကလိုချင်တဲ့ method signature နဲ. ကိုသုံးမဲ့ plugin အတွက် code (ဒီနေရာမှာ method signature ကိုပြင်လို.မလွယ်ဘူး ဒါမှမဟုတ် binary form နဲ.ပဲရလို. ပြင်လို.အဆင်မပြေဘူး)သည် method singature မတူရင် Adapter Design pattern ကိုသုံးပါတယ်။

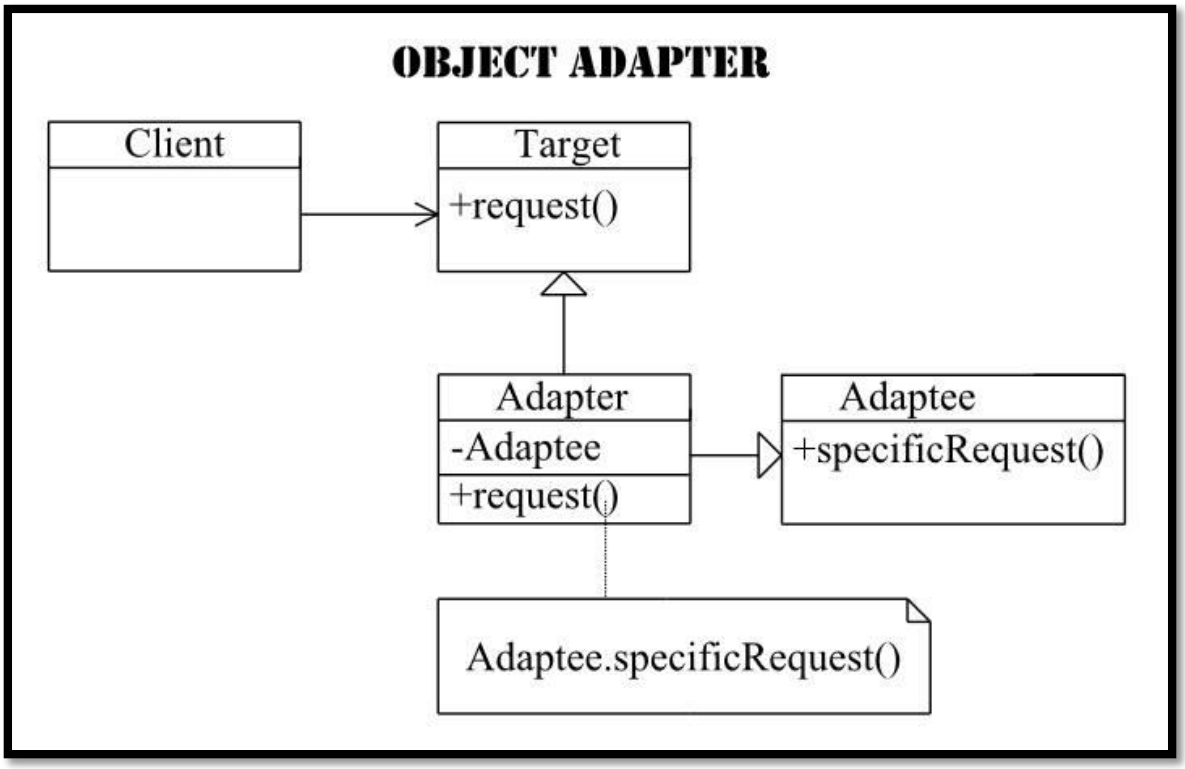
အပြင်လောက software မဟုတ်တဲ့ နယ်ပယ်မှာ ထင်ရှားတဲ့ Adapter တွေကတော့ ပါဝါကြိုးတွေမှာသုံးတဲ့ adapter ပါပဲ။ မြင်အောင်ပြောရမယ်ဆိုရင် ကွန်ပျူတာကြိုးတွေက 3 pin

ပေါ့. ဒါဆို plug ပေါက်က 2 ပင်ပဲရှိတယ်ဆိုရင် အဆင်မပြေပါဘူး။ ကြားထဲမှာ 3pin ကနေ 2 pin ကိုပြောင်းပေးနိုင်တဲ့ adapter ခေါင်းတခုလိုပါတယ်။ Adpater pattern ကလဲ အဲ့လိုပဲ ဟိုဘက်က code နဲ့.ဒီဘက်က code interface မတူညီကြရင် ကြားထဲကလုပ်ပေးတဲ့ Glue code ပါပဲ။



Class Adpater

Adapter မှာ ၂မျိုးရှိပါတယ် Class Adapter နဲ့. Object Adatper ဆိုပြီးတော့။ အပေါ်ကပြထားတဲ့ UML diagram ကတော့ Class Adapter ဖြစ်ပြီးတော့ အောက်ကကောင်ကတော့ Object Adapter ပါပဲ။



Object Adapter

Intent

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

ကျွန်တော်တို့မှာ Framework ဆိုတဲ့ interface တခုရှိတယ် သူ့ကို အားလုံးကလိုက်နာရတယ်ဆိုပါစို့. ။ သူမှာ add(Integer item) ဆိုတဲ့ method တခုပါတယ်ဆိုပါစို့. ဒါပေသိ Framework နဲ့.တွဲသုံးချင်တဲ့ legacy class တခုကျတော့ addItem(Integer item) ဖြစ်နေရော ဒါဆိုသူတို့. ၂ခုတွဲလုပ်လို့.မရဘူးပေါ့။ အဲ့ဒါဆိုရင် Class Adapter သုံးပြီးဖြေရှင်းလို့.ရပါတယ်။

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

ဒါကတော့ interface ပါ သူ.ကို အားလုံးကလိုက်နာရပါမယ် အောက်ကကောင်ကတော့ Old legacy code ပါသူ.ကိုလဲ တခြားကောင်တွေကသုံးနေတော့ ပြင်ခွင့်မရှိဘူးဆိုပါစို့.ဗျာ။

```
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 အဓိပ္ပာယ်သဘောတရားချင်းတူပေမဲ့ interface method signature ကွာတဲ့အတွက် တွဲသုံးလို့.အဆင်မပြေပါဘူး ။ ဒါဆို ကျွန်တော်တို့.က ClassAdapter လို့.ဆောက်ပြီး OldAPI ကို extend လုပ်မယ် Framework ကို implement လုပ်မယ် ပြီးတော့ framework method ကို override လုပ်မယ် အဲ့ကနေ OldAPI method addItem ကိုခေါ်ပေးလိုက်မယ်ဆိုရင် Class Adapater 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 သူ့.အလုပ်က OldAPI ကို extend လုပ်တယ် ဒါဆို inheritance နည်းအရ OldAPI ကပေးထားတဲ့ api တွေ method တွေကိုသူသုံးလို့.ရသွားတယ်ပေါ့. 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 ရဲ. method အဟောင်းဖြစ်တဲ့ add ကိုလှမ်းခေါ်လိုက်တာပါပဲ။ အဲ့ဒါသည် adapter pattern ရဲ့.အသက်ပါပဲ ဒါဆို ClassAdapter သည် Framework ကလိုချင်နေတဲ့ interface နဲ့.လဲကိုက်ပြီ OldAPI ကိုလဲ လှမ်းခေါ်နိုင်ပြီ ဖြစ်တဲ့အတွက် adapter ဖြစ်ပါပြီ။ Client ကတော့ဒီလိုသုံးရမှာပါ။

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

အပေါ်က ClientDemo example မှာ client သည် OldAPI ကို တိုက်ရိုက်မသုံးဘူးဆိုတာပါပဲ.။ ဘာလို့.လဲဆိုတော့ ရည်ရွယ်ချက်ကိုက OldAPI ကိုသုံးချင်တယ် ဒါပေမဲ့ method အဟောင်းအနေနဲ့.မဟုတ်ပဲ Framework ကနေပေးထားတဲ့ method အတိုင်းသုံးချင်တာ။ ဒါကြောင့် ClassAdpater ကနေတဆင့် လှမ်းခေါ်လိုက်ရင် ဒီပြဿနာက အဆင်ပြေပါပြီ။

Object Adapter

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

```
public class ObjectAdpater  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 inheirtance မသုံးပဲ Object composition သုံးပြီး oldAPI object ကိုဆောက်ပါတယ် ပြီးတော့ အဲ့ကနေတဆင့် OldAPI ရဲ. method addItem ကိုခေါ်ပါတယ်။ Client ကနေသုံးရင် ClassAdapter အစား ObjectAdapter လေးပြောင်းသုံးရုံပါပဲ။

Adapter ကဘာကောင်းလဲဆိုရင်တော့ Legacy code တွေမှာ interface နဲ့.မကိုက်လို. အဆင်မပြေဘူး ခေါ်လို.မရဘူးဆိုရင် ကြားခံ Glue Code အနေနဲ့.သုံးလို.ရအောင် ပြောင်းလိုက်လို.ရပါတယ်။ ဒါပေမဲ့ Design အရ မှားနေတဲ့ class ဆိုရင်တော့ ဆိုချင်တာက OldAPI class ကို ထိလို.ရမယ်ဆိုရင် Adapter မသုံးပဲဖြေရှင်းတာပိုကောင်းပါတယ်။

Object-Oriented Design Pattern Series Part-7 Bridge Design Pattern

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

Intent

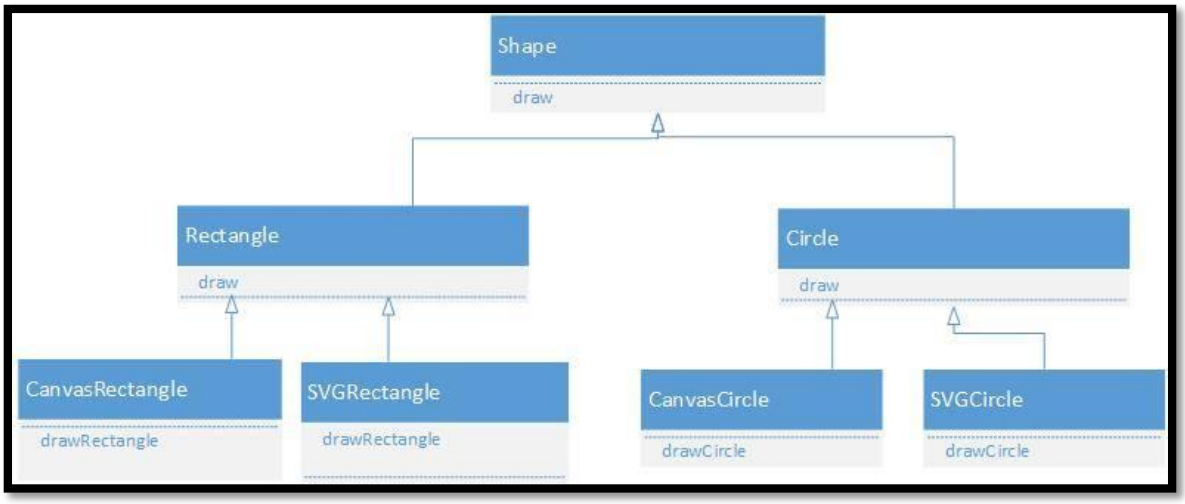
သူ. intent ကိုတော့ GoF Book မှာဒီလိုရေးထားပါတယ်။

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

ဒီနေရာမှာ Decouple လုပ်တယ်ဆိုတာ တခုနဲ့.တခု ကိုအသေချိတ်ထားတာဲ (class တခုမှာ နောက် class တခုကိုပဲသုံးထားတယ် abstract,interface ကိုမသုံးဘူး ဒါဆိုနောင်တချိန်မှာ အစားထိုးသုံးလို.မရဘူး။ extend လုပ်ဖို.ခက်မယ်, abstract,interface ကိုသာသုံးထားရင် သူတို.က ဆင်းလာတဲ့ကောင်တွေနဲ.သုံးလို.ရလို. extend ရတာလွယ်မယ်). ကိုဆိုချင်တာပါ။ Abstraction ဆိုတာတော့ implementation detail ကို hide လုပ်ထားတာကို ပြောချင်တာပါ။ Implementation detail ဆိုတာကတော့ ဒီ code ကို implment လုပ်ဖို.အတွက် ဘယ် class ဘယ် method ဆိုတာကို ခေါ်တဲ့သူက သိနေရမယ် ဒီလိုဆိုချင်တာပါ။ တကြောင်းတည်းကိုရှင်းတာ တော်တော်ရှုပ်နေပါပြီ။နောက်အဲ့တကြောင်းထဲကလဲ General ကျတဲ့အတွက် ဘာကိုဆိုလိုတယ်ဆိုတာ ရုတ်တရုတ်သဘောပေါက်မှာမဟုတ်ပါဘူး။

Motivation

ကျွန်တော်တို.က Drawing application တခုဆောက်မယ်ဆိုပါစို.ဗျာ။ အဲ့မှာ Shape တွေအများကြီးရှိမယ်ပေါ့။ ဥပမာ Cricle, Rectangle အစရှိသဖြင့်ပေါ့ဗျာ။နောက် ခုနက Shape တွေဖြစ်တဲ့ Circle, Rectangle တွေကို ကျွန်တော်တို.က drawing method ၂မျိုးသုံးပြီး ဆောက်ချင်တာက ဆိုပါစို.ဗျာ SVG နဲ.ရယ် Canvas နဲ.ရယ်ပေါ့။ အဲ့တော့ ကျွန်တော်တို.က ခုနကလိုတာတွေကို class diagram ဆောက်လိုက်မယ်ဆိုရင် ဒီလိုရမယ်ဆိုပါစို.။

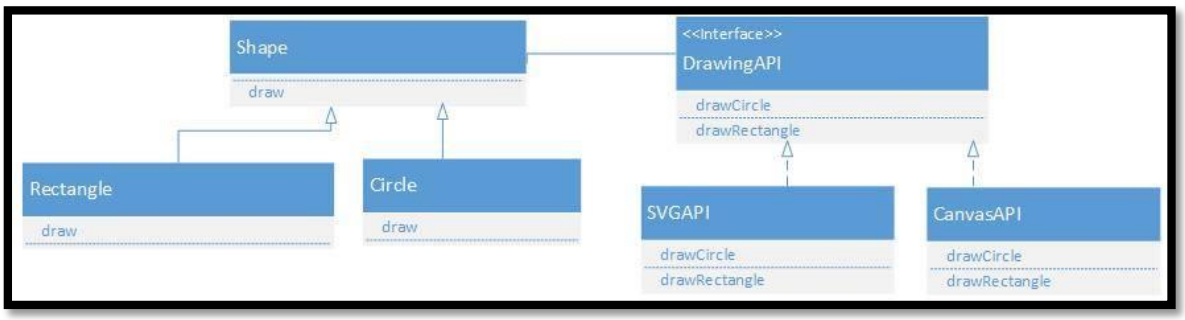


အပေါ်ကပုံမှာ Shape သည် abstract class ဖြစ်မယ် Diagram မှာတော့ထဲ့မထားဘူး။ Shape မှာ Rectangle, Circle သည် child shape ကနေ inherit လုပ်လာတာ။ နောက် ခုနက DrawingAPI Canvas or SVG ပေါ်မူတည်ပြီး CanvasRectangle,SVGRectangle အဲ့လိုခွဲထားတယ်။ ဒီ class diagram မှာ ဘာအားနည်းချက်ရှိလဲကြည့်ရအောင်။ ဒီ diagram အရဆိုရင် Rectangle,Circle သည် abstractionလို.ပြောရမယ် ဘာလို.လဲဆိုတော့ client ကသုံးမှာ SVG နဲ.ဆွဲထားတာလား canvas နဲ.ဆွဲထားတာလား သိစရာမလိုဘူး သူလိုချင်တာ conceptual ဖြစ်တဲ့ rectangle လား circle လားပဲ။ အဲ့တော့ conceptual level အရကြည့်ရင် Rectangle, Circle,Shape သည် abstraction ပဲ။ တကယ့် တကယ်အလုပ်လုပ်တာကြတော့ CanvasRectangle ,SVGRectangle သူတို.ကျတော့ implementation လို.ဆိုရမယ် ဘာလို.လဲဆိုတော့ တကယ့် Canvas drawing,SVG drawing ကိုလုပ်လို.။ သူတို.မှာ ဘာအားနည်းချက်ရှိလဲဆိုတော့ client ကသုံးမယ်ဆိုရင် ဒီလိုတွေသုံးရမယ်

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

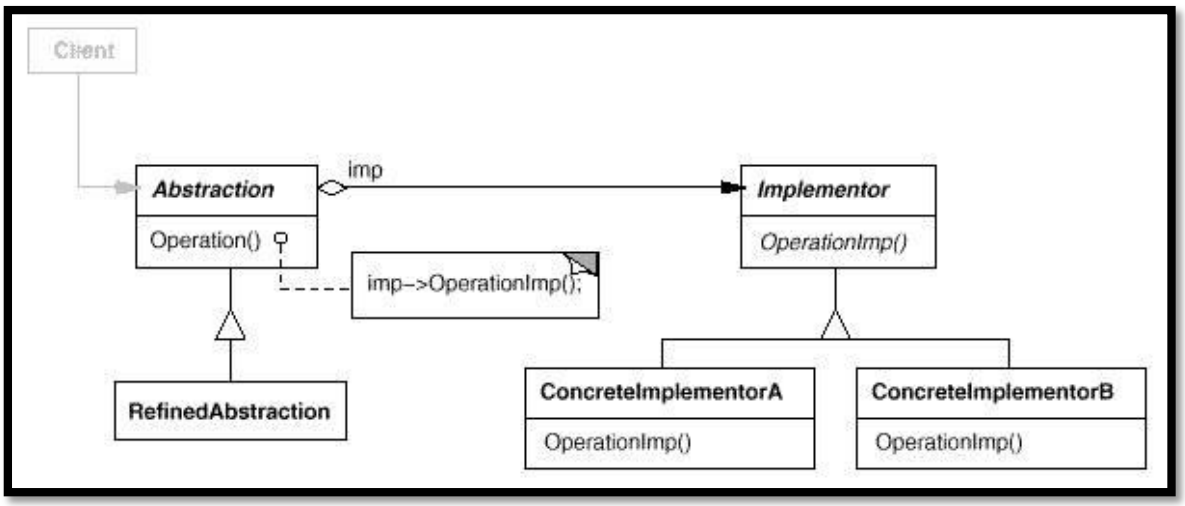
အဲ့တော့ ဘာအားနည်းချက်ဖြစ်လာမလဲဆိုတော့ client က သူသုံးရမဲ့ implemnetation (Canvas လား SVG လားပြောပြီး concrete class) နဲ. create လုပ်နေရတယ်။ OO Programming မှာ program to interface not to implementation လို.ဆိုကြတယ်။ ဆိုချင်တာက ဒီ implementation class တွေကို တိုက်ရိုက်ခေါ်နေရတော့ နောင်တချိန် 3D Drawing API လိုတခုခုဆောက်မယ်ဆိုရင် client က သပ်သပ်သိနေရဦးမယ် ဘယ် Object ဆောက်ဆိုတာကို။ client က implementation ကိုမသိပဲပေးဆောက်ချင်ရင် Bridge Pattern ကိုသုံးလို.ရတယ်။ ဥပမာ ဒီနေရာမှာ implementation ကို interface တခုအနေနဲ.ထားပြီး class Diagram

ကိုဒီလိုပြန်ဆောက်လိုက်မယ်ဆိုရင် implementation ကို runtime မှာပါချိန်းလို.ရမှာပါ။ Client ကတော့ Circle, Rectangle ဆိုတဲ့ abstraction တွေနဲ.အလုပ်လုပ်သွားနိုင်မှာပါ။ ဘယ် specific implementation ဆိုတာကိုသိစရာမလိုတော့ပါဘူး။



အပေါ် ပုံဆိုရင် Bridge pattern နဲ. Design ချထားတာပါ။ ခုနက Drawing API ကို interface သက်သက်ထားပြီးတော့ သူ.ကို shape ကနေတဆင့်ပဲယူသုံးခိုင်းတာပါ။ ဒါဆို client ကကြိုက်တဲ့ API ကို ပြောင်းပြီးသုံးလို.ရပါတယ်။ နောက်ပြီး drawing code တွေကို ဆိုင်ရာ class တခုထဲအောက်မှာပဲထားတဲ့အတွက် (ဥပမာ အရင်ပုံဆိုရင် SVG drawing အတွက်ဆိုရင် ရှိသမျှ class တွေလိုက်ကြည့်ရမယ်) ခုကျတော့ SVGAPI အောက်မှာကြည့်လိုက်တာနဲ.အကုန်ရပါပြီ။ ဒါကို SRP (Single Responsibility Principle) လို.ခေါ်ပါတယ်။ အဲ့တော့ ခုပုံစံအရဆိုရင် Abstraction ဖြစ်တဲ့ Rectangle ,Circle နဲ. implementation ဖြစ်တဲ့ SVGAPI နဲ. CanvasAPI ကိုခွဲထုတ်လို.ရပါပြီ။ Coupling မဖြစ်တော့ဘူး အဲ့တော့ နောက်ထပ် API တခုထပ်ထဲ့မယ်ဆိုရင် class hierarchy တွေမများပဲ class တခုပဲထဲ့လို.အဆင်ပြေပါပြီ။

အောက်ကတော့ GoF မှာပေးထားတဲ့ Bridge ရဲ. class diagram ပါ။



Code ကတော့ရှင်းပါတယ်။

```
public interface DrawingAPI {
   void drawCircle();
   void drawRectangle();
}
```

ဒါကတော့ DrawingAPI interface ပါ။ သူ.မှာ drawing method တွေအကုန်ထဲ့ထားပါတယ်။ Different Implementation တွေကသူ.ကို implement လုပ်ယုံပါပဲ။ ဒီမှာဆို SVGApi နဲ. 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 နဲ. CanvasAPI ၂ခုဟာ DrawingAPI ရဲ. 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 ဆိုပြီးထဲ့ထားတဲ့အတွက် ကြိုက်တဲ့ implementation ကိုသုံးလို့.ရမှာပါ။ ဒါကနောက်ပိုင်း DrawingAPI ကနေ တခြား different implementation တွေ ဥပမာThreeDAPI ထပ်ထဲ့ရင်လဲ အဆင်ပြေပါတယ်။ Shape constructor မှာ ဘယ် api နဲ့.ဆွဲမလဲဆိုတာကိုလက်ခံပါတယ်။

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 ,drawRectanlge ကိုခေါ်ပါတယ်ဒါပေမဲ့ ဒီမှာသတိထားရမှာက ဘယ် API ဆိုတာမပါပါဘူး ဒါသည် loose coupling ဖြစ်အောင်လုပ်ထားတာလို့.ဆိုရမှာပါ။ ကြိုက်တဲ့ API နဲ့.တွဲသုံးလို့.ရတယ်ပေါ့ဗျာ။ 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 ကျွန်တော်တို့.သုံးနေတာသည် ဟိုးပထဆုံးပြထားတဲ့ကောင်မှာလို SVG cirlce လား Canvas Circle လားသိစရာမလိုပါဘူး ဒါသည် abstractionပါ။ နောက် ကြိုက်တဲ့ drawing API ကို ပြောင်းလို.ရတဲ့အတွက် Decouple an abstraction from its implementation ဆိုတာကို လုပ်လို.ရနေတာပါပဲ။ ကောင်းတဲ့အချက်ကတော့ class hierarchy ဖောင်းပွမှု.ကို ကာပေးနိုင်တယ်။ Runtime မှာ different implementation ကို ပြောင်းလို.ရမယ်၊ specific implementation အပေါ် client code က မမှီခိုတဲ့အတွက် နောက် implementation class API တွေထပ်ထဲ့ရင်လွယ်မယ်ပေါ့ဗျာ။

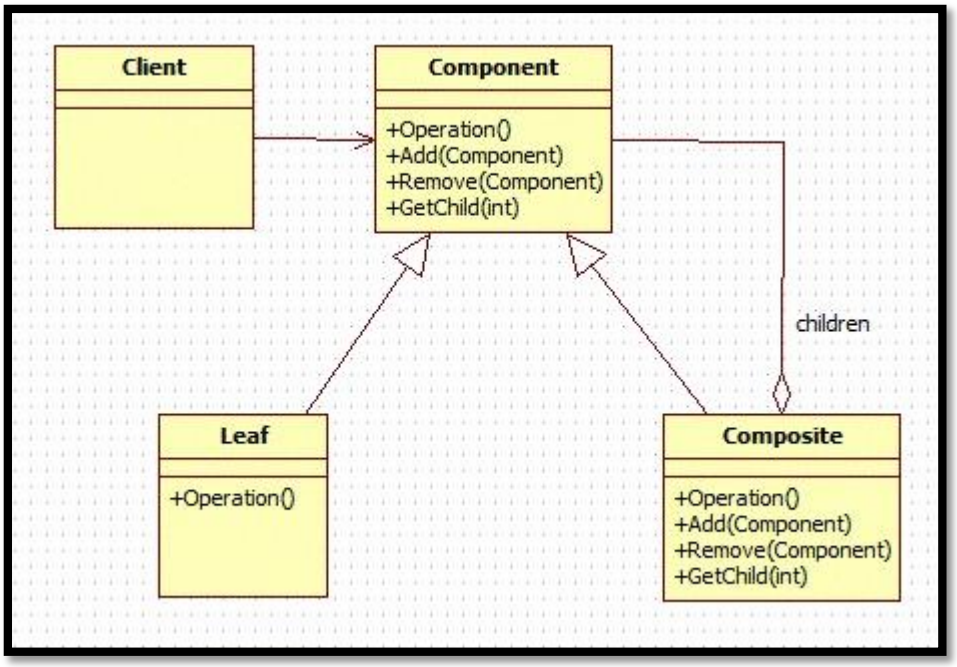
Object Oriented Design Pattern Series Part-8 Composite Design Pattern

Composite Design pattern က Structural pattern တခုပါ။ သူ.ကိုသုံးနေကျ မြင်နေကျပါ။ သူက Tree ပုံစံ Hierarchcy object တွေကို တည်ဆောက်ပြီး သူတို.ကို same interface နဲ.ထိန်းချုပ်ရင်သုံးပါတယ်။ ဥပမာ Java Swing မှာ Panel ထဲမှာ panel တွေ button တွေ input groupတွေ ထဲ့လို.ရပါတယ်။ နောက် Android မှာဆိုလဲ ViewGroup ထဲမှာ view တွေ ViewGroup တွေကို tree ပုံစံနဲ.ထဲ့လို.ရပါတယ်။ နောက် runtime system ကနေ သူတို.ကို screen ပေါ်ပြချင်တဲ့အခါ paint method ကိုခေါ်ရပါတယ်။ ဒီအခါမှာ ခုနက tree structure ဆောက်ထားတဲ့ object တွေကို traverse လုပ်ဖို.လိုပါပြီ ။ဒီလိုအခြေအနေမျိုးမှာ Composite pattern ကိုသုံးပါတယ်။ အဓိက ကတော့ object တွေဟာ hierarchical structure နဲ.အထဲမှာ တဆင့်ခြင်းရှိနေမယ်ဆိုရင် ဒီ pattern နဲ.တည်ဆောက်ယူလို.ရပါတယ်။

Intent

"Compose objects into tree structure to represent part-whole hierarchies.Composite lets client treat individual objects and compositions of objects uniformly".

GoF မှာတော့သူ. Intent ကိုဒီလိုပြထားပါတယ်။ Object တွေကို tree structure အနေနဲ. တည်ဆောက်ပြီး ဆောက်ထားတဲ့ object တွေကို uniformly treat လုပ်ချင်တာပါ။ ဆိုချင်တာက individual ရယ် Composite ရယ်မခွဲပဲ သုံးလို.ရချင်တယ် သုံးချင်တယ်ဆိုတဲ့ အဓိပ္ပာယ်ပါ။ ဥပမာ Android Rutime system ကနေ ViewGroup တခုကို paint ခေါ်လိုက်ရင် သူ.အထဲမှာရှိတဲ့ View တွေ ViewGroup တွေကိုပါ တခါတည်း paint လုပ်သွားစေချင်တယ်။ View လား ViewGroup လားခွဲခြားပြီး သိစရာမလိုချင်ဘူး အဲ့ဒါဆိုဒီ Composite pattern နဲ.ရေးလို.ရပါပြီ။ သူ.ရဲ. class diagram ကတော့ အောက်ကလိုပါ။



Component ကတော့ အားလုံးသုံးမဲ့ (Leaft, Composite) ကနေသုံးမဲ့ operation တွေစုထားတာပါ။ ကျွန်တော်တို. example အရ ဆို paint method ပေါ့။ နောက် leaf ကတော့ ထပ်ခွဲလို.မရတဲ့ nested child တွေ မပါတော့တဲ့ object တွေကို leaf လို.သတ်မှတ်ရမှာပါ။ Leaf ကတော့ View ဖြစ်မှာပါ။ Composite ကတော့ ViewGroup ပါပဲ။ သူ.မှာ child element တွေပါနိုင်ပါတယ်။ ViewGroup ထဲမှာ View တွေရော ViewGroup တွေပါ ကြိုက်သလောက်ပါနိုင်ပါတယ်။ ဒီနေရာမှာ Composite မှာ add ရယ် remove ရယ် getChild ဆိုတာရယ်ရှိပါတယ်။ သူတို.ကတော့ Compositve ထဲကို child element တွေ ထဲ့ချင်တဲ့အခါ ရယ် traverse လုပ်ဖို.ရယ်သုံးတာပါ။ Code ကိုကြည့်ရအောင်။ Uniform interface ရဖို.အတွက်ကျွန်တော်တို.အောက်က UIWidget class ကိုဆောက်လိုက်ပါတယ်။

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

သူ.မှာ id ပါပါတယ် ဒါကတော့ ပြရလွယ်အောင်ပါ။ Variable name ပေးထားသလိုပေါ့ဗျာ။ နောက် paint ပါပါတယ်။ ဒါကတော့ UIWidget တိုင်းက paint လုပ်ရမယ် ဒါကြောင့် သူ.ကို

abstract ထားပေးထားတာပါ။ သူ.ကို extends လုပ်တဲ့ Child တွေက override လုပ်စေချင်လို့.ပါ။ ဒါမှ child ပေါ်မူတည်ပြီး different implmenation ပေးလို့.ရမှာကိုး။ 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);
   }
}
```

သူ.မှာ ထူးထူးဆန်းဆန်း ဘာမှမပါပါဘူး Constructor ဆောက်တယ် parent ကို Id ပေးလိုက်တယ်. နောက် သူ. paint မှာ ဘယ်သူ.ကို 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 compositve ဖြစ်တဲ့အတွက်တခြား View, ViewGroup တွေပါလို.ရ ရမှာပါ။ ဒါကြောင့် အဲ့ကောင်တွေကိုသိမ်းဖို.

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

ဆိုပြီးရေးပါတယ်။ ဒီနေရာမှာ သတိထားရမှာက သိမ်းတာသည် UIWidget ပါ သူသည် parent ဖြစ်တဲ့အတွက် View ရော ViewGroup ပါသိမ်းလို.ရမှာပါ။ ဒါကအရေးကြီးပါတယ်။ နောက် View,ViewGroup တွေထဲ့ဖို.အတွက် add method ကိုရေးပါတယ်။ ဒါကလဲ ဘာမှမဆန်းပါဘူး။.children list ထဲကိုထဲ့လိုက်တာပါ။ နောက် paint method မှာကျတော့ ကျွန်တော်တို.က Composite ကိုယ်တိုင်ကို paint လုပ်ရပါမယ်။ နောက် သူ.ရဲ. 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("leve2child"));
      viewGroup.add(viewGroup2);
       viewGroup.paint();
   }
}
```

ViewGroup type viewGroup ဆိုပြီး parentGroup တခုဆောက်ပါတယ်။ သူ.ထဲကို View 2 ခုထဲ့ပါတယ်။ နောက်ပြီး ViewGroup တခုဖြစ်တဲ့ childGroup ကိုထဲ့ပါတယ်။ သူ.မှာ 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 တလျောက် paint လုပ်သွားမှာပါ။ ဒါဆိုရင် ဒီ pattern ကိုဘယ်လိုအလုပ်လုပ်တယ် ဘယ်နေရာမှာသုံးတယ် ဆိုတာ သိလောက်ပါပြီ။ လက်တွေ့.ကတော့ Android UI, Swing UI, အစရှိတဲ့ UI တွေမှာသုံးပါတယ်။ နောက်ထပ်သုံးလို.ရတဲ့နေရာကတော့ tree datastructure ဆောက်တဲ့နေရာမှာ ဒီ Composite pattern ကိုသုံးလို.ရတယ်ဆိုတာပါပဲ။ သူ.အားသာချက်ကတော့ ခု paint နေတာသည် Leaf လား composite လားသိစရာမလိုပဲ interface တခုတည်းတဲ့.အလုပ်လုပ်သွားမှာပါ။ ဒီနေရာမှာတော့ paint ပေါ့ဗျာ။

Object Oriented Design Pattern Series Part-9 Decorator Design Pattern

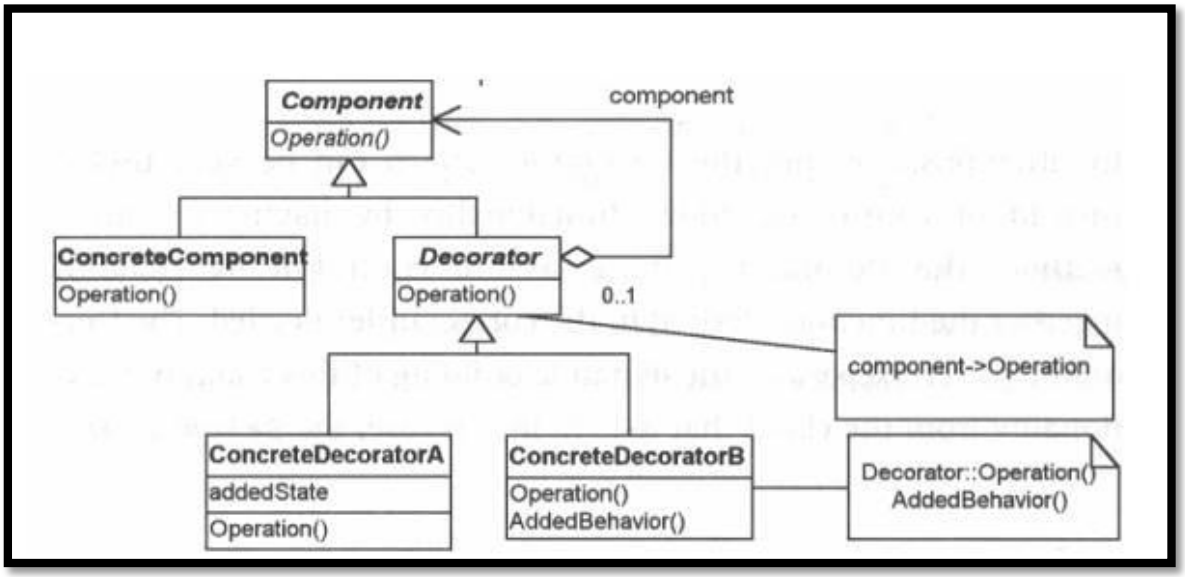
Decorator pattern ကတော့ Sturctural pattern ထဲကတခုပါ။ သူ.ကို အသုံးချပုံကတော့ runtime မှာ object တွေအတွက် responsibility တွေကို dynamically ထဲ့ချင်တဲ့အခါမှာသုံးပါတယ်။ ဒီနေရာမှာ responsibility ဆိုတာ a set of public method (some useful method) လို.ဆိုရမှာပါ။ OO programming မှာ class တခုကို behaviour တွေ ထဲ့ချင်တဲ့အခါမှာ သုံးလေ့ရှိတဲ့နည်းက Inheritance ၊ Composition ကိုသုံးကြပါတယ်။ ဥပမာ ကျွန်တော်တို.က basic logger တခုရှိတယ်ဆိုပါစို.သူက console ကိုပဲ logging ပဲလုပ်နိုင်တယ်။ နောက် HtmlLogger , XMLLogger ထပ်ရှိမယ်ဆိုပါစို. သူတို.တွေကတော့ HTML XML တွေနဲ.ပါ log ရိုက်နိုင်တယ်ဆိုပါစို. ဒါဆို ခုနက Logger တွေမှာ functionality အသစ်ထပ်ထဲ့မယ် ဥပမာ ဗျာရိုးရိုး log message လေးတင်မကပဲ အရှေ့.မှာ Time ထဲ့မယ် ပေါ့ အဲ့ဒီကောင်ကိုHTML Logger မှာရော XML logger မှာရောသုံးစေချင်တယ်။ ဒါဆိုရင် ခုနက Inheritance အရလုပ်မယ်ဆိုရင် TimeAddedLogger ဆိုတာက HTMLLogger XMLLogger ဆိုတာတွေကို extend လုပ်ပစ်ရမှာပါ ဒါဆို ကျွန်တော်တို.က class တွေ ဖောင်းပွကုန်ပါလိမ့်မယ် ။နောက်ပြီး တကယ်လို. JSON ထုတ်တဲ့ logger ဆိုပါစို. အဲ့လိုနောက် Logger class တခုထပ်ထဲ့မယ်ဆိုရင် ခုနက Time ပါပါတဲ့ Logger ကို ထပ် extend လုပ်ရဦးမယ် အဲ့တော့ အဆင်မပြေဘူး ။ ဖြစ်စေချင်တာက ခုနက TimeAddedLogger ဆိုတာကို decorator class တခုအနေနနဲ.ထားလိုက်မယ် ပြီးတော့မှ အဲ့ဒီကောင်ကို တခြား Logger တွေနဲ. tight coupling မဖြစ်စေပဲနဲ. Decorator pattern သုံးပြီး သုံးလိုက်မယ်ဆိုရင် Logger တွေဘယ်လောက်များများ အဆင်ပြေနိုင်ပါတယ်။ နောက်ပြီး Decorator ကလဲ time တခြား functionality တွေပါ ထဲ့ဖို. ထပ်လုပ်နိုင်ပါတယ်။ အဲ့တော့ရှိပြီးသား Logger class တွေကို မထိခိုက်စေပဲနဲ. သူတို. functionality ကို runtime မှာထပ်ခဲ့လို.ရမယ် ။ Tightly coupled လဲမဖြစ်တော့ဘူးပေါ့။ဒါက Decorator ရဲ. intent ပါ။ Decorator က ဘာလဲဆိုတာကို သေချာ define လုပ်ရရင်တော့ object တွေအတွက် additional beahviour ကို runtime မှာ ပေါင်းထဲ့ပေးနိုင်တဲ့ pattern လို.ပဲပြောရမှာပါ။

Intent

သူ. Intent ကိုတော့ GoF မှာ ဒီလိုပြထားပါတယ်။

Attach additional responsibilities to an object dynami cally. Decorat ors provide a flexible alternative to subclassing for extending functionality.

Java IO library မှာဆိုရင် Decorator တွေကိုတော်တော်သုံးထားပါတယ်။ ဥပမာ BufferedInputStream ဆိုတာ Decorator class ပါ။ သူက ဘာကို decorate လုပ်သလဲဆိုရင် InputStream ကို decorate လုပ်ပါတယ်။ နဂိုမူရင်း InputStream မှာ buffer မသုံးပါဘူး။ ဒါကြောင့် သူ.ကိုသုံးရင် နှေးပါတယ်။ BufferedInputStream ကျတော့ buffer သုံးပြီး InputStream ကို decorate လုပ်လိုက်တဲ့အတွက် ပိုမြန်ပါတယ်။ ဒါပေမဲ့ BufferedInputStream သည် InputStream ကို inherits လုပ်ပြီး သုံးတာမဟုတ်ပဲနဲ့. Decorator pattern အနေနဲ့.သုံးတာပါ။ ဒါကြောင့် BuffereedInputStream ကို FileInputStream ကို decorate လုပ်ဖို့.သုံးမယ်ဆိုလဲရသွားပါတယ်။ ဘာလို့.လဲဆိုတော့ FileInputStream သည် InputStream ရဲ့. subclass ဖြစ်နေလို့.ပါပဲ။ ဒီလောက်ဆို Decorator က class hierarchy ကိုမဖောင်းပွစေပဲ tight coupling မဖြစ်စေပဲ ဘယ်လို rutime behaviours တွေကို add လုပ်နိုင်တယ်ဆိုတာကို သဘောပေါက်မှာပါ။ နောက်ပြီး Decorator တွေကို တဆင့်ပြီးတဆင့်သုံးလို့.လဲရပါတယ်။ သူ. class diagram ကိုတော့ အောက်မှာပြထားပါတယ်။



Code ကိုကြည့်ရအောင် အောက်ကတော့ Logger interface ပါ။

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

ဘာမှ ထူးထူးထွေထွေမပါပါဘူး log() ဆိုတဲ့ method လေးပဲထဲ့ထားတာပါ။ ကျွန်တော်တို့. 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 ပို့လာတဲ့ log message ကို return ပြန်ရုံပါပဲ။ ကျွန်တော်တို့က HTML နဲ့ log ထုတ်ချင်တယ် 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။ သူ.ထဲမှာ 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။ သူ.ထဲမှာ log method ကိုသေချာကြည့်ပါ။ Log method မှာ သူက ဒီလိုရေးထားပါတယ်။

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

သူက html start tag ,end tag တွေထဲ့ပြီး decorate လုပ်ပါတယ် အလည်မှာတော့ logger.log ဆိုပြီး သူ.ကို parameter ပေးထားတဲ့ object method ကိုသုံးပါတယ်။ ဒါသည် Decorator ရဲ.အသက်ပါ။ နဂိုမူလက ရှိပြီးသား behaviour ဖြစ်တဲ့ log ကို ကျွန်တော်တို.က html tag တွေထဲ့ပြီး 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 နဲ့.သဘောတရားတူတူပါပဲ။ 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 ကျွန်တော်တို.က decorator တွေကို chain လုပ်ပြီးသုံးသွားတာပါ။ တကယ်လို. HTMLDecorator ကိုမလိုဘူးဆိုရင် ဖြုတ်ပစ်လိုက်လို.လဲရပါတယ်။ ဒါဆိုရင် Decorator သည် ဘာကြောင့် အသုံးတဲ့တယ် ဆိုတာ သဘောပေါက်မယ်ထင်ပါတယ်။

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

နောက်လာမဲ့ JavaScript ES7 မှာ decorator တွေကို propose လုပ်ထားပါတယ်။ Typescript မှာလဲ decorator တွေကို language construct အနေနဲ့ထဲ့ပေးထားပါတယ်။ အဲ့တော့ Angular2 မှာ Decorator တွေကို Component တွေကို decorate လုပ်ဖို့.သုံးကြပါတယ်။ ဆိုချင်တာက language feature အနေနဲ့. pattern တွေကို ပေးထားလိုက်ရင် တော်တော်အသုံးတဲ့မယ် ဒီလို pattern တွေပေါ်မူတည်ပြီး code တွေသီးသန့်.လိုက်ရေးစရာမလိုတော့ဘူးပေါ့ဗျာ။

Object Oriented Design Pattern Series Part-10 Observer Design Pattern

Observer pattern က behaviour pattern ထဲက တခုပါ။ သူ.က GoF 23 မျိုးထဲမှာ တော်တော်သုံးတဲ့ pattern လို.ဆိုရပါလိမ့်မယ်။ MVC (Model View Controller) design pattern ထဲမှာ Observer ကိုထဲ့သုံးထားရပါတယ်။ နောက် Reactive Programming မှာ Reactive stream တွေကို Observer တွေကိုသုံးပြီး implement လုပ်ထားတဲ့အတွက် Reactive Programming ကိုနားလည်ချင်ရင် 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

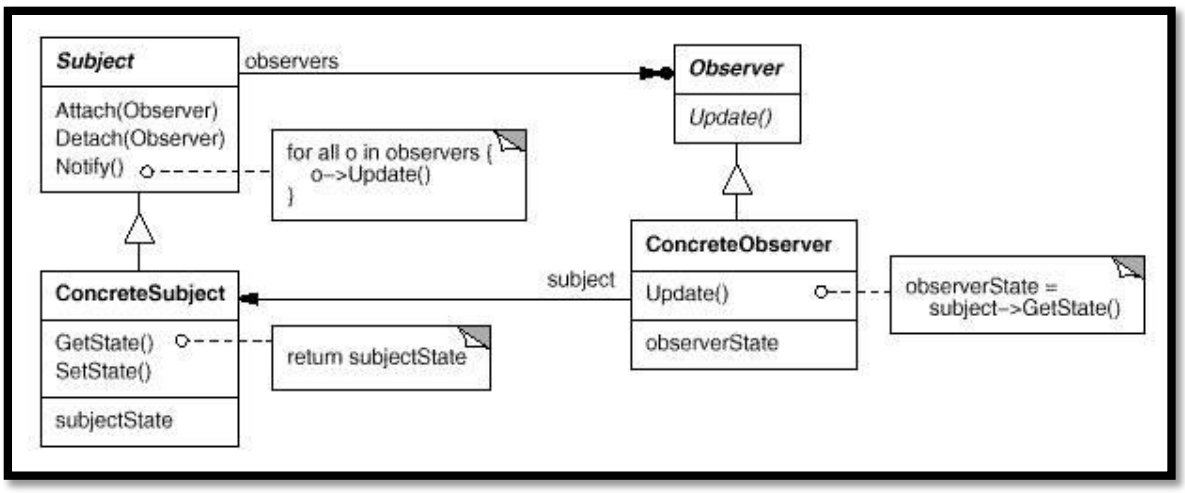
အပေါ်ကတော့ Observer pattern ရဲ. intent ကို GoF မှာရေးထားတဲ့အတိုင်းပြထားတာပါ။ ဆိုချင်တာက one to many dependency ရှိတဲ့ Object တွေကြားက behavriour ကိုထိန်းတဲ့ pattern ပါ။ Object တခုက သူ.ရဲ. state ကိုပြောင်းလိုက်တာနဲ. သူ.ကိုမှီခိုနေရတဲ့ dependency ရှိနေတဲ့ကောင်တွေကို ငါရဲ. state တော့ပြောင်းသွားပြီဆိုပြီး notify (အသိပေးတဲ့) pattern ပါ။ စာကိုစာအတိုင်းရှင်းတော့ရှုပ်ပါတယ်။ ဥပမာပြောရအောင်ပါ။

ဥပမာ ကျွန်တော်တို.မှာ model တခုရှိတယ်ဆိုပါစို.။ Model ဆိုတာ အမှန်တော့ဘာမှမဟုတ်ဘူး business data ကိုသိမ်းထားတာ။ ဆိုပါစို.ဗျာ မြင်သာတဲ့ဥပမာ ပေါ့ stock price ဆိုတဲ့ model ပေါ့။ အဲ့ဒီ Stock price တွေကိုပြဖို.ကြတော့ View တွေလိုတယ်။ View ကလဲကြည့်တဲ့သူအပေါ်မူတည်ပြီး Web ကကြည့်တာရှိမယ်။ နောက် desktop ကကြည့်တာရှိမယ်။(ဒီနေရာမှာ client server တွေထဲ့မစဉ်းစားပဲခနမေ့ထားလိုက်ပါ)။ အဲ့တော့ StockPrice ဆိုတဲ့ model ရဲ. data ပေါ်မူတည်ပြီး View1 View2 ပြရမယ်ဆိုပါစို.။ View1 က StockPrice ကိုပုံစံတမျိုးနဲ.ပြမယ် View2 က StockPrice ကိုနောက်ပုံစံတမျိုး(တူလဲရတာ ကိစ္စမရှိဘူး)ပြမယ်ပေါ့။ ဒါဆိုရင် ခုနက StockPrice ဆိုတဲ့ View1,View2 Object ၂ ခုသည် StockPrice ဆိုတဲ့ model object ကိုမှီခိုနေရတယ်။ Dependence ဖြစ်နေတယ် ဘာလို.လဲဆိုတော့ StockPrice က data ကိုသုံးပြီးတော့ သူတို. View တွေကို render လုပ်ရလို။ StockPrice က တခုတည်း View1,View2 က ၂ခု multi ပေါ့။ ဒါသည် one-to-many relation လို.ဆိုရမှာပေါ့။ နောက်ထပ် View တွေကို dynamically ထဲ့ပြီး 2

ခုထက်မကလဲဖြစ်ချင်ရင်ဖြစ်နိုင်သေးတယ်။ View တွေလျော့သွားတာလဲဖြစ်ရင်ဖြစ်မယ်။ အဲ့တော့ StockPrice ရဲ. data ဟာ အကြောင်းတခုခုကြောင့် ပြောင်းသွားတယ်ဆိုရင် ခုနက သူ.ကိုမှီခိုနေရတဲ့ View1 နဲ. View2 သည် လိုက်ပြောင်းပေးရမှာပါ။ ဒါဆိုရင် View1 နဲ. View2 သည် StockPrice ပြောင်းလား မပြောင်းလားဆိုတာ ကို တချိန်လုံးထိုင်ကြည့်နေရမယ်ဆိုပါစို့။ ဒါဆိုရင် သိပ်အဆင်မပြေပါဘူး။ အဲ့လိုမဟုတ်ပဲ Hollywood principle အရ Don’tc all us , we will call you သုံးရင်ရော ။ဆိုချင်တာကဗျာ View1 ရော View 2ရောက StockPrice ကိုကြည့်မနေပဲနဲ. pulling မလုပ်ပဲနဲ့။ StockPrice ကနေ data change တော့မှ View1 ကိုရော View2 ကိုရော အကြောင်းကြားရင်ရော (notify) လုပ်တယ်ဆိုပါစို့. ။ဒါဆိုပိုအဆင်ပြေပါတယ်။ ဘာလို.လဲဆိုတော့ တချိန်လုံး pulling လို ပြောင်းသွားပြီလား ပြောင်းသွားပြီလား ကြည့်စရာမလိုပဲနဲ. တကယ်ပြောင်းတော့မှ notify (push) လုပ်ပေးတော့ပိုအဆင်ပြေတာပေါ့။ ဒီလို situation မျိုးဆို Observer pattern ကိုအသုံးပြုရပါလိမ့်မယ်။

အတိုချုပ်ပြောရရင် Object တွေမှာ 1-to-many dependency ရှိမယ်။ Object ဒခုကပြောင်းလိုက်တာနဲ. နောက် Object တွေကို effect ဖြစ်မယ် changes လုပ်ရမယ်ဆိုရင် Observer pattern ကိုသုံးရပါလိမ့်မယ်။ MVC မှာ model တွေပြောင်းသွားတာနဲ. 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

လို.ဆိုချင်တာပါ။ ခုနက Example အရဆိုရင်တော့သူကတော့ View ပေါ့ဗျာ။ Subject တခုမှာ Observer တွေအများကြီးပါနိုင်ပါတယ်။ Observer တွေကိုလဲ dynamically add ,remove လုပ်လို.ရရပါလိမ့်မယ်။ Subject က တခုခုပြောင်းသွားတာနဲ. Subject ကို dependence ဖြစ်နေတဲ့ Observer တွေကို notify လုပ်မှာပါ(Observer object တွေရဲ. callback function တခုခုကို လှမ်းခေါ်လိုက်တာပါ) အဲ့တော့မှ Observer တွေက Subject change လုပ်တာကိုသိပြီး သူတို.လိုအပ်တဲ့ function တွေ (ဥပမာ View ကို model subject အပေါ်မူတည်ပြီး လိုက်ပြောင်းတာမျိုး) လုပ်မှာပါ။ နောက်တခုက Observer pattern ကို publish subscribe pattern ရယ်လို.လဲခေါ်ကြပါသေးတယ်။ Subject ကတော့ publish လုပ်တဲ့အတွက် publisher ပေါ့ဗျာ။ Observer ကတော့ Subscriber ပေါ့။ FB ဥပမာနဲ.ဆိုရင် Author နဲ. Follower ပေါ့။ Author ကတော့ Subject ဖြစ်ပြီး Follower တွေကတော့ Subscirber or Observer ပေါ့။ Author ကတခုခု publish လုပ်တာနဲ. Follower တွေဆီမှာလာပေါ်နေမှာပါ။ notify လုပ်တယ်ပေါ့ဗျာ။

Java API မှာ Observer,Observable ဆိုပြီးပါလာပြီးသားရှိပါတယ်။ ကျွန်တော်ကတော့သူတို.ကိုမသုံးပဲ အစအဆုံး pattern ကို implement လုပ်ပါမယ်။

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

ဒါကတော့ Subject သို.မဟုတ် Publisher interface ပါ။ Subject ရဲ.တာဝန်က Subscriber သို.မဟုတ် Observer တွေကို လက်ခံရပါမယ်။ ဒါကြောင့် attach(Subscriber) ဆိုတာကိုရေးထားတာပါ။ နောက် Publisher or Subject သည် data or state change လုပ်နိုင်ဖို.ပါရပါမယ်။ ဒါကို change အနေနဲ.ရေးထားပါတယ်။ နောက် class ကတော့ Subscriber ပါ။ ဒီနေရာမှာ publisher က deattach လဲလုပ်နိုင်ရမှာပါ။ ဒီမှာတော့ခနထားခဲ့ပါမယ်။

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

Subject or Publisher က တခုခု change ပြီဆိုတာနဲ. Subscriber ရဲ. callback method တခုခုကိုခေါ်ရမှာပါ။ ဒါကြောင့်ဒီနေရာမှာ update method ဆိုပြီး callback ထားလိုက်ပါတယ်။ဒီ

update ကို Publisher ကခေါ်မှာပါ။ဒါကြောင့် calllback method (ကိုယ်ကိုယ်တိုင်သုံးတာမဟုတ်ပဲ သူများက ကိုယ့်ကိုပြန်ခေါ်ဖို. ပေးထားတဲ့ method) လို.ဆိုလိုတာပါ။ အောက်က class ကတော့ DataSource ပါ အပေါ်က ပြခဲ့တဲ့ ဥပမာ အရဆိုရင် StockPrice model နဲ. 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 သူ.ထဲမှာ Subscriber တွေကိုသိမ်းဖို.ဒီလိုရေးထားပါတယ်။

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

နောက် attach method မှာ ခုနက subscriberList ထဲကိုဒီလိုထဲ့ပေးလိုက်ရုံပါပဲ။

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

အဲ့တော့ Subscriber ကသူ.ကို connect လုပ်ချင်ရင် attach method ကို လှမ်းခေါ် ပြီး connect လုပ်လိုက်ရမှာပါ။ အများကြီး one-to-many လုပ်ချင်တဲ့အတွက် List နဲ့သိမ်းထားတာပါ။ နောက်အရေးကြီးတဲ့ method က change ပါ။ ဆိုပါစို. Publisher ,Subject ကပြောင်းပြီဆိုရင်

ကျွန်တော်တို့.က သူ.ကိုမှီခိုနေတဲ့ Observer or Subscriber တွေကို လှမ်း notify လုပ်ရပါမယ်။ ဒါကလွယ်ပါတယ်။ change method မှာဒီလိုရေးထားလိုက်ပါတယ်။

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

သဘောကတော့ သူ.ကို subscribe or connect လုပ်ထားတဲ့ subscriber တွေကို သူchanges တခုခုဖြစ်သွားပြီဆိုတာကို လှမ်းအကြောင်းကြားတာပါ။ ဒါဆို Observer , Subscriber တွေက သူတို. data ကိုသူတို.ပြန် ပြောင်းဒါမှမဟုတ် တခုခုလုပ်ရမှာပါ။ Subcriber or Observer ကိုဒီလိုရေးထားပါတယ်။

```
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။တကယ်လို. Subject or Publisher ကပြောင်းတာနဲ. Subject or Publisher က View (Observer or Subscriber)ရဲ. update ကိုလှမ်းခေါ်မှာပါ။ အဲ့မှာ View ကသူ.ဟာသူပြန် render လုပ်တာမျိုးလို.လုပ်ပေါ့။ခုကတော့ output ပဲထုတ်ပြထားတာပါ။ 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 ပေါ့ဗျာ။ 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 ရဲ့.လိုရင်းဖြစ်တဲ့ Object တခု state change တာနဲ့. သူ.ကိုမှီခိုနေတဲ့ကောင်တွေကို 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) တွေလိုက်ပြောင်းတယ်ဆိုတာကိုတွေ့.မှာပါ။

Obeserver pattern ရဲ့.ကောင်းတဲ့အချက်တွေကတော့ Loosely Couple code ဖြစ်တယ် interface နဲ့.ပဲသွားတဲ့အတွက် Subject သည်သူ.ကိုဘယ်သူတွေလာ Subscribe လုပ်ထားလဲ ဘယ် Object တွေလဲသိစရာမလိုဘူး interface သိရင်ရပြီ ဒါက decoupled ဖြစ်စေတဲ့အကြောင်းရင်း ။ Statically attached ဖြစ်မနေဘူး။ runtime မှာ attach deattached လုပ်လို့.ရတယ်။ one-to-many relationship မှာ View တွေသည် dyanmically add or remove

လုပ်လို.ရတယ်။ Data Change တာနဲ. push ကိုသုံးတဲ့အတွက် ပြောင်းပြီလားဆိုတာ လိုက်ကြည့်စရာမလိုဘူး ဒါကြောင့် efficient ဖြစ်တယ်။

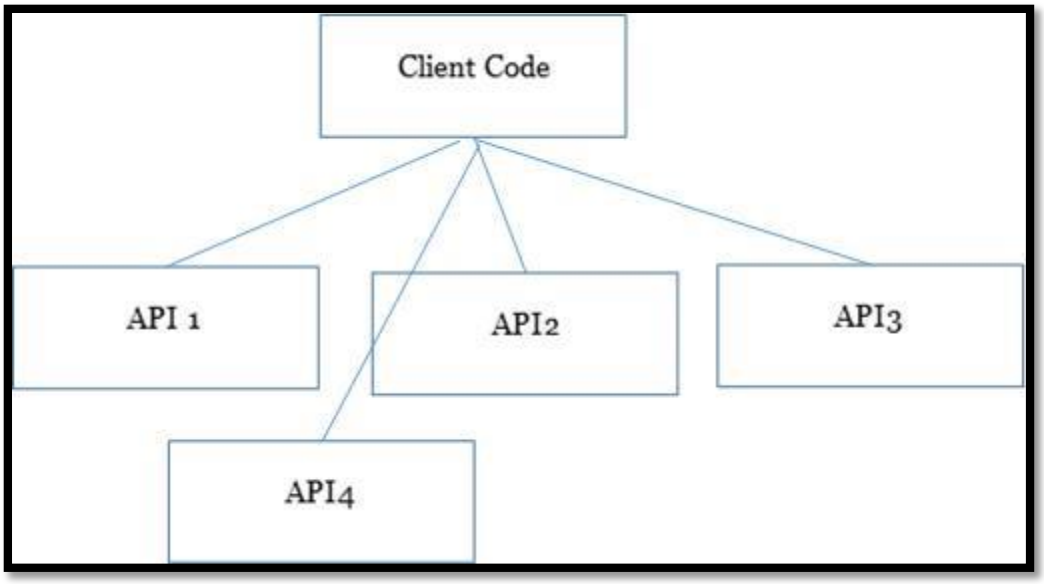
မကောင်းတာတွေကတော့ သတိထားပြီးမသုံးရင် complexity ဖြစ်နိုင်တယ်။ နောက်တခုက deattach မလုပ်မိရင် memory leak ဖြစ်နိုင်တယ်။ ဥပမာ မရှိတော့တဲ့ View ကို deattach မလုပ်မိတာကြောင့် Subject ကနေ call back ကိုလှမ်းခေါ်နေတာမျိုး။

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

Facade က Structural design pattern တွေထဲက တခုပါ။ ကျွန်တော်တို့. code တွေရေးတဲ့အခါမှာ ဆိုပါစို့. module တခုသို့.မဟုတ် package တခုထဲကနေ Class တွေအများကြီးဆီကနေ functionaltiy ကိုတခါတလေ access လုပ်ဖို့.လိုပါတယ်။ ဘာလို့.လဲဆိုတော့ အဲ့ဒီ functionality သည် class တခု API တခုတည်းကနေ ယူသုံးလို့.ဂတာမဟုတ်ပဲနဲ့. တခုထက်ပိုတဲ့ class တွေ API တွေ ကို ချိတ်ခေါ်မှသာရမဲ့ functionaltiy မျိုးဖြစ်နေလို့.ပါ။ အဲ့ဒီကျရင်ကျွန်တော်တို့.သုံးနေတဲ့ client code သည် အောက်ကပုံစံလိုဖြစ်နေမှာပါ။



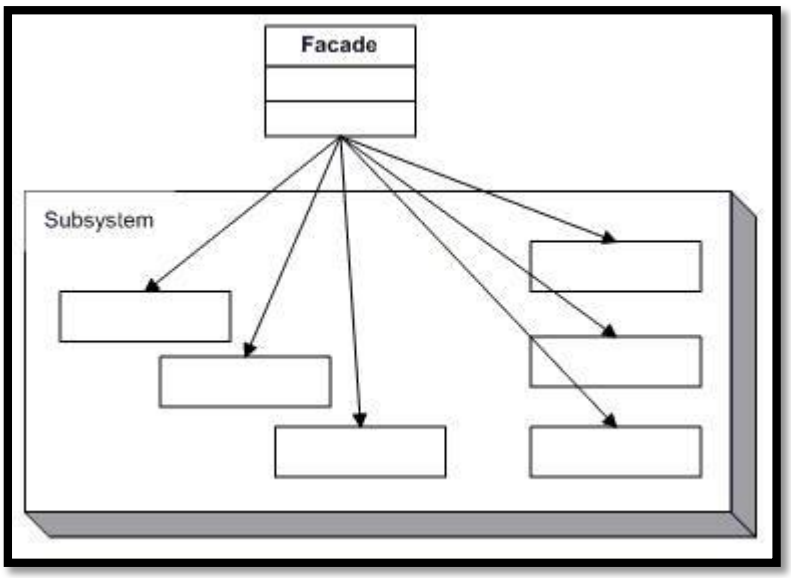
အပေါ်က code ပုံစံမျိုးမှာ ဘာပြဿ နာရှိလဲဆိုတော့ Client သည် API1, 2,3,4 စတဲ့ class တွေကိုတိုက်ရိုက်သုံးနေရတယ်။ ဆိုချင်တာက functionality တခုခု ခေါ် ဖို့.ဆိုရင် ခုနက class 4 ခုလုံးထဲက method တွေကို လှမ်းခေါ်နေရတယ် ဒါဆိုရင် code သည် တော်တော်ရှုပ်သွားပါတယ်။Client ဘက်ကို လိုချင်တဲ့ functionality လေးကို ပေးနိုင်တဲ့ class အသစ်ဆောက်မယ် အဲ့ဒီ class ကနေခုနက API 1,2,3,4 အစရှိတာတွေကို ထပ်ဆင့်ခေါ်သွားမယ်ဆိုရင် client သည် API 1,2,3,4 ကိုတိုက်ရိုက်ထိစရာမလိုတော့ပါဘူး Code တွေလဲရှုပ်ပွစရာမလိုတော့ပါဘူး။ ဒီသဘောတရားကို Facade design pattern လို.ခေါ်ပါတယ်။

Intent

Facade ရဲ. 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

သူ.ရယ်ရွယ်ချက်က ရှုပ်ထွေးနေတဲ့ Class တွေအများကြီးပါတဲ့ method တွေ functionality တွေကို client ဘက်ကနေသုံးလို.အဆင်ပြေအောင် higher level interface တခုထားပြီး ခုနက complex code တွေကို hide လုပ်လိုက်တဲ့သဘောပါပဲ။ အဲ့လိုအခြေအနေရောက်လာရင် Facade ကိုသုံးလို.ရပါတယ်။ Complex class, object interaction တွေကို method တခုသို.မဟုတ် API class တခုအောက်မှာပဲ hide လုပ်ပြီး abstraction ကိုပိုကောင်းအောင်လုပ်လိုက်တဲ့သဘောပါ။ သူ.ရဲ.classDiagramကအောက်မှာပြထားသလိုပါ။ခုနကအပေါ်က
ကောင်နဲ.ဘာကွာသွားသလဲဆိုတော့ client သည် API class တွေကို တိုက်ရိုက်မသုံးတော့ပဲ Facade class ကနေတဆင့်သုံးမှာပါ။ အဲ့တော့ပိုရှင်းသွားသလို decouple လဲပိုဖြစ်သွားစေပါတယ်။



Code ရေးကျရအောင်။ ဆိုပါစို. ကွန်တော်တို. Computer တခု ကို code နဲ. example ရေးမယ်ဆိုပါစို.ဗျာ။ Computer မှာဆို CPU, Memory, HardDisk ရှိမယ်ဆိုပါစို.။ Computer ကိုစဖွင့်ရင် Computer က boot loader ကို run ပြီး OS ကိုခေါ်တင်ရပါတယ်။ ပထမဆုံး HardDisk က boost sector ကိုဖတ်မယ် အဲ့ကနေ 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, အဲ့တာကို Facade class တခုထားပြီးတော့ hide လိုက်မယ်။ ဘာမှအခက်ကြီးမဟုတ်ပါဘူး။ ဒီလို code ရုပါလိမ့်မယ်။

```
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");
   }
}
```

အပေါ်ကCPU, 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။အဲ့methodမှာ bootloading process ရဲ့ complexity တွေကို user အဆင်ပြေအောင် start ဆိုတဲ့ method တခုတည်းနဲ့ hide လိုက်တာပါ။ အဲ့တော့ user က bootloading ကိုသုံးချင်ရင် 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 တို့.ကို တိုက်ရိုက်ယူသုံးတာမဟုတ်တဲ့အတွက် loose coupling ဖြစ်မယ်။အဲ့တော့ easy to change,modify ဖြစ်မယ်ပေါ့ဗျာ။

ရှိနိုင်တဲ့ပြဿနာကတော့ ခုနက CPU, Memory, Harddisk အစရှိတဲ့ subsytem တွေဟာ Facade နဲ့ tightly copuled ဖြစ်နေတဲ့အတွက် သူတို့.ကိုပြင်ရင် Facade ကိုလာထိနိုင်တယ်။ ဒါပေမဲ့ user ဘက်ကနေကြည့်ရင်တော့ Facade interface မပြောင်းမချင်း ဟိုဘက်က subsytem ပြောင်းလဲသိစရာမလိုဘူးပေါ့ဗျာ။

Code တွေကတော့ github မှာသွားယူပေါ့ဗျာ။

https://github.com/mrthetkhine/designpattern

Object-Oriented Design Pattern Series Part-12 Flyweight Design Pattern

Flyweight pattern က structural design pattern တခုပါ။ သူကဘယ်နေရာအမှာအသုံးဝင်လဲဆိုတော့ Object တွေဟာ create လုပ်ဖို့. memory cost ကြီးမယ် (ဆိုချင်တာက object atribute တွေသိမ်းဖို့. ဖြစ်စေ Object တည်ဆောက်ဖို့.အတွက် memory ကြီးတာဖြစ်စေပေါ့) ပြီးတော့ အဲ့ဒီ Object တွေကိုလဲအများကြီးသုံးရမယ်။ ဒါဆိုရင် ခုနက Object create လုပ်ဖို့. costly ဖြစ်တဲ့ကောင်တွေကို cache သဘောမျိုးမှာထားပြီး reuse လုပ်မယ်ဆိုရင် ခနခန Object ပြန်ဆောက်စရာမလိုဘူး။ နောက်ပြီး တခါတလေ မှာ share လုပ်ပြီးသုံးချင်လဲသုံးမယ်။ အဲ့လိုအခြေအနေတွေမှာဆို 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)။ အဲ့တော့ ဘာကောင်းသလဲဆိုတော့ memory အစားသက်သာမယ်။ Heap fragmenation ကိုသက်သာမယ်။ ဘာလို. Heap Fragmentation သက်သာလဲဆိုတော့ Object တွေ ခနခန ဆောက်တာသည် Heap memory မှာ allocation လုပ်ရတယ်။ အဲ့ကျရင် fragementation ဖြစ်တယ် (heap memory သည် အပိုင်းပိုင်း အစိတ်စိတ်လေးတွေဖြစ်သွားတယ် Large collection ပြင်ညီမရတော့ဘူး အဲ့တော့ 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 ရဲ့.သဘောသည် costly ဖြစ်မဲ့ Object တွေ သူတို့.တွေမှာ content တူမယ် ဒါဆိုရင် ခနခန မဆောက်ပဲနဲ့. ဆောက်ဖြီးသား ရှိပြီးသား Object ကို reuse လုပ်တာ ဆိုတာကို သဘောပေါက်မယ်ထင်ပါတယ်။

နောက်ထပ်ဥပမာတခုပေါ့ ကျွန်တော်တို့. Plant and Zoombie ဆော့ဖူးမှာပါ အဲ့ထဲမှာ zoombie တွေပါတယ် plant တွေပါတယ်။ Plant တွေလဲသေတယ်။ zoombie တွေလဲသေတယ်။ ဒါဆိုရင် သေသွားတဲ့ Plant Object တွေ zoombie Object တွေကို GC ဖြစ်အောင် မထားပဲနဲ့. cache ထဲမှာသိမ်းထားမယ် နောက်မှ ပြန်ပြီး reuse လုပ်မယ်ဆိုရင် performance အားဖြင့်သော်လည်းကောင်း speed အားဖြင့်သော်လည်းကောင်း မြန်လာနိုင်ပါတယ်။ နောက်

example တခုကတော့ အဲ့ဒီ Game ထဲမှာပဲ တချို့. Object တွေရဲ. state သည် share လုပ်လို.ရတယ် ဥပမာ နေကြာပန်းရဲ. geomerty location သည် နေကြာပန်းအားလုံးမှာတူတူပါပဲ။ ဒါဆိုရင် ကွာတာက ဘယ်နေရာဆိုတာလေးပဲကွာတာ အဲ့တာဆိုရင် Flywieght သုံးပြီး မတူတဲ့နေရာလေးပဲ ထပ်သုံးမယ်ဆိုရင် large object collection တွေဆောက်စရာမလိုတော့ပါဘူး။

Intention

GoF မှာရေးထားတဲ့ သူ. intention ကဒီလိုပါ။

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

ဆိုချင်တာကတော့ Object တွေအများကြီးမဆောက်ပဲ တူတဲ့ Object တွေဆို sharing လုပ်ပြီးသုံးမယ် ဥပမာ String ဆိုရင် "Hello World" ဆိုတဲ့ String သည် ဘယ်နေရာမှာပါပါ အတူတူပဲ ဒါကို Object အသစ်ဆောက်မနေတော့ပဲ Object တခုတည်းဆောက်ပြီးမျှသုံးမယ်ဆိုရင် application သည် performance ကိုပိုကောင်းလာပါလိမ့်မယ်။

Flyweight ကို အဓိပ္ပာယ်ဖွင့်ရရင် shared object လို.ဆိုရမှာပါ။သူ.မှာ state 2 မျိုးရှိနိုင်ပါတယ်။ intrinsic နဲ. extrinsic ဆိုပြီးတော့ပါ။ Inrinsic ဆိုတာကတော့ Flyweight ထဲမှာပဲသိမ်းတဲ့ state မျိုးပါ ။Share လုပ်လို.ရပါတယ်။ ဥပမာ ခုနက zoombie ထဲက နေကြာပန်းရဲ. ပုံစံကာလာဆိုတာမျိုးက ဂိမ်းထဲမှာရှိတဲ့နေကြာပန်း Object တွေအကုန်မှာအတူတူပါပဲ။ ဘာလေးပဲကွာသွားမလဲဆိုရင် နေကြာပန်းက ဘယ်နေရာမှာရှိသလဲဆိုတော့ physical location on grid လေးပဲကွာမှာပါ(တခြား extrinsic stateတွေရှိနိုင်ပါသေးတယ်)ဒါမျိုးကိုတော့extrinsic ဆိုပြီးခေါ်ပါတယ် ။ သူကတော့ share လုပ်လို.မရပါဘူး။

Flyweight ကိုဘယ်နေရာမှာသုံးလို.ရသလဲ။

အောက်က အခြေအနေတွေအားလုံး ဖြစ်နေပြီဆိုရင် Flyweight ကိုသုံးလို.ရပါတယ်။

Application မှာ object တွေအများကြီးသုံးနေရတယ် (တူညီတဲ့ Object အမျိုးအစားတွေကိုဆိုလိုတာပါ ဒီနေရာမှာ မတူညီတဲ့ object type တွေဆို များလဲ flyweight နဲ့.သုံးလို့.အဆင်မပြေပါဘူး တူတယ်ဆိုတာ type တူတယ်ကိုပြောတာပါ)

Object တွေများလာတာနဲ့.အမျှ memory မှာ storage costတွေပိုများလာမယ်(ဥပမာ android device လို phone လို limit နဲ့.လုပ်နေရတာမျိုးဆိုပိုသိသာမယ်)

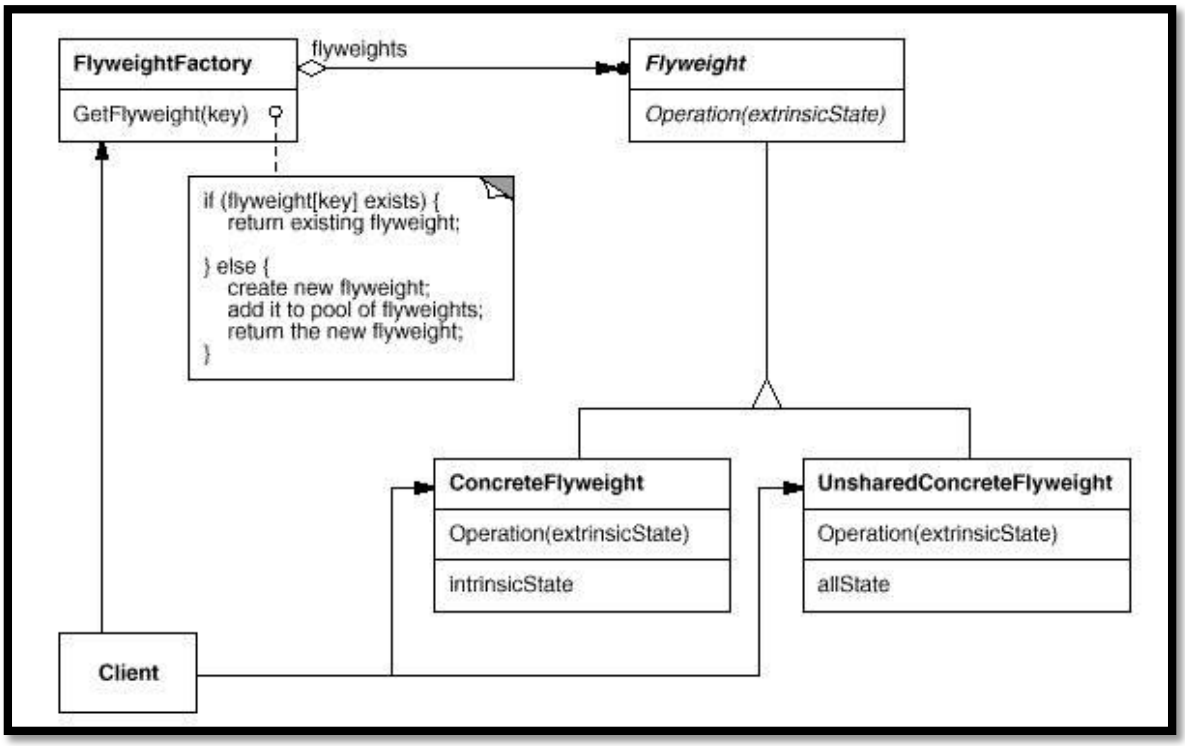
Object ရဲ့. state အများစုကို အပြင်ကိုခွဲထုတ်လို့.ရမယ်။(extrinsic လုပ်လို့.ရမယ် တူတာတွေကို share လုပ်ပြီး မတူတာလေးတွေကိုပဲ အသစ်ယူပြီးသုံးမယ်)

Extrinsic state ကိုဖယ်လိုက်လို့. Object အများစုကို Object အနည်းငယ်နဲ့. အစားထိုးလို့.ဂ်မယ်ဆိုရင်။

Application သည် Object identity အပေါ်မှာ မမူတည်ဘူး ဆိုချင်တာက Object နှစ်ခုသည် တူလားမတူလား စစ်ရတဲ့ operation မျိုးမသုံးဘူး (ဒီနေရာမှာ ဒီအချက်ကြောင့် Java String တွေကို == (object equality operator) နဲ့.မစစ်ရပဲ equals method နဲ့.စစ်ရတာ) ဆိုရင်

အပေါ်ကအချက်အားလုံးဖြစ်နေရင် Flyweight ကိုသုံးလို့.ရပါပြီ။

Flyweight ရဲ့. class diagram ကတော့အောက်မှာပြထားသလိုပါ။



အပေါ်က class diagram မှာအဓိက ကျတာသည် FlyweightFactory ပါ။ သူသည် Factory pattern ကိုသုံးပါတယ်။ Object တွေကို cached လုပ်ပြီးသိမ်းထားတယ် နောက်အဲ့လို object မျိုး request လုပ်ရင် အသစ်မဆောက်တော့ပဲနဲ. cache ကနေပြန်ပေးလိုက်တယ်။ ဒါသည် Flyweight ရဲ.အသက်ပါ။ ကျန်တဲ့ class တွေကတော့ မလိုက်နာလဲ သဘောပါ။ design pattern ဆိုတာ concept သဘောသာဖြစ်တဲ့အတွက် တချို့. pattern တွေမှာ တိုက်ရိုက်ကူးရမယ်ဆိုတဲ့ သဘောမရှိပါဘူး။

Example code ရေးကြရအောင်။

ကျွန်တော်တို.က Java Platform ရယ် .NET platform ရယ်အပေါ်မှာ code တွေ run ကြမယ်ဆိုပါစို. ခုနက Java Platform သော်လည်းကောင်း .NET paltform သော်လည်းကောင်းကို code run ဖို.လုပ်တိုင်းမှာ Object အသစ်တွေပြန်မဆောက်ပဲ Factory ထဲကနေ cache ကနေပြန်ယူသုံးမယ်ပေါ့။ (အမှန်ကဒီနေရာမှာ platform 2 ခုသည် large collection of object ဖြစ်ရမှာ ဒါပေမဲ့ example ကရှုပ်သွားမှာစိုးလို. 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 ကိုအောက်လိုဆောက်မယ်။

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

Platform ၂ခု run မှာဆိုတော့ ဘုံထုတ်ပြီး interface ဆောက်လိုက်တာပါ။ Interface ဆောက်လိုက်တော့ နောက်ထပ် Platform ထပ်ထဲ့ချင်ရင်လဲပိုလွယ်တာပေါ့။

နောက် Platform တခုချင်းဆီအတွက်ရေးမယ်။ Java နဲ. DotNET တခုစီပေါ့။

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

ဘာမှမထူးခြားပါဘူး interface ထဲက method ကို implement လုပ်ထားရုံပါပဲ။ နောက် DotNet အတွက်ကလဲဒီလိုပါပဲ။

```
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 ရဲ.အသက် 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။ Paltform တခု request လုပ်ပြီဆိုရင် map ထဲမှာရှိလား အရင်ထုတ်ပါတယ် ရှိရင်ပြန်ပေးပါတယ် ဒီနည်းနဲ့. sharing ကိုလုပ်သွားတာပါ။ မရှိရင်တော့ Object ဆောက်ပြီး map ထဲကိုထဲ့သိမ်းပါတယ်။ သူ.ကိုသုံးမဲ့ 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 တခုကို create လုပ်မယ်ဆိုရင် PlatformFactory ရဲ့. getInstance ကိုသုံးပါတယ်။ getInstance ကနေ object ဆောက်ပြီးသားဆိုရင် အသစ်ထပ်မဆောက်တော့ပဲရှိပြီးသား Platform ကိုပဲပြန်ပေးမှာပါ။ ဒါဆိုရင် platform တွေ ဥပမာ Java Platform ကိုဘယ်နှခါ request လုပ်လုပ် Object အများကြီးသုံးစရာမလိုပဲနဲ့. တခုတည်းနဲ့. သုံးလို.ရပါပြီ အဲ့ဒီအတွက် application သည် performance အားဖြင့်လည်းကောင်း speed အားဖြင့်လည်းကောင်းပိုမြန်လာမှာဖြစ်ပါတယ်။

https://github.com/mrthetkhine/designpattern

Object Oriented Design Pattern Series Part-13 Proxy Design Pattern

Proxy design pattern က Java မှာရော တခြား langauge တွေမှာ အတော်အသုံးများတဲ့ pattern လို.ပြောရမှာပါ။ Proxy ရဲ.သဘောက Object တခုကို proxy နောက်မှာထားပြီး client သည် real object ကိုတိုက်ရိုက်မထိပဲ Proxy ကနေ ယူသုံးရတာမျိုးပါ။ ဘာကောင်းလဲဆိုတော့ Proxy သည် real object အတွက် access control လုပ်တာမျိုး နောက်လိုအပ်တဲ့ functionality တွေထပ်ထဲ့ပေးတာမျိုးလုပ်ဖို.ရာအတွက်အသုံးဝင်ပါတယ်။ Spring လို framework တွေမှာ AOP (Aspect Oriented Programming) လို functionality မျိုးရဖို.အတွက် JDK dynamic proxy တွေကိုသုံးရပါတယ်။ Interface နဲ.ရေးထားတဲ့ component တွေဆိုရင်တော့ JDK Dynamic proxy နဲ. လိုအပ်တဲ့ code ကို inject လုပ်ပါတယ်။ မဟုတ်ဘူးဆိုရင်တော့ byte code generation library တွေဖြစ်တဲ့ cglib လိုကောင်တွေသုံးပြီး runtime မှာ component တွေကို modified လုပ်ပါတယ်။ ဒါမှသာ runtime ရောက်တဲ့အခါကျရင် controller တွေ component တွေမှာ AOP လိုကောင်တွေသုံးလို.ရမှာဖြစ်ပါတယ်။ ဥပမာ contorller method တွေ မခေါ်ခင်နဲ. ခေါ် ပြီးရင် ဘယ် method ကိုထ run ပေးပါဆိုပြီးလုပ်လို.ရပါတယ်။ User ကတော့ရိုးရိုး POJO လောက်ရေးထားပေမဲ့ runtime မှာ POJO component တွေကို (Interface နဲ. dependecny 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 များသောအားဖြင့်ပေါ့။)

နောက် JavaScript မှာဆိုရင် Proxy တွေကို MVC framework တွေ မှာသုံးလေ့ရှိပါတယ်။ ဥပမာ ကျွန်တော်တို.က ရိုးရိုး JS object တခုကိုဆောက်လိုက်မယ်။ အဲ့ Model object ထဲကနေ Property တခုခုကို write လုပ်တာနဲ. View ကိုပြောင်းပြစ်ရမယ်။ ဒါဆို framework တွေရဲ.အထဲမှာဘယ်လို implement လုပ်ထားလဲဆိုရင် ခုနက User ရဲ. model object ကို proxy နဲ. wrap လုပ်လိုက်တယ်။ User က object.property = something ; အဲ့လိုထဲ့လိုက်တာနဲ. proxy method ကို လှမ်း run မယ်။ Proxy ကနေ View ကို update လုပ်ဖို.ခိုင်းမယ်။ ဒီလိုနေရာတွေမှာလဲ Proxy ကိုသုံးကြတယ်။

အဓိက ကတော့ Proxy ကိုသုံးရခြင်းသည် မူရင်း Object ကို access သို.မဟုတ် functionality အရ ထိန်းချုပ်ဖို. လုပ်ဆောင်တာပဲဖြစ်ပါတယ်။ GOF reference အရတော့ Proxy တွေကိုဒီလိုခွဲချလို.ရပါတယ်။

Remote Proxy

Remote location မှာရှိတဲ့ Object တွေကိုထိန်းဖို.သုံးတဲ့ Proxy မျိုးပါ။ ဥပမာ Java RMI မှာသုံးတဲ့ RMI stub, skelton code တွေသည် Remote proxy တွေပါ သူတို.ကဘာလုပ်ပေးလဲဆိုရင် တခြားစက်ရဲ. JVM မှာရှိတဲ့ Object ရဲ. method တွေကို invoke လုပ်ဖို. လိုအပ်တဲ့ parameter တွေကို marshalling လုပ်ပေးတာမျိုး (searialization နဲ.သဘောတရားတူတူပဲ) နောက် network communication တွေလုပ်ပေးတာမျိုးကိုလုပ်ပေးတယ်။ အဓိက ကတော့ Remote proxy သည် remote location မှာရှိတဲ့ object ကို ထိန်းချုပ်တာမျိုးလုပ်တာပါ။

Virtual Proxy

Virtual Proxy ဆိုတာကတော့ real object သည် ရှိချင်မှရှိနေမယ် ဘာလို.လဲဆိုတော့ သူ.ကို create လုပ်ဖို. memory အရ CPU အရ costly ဖြစ်နေမယ်ဆိုရင် လိုအပ်မှပဲ create လုပ်မယ် ဒါမျိုးကို virtual proxy လို.ခေါ်ပါတယ်။

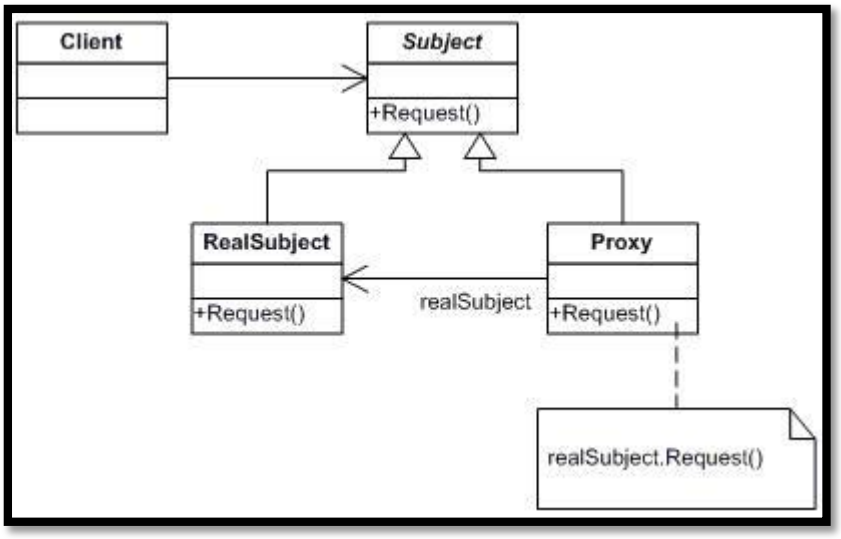
Protection Proxy

သူကတော့ Object တခုရဲ. 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 နဲ.ပြမှာပါ။

Smart Reference

ဒီကောင်ကတော့ အပေါ်က ဥပမာတွေမှာပြသွားတဲ့ Spring တို. Hibernate တို.မှာသုံးတဲ့ proxy တွေလို.ဆိုရမှာပါ။ သူတို.က မူရင်း Object ရဲ. method တခုခု ဒါမှမဟုတ် properties တခုခုကို access လုပ်ပြီ invoke လုပ်ပြီဆိုရင် မူရင်း Object ရဲ. code အပြင် Proxy ကနေ ထပ်ပေါင်းထားတဲ့ code ပါ invoke လုပ်ပေးတဲ့ကောင်မျိုးပါ။ ဥပမာ Model object ရဲ. proxy ဆိုရင် model object ရဲ. state ပြောင်းသွားတာနဲ. view ကို change လုပ်ဖို. code ပါ invoke လုပ်မှာပါ။ ဒါမျိုးကို smart reference proxy လို.ခေါ်ပါတယ်။

အောက်ကတော့ Proxy ရဲ. 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 တခုခုကို runtime မှာ ထိန်းချုပ်တော့မယ် access အရသော်လည်းကောင်း creation အရသော်လည်းကောင်း functionality အရထပ်ဖြည့်တော့မယ်ဆိုရင် Proxy ကိုသုံးလို့.ရပါတယ်။

Example code အနေနဲ့. Protection proxy တခုရေးရအောင်။ 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 တခုအောက်ကလိုဆောက်လိုက်ပါမယ်။

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

သူက command တခုကို execute လုပ်ဖို့.အတွက်ပါပဲ။ RealObject ရောProxy ရော သူ့.ကို implement လုပ်ရမှာပါ။ Access right မရှိပဲ command တွေ run ရင် exeception 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 အစရှိတာတွေနဲ့.တူပါတယ်။ ဒီနေရာမှာ တခုသတိထားရမှာက Realsubject ဖြစ်တဲ့ CommandExecutorImpl သည်သူ.ဆီရောက်လာတဲ့ command ကို ဘာမှမစစ်ပဲ execute လုပ်တယ်ဆိုတာပါပဲ။ စစ်တဲ့တာဝန်ကဘယ်သူကယူရမှာလဲဆိုတော့ Proxy ကပါ။ ဒါကြောင့် Proxy သည် မူလ Object ရဲ. access right ကိုထန်းဖို့.သော်လည်းကောင်း funcitonality ကို add လုပ်ဖို့.သော်လည်းကောင်း သုံးတယ်ဆိုတာ သဘောပေါက်ရမှာပါ။ Proxy code ကအောက်မှာပါ။ သူ.မှာတော့ user, admin ၂ယောက်ခွဲပြီး access right ကိုပါထိန်းပါတယ်။

```
public class CommandExecutorProxy implements CommandExecutor{
private boolean isAdmin;
private CommandExecutor executor;
public CommandExecutorProxy(String user, String pwd){
if("admin".equals(user) && "admmin".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 နဲ. user ကိုခွဲပြီး admin ဆိုရင် isAdmin ထဲကို true ထဲ့ပါတယ်။ နောက် RealSubject CommandExecutorImpl ကိုဆောက်ထားပါတယ်။ နောက် method ဖြစ်တဲ့ runCommand ကတော့စိတ်ဝင်စားစရာပါ။ Access right ကိုသူ.မှာစစ်ပါတယ်။ isAdmin ဆိုရင် command ကို subject ဆီပို့.ပြီး run ပါတယ်။ မဟုတ်လို. rm command ကိုတွေ.ရင် user ဆိုရင် ပေးမ run ပါဘူး။ Execption ကို throw လုပ်ပစ်ပါတယ်။ တခြား command ဆိုရင်ပေး run ပါတယ်။ဒါကို if else နဲ.စစ်ထားတာပါ။ 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 မှာကျွန်တော်တို.က Realsubject ကိုမသုံးရပါဘူး။ Proxy ကိုပဲသုံးရပါတယ်။ Framework တွေမှာတော့ runtime မှာ real object နဲ. proxy တွေကို change ပေးပါတယ်။ အဲ့မှာ 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 ဘာလို. Proxy ကိုသုံးလဲပေါ့. Acces control ကို RealSubject မှာပဲထိမ်းမယ် မရဘူးလားပေါ့။ ရပါတယ်။ ဒါဆိုရင် RealSubject သည် responsibilites 2 ခုဖြစ်သွားပါပြီ။ ဒါသည် OO principle အရ မကောင်းပါဘူး။ နောက်တခုက Acces control ကိုပြောင်းချင်တိုင်းမှာ RealSubject code ကိုထိနေရတာပါ။ ဒါကမကောင်းပါဘူး။ Proxy နဲ.ဆိုထိစရာမလိုပါဘူး နောက်တချက်က လောလောဆယ် ခုရှိတာက Access Control proxy တခုပေါ့ တခြား proxy တွေ access control method တွေ ထပ်ထဲ့ချင်ရင် Hard code လုပ်ထားတဲ့နည်းနဲ.ဆိုအဆင်မပြေဘူး။ AOP မှာ အဓိက key သည် မတူညီတဲ့ logic code တွေကို စုမထားပဲ တနေရာဆီခွဲထုတ်လိုက်တာပါ။ ဒါမျိုးကို Proxy pattern နဲ.မှအဆင်ပြေမှာပါ။

Proxy code ကတော့ဒီမှာ

https://bit.ly/35SoMeG

JDK Dynamic Proxy စိတ်ဝင်စားရင်တော့ code ကဒီမှာ

https://bit.ly/3mSTw5k

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 တခုလို.ပြောရမယ် ။ဘာလို. behavioural design pattern လို.ခေါ်ရသလဲဆိုရင် Object တွေရဲ. runtime မှာ interaction နဲ.ပတ်သတ်ပြီး သက်ဆိုင်တဲ့ design pattern တွေကို behavioural design pattern လို.ဆိုကြတယ်။

COR (Chain of responsibility) pattern က ဘယ်လိုကောင်မျိုးလဲဆိုရင် command request တခုရှိမယ်။ နောက် အဲ့ဒီ command or request ကို process လုပ်ဖို. action handler object တွေသည် chain ပုံစံနဲ.ရှိမယ်။ ပထမဆုံး Command request သည် chain ထဲကထိပ်ဆုံးက Action Handler ကို run မယ် Action Handler က သူ handle လုပ်နိုင်တယ်ဆိုရင် သူနဲ.ပဲပြီးမယ် နောက်ကိုမပို.တော့ဘူး ။ဒါမှမဟုတ်သူလဲ handle မလုပ်နိုင်ဘူး နောက်တချက်က တခြား handler တွေဆက်ပြီး process လုပ်စေချင်သေးတယ်ဆိုရင် chain ထဲမှာရှိတဲ့နောက် handler တခုကိုပို.မယ် ဒီလိုနည်းနဲ. Object interaction လုပ်ပုံကို COR pattern လို.ခေါ်တယ်။ ဥပမာ Linux မှာဆို command တွေဟာ chain လုပ်လို.ရတယ် vertical bar or pipe လေးသုံးပြီး command တခု run ပြီး ရင် နောက် command ကို run မယ် နောက် command သည်ရှေ.က command ရဲ. output ကိုသုံးမယ်။ ဒါသည်လဲ COR pattern ပဲ။ နောက် Java မှာရှိတဲ့ logger , logger တွေသည် hierarchical ထားလို.ရတယ် အဆင့်ဆင့်သွားလို.ရတယ်။ နောက် Servlet မှာရှိတဲ့ filter တွေ ဥပမာ Security တွေကိုထိန်းတဲ့အခါ filter တွေနဲ.ထိန်းတယ် တကယ့် operation ကိုမလုပ်ခင်မှာ sercurity filter တွေ က ထ run တယ်။ ဒါမျိုးသည် COR ရဲ.သဘာဝ အဆင့်ဆင့် process လုပ်တယ်ပေါ့။ နောက် Java, JavaScript မှာရှိတဲ့ object တခုရဲ. method တွေခေါ် ပုံပေါ့။ Object တခုရဲ. method ကိုခေါ်မယ်ဆိုရင် အဲ့ဒီ Object မှာ method ရှိရင် အဲ့ method ကိုသုံးတယ်။မရှိဘူးဆိုရင် parent ကိုဆက်ရှာတယ်။ parent မှာမရှိရင် parent ရဲ. parent ကိုဆက်ရှာတယ်။ ဒါသည် သဘာဝအားဖြင့် COR pattern လို.ဆိုရမယ်။

ခုနက လို အဆင့်ဆင့် လုပ်သွားရမဲ့ လက်လွှဲသွားရမဲ့ object interaction လိုတဲ့အခါ COR ကိုသုံးလို.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 ရဲ. sender နဲ. receiver ကို decouple လုပ်ထားတာ။ ဆိုချင်တာက ဒီ request ကိုဘယ်သူက handle လုပ်ရမယ်ဆိုပြီးသတ်မှတ်ထားတာမဟုတ်ဘူး ။ Request နဲ. handler သည် tightly coupled ဖြစ်မနေဘူး။ Handler chain ထဲက အဆင်ပြေတဲ့ hanlder ကနေ ကောက်ပြီးတော့ handle လုပ်မယ်။ သူလုပ်လို.အဆင်မပြေရင် သူ.နောက်က Handler object ကိုပို.မယ်။ ဥပမာ Browser မှာ event တခုတက်ပြီဆိုပါစို. click event ပေါ့ ဒါဆိုရင် လက်ရှိ target element မှာ client event bind ထားလားကြည့်တယ် ။ဆိုကြပါစို. body ထဲက div ထဲက span လေးကို နှိပ်လိုက်တယ်။ span မှာ click event bind လိုက်ရင် span ရဲ. click event ကို execute လုပ်တယ်။ ဒါနဲ့.ရပ်သလားဆိုတော့မရပ်ဘူး။ သူ.အပေါ် div ကိုလှမ်းပို.တယ်။ div မှာ click event ရှိရင် run မယ် မရှိဘူးဆိုရင် body ဆီကိုပို.တယ်။ဒါကိုကျတော့ browser တွေမှာ event bubbling လို.ဆိုကြတယ်။ သဘောတရားကတော့ COR နဲ.ခပ်ဆင်ဆင်တူတယ်။

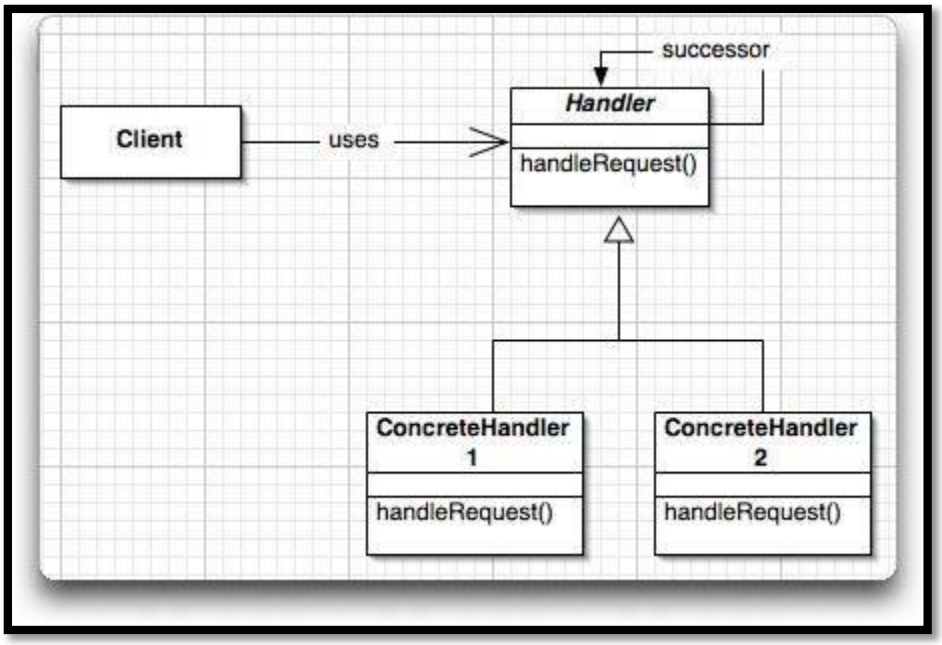
အဲ့တော့ဘယ်လိုအခြေအနေတွမှာCORကိုသုံးရမလဲဆိုရင်အောက်ကအခြေအနေတွေမှာသုံးလို.ရတယ်။

Request ကို handle လုပ်ဖို. Object တခုထက်ပိုရှိနိုင်တယ်။ ဘယ်သူက handle လုပ်ရမယ်ဆိုတာလဲကြိုမသိနိုင်ဘူး။ Handler ကိုလဲ သူ.ဟာသူရွေးပေးရမယ်ဆိုပါစို.

Request တခုကို ဘယ်သူဘယ်ဝါ handle လုပ်ပါမပြောပဲ Handler အများကြီးကို ပို.ချင်ရင်

Handler Object တွေသည် dynamically သက်မှတ်ရမယ်ဆိုရင် (ဆိုချင်တာ runtime ရောက်မှသိနိုင်မယ် နောက်တချက်က အရင်က သူက handle လုပ်ပေမဲ့နောက်သူဖြစ်ချင်မှဖြစ်မယ်။)

အဲ့လိုတွေဆိုရင် COR ကိုသုံးလို.ရပါတယ်။



အပေါ်က တော့ COR ရဲ. class diagram ပုံပါ။ သူ.မှာ handler interface တခုရှိမယ်။ handleRequest ရှိမယ်။ နောက် hanlder တွေကို အဆင့်ဆင့်ချိတ်ထားမယ်။

Example တခုအနေနဲ. sale နဲ.ပတ်သတ်တာတခုရေးကြည့်ရအောင် ဥပမာ ပိုက်ဆံနည်းနည်းဆို Manager က handle လုပ်လို.ရမယ်။ Manager handle လုပ်လို.မရတဲ့ 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 လို.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. hanlde 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.နောက်မှာရှိသေးတယ်ဆိုရင် နောက်ကို request ကိုလွှဲပေးမယ် ဒါသည် COR ရဲ့အသက်ပဲ့ါ။ 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 ပမာဏပဲကျန်တာက သဘောတရားအတူတူပဲ နောက်ထက် VicePredient လဲဒီလိုပဲရေးတယ်ပေါ့ဗျာ ။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 တွေချိတ်သွားပုံကိုကြည့်ပါ manager ရဲ. succesor ထဲကို director ထဲ့တယ်နောက် director ရဲ. succesor အနေနဲ. vice president ကိုထဲ့တယ်။ နောက် request ကိုပို့.တာကျတော့ manager ဆီအရင်ပို့.တယ် Manager က handle လုပ်လို့.ရတဲ့ amount ဆိုရင် သူလုပ်မယ် မဟုတ်ရင် သူ.နောက် က director , vice president အဲ့လိုအဆင့်ဆင့်လွှဲသွားမယ် ။ဒါသည် COR ပါပဲ။

Source code ကတော့ ဒီမှာပါ။

https://bit.ly/399uHhs

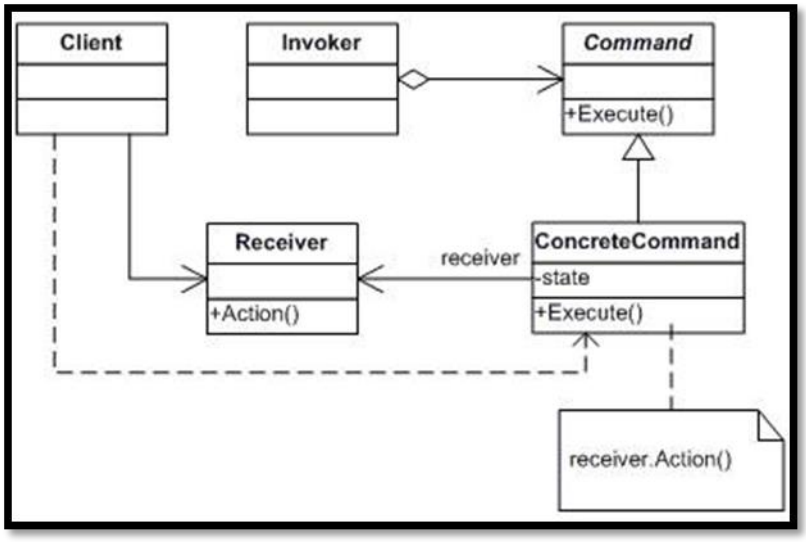
Object Oriented Design Pattern Series Part-15 Command Design Pattern

Command design pattern ဆိုတာ behavioural design pattern တွေထဲကတခုပါ။ သူ.ကိုအတွေ.ရများပါတယ်။ ဘယ်မှာအများဆုံး သုံးကြလဲဆိုရင် UI action တွေမှာ command pattern ကမပါမဖြစ်ပါ။ Java Swing Framework က ActionListener တို. Android က View ရဲ. OnClickListener ကို implement လုပ်တဲ့ကောင်တွေဟာ Command pattern ကိုသုံးထားတာတွေပါ။

Motivation

Command pattern ကိုဘယ်နေရာမှာသုံးလဲဆိုရင် request တခုခုကို (ဥပမာ menu ကိုနှိပ်ရင် လုပ်မဲ့ action တခုခုပေါ့ဗျာ) တယောက်ယောက်ကိုလုပ်ခိုင်းမယ် (button နှိပ်ရင်လဲဖြစ်ရင်ဖြစ်မယ် mouse click ရင်လဲ ဖြစ်ရင်ဖြစ်မယ်) ဒါပေမဲ့ ခုနက command ကိုတကယ် invoke လုပ်မဲ့သူက Command သည်ဘယ်ကလာတယ်ဆိုတာ သိစရာမလိုဘူး။ နောက်တခုက Command ထဲမှာ ဘာတွေပါလာတယ်ဘယ်လို context information တွေပါလာတယ်ဆိုတာ သိစရာမလိုဘူး ။သူ.အလုပ်သည် ပေးထားတဲ့ command ကို execute လုပ်ပေးလိုက်ယုံပဲ။ ဘယ်သူကပို.တာ အဲ့ထဲဘာတွေပါတယ် (parameter or context information) အဲ့လိုအခြေအနေမှာ Command pattern ကိုသုံးမယ်။ မြင်သာအောင်ပြောရရင် button တခုခုကို click ရင် ကျွန်တော်တို.က action တခုခုကို execute လုပ်စေချင်တယ်။ အဲ့ဒါကို command object ထဲမှာထဲ့လိုက်မယ် button နှိပ်လိုက်ရင် ခုနက command ကို handle လုပ်ရမဲ့ object သည် command ကို execute လုပ်ယုံပဲ။ ဥပမာ အဲ့ဒီ action command ကိုပို.တာသည် keyboard action ကနေလဲဖြစ်နိုင်သလို mouse ကနေလဲဖြစ်နိုင်တယ်။ Command ကိုဘယ်သူလွှတ်လဲဆိုတာသိစရာမလိုဘူး။ ဥပမာ Copy command ပေါ့ သူ.ကို keyboard ကနေလဲ လွှတ်နိုင်သလို mouse menu ကနေလဲလွှတ်နိုင်တယ်။ ဒါပေသိအလုပ်က execute လုပ်ယုံပဲ။ ဘယ်သူဆိုတာအရေးမကြီးဘူး run ပေးယုံပဲ။ သူ.ထဲဘာပါလဲ ဆိုတာလဲ သိစရာမလိုဘူး။ အဲ့လိုအခြေအနေမှာ သူ.ကိုသုံးတယ်။ Java ရဲ. ActionListener က actionPerformed သည် command execute လုပ်တဲ့ method ပါပဲ၊ ActionListener ကို implement လုပ်တာသည် command request object ပါပဲ။ အဲ့ဒီ Command Object ကို form control ကိုပို.လို.ရသလို တခြား element တွေကိုလဲပို.လို.ရတယ်။ ဒါလောက်ဆို Command ကိုဘာကြောင့်သုံးရတယ်ဆိုတာ သဘောပေါက်ပါပြီ။

နောက်ဥပမာတခုအနေနဲ့. စစ်ဘက်ဆိုင်ရာ အဖွဲ့.အစည်းတွေမှာ အမိန့်.သည် အထက်ကလာတယ် ဒါပေမဲ့ ဘယ်သူ.ဆီကလာတယ်ဆိုတာ သိချင်မှသိမယ်။ လုပ်ကွာဆို လုပ်လိုက်ရတာပဲ ။ Command pattern သည်လဲ သည်သဘောပါပဲ။ နောက်တခုက Command pattern သည် undoable operation တွေကို လုပ်ဖို. အသုံးဝင်တယ်။ Series of command object တွေကိုမှတ်ထားလိုက်ရင်အဆင်ပြေတာကိုး။နောက်တခုက မတူညီတဲ့ command တွေကို ဘုံထားထားတဲ့ interface သုံးပြီး execute လုပ်လို.ရမယ်။ ဥပမာ keyboard ကလာတဲ့ command ဖြစ်ဖြစ် mouse ကလာတဲ့ command ဖြစ်ဖြစ် အားလုံးကို same interface နဲ့.ထားပြီး execute လုပ်နိုင်မယ်။ ဒါတွေသည် command pattern ရဲ. ရည်ရွယ်ချက်နဲ့.အသုံးဝင်ပုံပါပဲ။ အောက်ကပုံမှာ command pattern ရဲ. class diagram ကိုပြထားပါတယ်။



အပေါ်ကပုံမှာ client သည် command pattern ကိုသုံးမဲ့ class ပါ။ နောက် Comand ဆိုတာက Command request interface သူ့.မှာ execute ဆိုတဲ့ကောင်ပါမယ်။ Command ကို handle လုပ်ရမဲ့ကောင်တွေက execute ကို run ယုံပဲ။ Command သည် Interface ဖြစ်တဲ့အတွက် သူ.ကနေ Concrete Command တွေအများကြီးဆောက်လို.ရမယ်။ ဒါမှအမျိုးမတူတဲ့ command တွေ ဆောက်လို.အဆင်ပြေမှာကိုး။ Invoker ကတော့ Command တွေကို တကယ် handle လုပ်ပေးတဲ့သူ execute လုပ်မဲ့သူပေါ့. သူက Command တွေကို execute လုပ်ယုံတင်မကပဲ history ကိုပါမှတ်ပေးထားပါတယ်။ ဒါမှ undo လုပ်လို.ရမှာကိုး။ Code တွေကတော့ အောက်မှာပါ။

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

ဒါကတော့ Command request interface ပါ ဘာမှမပါပါဘူး Command လုပ်ချင်တဲ့ class တွေသည် သူ.ကို implement လုပ် execute ကို override လုပ်ယုံပါပဲ။ ConcreteCommand class ကိုဒီလိုဆောက်ပါမယ်။

```
public class CopyCommand implements Command{
@Override public void execute() {
System.out.println("Copy Executed");
}
}
```

CopyCommandသည် Command interface ကို implement လုပ်ပြီး execute မှာသူလုပ်ချင်တာလေးကိုရေးယုံပါပဲ။. အောက်က EditCommand သည်လဲဒီလိုပါပဲ။

```
public class EditCommand implements Command{
@Override
public void execute() {
System.out.println("Edit execute");
}
}
```

နောက် Command တွေကို execute လုပ်ဖို. Invoker ကိုရေးပါမယ်။ Invoker သည် history ကိုမှတ်ထားပါမယ်။ ဘာလို.လဲဆိုတော့ undo operation တွေလုပ်ချင်ရင်လုပ်လို.ရအောင်ပါ။

```
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 ကိုသိမ်းဖို. ArrayList ကိုသုံးပါတယ်။ သူ.ရဲ. invoke မှာ command တခုကို invoke မလုပ်ခင် history ထဲထဲ့ပါတယ်။ နောက် command ကို execute လုပ်ပါတယ်။ ဒါဆိုရင် ဒီ invoker ထဲကို မတူညီတဲ့ command တွေကို series အလိုက် ထဲ့ပြီး execute လုပ်ခိုင်းနိုင်ပါပြီ။. 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 ပို.ပို. ဘယ်လို မတူညီတဲ့ command တွေဖြစ်ပါစေ execute လုပ်နိုင်ပါတယ် ဘာလို.လဲဆိုတော့ command pattern ကြောင့်ပါ။ ဒါသည် command ကို issue လုပ်တဲ့သူကို သိစရာမလိုပဲ command ကို execute လုပ်နိုင်ပါတယ်။ Loose coupling ဖြစ်တယ်ကောင်းတယ်လို.ဆိုရမှာပါ။ 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။ နောက်သူတို.ကို undo ပြန်လုပ်တယ်။ ဒါပါပဲ။ Command pattern ရဲ.အသုံးအများဆုံးကတော့ GUI application တွေမှာပါပဲ။ source code ကတော့ အောက်မှာပါ။

https://bit.ly/2J0uUsF

Object Oriented Design Pattern Series Part-16 Strategy Design Pattern

Strategy design pattern က behaviour design pattern တွေထဲက တခုပါ။ သူ.ရဲ. 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.

ဆိုချင်တာက program မှာ interface အားဖြင့်တူညီပေမဲ့ မတူညီတဲ့ Algorithm တွေကိုပြောင်းပြီး သုံးရတဲ့အချိန်ကျရင် strategy ကိုသုံးလို.ရတယ်။ ဥပမာ Sorting စီမယ်။ ဒါပေမဲ့ sorting algorithm က တခုတည်းမဟုတ်ဘူး data အပေါ်မူတည်ပြီး ဘယ် algorithm နဲ့. စီမလဲဆိုတာမျိုး Algorithm ကို dynamically ပြောင်းချင်တယ် ဒါဆိုရင် strategy ကိုသုံးလို.ရတယ်။ နောက် ဥပမာ တခုပေါ့ Payment Gateway တွေ အများကြီးရှိနေတယ် user ရဲ. payment အပေါ်မူတည်ပြီး ဘယ် gateway နဲ့.သုံးမယ်ဆိုပြီးရေးရမယ် ဒီအခါ if နဲ့. hard code မလုပ်ပဲ Strategy pattern ကိုသုံးပြီးရေးလို.ရတယ်။ အဲ့ဒီတော့ algorithm ကိုပြောင်းရလွယ်တယ်။ Strategy pattern ရဲ. ရည်ရွယ်ချက်က family of algorithm တွေကို ပြောင်းပြီးသုံးလို.ရအောင် ရေးထားတာ အဲ့သဘောပဲ။ သူ.ကိုသုံးနေတဲ့ client code သည် နောက်ကွယ်က algorithm ပြောင်းလား မပြောင်းလား သိစရာမလိုပဲနဲ့. သုံးလို.ရအောင် ရေးချင်ရင် strategy ကိုသုံးလို.ရတယ်။

အောက်က အခြေအနေတွေမှာ strategy pattern ကိုသုံးလို.ရတယ်။

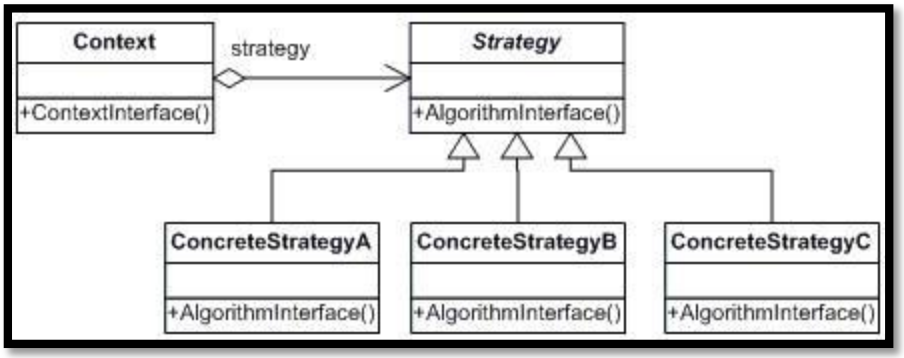
ဆင်တူဖြစ်နေတဲ့ classes တွေသည် behaviours ပိုင်းအရ ပဲကွဲတယ်။ ဥပမာ selection sort, merge sort သည် sorting လုပ်ထုံးလုပ်နည်းသာ ကွာမယ်။ တကယ့်အလုပ်က sorting စီတာပဲ ဒါမျိုးကိုဆိုချင်တာ။ အဲ့လို behaviour မတူတဲ့ class တွေကို တစုတည်း သုံးဖို.အတွက် Strategy ကိုသုံးလို.ရတယ်။

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

Client code က မသိသင့်တဲ့ data တွေ complex operation တွေကို encapsulate လုပ်ဖို.အတွက်လဲသုံးလို.ရတယ်။

တခါတလေ မှာ multiple if တွေနဲ.အများကြီး hard code စစ်နေရမဲ့အချိန်မျိုးမှာလဲ သုံးလို.ရတယ်။ မေးစရာရှိတယ် if သုံးလို.ရနေသားနဲ.ဘာလို. strategy သုံးခိုင်းလဲပေါ့။ Maintenance ကပိုကောင်းသွားတယ်။ If သည် hard code လုပ်ထားတာ အသေ ဖြစ်သွားတယ်။ Strategy ကျတော့ extend လုပ်လို.ရသွားတယ်။ ထပ်ထဲ့လို.အဆင်ပြေတယ်။ ဒါကို Open Close principle လို.ခေါ်တယ်။ open for extension , close for implmentation ကိုပြောတာ။

အောက်က သူ.ရဲ. pattern structure ပုံ။



Context သည် client code သူသည် strategy ကိုထိန်းထားတယ်။ Client ကနေ Context ကိုပဲသုံးမယ်။ strategy ကိုတိုက်ရိုက်မသုံးဘူး။ ဒီနည်းနဲ. encapsulation ကိုထိန်းထားတယ်။ Strategy က interface သူ.ကနေ ConcreteStrategy တွေ အများကြီး ဆောက်ထားလို.ရတယ်။ Interface ကိုသုံးထားပြီး extend လုပ်လို.ရတဲ့အတွက် Open for extension

ကိုပေးပြီးသားဖြစ်သွားမယ်။ ဆိုချင်တာက Algorithm အသစ်တခုထပ်ထဲ့ချင်ရင် strategy ကို implement လုပ်ယုံပဲ။ Context ကနေ Strategy ကို composition အနေနဲ.သုံးတယ်။ ဘာလို. composition ကိုသုံးလဲဆိုရင် ပြောင်းရ ပြုရလွယ်တယ်။ Favor composition over inheritance ဆိုတဲ့ principle အရ ကိုသုံးတာ။

ကဲ Code ရေးရအောင် sorting algorithm တွေလိုမယ်ပေါ့။ Data အပေါ်မူတည်ပြီး ဘယ် Algorithm ကိုသုံးမလဲဆိုတာ ဆုံးဖြတ်ရမယ်။ Algorithm တွေကို runtime မှာ ပြောင်းချင်တယ်ပေါ့။ ဒါကို strategy နဲ့.ရေးရင် ဒီလိုရမယ်။ ပထမဆုံးကတော့ Stategy inferface ဆောက်လိုက်မယ်။

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

သူ.မှာ sort ဆိုတာပဲ ပါမယ်။ interface နဲ့.ထားတဲ့အတွက်ကြောင့် SortStrategy ကို implement လုပ်တဲ့ class တိုင်းသည် SortStrategy မှာ အစားထိုးသုံလို.ရမယ်။ 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 ကိုယူမသုံးပဲနဲ့. interface ဖြစ်တဲ့ SortStrategy ကိုပဲ composition အနေနဲ့.သုံးထားတာ။ အဲ့ဒီအတွက် SortStrategy ကနေဆင်းလာတဲ့ class အားလုံးကို exchange လုပ်ပြီး သုံးလို့.ရမယ်။ 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 ဖြစ်ချင်တဲ့ behaviour ကို class တခုချင်းဆီမှာလိုက်ထဲ့လိုက်တာပဲ။ ဒါပေဲ့ အားလုံးက SortStrategy ကို implement လုပ်ရမယ်။ ဒါမှ context ကနေ သုံးလို့.ရမှာကိုး။ Client code ကတော့ဒီမှာ။

```
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 ရဲ. 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 နဲ. logic ကို hard code လုပ်စရာမလိုဘူး။ Open for extension လုပ်ထားတဲ့အတွက် နောက်ပိုင်း Algorithm တွေထပ်ထဲ့ရတာလွယ်မယ်။ ပြင်ရင်လဲ client code လေးလောက်ပဲထိရမယ်။ အဲ့တော့ maintenance ကောင်းမယ်။ Code ကတော့ဒီမှာ

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