IMPLEMENTATION EXAMPLES WRITTEN IN JAVA

OOP DESIGN PATTERNS

ALL THE PATTERNS ARE EXPLAINED IN

Myanmar Language

Table of Contents

Design Pattern ဆိုသည်မှာ...

ပြဿနာတစ်ခု ဖြစ်တယ်။ ဖြစ်လာတဲ့ ပြဿနာကို ဖြေရှင်းနည်းတစ်ခု ကိုသုံးပြီး ဖြေရှင်းလိုက်တယ်။ နောင်တစ်ခေါက် အဲ့လို ပြဿနာမျိုးထပ်ဖြစ်တယ်။ အရင် ဖြေရှင်းနည်းကို ပြန်သုံးပြီးထပ်ဖြေရှင်းတယ်။ ပြဿနာက ပြေလည်သွားတယ်။ အဲ့လိုမျိုး ထပ်ခါထပ်ခါ ဖြစ်လေ့ဖြစ်ထရှိတဲ့ ပြဿနာမျိုးကို ဖြေရှင်းတဲ့ ဖြေရှင်းနည်းကို လူအများသိအောင် စံတစ်ခုအနေနဲ့ သတ်မှတ်ပြီး စနစ်တကျ အကျယ်ချဲ့ ရှင်းပြထားခြင်း ကို Desing Pattern လို့ခေါ်ပါတယ်။ အလားတူပြဿနာမျိုးကို နှောင်းလူတို့ လွယ်လင့်တကူ ဖြေရှင်းနိုင်စေဖို့ အတွက် ရည်ရွယ်ပါတယ်။

Design Pattern ဆိုတဲ့ concept ကိုဘယ်သူစတီထွင်ခဲ့တာလဲ

1977 ခုနှစ်မှာ Christopher Alexander ရဲ့ “in A Pattern Language: Towns, Buildings, Construction” ဆိုတဲ့ စာအုပ်က စစချင်း design pattern ဆိုတဲ့ concept ကို မိတ်ဆက်ပေးခဲ့တာဖြစ်ပါတယ်။ သူကတော့ မြို့ပြတည်ဆောက်တဲ့ အခါမှာ တွေ့ကြုံရတဲ့ အခက်အခဲတွေကို ဖြေရှင်းပုံ Design patterns တွေကို ရှင်းပြထားပါ တယ်။ ဆိုပေမယ့် သူရဲ့ concept ဖြစ်တဲ့ Design pattern ဆိုတာကို တခြားနယ်ပယ် တွေကပါ အသုံးချလာကြပါတယ်။ Software တွေရေးတဲ့ Software Industry မှာ လည်း 1995 ခုနှစ်မှာ four authors: Erich Gamma, JohnVlissides, Ralph Johnson, and Richard Helm တို့က “Design Patterns: Elements of Reusable Object-Oriented Software” ဆိုတဲ့ စာအုပ်နဲ့ စတင်ပြီး မိတ်ဆက်ပေးခဲ့ကြပါတယ်။ author ၄ ယောက်ဖြစ်တဲ့ အတွက် သူတို့ရဲ့ pattern တွေကို Gang of four : GOF patterns လို့လည်း တွင်တွင်ကျယ်ကျယ် ခေါ်ဝေါ်ကြပါတယ်။ သူ့နောက်မှာတော့ သူ့ သူရဲ့ pattern တွေပေါ် အခြေခံပြီး တခြားစာအုပ် များစွာလည်း ထပ်ထွက်လာပြီး Software development မှာ Desing Patterns တွေကို အသုံးပြုမှုဟာလည်း များလာ ပါတယ်။

Design Pattern တွေကို ဘာကြောင့်လေ့လာသင့်တာလဲ

တစ်ကယ်တော့ developer တစ်ယောက်အနေနဲ့ design patterns တွေတစ်ခုမှ မသိရင်လည်း ရပ်တည်နိုင်ပါတယ်။ ကိုယ်တိုင် မသိဘဲနဲ့ ကိုယ့် code တွေထဲမှာ သုံးမိမှန်းမသိဘဲ design patterns တွေကို သုံးနေမိပါမယ်။ ဆိုပေမယ့် တကယ်လို့ design patterns တွေကို စနစ်တကျ လေ့လာထားမယ်ဆိုရင် ကိုယ်ရေးတဲ့ code ထဲမှာ ဘယ်နေရာမှာ ဘယ် pattern ကိုအသုံးချလိုက်ရင် ပိုကောင်းသွားမယ် ဆိုတာမျိုး ကိုယ့်ဘာသာ မြင်လာပါလိမ့်မယ်။ ဒါ့အပြင် အခြား developer တွေနဲ့ communication ပိုင်းမှာလည်း design pattern တွေက အထောက်အကူပြုပါတယ်။ အသေးစိတ် ရှင်းပြနေစရာမလိုဘဲ ဒီနေရာမှာ "Singleton pattern” သုံးလိုက်ဆိုတာမျိုး အပြန်အလှန်နားလည် လွယ်အောင် အထာနဲ့ ပြောလို့ရလာပါလိမ့်မယ်။

Classification of Design Pattern

အသုံးပြုတဲ့ ရည်ရွယ်ချက်ပေါ်မူတည် ပြီး design pattern တွေကို သုံးမျိုး ခွဲ၍ သတ်မှတ်ထားပါတယ်။

1. Creational Pattern

Object တွေ တည်ဆောက်တဲ့ အပိုင်း (Object creation) ကို အထောက်အကူပြုတဲ့ patterns တွေကို creational pattern လို့သတ်မှတ်ထားပါတယ်။

eg. Factory Pattern, Singleton Pattern

2. Structural patterns

Object တွေ Class တွေကို ပိုမို ကြီးမာလာအောင် ပြုလုပ်တဲ့ နေရာ တနည်းအားဖြင့် ဆိုရလျှင် Object တွေ Class တွေကို စုစုစည်းစည်း ပြုလုပ်တဲ့နေရာမှာ အသုံးပြုရတဲ့ pattern တွေကို Structural Pattern လို့သတ်မှတ်ထားပါတယ်။

eg. Decorator Pattern, Adaptor Pattern, Facade Pattern, Composite Pattern, Proxy Pattern

3. Behavioral patterns

Object တစ်ခုနှင့်တစ်ခုကြားက ဆက်သွယ်မှု နဲ့ အချင်းချင်း အလုပ် ခွဲဝေမှု ကို အထောက်အပံ့ပြုလုပ်ပေးတဲ့ pattern တွေကို Behavioral Pattern လို့ခေါ်ပါတယ်။

eg. Strategy Pattern, Observer Pattern, Command Pattern, Template Method Pattern, Iterator Pattern, State Pattern

How to read this book

ဤစာအုပ်တွင် စစချင်း OOP အကြောင်းအကျယ်တဝင့် ရှင်းပြထားပါသည်။ OOP Concept ကို သိနားလည်ထားပြီးသူဖြစ်လျှင် Design Pattern တွေကို ကျော်၍ ဖတ်နိုင်သည်။ နောက်ထပ် ဤစာအုပ်၏ Design Pattern တွေကို ဖွဲ့စည်းထားပုံသည် Head First: Design Patterns စာအုပ် first edition နှင့်တူညီသည်။ ဤစာအုပ်သည် Head First စာအုပ်မှ တသွေမတိမ်း ကူးချထားသည်မျိုးမဟုတ်ပါ။ Head First မှ example များထက် ပိုမိုနားလည်လွယ်အောင် တကယ်လိုအပ်သည့် concept များလည်း မကျန်ခဲ့အောင် ဂရုတစိုက် example များကို ဖန်တီး၍ရေးသားထားပါသည်။ ဆိုပေမယ့် ဤစာအုပ်ရှိ example များသည် အနှစ်ချုပ်ပုံစံမျိုး ဖြစ်သောကြောင့် ပထမ Head First: ရှိ chapter တစ်ခုကို ဖတ်ပါ။ ထို့နောက်မှ ဤစာအုပ်ရှိ chapter ကို ဆက်ဖတ်လျှင် ပိုမို ပြည့်ပြည့်စုံစုံ နားလည်သွားပါမည်။ စာအုပ်နှစ်အုပ် ကို ယှဉ်ဖတ်ရန် တိုက်တွန်းလိုပါသည်။

OOP ဆိုသည်မှာ....

OOP မတိုင်ခင်မှာ Program တွေ ရေးတဲ့ အခါမှာ သုံးခဲ့တဲ့ နည်းစနစ် ကတော့ Procedural ဆိုတဲ့ နည်းစနစ် ဖြစ်ပါတယ်။ Procedural ကို မြန်မာမှုပြုမယ်ဆိုရင် တော့ အစီအစဉ်လိုက် ဆိုတဲ့ သဘောပေါ့။ နာမည် အတိုင်းဘဲ အစီအစဉ်တကျ အပေါ် ကနေ အောက် code တွေကို တလိုင်းချင်းစီ ရေးရမယ် execute လုပ်ရင် လည်း အစီ အစဉ် တကျ execute လုပ်သွားမယ်။ သူရဲ့ အားသာချက်က OOP လိုမျိုး Design ပိုင်း ကို စဉ်းစားဖို့ အချိန် မပေးရတဲ့ အတွက် ကြောင့် မြန်မြန် ပြီးနိုင်တယ်။ ဆိုပေမယ့် တခုခု error ဖြစ်တဲ့ အချိန် code ကို ပြန် ပြင်ချင်တဲ့ အချိန် မျိုးမှာ အစကနေ အဆုံး code အားလုံးကို ဖတ်ရတဲ့ အတွက် အချိန် ကြာတယ်။

Program ကြီးလေ အချိန် ပိုကြာလာနိုင်တဲ့ အတွက် ခဏခဏ ပြင်ဖို့လိုအပ်တဲ့ Program မျိုးမှာ Procedural စနစ်နဲ့ ရေးလို့မရတော့ဘူး။ အဲ့ဒီ့ အတွက် OOP ဆိုတဲ့ ပြုပြင်ရလွယ်တဲ့ နည်းစနစ် ကို တီထွင် ပြီးအသုံးပြုခဲ့ကြတယ်။ ပုံမှန်အားဖြင့် တော့ ဘယ် OOP ကို support ပေးတဲ့ language မှာမဆို Procedural စနစ် ကို ဘဲ သုံးပြီး ရေးမယ်ဆိုလည်း ရပါတယ်။ OOP စနစ်ဆိုတာ တကယ်တော့ Procedural ကို အဆင့် မြှင့်ထားတဲ့ သဘောပါဘဲ။ တစ်နည်းအားဖြင့်ဆိုရလျှင် Procedural မှာမပါတဲ့ အရာ လေးခု ဖြစ်တဲ့ (“Encapsulation”, “Abstraction”, “Inheritance”, “Polymorphism”) တို့ကို ထပ်ပေါင်းထည့်ထားခြင်းဘဲဖြစ်ပါတယ်။

၁။ Encapsulation (Security)

Encapsulation "ထုပ်ပိုးမှု" ဆိုသည်မှာ ဆေးအမှုန့်လေးတွေကို ရာဘာအတောင့် လေးထဲ ထည့်ထားသလိုပါဘဲ။Variable လေးတွေနဲ့ method တွေကို တစ်ခုတည်းဖြစ်အောင်စုစည်းထားခြင်းပါ။ ဆေးမှုန့် ကတော့ ပြင်ပလေထုနဲ့ ထိတွေ့ရင် အာနိသင်ပြောင်းသွား နိုင်သောကြောင့်ထည့်ထားခြင်း ဖြစ်ပါတယ်။ ထိုနည်းတူပင် variable တွေကို ပြင်ပကနေ အလွယ်တကူ ပြုပြင်ချင်းမပြုနိုင် ရန် ကာကွယ်ထားခြင်း ဖြစ်ပါတယ်။

Example

Game တစ်ခုရဲ့ level တိုးတဲ့ ပုံစံလေးနဲ့ ဥပမာပြပေးထားပါတယ်။

```
class Game {
    int level;
}

public class Test {
    public static void main (String [] args){
        Game latestGame = new Game();
        latestGame.level = 10;
        latestGame.level = -10;
        System.out.println("Your level is : " +
         latestGame.level );
    }
}
```

အပေါ်က code ကို ကြည့်ရင် Encapuslation concept ကို အသုံးမချထားတဲ့အတွက် စစချင်း မှာ level 10 ကို တန်းပြီးသွင်းလိုက်လို့ရသလို မဖြစ်နိုင်တဲ့ “-” နဲ့စတဲ့ level မျိုးလည်း သွင်းလိုက်နိုင်ပါတယ်။ အဲ့ပြဿနာကို ဖြေရှင်းဖို့ Encapsulation concept ကို သုံးထားတဲ့ code ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
class Game {

    private int level;

    public void increaseLevel() {
        level = level + 1;
    }

    public int getLevel() {
        return level;
    }

}

public class Test {
    public static void main(String[] args) {
        Game latestGame = new Game();
        latestGame.increaseLevel();
        latestGame.increaseLevel();
        System.out.println("Your level is : " +
        latestGame.getLevel());
    }
}
```

အပေါ်က code မှာ Game class ရဲ့ level variable ကို private ကြေငြာထားပြီး private variable ကို အပြင်ဘက်ကနေ access လုပ်နိုင်ဖို့ public methods နှစ်ခု ထပ်ရေးထားတယ်။ Test class မှာ အရင်ကလို တိုက်ရိုက် variable ကို access မလုပ်နိုင်ဘဲ method တွေကနေဘဲ access လုပ်နိုင်တဲ့အတွက် security ပိုင်းပိုကောင်းသွားပြီး မလိုအပ်တဲ့ အမှားတွေမဖြစ်နိုင်အောင် ထိန်းသွားနိုင်တာကို တွေ့ရပါလိမ့်မယ်။ အပေါ်ကလို variable တွေကို private ပြောင်းထားပြီး public method တွေကနေတဆင့် access လုပ်ခြင်းကို Encapsulation လို့ခေါ်ပါတယ်။

Abstraction (Design)

Abstraction "အကျဉ်းချုံ့ခြင်း" ဆိုသည်မှာ အသေးစိတ်ဖြစ်နေတဲ့ class အများကြီးကို အသုံးပြုရလွယ်ကူအောင် သဘောသဘာဝ တူညီရာ class တွေအတွက် abstract class တစ်ခုကို parent အဖြစ်ထားပြီး ထို parent ကို ပြန်ခေါ်သုံးအားဖြင့် code complexity ကို လျော့ကျအောင်လုပ်သည့် ပုံစံမျိုး ဖြစ်ပါသည်။

Example

အပေါ် Encapsulation က example လိုမျိုး game နဲ့ ဆိုင်တဲ့ example တစ်ခုနဲ့ဘဲ ရှင်းပါမယ်။ စစချင်း Abstraction concept ကို မသုံးထားတဲ့ code ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
class Game {
    void move(Male man) {
        man.walk();
    }
    void move(Female woman) {
        woman.walk();
    }

}

class Male {
    void walk() {
        System.out.println("Man is walking!");
    }
}

class Female {
    void walk() {
        System.out.println("Woman is walking!");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Male man = new Male();
        Female woman = new Female();
        Game game = new Game();
        game.move(man);
        game.move(woman);
    }
}
```

အပေါ်က Game class ရဲ့ move method နှစ်ခုကို တစ်ခုတည်းဖြစ်သွား အောင် Abstraction concept သုံးထားတဲ့ code ကိုအောက်မှာဖော်ပြပေးထားပါတယ်။

```
class Game {
    void move(Player player) {
        player.walk();
    }


}

abstract class Player {
    abstract void walk();
}

class Male extends Player {
    void walk() {
        System.out.println("Man is walking!");
    }
}

class Female extends Player {
    void walk() {
        System.out.println("Woman is walking!");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Male man = new Male();
        Female woman = new Female();
        Game game = new Game();
        game.move(man);
        game.move(woman);
    }
}
```

Male ကော Female ကိုပါ ယေဘူယျ abstract class တစ်ခုကို extends လုပ်ခိုင်းထားလိုက်ပါတယ်။ ပြန်ခေါ်သုံးတဲ့အခါမှာ အဲ့ abstact class ကို data type အဖြစ်သုံးလိုက်တဲ့ အတွက်ကြောင့် နဂိုရ်တူညီနေတဲ့ Game class ရဲ့ move methods နှစ်ခုနေရာမှာ တစ်ခုတည်း သုံးလို့ ရသွားပါတယ်။ အခုလိုမျိုး Male နဲ့ Female ကို Player အဖြစ်ချုံ့လိုက် ခြင်းကို Abstraction လို့ခေါ်ပါတယ်။ သတိထားရမှာ က Abstaction concept အတွက် abstract class တစ်ခုတည်းကိုသာ သုံးနိုင်တာမျိုး မဟုတ်ပါဘူး။ Interface တို့ Inheritance တို့ နဲ့လည်း Abastraction concept ကို လုပ်ယူနိုင်ပါသေးတယ်။

၃။ Inheritance (Code Reuse)

Inheritance "အမွေဆက်ခံခြင်း" ဆိုသည် မှာ ပြင်ပမှာ အမွေဆက်ခံသကဲ့ သို့ ပင်ဖြစ်သည်။ မိဘအမွေကို သားသမီးက ရသကဲ့သို့ပင် နဂိုရေးထားပြီးသား class အတွင်းရှိ variable တွေ method တွေ ကို ထပ်ရေးစရာ မလိုဘဲ အမွေဆက်ခံလိုက်သည်နှင့် အမွေဆက်ခံသည့် class က အလိုအလျောက် ရရှိသွားသည့် သဘော ဖြစ်သည်။

Example

အပေါ် Abstraction ရဲ့ code ကိုဘဲ ထပ်ဖြည့်ပြီးရှင်းပါမယ်။

```
class Game {
    void move(Player player) {
        player.walk();
    }
}

abstract class Player {
    abstract void walk();
}

class Male extends Player {
    void walk() {
        System.out.println("Man is walking!");
    }
}

class Boxer  {
    void walk() {
        System.out.println("Man is walking!");
    }

    void fight() {
        System.out.println("Boxer is fighting!");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Male man = new Male();
        Boxer boxer = new Boxer();
        boxer.walk();
        boxer.fight();
    }
}
```

အပေါ်က code မှာ Boxer class တစ်ခုထပ်ဖြည့်ထားပါတယ်။ Boxer class ကော Male class မှာပါ တူညီနေတဲ့ walk method ကို ထပ်ခါထပ်ခါ သုံးထားတာ (code duplication ဖြစ်နေတာ) ကို တွေ့ရပါတယ်။ Inheritance ကို သုံးပြီး code duplication ကို လျော့ချပါမယ်။

```
class Game {
    void move(Player player) {
        player.walk();
    }
}
abstract class Player {
    abstract void walk();
}


class Male extends Player {
    void walk() {
        System.out.println("Man is walking!");
    }
}

class Boxer extends Male {
    void fight() {
        System.out.println("Boxer is fighting!");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Male man = new Male();
        Boxer boxer = new Boxer();
        boxer.walk();
        boxer.fight();
    }
}
```

အပေါ်မှာ highlight ပြထားတဲ့ အတိုင်း Boxer class မှာ Male class ကို extends လုပ်ခြင်းအားဖြင့် Male class ရဲ့ method တွေကို ထပ်ရေးစရာ မလိုတော့ဘဲ သုံးလို့ရသွားတာကို တွေ့ရပါတယ်။ ထိုကဲ့သို့ ပြုလုပ်ခြင်းကို "Inheritance" လုပ်တယ် လို့ခေါ်ပါတယ်။

၄။ Polymorphism (Method Overriding)

Polymorphism "ပုံစံမျိုးစုံ" ဆိုသည်မှာ method တစ်ခုရဲ့ implementation မျိုးစုံ ရှိနေတာကို ဆိုလိုသည်။ Inheritance ကို function ထပ်တိုးလိုက် ခြင်းပင် ဖြစ်သည်။ ပုံမှန် အမွေဆက်ခံ လျှင်ရရှိလာမည့် method တစ်ခုကို အမွေဆက်ခံမည့် class ၏ method နှင့် အစားထိုးလိုက်ခြင်းပင် ဖြစ်သည်။

Example

အပေါ်က code နဲ့ဘဲ ဆက်ရှင်းပါမယ်။ အခုအချိန်မှာ အပေါ်က Boxer class က parent class ရဲ့ walk() method ကို တိုက်ရိုက်သုံးတယ် ဆိုပေမယ့် နောက်ပိုင်း အပေါ်က parent class ကို မသုံးချင်တာမျိုးလည်း ဖြစ်နိုင်ပါတယ်။ အဲ့လို အခါမျိုးမှာ Boxer class ထဲမှာ ကိုယ်ပိုင် walk() method ကို ရေးပေးလိုက်ရုံပါဘဲ။ ခေါ်တဲ့အခါမှာ child class ထဲက method ကိုဘဲ ဦးစားပေး ခေါ်သွားပါလိမ့်မယ်။ ထိုကဲ့သို့ ပုံစံမျိုးကို polymorphism ဟုခေါ်ပါသည်။ code နမူနာကို တဖက်စာမျက်နှာမှာ ဖော်ပြပေးထားပါတယ်။

```
class Game {
    void move(Player player) {
        player.walk();
    }
}
abstract class Player {
    abstract void walk();
}


class Male extends Player {
    void walk() {
        System.out.println("Man is walking!");
    }
}

class Boxer extends Male {
    void walk() {
        System.out.println("Boxer is walking!");
    }
    void fight() {
        System.out.println("Boxer is fighting!");
    }
}

public class Test {
    public static void main(String[] args) {
        Male man = new Male();
        Boxer boxer = new Boxer();
        boxer.walk();
        boxer.fight();

    }
}
```

ဒီအတိုင်း တစ်ခုချင်းစီ ကြည့်လျှင်တော့ ထူးခြားမှုမရှိပေမယ့် အပေါ်က feature လေးခု (“Encapsulation”, “Abstraction”, “Inheritance”, “Polymorphism”) ကို စနစ်တကျ ပေါင်းစပ် နိုင်လျှင်တော့ ဘယ်လောက်ကြီးသည့် program မဆို ရေရှည် မှာ ပြုပြင် ထိန်းသိမ်းရလွယ်အောင် ပြုလုပ်၍ရပါသည်။

Strategy Pattern ဆိုသည်မှာ...

OOP Design Pattern တွေထဲက တစ်ခုပါ။ Design Pattern ဆိုတာက အဖြစ်များတဲ့ ပြဿနာတစ်ခုကို ဖြေရှင်းနိုင်တဲ့ ဖြေရှင်းနည်းတစ်ခုဘဲ ဖြစ်ပါတယ်။ကျတော်တို့ software development ဘက်မှာ အဖြစ်များတာကတော့ software တွေကို နေ့စဉ်နှင့် အမျှ ပြောင်းလဲမှု တွေကို အဆင်ပြေပြေ လုပ်ဆောင် နိုင်အောင် တနည်းအားဖြင့် တစ်ခု ပြင်လိုက်ရင် တခြားနေရာမှာ error မဖြစ်အောင်ထိန်းသိမ်းရတာ အဲ့လိုမျိုး ပြဿနာ တွေကို ဖြေရှင်းဖို့ အရင်လူတွေသုံးခဲ့တဲ့ နည်းတွေကို ပြန်သုံးနိုင်အောင် စုစည်းထားတဲ့ နည်းတွေကို ခေါ်ပါတယ်။ Stratety Pattern ဆိုတာက လည်း Design Pattern တွေထဲက တစ်ခုဘဲဖြစ်ပါတယ်။

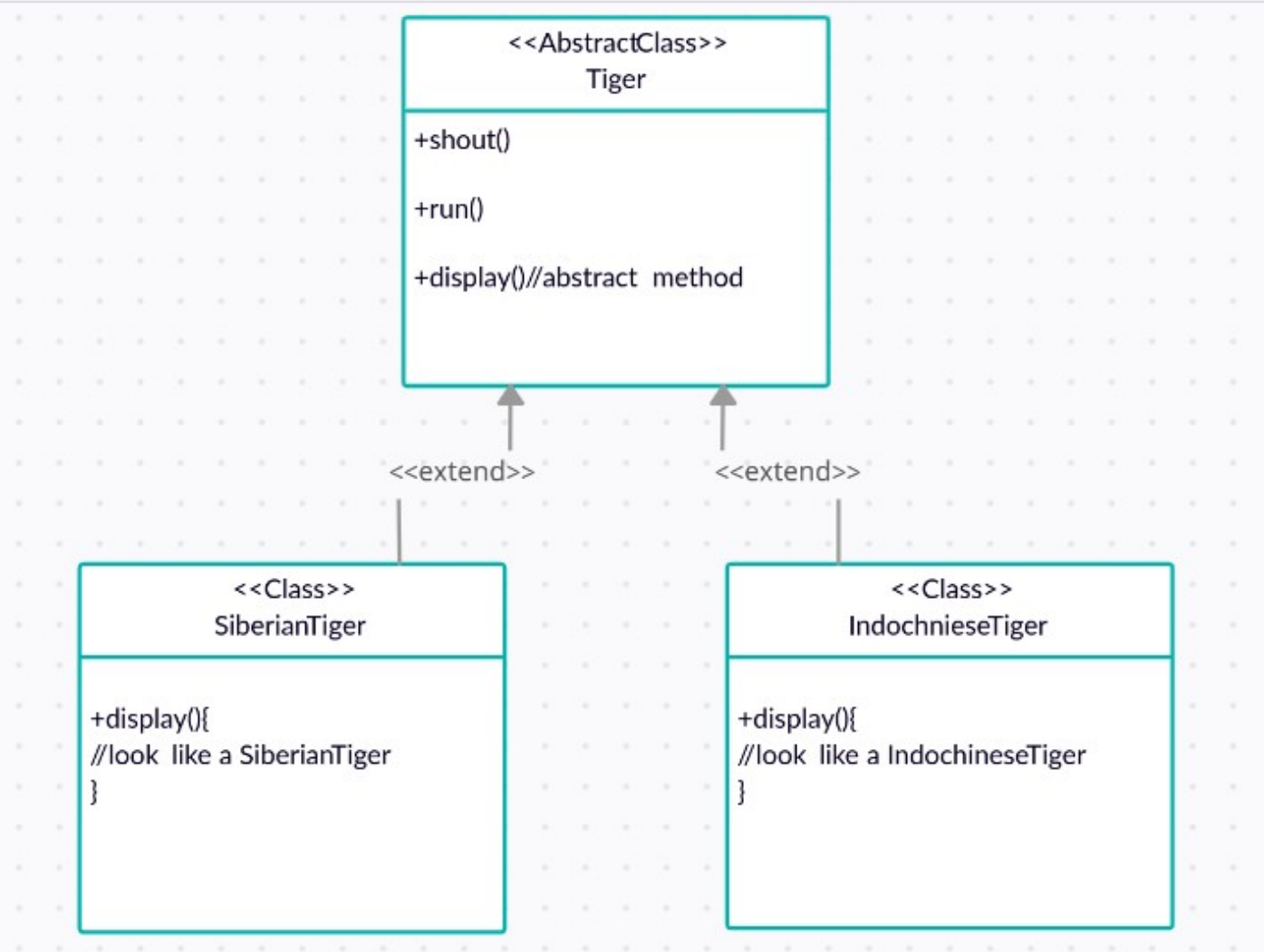
Definition of Strategy Pattern

Code ကို နှစ်ပိုင်းခွဲမယ်။ ပြောင်းလဲလေ့ရှိတဲ့ အပိုင်းနဲ့ မပြောင်းတဲ့ အပိုင်းဆိုပြီး နှစ်ပိုင်း အရင် ခွဲမယ်။ ပြီးရင် ပြောင်းတဲ့အပိုင်း သက်သက် မှာဘဲ ပြင်စရာရှိတာပြင်မယ်။ အဓိပ္ပာယ်ဖွင့်ဆိုချက်ကတော့ အဲ့လောက်ပါဘဲ ဆိုပေမယ့် ဥပမာလေး တစ်ခုနဲ့ တွဲကြည့်မှ သေချာ သဘောပေါက်မှာမို့လို့ ဥပမာတစ်ခုနဲ့ ရှင်းပါမယ်။

diagrams တွေပါဝင်တဲ့အတွက် images အဖြစ်ပြောင်းပြီး post နဲ့တွဲထားပါတယ်။

Example

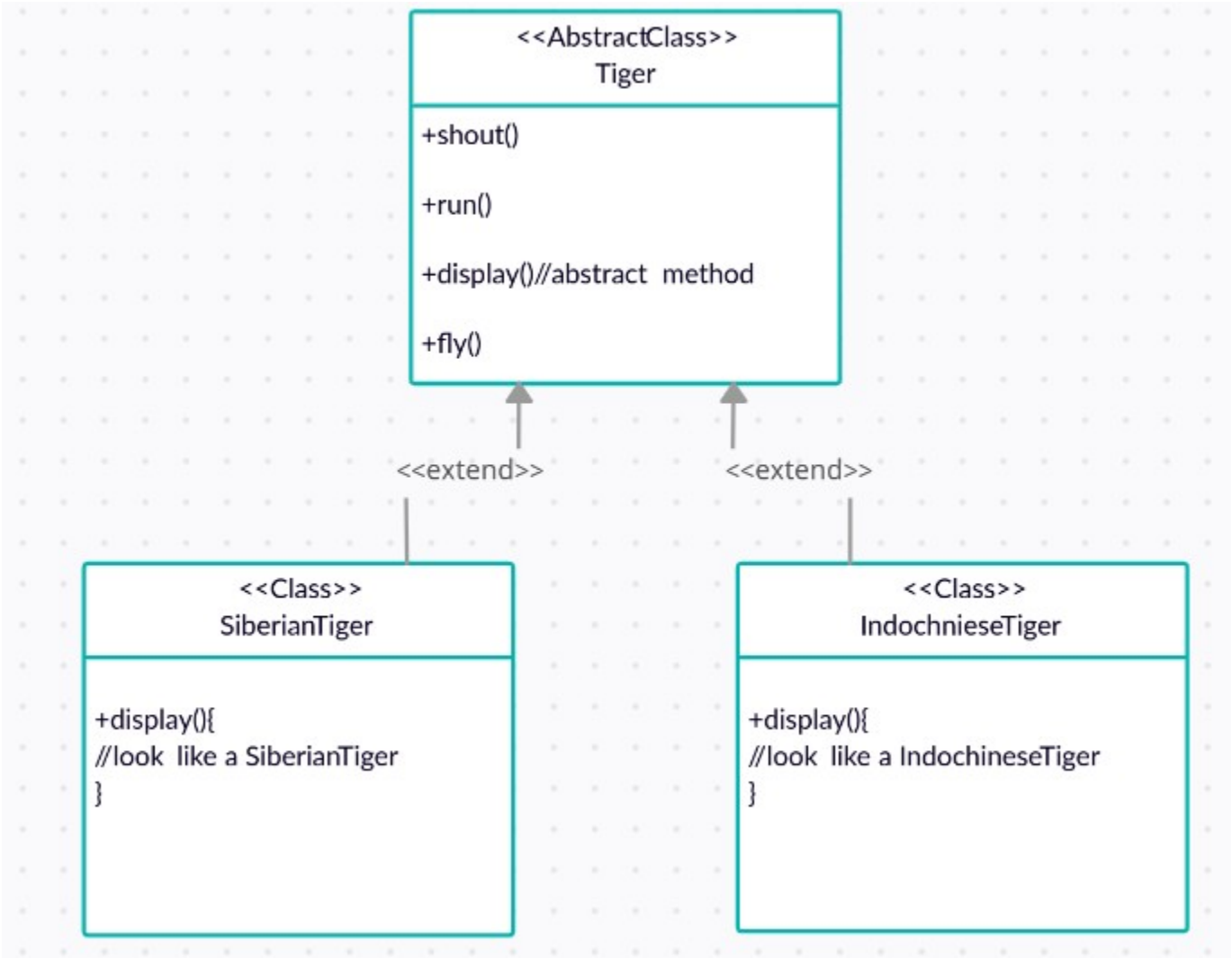
Game Company တစ်ခုက Game Software တစ်ခု နှင့် ဥပမာပေးပါမယ်။ Shooting Game တစ်ခု ဖြစ်ပြီး လက်ရှိ class တည်ဆောက်ပုံက -



Tiger ဆိုတဲ့ abstract class ကို SiberianTiger နဲ့ IndochineseTiger က extends လုပ်ထားမယ်။ Abstract class ထဲမှာကတော့ tiger တိုင်းမှာ တူညီတဲ့ shout နဲ့ run methods နှစ်ခု ကိုတော့ implement လုပ်ထားပါတယ် display mehtod ကတော့ Tiger Class မှာ abstract mehtod ဖြစ်တယ်။ extends လုပ်တဲ့ child class တွေက သီးခြားစီ display method ကို implement လုပ်ထားတယ်။

ဒါကတော့ စစချင်း ရှိနေတဲ့ ပုံစံမျိုး ဖြစ်ပါတယ်။

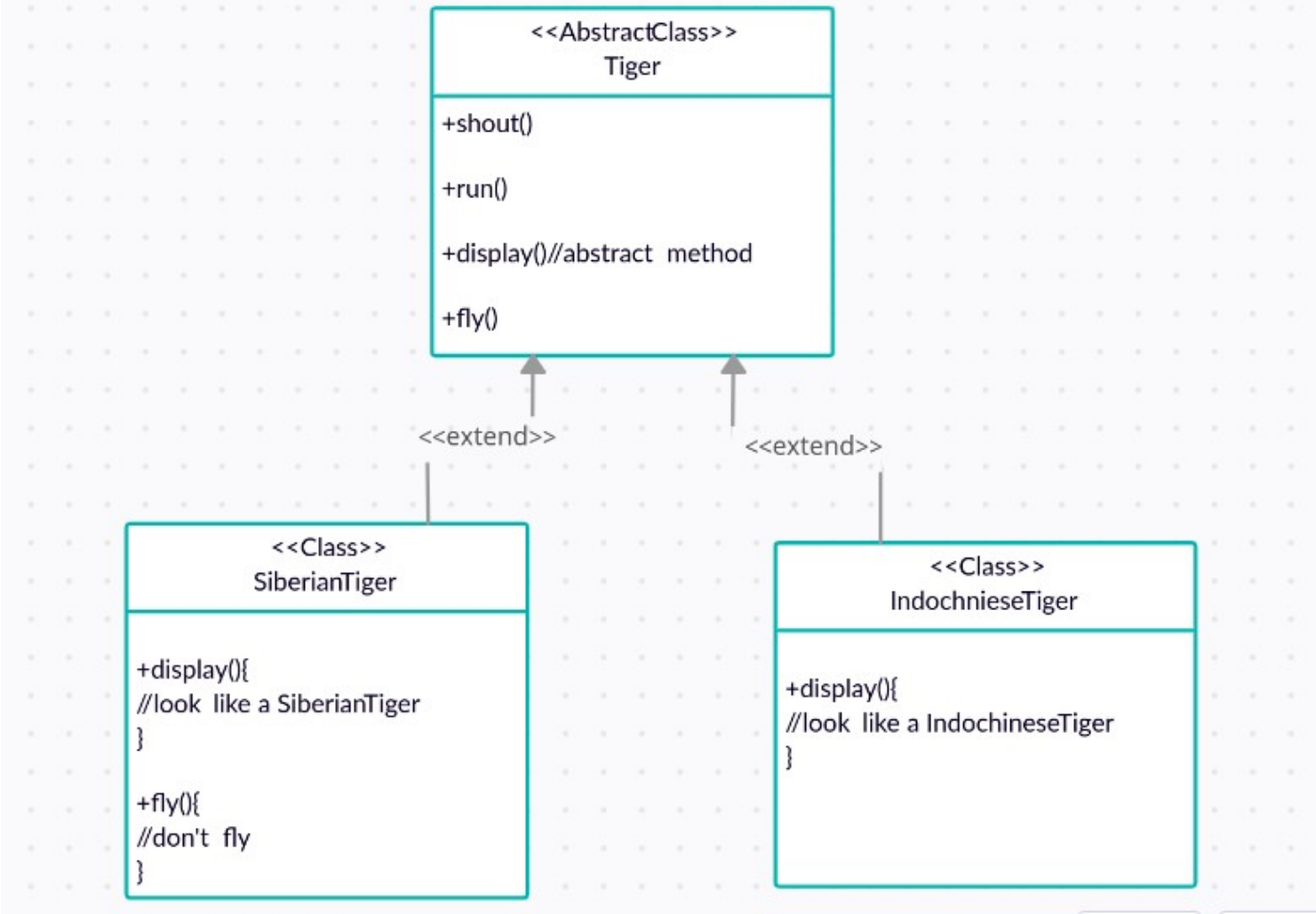
လက်ရှိ အခြေအနေမှာ Game Company က Feature အသစ် တစ်ခု ထပ်ထည့်ဖို့ programmer ကို ပြောပါတယ်။ အသစ်ထည့်မယ့် feature က fly ဘဲ ဖြစ်ပါတယ်။ programmer ကလည်း လွယ်လွယ်ကူကူဘဲ fly method လေးကို ထည့်ပေးလိုက်ပါတယ်။ ပြင်ဆင် ပြီးတဲ့ အခြေအနေက Class Diagram ကတော့-



game ကို ကြည့်ပြီး တော့ Company ကလူတွေက ကျားတွေ အကုန် လုံးပျံသန်းနေတာက ကြည့်မကောင်းဘူးဆိုပြီး တချို့သော ကျားတွေဘဲ ပျံလို့ရတဲ့ပုံစံမျိုးပြန်ပြင်ခိုင်း ပါတယ်။ အဲ့ အချိန် မှာ Programmer အတွက် အကြပ်ရိုက်စရာတွေ စဖြစ်ပါပြီ။လက်ရှိ အနေအထားအရ parent class ထဲ မှာ fly ကို ထည့်လိုက်တဲ့ အတွက်

child တိုင်းက fly ဆိုတဲ့ method ကို အမွေရပြီး ပျံလို့ရသွားကြပါတယ်။ တကယ်လို့ တချို့ child က မပျံနိုင်ဘူးဆိုရင် method overriding ကို သုံးရပါတော့မယ်။

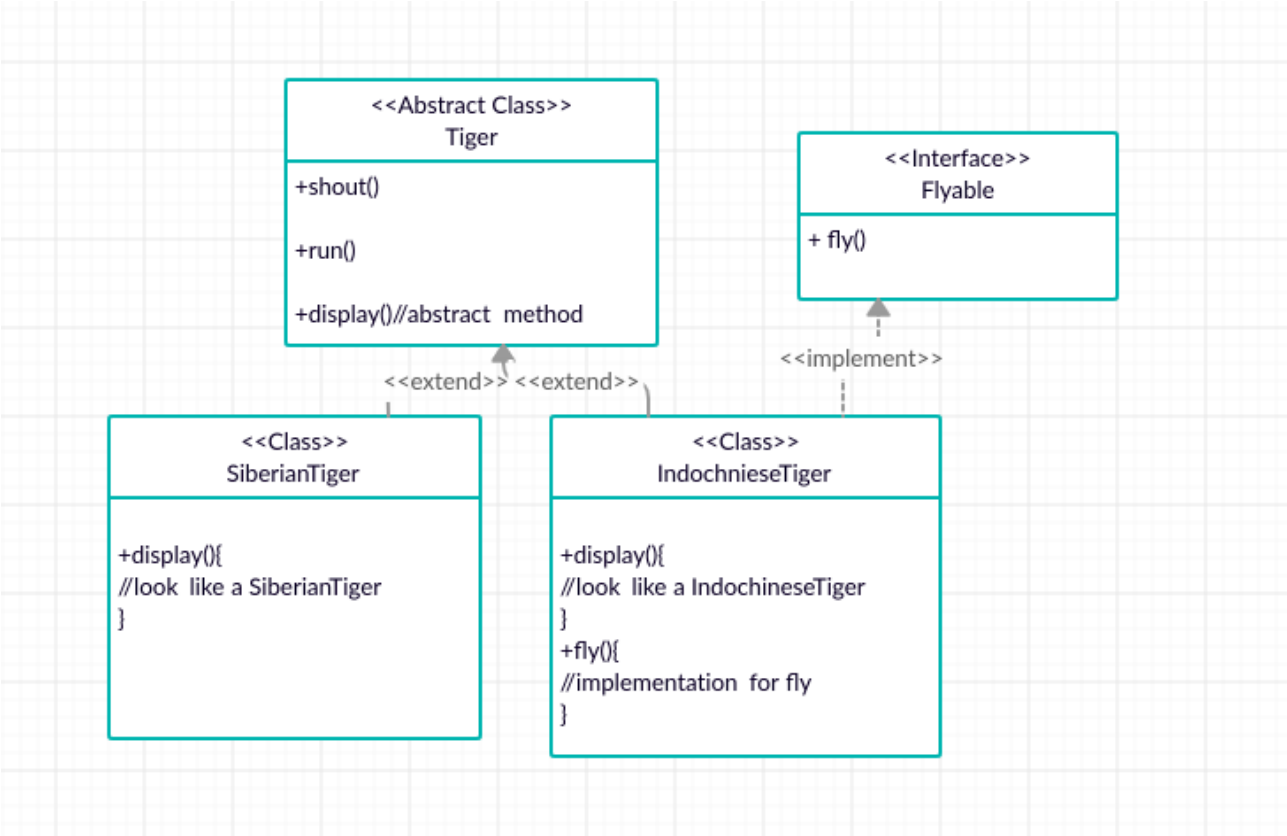
mehtod overriding ကို သုံးပြီး ယာယီ ဖြေရှင်းထားတဲ့ ပုံစံက-



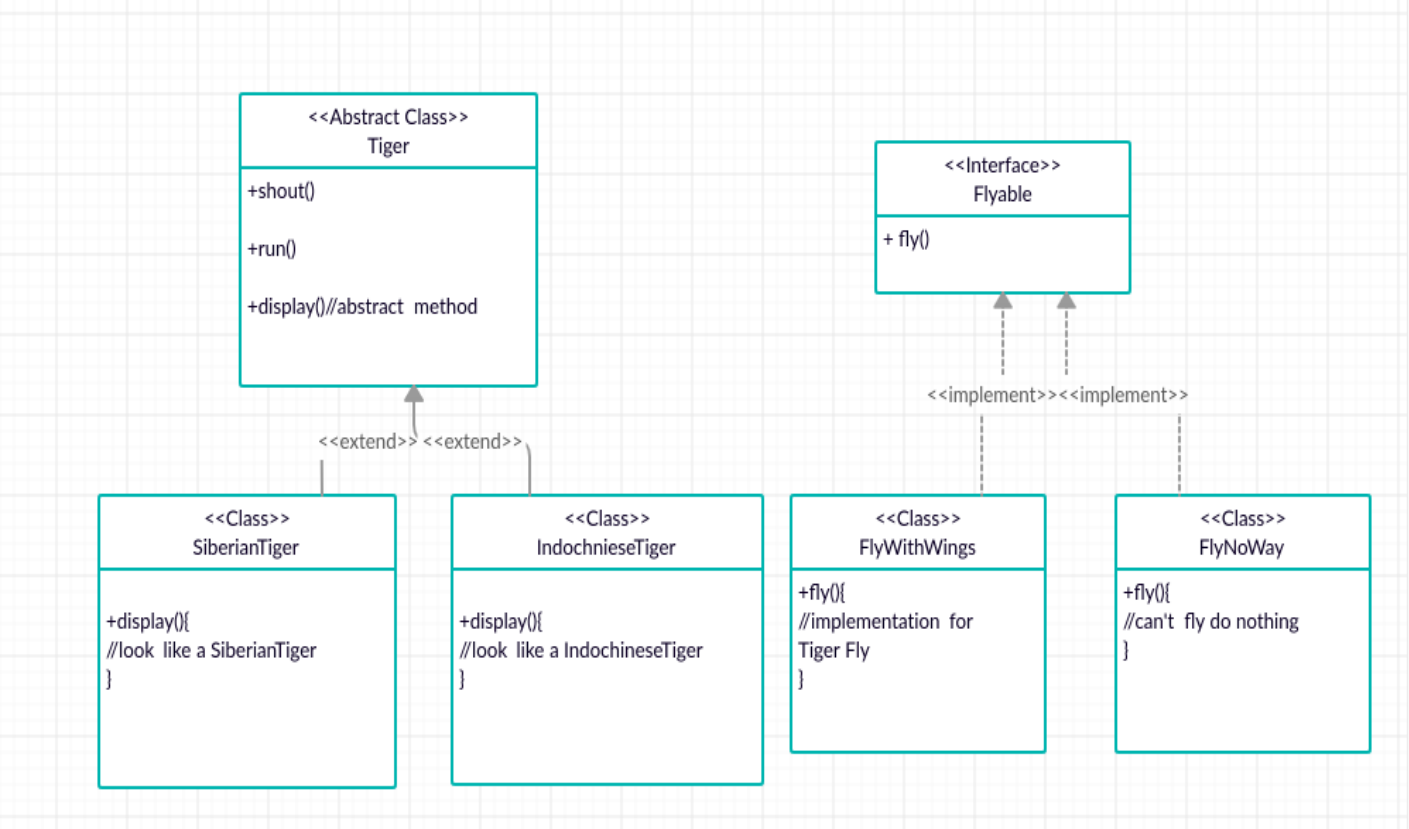
လောလောဆယ်တော့ အဆင်ပြေသလို ရှိနေပေမယ့် ကျားအမျိုးအစားပေါင်းရာချီ လာတဲ့ အခါ ပျံတာမပျံတာ ကို တစ်ခု ချင်းစီမှာ ပြန်ပြင်ပြီးရေးနေရရင် အဆင်မပြေပါဘူး။ တကယ်တော့ ဒီလိုမျိုးဖြေရှင်းတာက နည်းလမ်းမကျပါဘူး။ အဓိက ပြဿနာက inheritance ကြောင့်ပါ လိုတာကောမလိုတာကော အကုန်လုံးက inheritance လုပ်လိုက်ရင် အမွေရသွားတဲ့ အတွက် ဒီလိုပြဿနာမျိုးတွေပေါ်လာတပါ။ တကယ်တော့ fly ဆိုတဲ့ method က change ဖြစ်နေတဲ့ အရာမျိုးပါ။ အဲ့လို change ဖြစ်နေတဲ့ အရာမျိုးကို သီးသန့်ခွဲထုတ်ပါမယ် Strategy Pattern အရပေါ့။

တော်တော်များများနေရာတွေမှာ inheritance နဲ့ ဖြေရှင်းလို့ မရတာတွေကို interface နဲ့ ဖြေရှင်းလို့ရပါတယ်။ change ဖြစ်တဲ့ဟာတွေကို သီးသန့်ဖယ်ထုတ်နိုင်ဖို့ ဆိုရင် interface က အကောင်းဆုံးပါဘဲ။

interface သုံးထားတဲ့ class-diagram ကတော့-



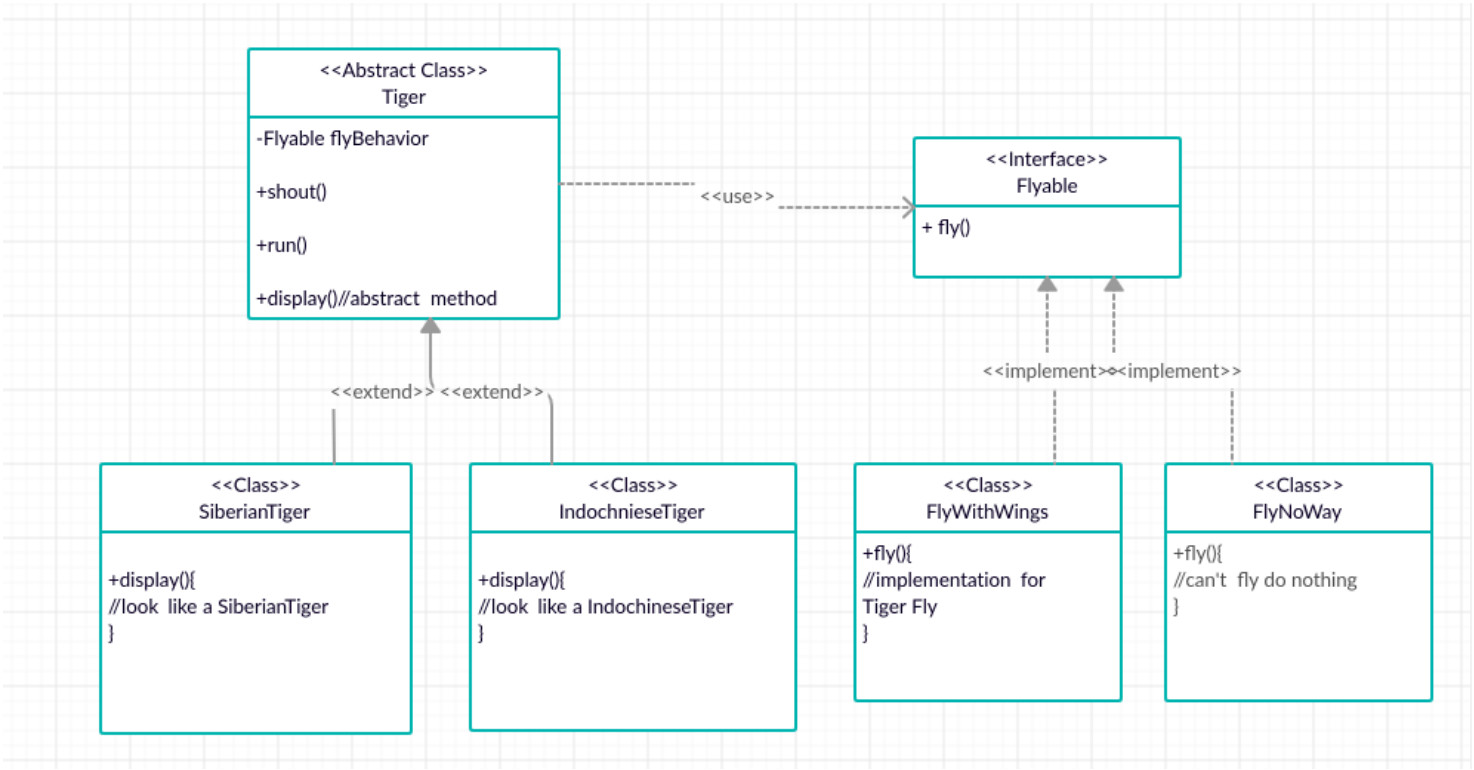
changes ဖြစ်တဲ့ အပိုင်းလေးကို တော့ ခွဲထုတ်လိုက်ပြီ ဆိုပေမယ့် Tiger အမျိုး အစာတစ်ခုချင်းစီ မှာ interface ကို ပြန် ပြီး implement လုပ်နေရရင် code reuse ဖြစ်မှာ မဟုတ်ဘူး အဲ့အတွက် code reuse ဖြစ်အောင် interface ကို implement လုပ် တဲ့ class သီးသန့်ထပ်ရေးမယ်။



လက်ရှိအနေအထားက အပေါ်ကပုံစံမျိုးပေါ့ ဒီနေရာမှာ အရေးကြီးတာက fly mehtod နဲ့ပတ်သက် ပြီး လက်ရှိ ရှိနေတဲ့ code မှာ ဘယ်လိုပုံစံမျိုးပြန်ပေါင်းမလဲပေါ့။ ကန့်သတ်ချက် ကတော့တစ်ခု ဘဲရှိတယ် Code reuse ဖြစ်ရမယ် Code မထပ်ရဘူး။ ဒီနေရာမှာ ဆိုရင် flyable interface အတွက် implementation ကတော့ ပျံသန်း နိုင်တာကော မပျံသန်းနိုင်တာကော နှစ်ခုလုံးရှိနေပြီ ဖြစ်တဲ့အတွက် ကျားတိုင်းအတွက် flyable interface ကို သုံးလို့ရနိုင်တဲ့ အနေအထားကို ရလာပြီ။

အဲ့အတွက်ကြောင့် parent abstract class ထဲမှာ flyable interface ကို field တစ်ခု အဖြစ်ထည့်သွင်းပြီး object ဆောက်ကာနီးကျမှ flyable ကို implement လုပ်ထားတဲ့ flywithwings ဖြစ်ဖြစ် flynoway ဖြစ်ဖြစ် ကို assign လုပ်ပေးလိုက်ရင် ရပါပြီ။

Final Class-Diagram ကတော့ -



Class Diagram ကိုကြည့်ရုံနဲ့တော့ ဘယ်လိုအလုပ်လုပ်လဲဆိုတာ အသေအချာ မသိနိုင်ပါဘူး။ Example code အနေနဲ့ SiberianTiger နဲ့ IndochineseTiger class တွေ ရဲ့ code ကို အောက်မှာပေးထားပါမယ်။

```
class SiberianTiger extends Tiger {

    SiberianTiger() {
        flyBehavior = new FlyNoWay();
    }

    display() {
        // look like a Siberiantiger
    }

}
=== === === === === === === === === === === === === === === ==

class IndochineseTiger extends Tiger {

    IndochineseTiger() {
        flyBehavior = new FlyWithWings();
    }

    display() {
        // look like a Indochinesetiger
    }

}

=== === === === === === === === === === === === === === === ==
```

Observer Pattern ဆိုသည်မှာ.....

OOP Desing Pattern တွေထဲက တစ်ခုပါဘဲ။ သူရဲ့ အဓိပ္ပါယ်ဖွင့်ဆိုချက်ကတော့- objects တွေကို တစ်ခုနဲ့တစ်ခု one-to-many relationship ချိတ်ထားမယ်။ object တစ်ခုက အပြောင်းအလဲ ဖြစ်သွားရင် သူနဲ့ချိတ်ထားတဲ့ object တွေအကုန်လုံးမှာ အပြောင်းအလဲ ကို အလိုအလျောက် ပြင်ဆင်ပေးနိုင်တဲ့ ပုံစံမျိုးကိုဆိုလိုသည်။

အကျယ်ချဲ့ရှင်းရလျှင် Object တစ်ခုရဲ့ field တစ်ခုမှာ တန်ဖိုး အပြောင်းအလဲ ဖြစ်သွားရင် အဲ့ဒီ object ကို ယူသုံးထားတဲ့ ကျန်တဲ့ objects တွေအားလုံးမှာပါ အဲ့ field တန်ဖိုးက auto ပြောင်းလဲသွားသလိုမျိုးပေါ့။

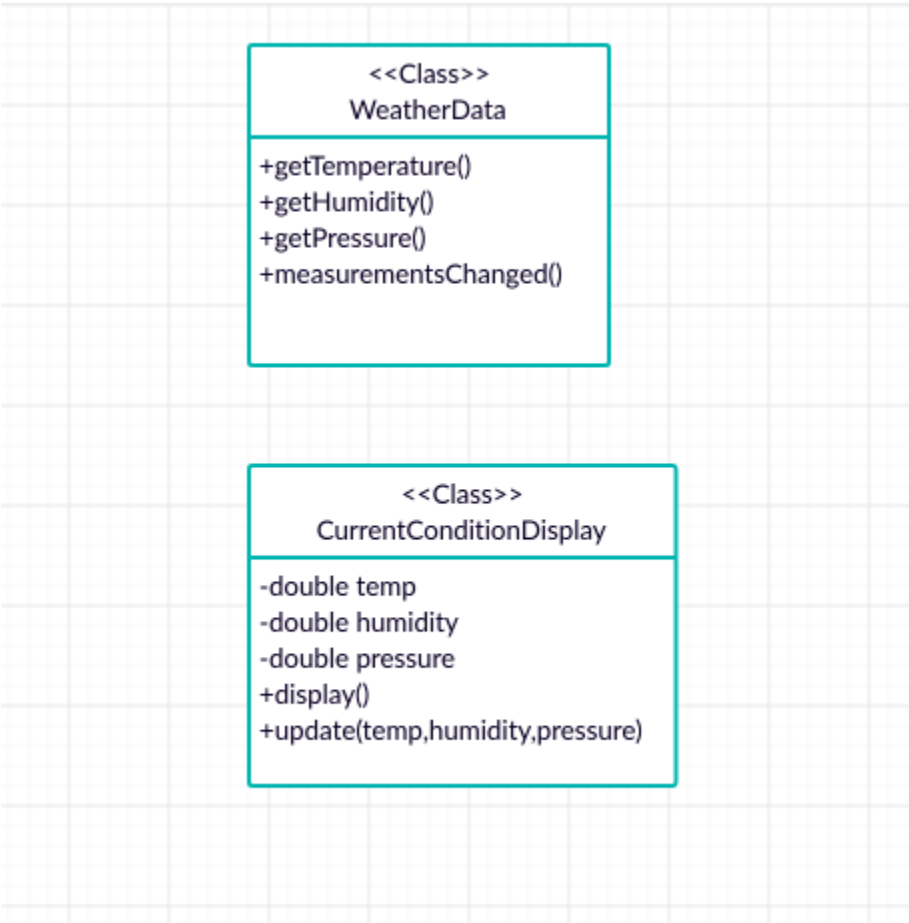
Observer Pattern က JDK (Java Development Kit) မှာလည်း တော်တော်လေး ထည့်သုံးထားတဲ့ pattern ဖြစ်ပါတယ်။ ဒီ pattern ကိုသေချာနားလည်ရင် JDK ရဲ့ တည်ဆောက်ထားပုံနဲ့ အလုပ်လုပ်ပုံကိုလည်း ပိုနားလည်လွယ် သွားမှာဖြစ်ပါတယ်။

သေချာနားလည်သွားအောင် ဥပမာလေးတစ်ခုနဲ့ ဆက်ရှင်းပြပါမယ်။

Example

Weather application တစ်ခုရဲ့ တည်ဆောက်ပုံလေးနဲ့ စလေ့လာပါမယ်။ weather application ကို အပိုင်းနှစ်ပိုင်း ခွဲလိုက်မယ်။ တစ်ပိုင်းက weather data တွေ ဥပမာ အပူချိန်၊လေတိုက်နှုန်းစတာ တွေကို အချိန်နှင့်အမျှ တိုင်းတာတဲ့ ကရိယာတွေဆီကနေ ဖတ်ပြီး store လုပ်ထားမယ်။ နောက်တပိုင်းက stroe လုပ်ထားတဲ့ data တွေကို အဆင်သင့်ယူပြီး နောက်ထပ်တွက်ချက်မှုတွေလုပ်ပြီး user ကို ပြသပေးမယ့် အပိုင်းဖြစ်တယ်။

အခုနှစ်ခုကို ပထမ observer pattern ကို အသုံးမပြုဘဲ တည်ဆောက်ထားတဲ့ class diagram ပုံစံကတော့ -



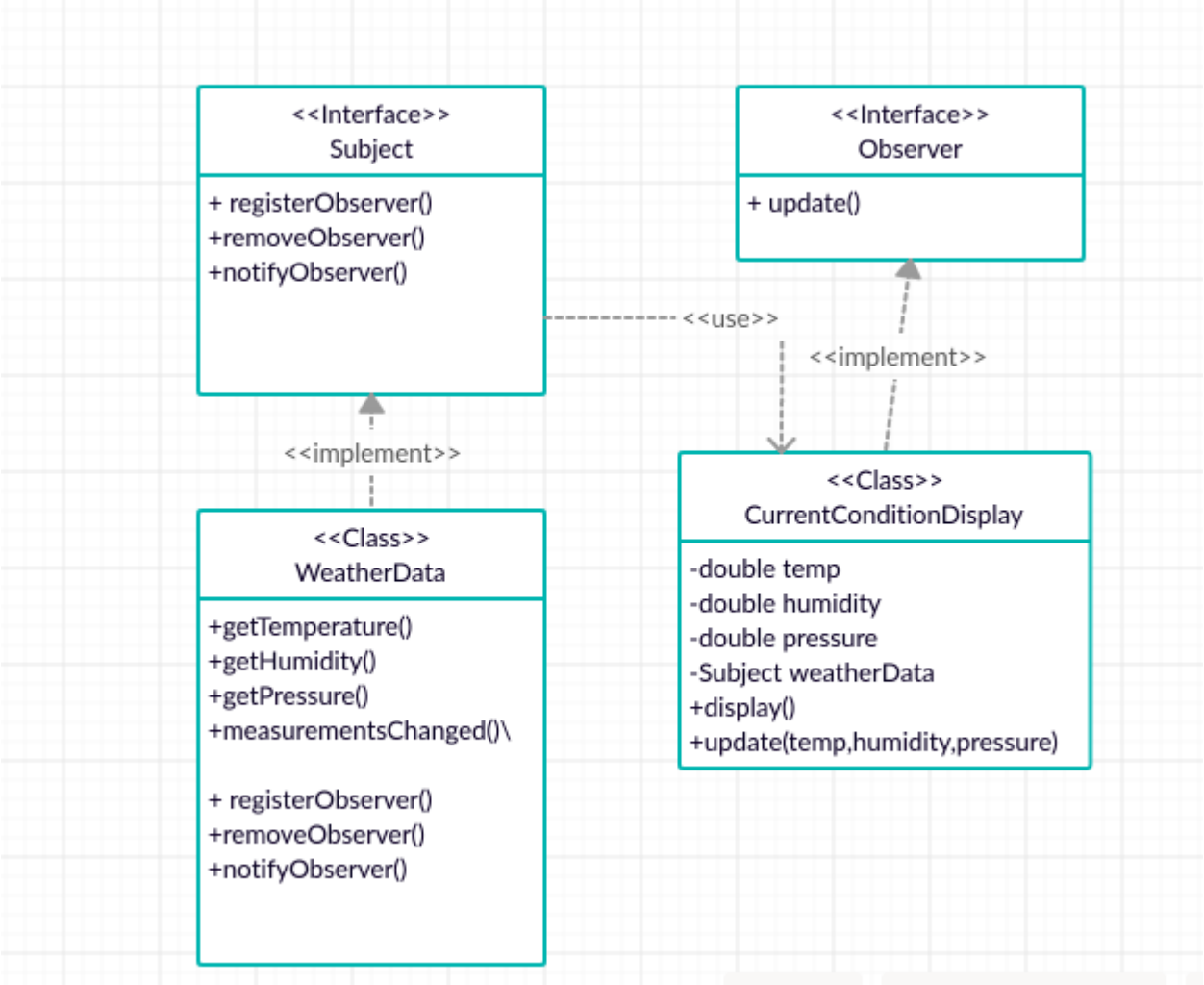
လက်ရှိ class diagram မှာက လိုတဲ့ method လေးတွေဘဲ ထည့်ထားပါတယ်။ WeatherData class ရဲ့ getmethod တွေအားလုံးက devices ကနေ data ကိုယူလာမယ် တကယ်လို့ မူလတန်ဖိုးနဲ့မတူရင် measurementsChanged() method ကို သုံးမယ်။

MeasurementChanged() method ရဲ့ code ကတော့-

```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    //currentConditonsDisplay is an object of
//CurrentCoditionDisplay class.

}
```

ဒီ method ရဲ့ အားနည်းချက်ကတော့ တကယ်လို့ display class ရဲ့ object သာ ဆယ်ခုရှိမယ်ဆိုရင် ဆယ်ခါတိတိ object တစ်ခုချင်းစီရဲ့ update method ကို ခေါ်သုံးရမယ်။ နောက်တစ်ခု တကယ်လို့ data တစ်ခုခုသာထပ်တိုးလာမယ်ဆိုရင် လည်း ခေါ်သုံးထားတဲ့ update method တွေ အကုန်လုံးမှာ arguments တွေလိုက်ပြင်ရမယ်။ အဲ့ဒီအချိန် ကျရင် အဆင်မပြေနိုင်တော့ဘူး အဲ့ တာတွေဖြေရှင်းနိုင်ဖို့ observer pattern ကိုသုံးပြီး class diagram ပြန်ပြင်ဆွဲမယ်။

Observer Pattern အရ Class Diagram ကတော့ -



Observer pattern အရ data class ကို subject interface ကို implement လုပ်ခိုင်းလိုက်တယ်။ subject interface မှာ observer pattern အတွက် လိုအပ်တဲ့ method သုံးခုပါမယ်။ observer ဆိုတဲ့ အတိုင်း observe လုပ်မယ့် objects တွေကို register လုပ်တာ register လုပ်ပြီးသား class တွေကို ပြန်ပြီး remove လုပ်တာရယ် တကယ်လို့ အပြောင်းအလဲ ဖြစ်လာရင် register လုပ်ထားတဲ့ object တွေကို notify လုပ်တာရယ် ဆိုပြီး method သုံးခုပါမယ်။ အဲ့ method တွေကို WeatherData Class က implement လုပ်ထားမယ်။ အဲ့ interface နဲ့ class ရဲ့ code ကို လေ့လာကြည့် ပါမယ်။

```
public class WeatherData implements Subject {

    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer) observers.get(i);
            observer.update(temperature, humidity, pressure);
        }

        public void measurementsChanged() {
            notifyObservers();
        }

    }
```

အပေါ်က code မှာ Arraylist တစ်ခုထဲ မှာ register လုပ်တဲ့ observers တွေကို ထည့်ထားပြီး remove လုပ်ရင် arraylist ထဲကနေ ပြန်ထုတ် update လုပ်မယ်ဆိုရင်လည်း arraylist ကို loop ပတ်ပြီး တစ်ခုချင်းစီ update လုပ်သွားပါတယ်။ အရင် code ထက်စာရင် အများကြီး ပိုအဆင်ပြေပြေ အပြောင်းအလဲကို ကိုင်တွယ်သွားနိုင်ပါပြီ။ ဆက်ပြီး Observer Interface နဲ့ သူ့ကို implements လုပ်ထားတဲ့ class ရဲ့ code ကတော့ -

```
public interface Observer {
    public void update(float temp, float humidity, float
pressure);
}

public class CurrentConditionsDisplay implements Observer {

    private double temperature;
    private double humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity,
float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }


    public void display() {
        System.out.println("Current conditions: "+temperature
+
            "F degrees and" + humidity + " % humidity");

    }
}
```

အပေါ်က code မှာ CurrentConditionDisplay class ကို object ဆောက်လိုက်တာနဲ့ weatherData Object ရဲ့ observers arraylist ထဲမှာ တခါတည်း register လုပ်သွားတာ လေးကို သေချာလေးကြည့်ပေးပါ။ highlight လုပ်ပြထားပါတယ်။ အဲ့တာ keypoint ပါ အဲ့နည်းနဲ့ Object တွေကို one-to-many ချိတ်ထားလိုက်တာပါ။

အခုပြတဲ့ Example ကတော့ ကိုယ်တိုင် Observer Pattern ကို အစအဆုံး ဖန်တီးတဲ့ ထားတာပါ။ တကယ်လို့ Java မှာဆိုရင်တော့ သူ့ language က ပေးထားတဲ့ class တွေရှိပါတယ်။ java.util.Observable နဲ့ java.util.Observer ဆိုပြီး class နဲ့ interface နှစ်ခု ကိုသုံးပြီး လည်း observer pattern ကို လုပ်ယူနိုင်ပါတယ်။

Jdk မှာသုံးထားတာကတော့ button ရဲ့ actionListener() method တွေမှာ သုံးထားတာတွေက Observer Pattern ပါဘဲ။ button ကို actionevent တစ်ခုခု ဝင်လာတာနဲ့ listener အနေနဲ့ ထည့်ထားတဲ့ object တွေ အကုန်လုံးကို notify() လုပ်တဲ့ ပုံစံမျိုးပါ။

Decorator Pattern ဆိုသည်မှာ……

Decorator ဆိုတဲ့ နာမည် က ကျတော်တို့ လက်တွေ့ဘဝမှာ သုံးနေတဲ့ အိမ်တွေ ရုံးခန်းတွေ ကို decoration လုပ်တယ်ဆိုတဲ့ ပုံစံနဲ့ ဆင်တူပါဘဲ။ ကျတော်တို့ ပိုမိုလှပ အောင် နဂိုရှိပြီးသား နံရံပေါ်မှာ ဆေးထပ်သုတ်သလိုပါဘဲ။ OOP Design အရပြောမယ် ဆိုရင် နဂိုရှိပြီးသား object တစ်ခု ကို အခြေခံပြီး အဲ့ဒီ object ရဲ့အပေါ်မှာ နောက်ထပ် object အသစ်တစ်ခုကို တည်ဆောက် လိုက်တာပါဘဲ။ ပြောရမယ်ဆိုရင် class တွေကို inheritance လုပ်သလိုဘဲ object တွေကို inheritance လုပ်လိုက်တဲ့ သဘောပါဘဲ။

အကျယ်ချဲ့ရှင်းရလျှင် class a နဲ့ b ဆိုပြီး class နှစ်ခုရှိမယ်။ class b ထဲမှာ class a ရဲ့ object တစ်ခု ကို field variable အဖြစ်ထားထားမယ်။ class b ရဲ့ constructor မှာ class a ရဲ့ object ကို parameter အဖြစ်တောင်းထားမယ်။ အဲ့တော့ class b ကို object တည်ဆောက်ဖို့ class a object တစ်ခု မရှိမဖြစ် လိုအပ်မယ်။ အဲ့ လိုမျိုး class တစ်ခု ရဲ့ object ပေါ်အခြေခံပြီး နောက်တစ်ခု တည်ဆောက်တာမျိုးကို Decorator Pattern လို့ခေါ်တယ်။

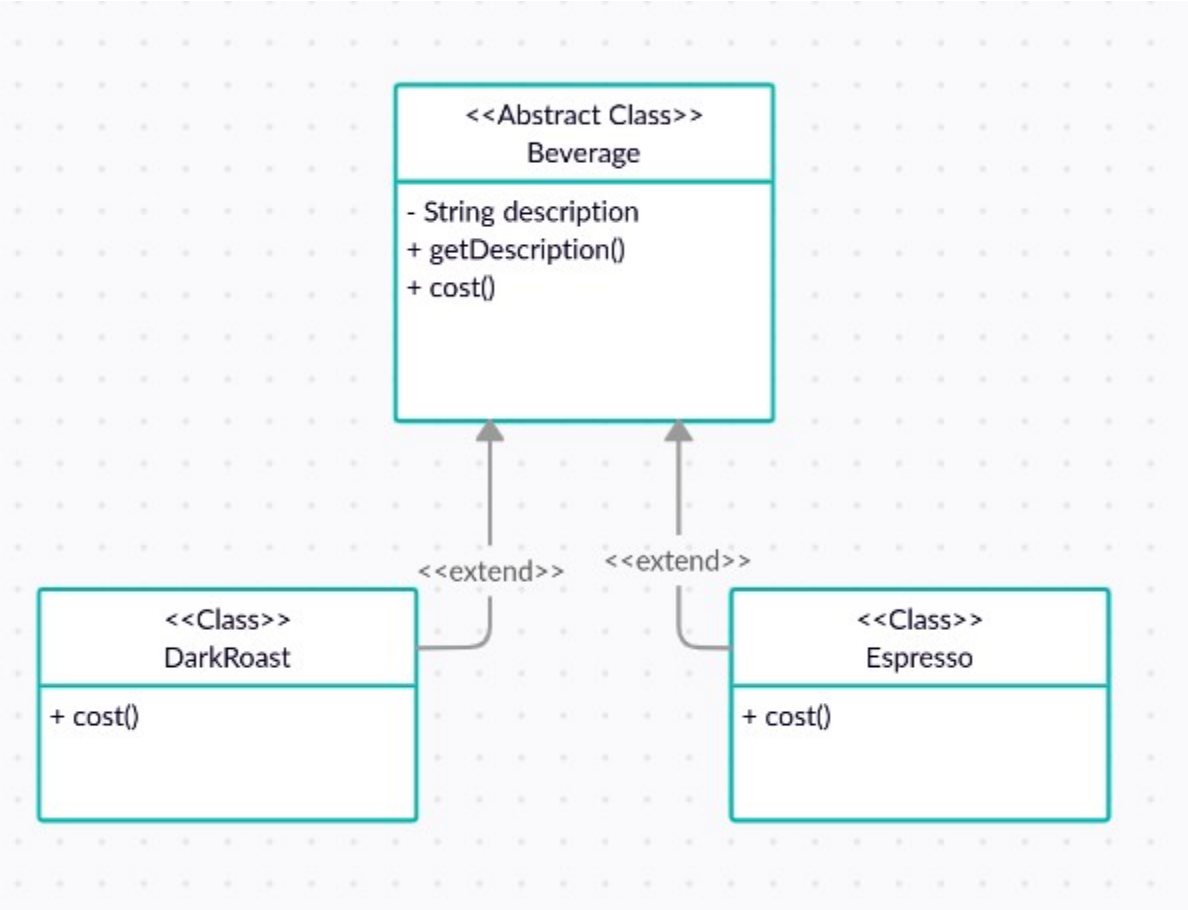
Java ရဲ့ java.io Package မှာ Decorator pattern ကို သုံးပြီးတည်ဆောက် ထားပါတယ်။ FileReader object တစ်ခု ဆောက်မယ်ဆိုရင် file object ကို အရင် ဆောက်ပြီး FileReader object ဆောက်တဲ့အချိန်မှာ file object ကို ထည့်ပေးရတာ နောက် အလားတူ BufferReader မှာ လည်း FileReader object တစ်ခု ထည့်ပေးရ တာတွေ က Decorator Pattern ကို သုံးထားလို့ပါ။ အောက်မှာ code ထည့်ပေးထားပါ တယ်။

```
File f = new File(“test.txt”);
FileReader fReader = new FileReader(f);
BufferedReader br0 = new BufferedReader(fReader);
```

ဆက်ပြီး သေချာနားလည်သွားအောင် example လေးတစ်ခုနဲ့ ဆက်ရှင်းပါမယ်။

Example

ကော်ဖီဆိုင်တစ်ခု မှာ သုံးမယ့် software နဲ့ ဥပမာ ပေးသွားပါမယ်။
လက်ရှိ သုံးနေတဲ့ class diagram ကို အောက်မှာပြထားပါတယ်။



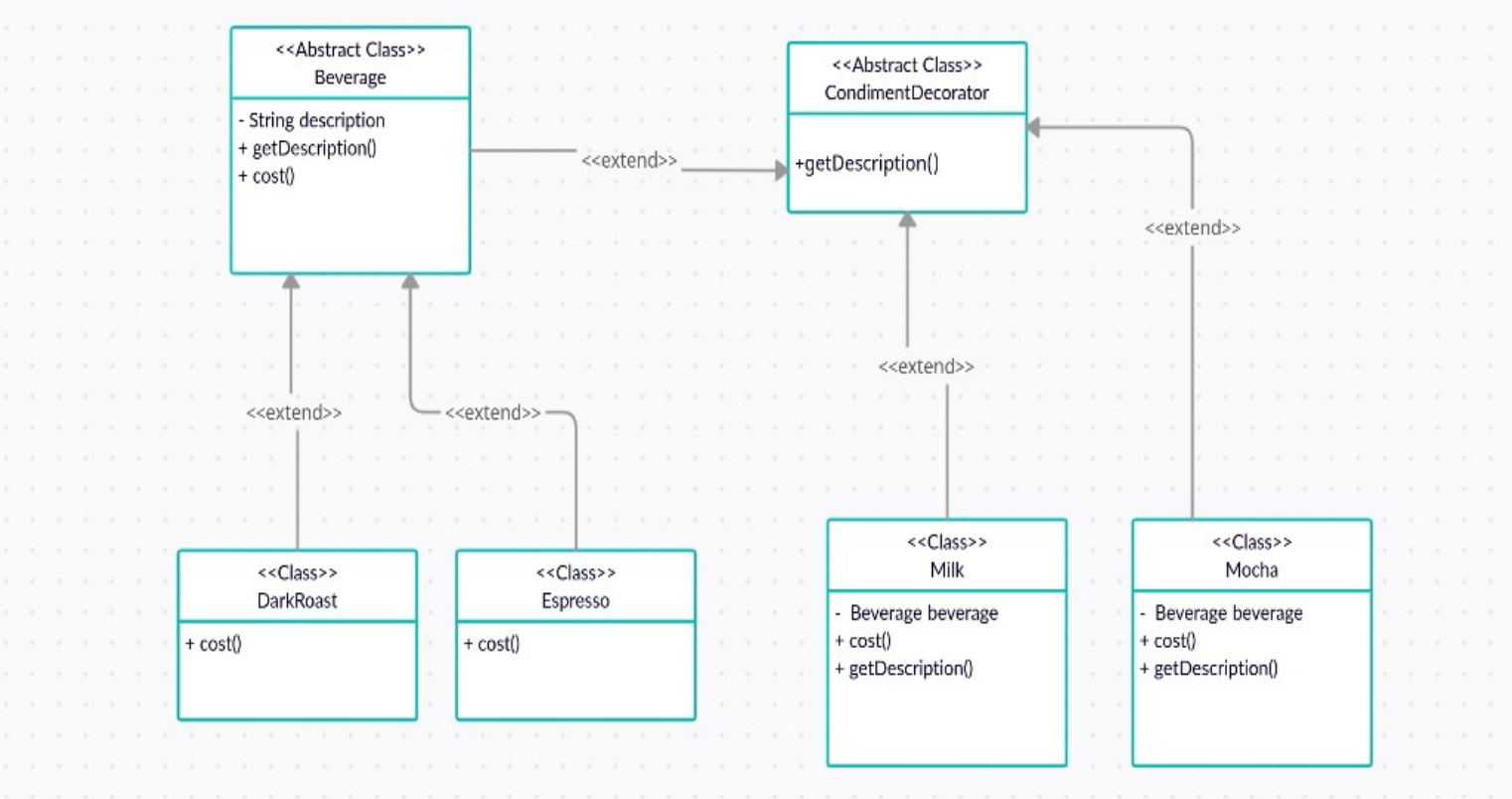
လက်ရှိ မှာက abstract class တစ်ခု ကို ပုံသေထားထားပြီး အဲ့ abstract class ကို coffee အမျိုးအစား အားလုံးက extends လုပ်တဲ့ ပုံစံမျိုးကို သုံးထားတယ်။ လက်ရှိ မှာ ပြဿနာ မရှိပေမယ့် ထပ်ပြီးတော့ function ထပ်တိုးဖို့လိုလာပြီ ထပ်တိုးချင်တာ က coffee အမျိုးအစားတစ်ခုချင်းဆီမှာ ပါဝင်တဲ့ ingredients ပြောရမယ်ဆိုရင် အရသာ ထပ်ဖြည့်ချင်တာ milk, mocha, soy စတဲ့ အရသာတွေထပ်ဖြည့်မယ်။ အရသာတစ်ခု ချင်းဆီမှာ လည်း cost ထပ်ရှိမယ်။ အရသာထပ်ဖြည့်လိုက်တိုင်း total cost ပါ ပေါင်း သွားရမယ်။

အဲ့တော့ ပထမတစ်နည်း က သက်ဆိုင်ရာ အမျိုးအစားလိုက် class တွေဖြစ်တဲ့ DarkRoast တို့ Espresso တို့ မှာ field variable တွေ ထပ်ထည့်လိုက်မယ်။ milk,

mocha, soy ဆိုပြီး boolean ထားထားလိုက်မယ် ပြီးရင် milk ထပ်ထည့်မယ်ဆို true မထည့်ဘူးဆို false အဲ့လိုသွားမယ်။

အဲ့တာကို ထပ်စဉ်းစားကြည့်ရအောင် တကယ်လို့ စုစုပေါင်း coffee အမျိုးအစား တစ်ရာ ရှိတယ်ဆိုပါတော့ တစ်ရာလုံးမှာ လိုက်ပြီး variable ထည့်ရမယ်။ နောက်ပြီး တကယ်လို့ စျေးနှုန်းတစ်ခုခု change ပြီဆိုတာနဲ့ အကုန်လုံးမှာ လိုက် change ရတော့ မယ်။ နောက် တကယ်လို့ အရသာ အသစ်တစ်ခု ထပ်ထည့်ချင်ပြီဆိုရင်လည်း coffee အမျိုးအစား တစ်ရာလုံးမှာ ထပ်ထည့်ရတော့မယ်။ ဒီပြဿနာကို ရှင်းဖို့ design pattern ကို သုံးတတ်ဖို့ အရေးကြီးလာပြီ။

Design pattern တွေချတဲ့ နေရာမှာ Rule လေးတစ်ခု ကို ဂရုစိုက်ဖို့လို တယ်။"Open-Closed Principle” လို့ခေါ်တယ်။ "Open for Extenion and Close for modification” ဆိုလိုရင်းကတော့ fuction အသစ် ထပ်ထည့်နိုင်ရမယ်။ နဂိုရှိ နေတဲ့ ရေးပြီးသား code ကိုမပြင်ဘဲနဲ့။ အခု ဥပမာနဲ့ ထပ်ရှင်းရမယ်ဆိုရင် DarkRoast တို့ Espresso တို့မှာ code တစ်ခုကို မှ မထိဘဲ နဲ့ မပြင်ဘဲနဲ့ အသစ်ထပ်ထည့်ရမယ်။ အဲ့လို ထပ်ထည့် ဖို့ အခုသုံးမယ် pattern က decorator pattern ကို သုံးမယ်။ decorator pattern အရ နဂိုရှိနေတဲ့ DarkRoast တို့ Espresso တို့မှာ Functions တွေကို object inheritance နည်းနဲ့ ထပ်ဖြည့်လိုက်မယ်။ နောက်ဆုံး decorator pattern အရ ပြင်ဆွဲ ပြီးတဲ့ class diagram ကတော့-



နဂိုရေးပြီးသား code တစ်ခုမှ မထိထားဘဲ အသစ် CondimentDecorator ဆိုပြီး abstract class တစ်ခုကို ဆောက်ပြီး နဂိုရှိနေတဲ့ beverage class ကို extends လုပ်တယ်။ နောက်ဆုံးအနေနဲ့ milk တို့ mocha တို့စတဲ့ အရသာ ထပ်ဖြည့်တဲ့ class ကို သီးသန့်ရေးထားလိုက်ပြီး CondimentDecorator class ကို extends လုပ်လိုက်တယ်။

Class diagram မှာတော့ DarkRoast class နဲ့ Milk တို့ Mocha တို့ က ချိတ်ဆက်မှုမရှိပေမယ့် object ဆောက်ပြီး object ချင်းချိတ်ဆက်မှု မျိုးကို တော့ လုပ်နိုင်ပါတယ်။ အဲ့ချိတ်ဆက်မှုကတော့ coding ကိုထပ်ကြည့်ရမှာ ဖြစ်ပါတယ်။ လက်ရှိ class diagram မှာတော့ တကယ်လို cost change မယ်ဆိုရင် Milk cost ဆို Milk မှာ change ရုံပါဘဲ တကယ်လို့ အရသာ ထပ်ဖြည့်မယ်ဆိုရင်လည်း CondimentDecorator class ကို extends လုပ်ပြီး class အသစ်တစ်ခုထပ်ဆောက်လိုက်ရုံပါဘဲ။

ဆက်ပြီး Coding ပိုင်း ကို ပြပါမယ်။

```
public abstract class Beverage {
  String description = "Unknown Beverage";

  public String getDescription() {
      return description;
  }

  public abstract double cost();

}

public class Espresso extends Beverage{
    public Espresso(){
        description = "Espresso";
    }

    @Override
    public double cost() {
        return 1.99;
    }

}

public class DarkRoast extends Beverage{
      public DarkRoast(){
        description = "Dark Roast Coffee";
    }

    @Override
    public double cost() {
        return .89;
    }

}

public abstract class CondimentDecorator extends Beverage{
    public abstract String getDescription();
}
```

```
public class Mocha extends CondimentDecorator{
    Beverage beverage;

    public Mocha(Beverage beverage){
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
      return beverage.getDescription() + ",Mocha";
    }

    @Override
    public double cost() {
        return .20 + beverage.cost();
    }

}


public class Milk extends CondimentDecorator{
    Beverage beverage;

    public Milk(Beverage beverage){
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
      return beverage.getDescription() + ",Milk";
    }

    @Override
    public double cost() {
        return .40 + beverage.cost();
    }

}
```

အပေါ်ဘက်အထိကတော့ နဂိုရှိတဲ့ class diagram ထဲမှာ ပါတဲ့ class တွေပါဘဲ Runtime မှာ ချိတ်ဆက် သွားပုံကို အောက်မှာ Main class မှာပြပေးထားပါတယ်။

```
public class Main {
    public static void main(String[] args) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                           + " $ "+beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Milk(beverage2);
        System.out.println(beverage2.getDescription()
                + " $ "+beverage2.cost());
    }
}
```

အဓိက အလုပ်လုပ်သွားတဲ့ code ကတော့ အပေါ်မှာ highlight လုပ်ထားတဲ့ နှစ်ကြောင်းပါဘဲ။ ပထမဆုံး DarkRoast object တစ်ခုကို beverage2 ဆိုတဲ့ နာမည်နဲ့ ဆောက်တယ်။ နောက်အဲ့ beverage2 မှာ အရသာနှစ်ခုထပ်ထည့်တယ်။ ထပ်ထည့်တဲ့ နေရာမှာ argument အနေနဲ့ အဲ့ဒီ beverage2 object ကိုထည့်ပေးလိုက်တယ်။ အဲ့နည်းနဲ့ object 3 ခုကို ချိတ်ထားလိုက်ပြီ နောက်ဆုံး getDescription() ကိုခေါ်လိုက်တဲ့အခါမှာ စစချင်း နောက်ဆုံး object ဖြစ်တဲ့ Milk ရဲ့ getDescription() ကိုခေါ်မယ် အဲ့ဒီ getDescription() method ကတခါ Mocha ရဲ့ getDescription() ကိုခေါ်မယ်။ နောက်ဆုံးအနေနဲ့ Mocha ရဲ့ getDescription() method ကတခါ DarkRoast ရဲ့ getDescription() ကိုခေါ်မယ်။ DarkRoast ရဲ့ getDescription() က execute လုပ်ပြီး သွားတဲ့အခါမှာ နောက်ဆုံး function call လည်းအဆုံးသတ်သွားမယ်။ စာဖတ်ပြီး code ကိုကိုယ်တိုင်ရေးပြီး Debug လုပ်ကြည့်ပါ တစ်ဆင့်ချင်းသွားပုံကို သေချာတွေ့ရပါမယ်။

Decorator Pattern ရဲ့အားနည်းချက် ကတော့ class အသေးလေးတွေအများကြီး ထွက်လာပြီး code complex ဖြစ်တာပါဘဲ။

Factory Pattern ဆိုသည်မှာ……

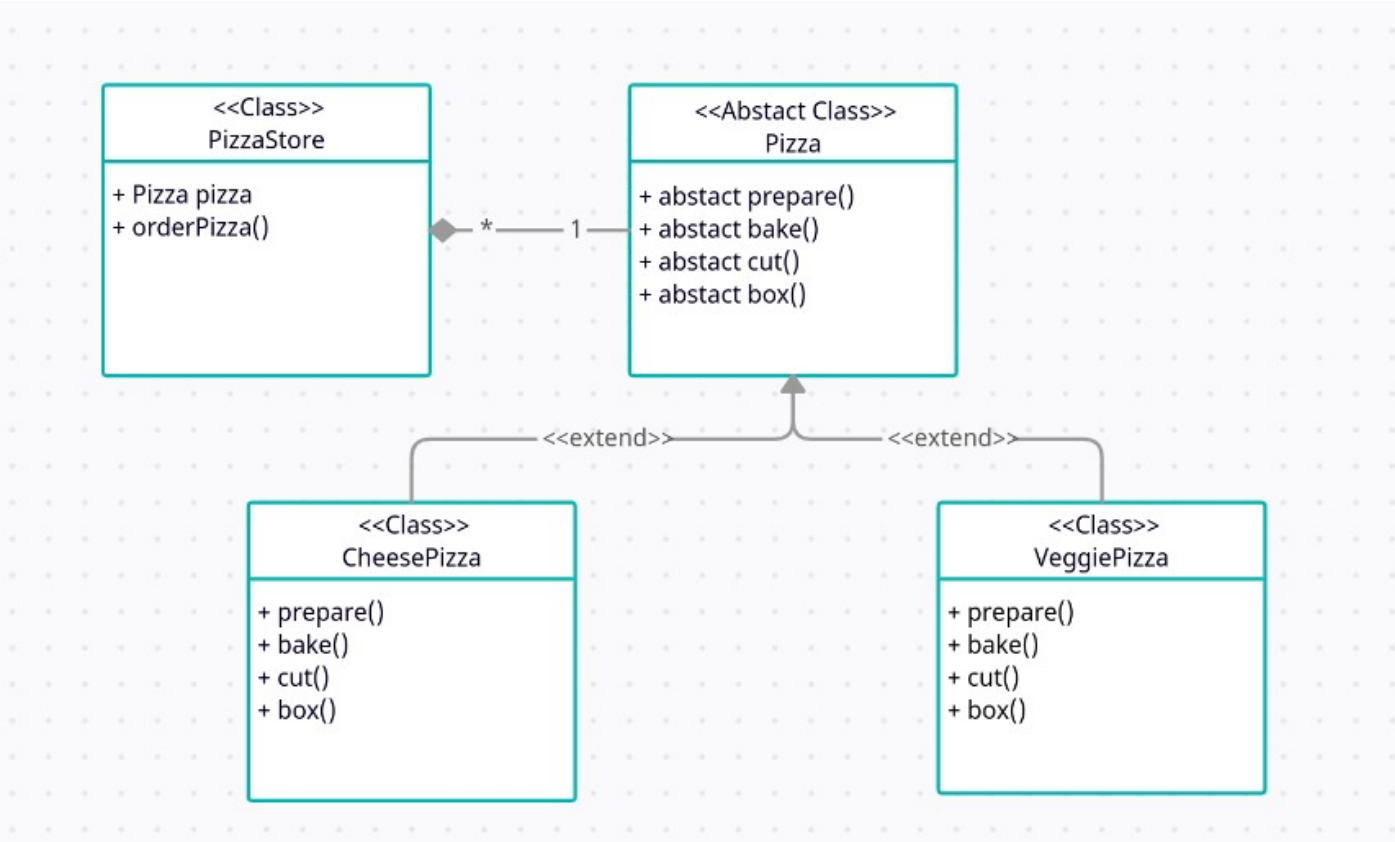
Factory Pattern ဆိုတာက တကယ်တော့ Simple Factory Pattern ရယ်၊ Factory Method Pattern ရယ် Abstract Factory Pattern ဆိုတဲ့ Design Pattern သုံးမျိုးကို ခြုံပြီး ခေါ်တဲ့ နာမည်တစ်ခုဘဲ ဖြစ်ပါတယ်။ အဲ့ဒီ design pattern သုံးမျိုးကို ဆက်ရှင်းပါမယ်။ တကယ်တော့ Simple Factory Pattern ဆိုတာက design pattern တစ်ခုမဟုတ်ပါဘူး။ Programming ဆိုင်ရာ idiom အသုံးတစ်ခု ဘဲ ဖြစ်ပါတယ်။ ကျန်တဲ့ နှစ်ခု ကတော့ design pattern မှာ ပါဝင်ပါတယ်။

Design Pattern တွေ ဆိုတာက ကျတော်တို့ code တွေကို ပြင်ဆင်ရာမှာ လွယ်ကူဖို့နဲ့ functions တွေတိုးတဲ့အခါမှာ အဆင်ပြေစေဖို့ လိုက်နာသင့်တဲ့ class ဖွဲ့စည်းပုံ ပုံစံငယ်တွေဘဲ ဖြစ်ပါတယ်။ Coding တွေရေးရာမှာ နဂိုရေးပြီးသား class ကို မပြင်ဘဲ အသစ်ထပ် functions တိုးဖို့ဆိုတာ တကယ့်တော့ မလွယ်ပါဘူး။ နဂိုရေးထားပြီးသား အပိုင်းကို ပြင်မိတဲ့အခါ အမှားတွေလုပ်မိနိုင်ချေများတဲ့ အတွက် code စစချင်းရေးတဲ့ အချိန် ကစပြီး ပြောင်းလဲတတ်တဲ့ အပိုင်းနဲ့ မပြောင်းလဲနိုင်တဲ့ အပိုင်းကို ခွဲခြားထား ရမယ်။ တစ်နည်းပြောရမယ်ဆိုရင် ထပ်ပြီး functions တိုးလာရင် ဘယ် class ကို တော့ code ပြင်ရမယ် ဘယ် class တွေကတော့ လုံးဝထိစရာမလိုဘူး အဲ့လိုမျိုး နှစ်ပိုင်းခွဲထားရမယ်။ အဲ့လို နှစ်ပိုင်းခွဲခြားပြီး code ရေးနိုင်ဖို့ design patterns တွေကို နားလည်ထားဖို့ လိုတယ်။

Factory Method Pattern ရယ် Abstract Factory Pattern နှစ်ခုလုံးကလည်း ပြောင်းလဲတတ်တဲ့ အပိုင်းနဲ့ မပြောင်းလဲနိုင်တဲ့ အပိုင်းကို ခွဲခြားထားဖို့ရယ် Coding တွေ ရေးရာမှာ နဂိုရေးပြီးသား class ကို မပြင်ဘဲ အသစ်ထပ် functions တိုးဖို့ရယ်ကို အထောက်အကူပြုတဲ့ design pattern တစ်ခုဖြစ်ပါတယ်။

Example

Pizza Software တစ်ခုရဲ့ တည်ဆောက်ပုံနဲ့ အသေးစိပ် ထပ်ရှင်းပါမယ်။ ဘာ pattern မှ မသုံးထားတဲ့ စစချင်း Class Diagram ကတော့ -



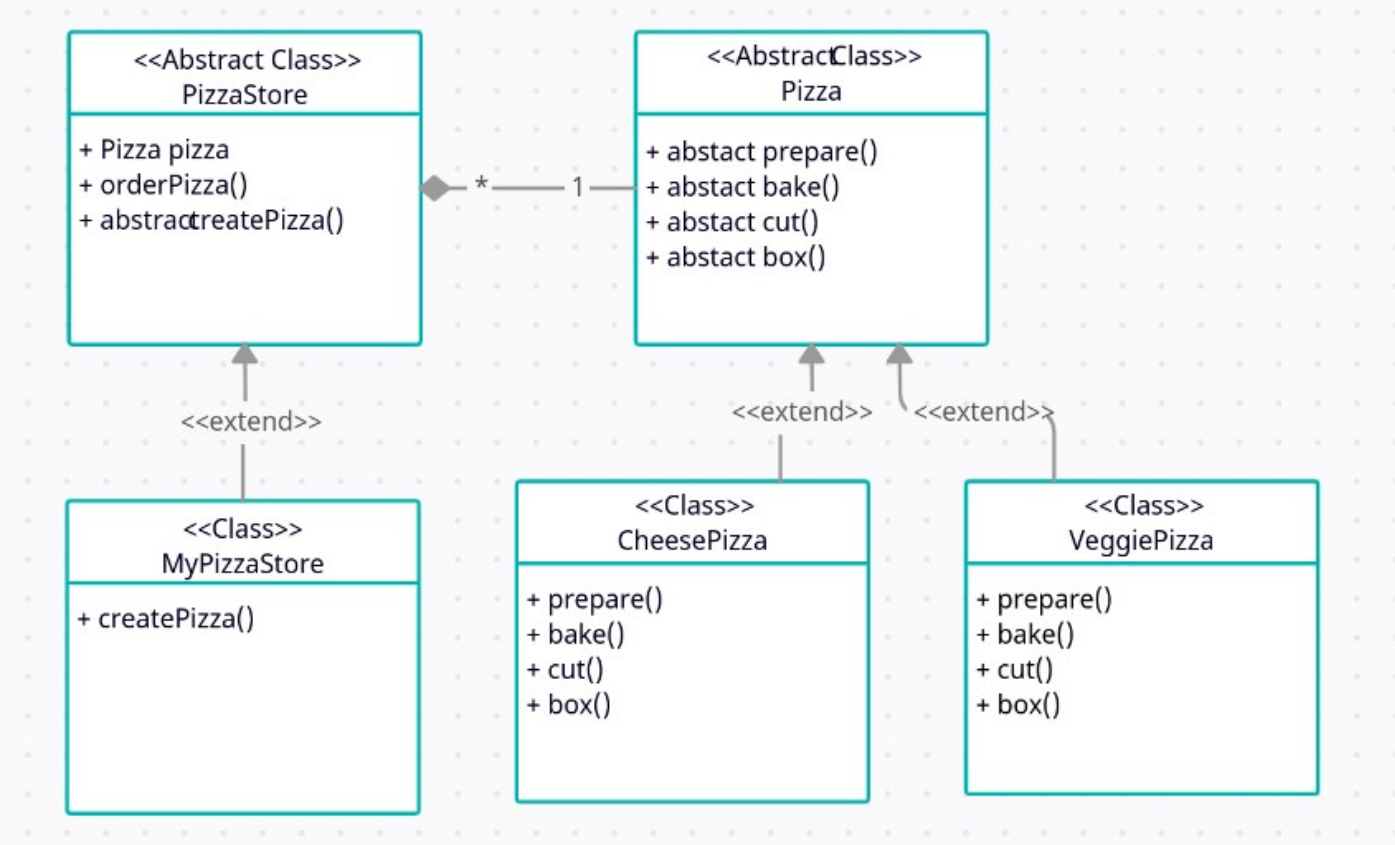
ပုံပါအတိုင်း Abstact Pizza class ကို pizza အမျိုးအစား နှစ်ခုက inheritance လုပ်ထားပြီး abstract method တွေကို implement လုပ်ထားတယ်။ PizzaStore class ကတော့ Pizza object တစ်ခုပါ ပြီး အဲ့ Pizza ကို order လုပ်တဲ့ method ပါ မယ်။ အဓိကအပိုင်းက Pizzastore ပိုင်းဖြစ်လို့ PizzaStore ရဲ့ code ကို အကျယ်တဝင့် ထပ်ရှင်းပါမယ်။

```
public class PizzaStore(){

    public void orderPizza(Stirng type) {
        Pizza pizza;
            if (type.equals(“cheese”)) {
                this.pizza = new CheesePizza();
        }
            if (type.equals(“veggie”)) {
                this.pizza = new VeggiePizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

}
```

အထက်ပါ code ကို ကြည့်ရင် တကယ်လို့ pizza အမျိုးအစား နောက်တခု ထပ်တိုးမယ်ဆိုရင် တိုးလာမယ့် pizza အမျိုးအစား အသစ်အတွက် PizzaStore ရဲ့ orderPizza method မှာ ပါ condition တစ်ကြောင်းထပ်တိုး စစ်ရမယ့် သဘောမှာရှိတယ်။ဆိုပေမယ့် အောက်က prepare တို့ bake တို့ကတော့ ပြောင်းစရာမလိုဘူး။ အဲ့လိုမျိုး ပြောင်းလဲတာနဲ့ မပြောင်းလဲတဲ့ အပိုင်း နှစ်ခုရော နေတဲ့ code ကို design pattern သုံးပြီးခွဲဖို့လိုလာပြီ။

အပေါ်က code ကို သေချာဂရုစိုက်ကြည့်မယ်ဆိုရင် ပြဿနာ ဖြစ်တဲ့ အပိုင်းက new ဆိုတဲ့ keyword ရှိနေတဲ့ လိုင်းတွေကြောင့်ဆိုတာ တွေ့ရလိမ့်မယ်။ Factory Method Pattern ဆိုတာ အဲ့လို object creation လုပ်တဲ့ code တွေကို သီးသန့်ခွဲထုတ်ထားပြီး sub class တစ်ခုက အဲ့ဒါကို inherit လုပ်ပြီးမှ object ဆောက်တဲ့ ပုံစံကို ခေါ်တာဖြစ်တယ်။

အပေါ်က code ကို Factory Method Pattern အရ class ဖွဲ့စည်းပုံ ကို ပြန်ပြင်ထား ပြီးတဲ့ class diagram ကို အောက်မှာ ဖော်ပြထားပါတယ်။



ပြောင်းလဲလိုက်တဲ့ PizzaStore class ရဲ့ code -

```
public abstract class PizzaStore {

    public Pizza orderPizza (String type) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
protected abstract Pizza createPizza (String type);
}
```

PizzaStore class ကို extends လုပ်ထားတဲ့ MyPizzaStore class ရဲ့ code -

```
public class MyPizzaStore extends PizzaStore {

    Pizza createPizza (String item) {

        if (item.equals(“cheese”)) {
            return new CheesePizza();
        }
        else if (item.equals(“veggie”)) {
            return new VeggiePizza();

        }
    }
}
```

အပေါ်က class နှစ်ခုရဲ့ တည်ဆောက်ပုံ ကို ကြည့်ရင်ကို သိသာပါတယ်။ အခု အချိန် မှာ တကယ်လို့ pizza အမျိုးအစားအသစ်တစ်ခု ကို ထပ်ဖြည့်ချင်ရင် MyPizzaStore ထဲက createPizza () method ရဲ့ Condition စစ်တဲ့နေရာမှာ တစ်ခု ထပ်ဖြည့်ရုံပါဘဲ။ မူရင်း PizzaStore class ထဲ ကနေ ပြောင်းလဲ့ရှိတဲ့ အပိုင်းကို Factory Method Pattern သုံးပြီး သီးသန့် ထုတ်ထားပြီးပြီဆိုတော့ design ပိုင်းက ပြည့်စုံသွားပါပြီ။

Factory Method Pattern မှာ ကတော့ object တစ်ခုတည်ဆောက်တာကို သီးသန့် ခွဲထုတ်တာဖြစ်ပြီး တကယ်လို့ တခုထက်ပိုတဲ့ objects တွေတည်ဆောက်တဲ့ အပိုင်း ကို သီးသန့်ခွဲထုတ် လိုက်ခြင်းကိုတော့ Abstract Factory Pattern လို့ခေါ်ပါ တယ်။

Singleton Pattern ဆိုသည်မှာ …

OOP Design Patterns တွေထဲမှာ အရိုးရှင်းဆုံး pattern ကတော့ Singleton Pattern ဘဲ ဖြစ်ပါတယ်။ သူ့ရဲ့ class Diagram တွေကလည်း class တစ်ခုသာပါဝင်တဲ့ အတွက် အရိုးရှင်းဆုံးဖြစ်ပါသည်။ Singleton Pattern ရဲ့ဆိုလိုရင်း ကတော့ Class တစ်ခုအတွက် Object တစ်ခုသာရှိခြင်းဖြစ်သည်။ ထပ်ပြီးအကျဲ့ချဲ့ ရှင်းရလျှင် class တစ်ခုအတွက် object တစ်ခုထပ်ပို တည်ဆောက်၍ မရအောင် ကန့်သတ်ထားခြင်း ဖြစ်သည်။

Example

Singleton Pattern က Design ပိုင်းအရ ရိုးရှင်းတဲ့အတွက်ကြောင့် coding ပိုင်းကို အဓိကထား ရှင်းပါမယ်။ ပုံမှန် object အသစ်ဆောက်တိုင်းမှာ သုံးလေ့ရှိတဲ့ code က “new ClassName()” ဆိုပြီး “new” ဆိုတဲ့ keyword နဲ့ class name တွဲခေါ်ပြီး တည်ဆောက်လေ့ ရှိတယ်။ အဲ့လို “new ClassName()” လို့ object တည်ဆောက် တဲ့အခါ တကယ် object တည်ဆောက်ပေးဖို့ နောက်ကွယ်က အလုပ်လုပ်ပေးတာကတော့ အဲ့ ClassName ဆိုတဲ့ class ရဲ့ default constructor က အလုပ်လုပ်ပေးသွားတာ ဖြစ်ပါတယ်။ မြင်သာအောင် code အသေးစိတ်ကို ဖော်ပြပါမယ်။

```
public MyClass () {
}

public Test () {
    MyClass object = new MyClass ();
}
```

အပေါ် က code မှာလိုမျိုး ဘာ constructor method ကိုမှ ထည့်မရေးပေးထားရင် default constructor ကို သုံးပြီး object တည်ဆောက်သွားပါလိမ့်မယ်။ ကိုယ်မရေးပေးထားပေမယ့် default အနေနဲ့ပါနေတဲ့ method ကို မြင်သာအောင်ဖော်ပြပါမယ်။

```
public MyClass () {
    MyClass(){}
}

public Test () {
    MyClass object = new MyClass ();
}
```

ပုံမှန် object အသစ် တည်ဆောက်တိုင်းမှာ highlight ပြထားတဲ့ အပေါ်က method လေးက နေတဆင့် တည်ဆောက်သွားလေ့ရှိသည်။ Singleton Pattern အရ object တည်ဆောက်ခွင့်ကို ကန့်သတ်ချင်ရင် အဲ့ method လေးကို ပြင်ဆင်ရုံပါဘဲ။

ပြင်ဆင်ပြီး code ကတော့ -

```
public MyClass () {
    private  MyClass(){}
    public static MyClass getInstance() {
        return new MyClass();
    }
}
public Test () {
    MyClass object =  MyClass.getInstance();
}
```

ပြောင်းလဲလိုက်တဲ့ code လေးတွေကို highlight ပြပေးထားပါတယ်။ ပထမဆုံး default constructor method ကို private ပြင်လိုက်ပါတယ်။ အဲ့တော့ တခြား class ကနေ object တည်ဆောက်လို့မရတော့ဘူး။ မူရင်း class ကနေ တည်ဆောက်ပေးမှ ရမှာဖြစ်လို့ မူရင်း class ထဲမှာ ClassName ကနေတဆင့် ခေါ်လို့ရနိုင်တဲ့ static method တစ်ခု ရေးပြီး object ကို return ပြန်ပေးထားလိုက်တယ်။ ဆိုပေမယ့် object construction ကို ထိန်းချုပ်လိုက်တာဘဲ ပြီးသေးတယ် အခု code က Singleton Pattern မဖြစ်သေးဘူး။ တကယ်လို့ MyClass.getInstance() ကို သာ နောက်တခါ ထပ်ခေါ်ရင် နောက်ထပ် object အသစ် တစ်ခုကို ထပ်ဆောက်နေမှာဖြစ်တယ်။ အဲ့အတွက် object တစ်ခုထပ်မပိုအောင် getInstance() မှာ condition ထပ်ဖြည့်မယ်။

```
public MyClass () {
        private static MyClass object;
        private  MyClass(){}
        public static MyClass getInstance() {
            if (object == null){
                object = new MyClass();
            }
            return object;
        }
    }
    public Test () {
        MyClass object =  MyClass.getInstance();
    }
```

ပြောင်းလဲလိုက်တဲ့ code လေးတွေကို highlight ပြပေးထားပါတယ်။ အပေါ်က code ပုံစံအတိုင်းဆိုရင်တော့ Singleton Pattern ဖြစ်သွားပါပြီ။ စစချင်း တစ်ကြိမ်မှ object မဆောက်ရသေးခင်မှာတော့ object ဆိုတဲ့ static variable ထဲမှာ ဘာမှ မရှိလို့ null ဖြစ်နေမှာ ဖြစ်ပြီး if condition ကာ true ဖြစ်ပြီး object ဆောက်သွားမှာ ဖြစ်ပြီး နောက်ပိုင်း တစ်ကြိမ် object ဆောက်ပြီး ချိန်မှာတော့ value တစ်ခု ရှိနေလို့ null မ ဖြစ်ဘဲ if condition က false ဖြစ်ပြီး ရှိပြီးသား object ကို ဘဲ return ပြန်ပေးသွားမှာ ဖြစ်ပါတယ်။

လက်ရှိ code က Singleton ဖြစ်ပြီဆိုပေမယ့် thread safe ဖြစ်ဖို့တော့လိုပါသေးတယ်။ တကယ်လို့ thread နှစ်ခု တပြိုင်နက်တည်း run လိုက်တဲ့ အချိန်မှာ if (object == null) လို့ တပြိုင်တည်းစစ်မိပြီး condition true ဖြစ်သွားမယ် ပြီးရင် thread နှစ်ခုလုံးက object နှစ်ခုကို ပြိုင်တူတည်ဆောက်လိုက်တဲ့ အခြေအနေမျိုးလည်း ဖြစ်လာနိုင်ပါတယ်။ အဲ့ အတွက် တစ်ကြိမ်မှာ thread တစ်ခုသာ method ကို ခေါ်နိုင်အောင် တစ်နည်းအားဖြင့် thread safe ဖြစ်အောင် method မှာ “synchronized” keyword ကို ထည့်သွင်း ပြီး lock ခတ်ထားလိုက်ပါမယ်။

```
public MyClass () {
        private static MyClass object;
        private  MyClass(){}
        public static synchronized MyClass getInstance() {
            if (object == null){
                object = new MyClass();
            }
            return object;
        }
    }
```

```
public Test () {
    MyClass object =  MyClass.getInstance();
}
```

အပေါ်က code က thread safe ဖြစ်သွားပြီ ဆိုပေမယ့် method တစ်ခုလုံးက synchronized ဖြစ်နေတဲ့ အတွက် performance ပိုင်းမှာ အားနည်းသွားပါလိမ့်မယ်။ တကယ်လို အပ်တဲ့ နေရာ ဖြစ်တဲ့ if (object == null) လို့ condition စစ်တဲ့အခါ true ဖြစ်တဲ့ အပိုင်းကိုသာ သီးသန့် synchronized block ထဲထည့်လိုက်မယ်ဆိုရင်တော့ therad safe လည်းဖြစ်သွားမယ် နောက်ထပ် performance ပိုင်းပါ ကောင်းသွားပါလိမ့်မယ်။ synchronized block မှာ parameter အနေနဲ့ lock ခတ်မယ့် class ကိုထည့်ပေးရပါတယ်။ နောက်တခု static variable တွေကို multi thread ကနေ access လုပ်တဲ့ အခါ thread safe ဖြစ်စေဖို့ “volatile” keyword လေးပါသုံးပါ။

နောက်ဆုံး code ပုံစံကတော့ -

```
public MyClass () {
        private volatile static MyClass object;
        private  MyClass(){}
        public static MyClass getInstance() {
            if (object == null){
                synchronized (MyClass.class){
                    if (object == null){ // double check
                        object = new MyClass();
                        }
                    }
                }
            return object;
        }
    }

    public Test () {
        MyClass object =  MyClass.getInstance();
    }
```

အပေါ်က example တွေ သေချာဖတ်ပြီးရင်တော့ Design ပိုင်းရိုးရှင်းပေမယ့် coding ပိုင်း မှာ သတိထားရမယ့် Singleton Pattern ရဲ့ implementation အပိုင်းတွေ ကိုပါသေချာ နားလည်သွားပါလိမ့်မယ်။

Command Pattern ဆိုသည်မှာ .....

Command ဆိုတာ က အမိန့်ပေးခိုင်းစေချက် လို့ မြန်မာလို ဘာသာပြန်ဆိုလို့ရ တယ်။ ဆိုပေမယ့် Computer အသုံးအနှုန်းနဲ့ ကိုက်ညီအောင် ဘာသာပြန်ရမယ်ဆိုရင် လုပ်ဆောင်ချက် (function) လို့ဘာသာပြန်ရင်တော့ ပိုနားလည်လွယ်မယ်။ ကျတော်တို့ computer software တစ်ခုချင်းဆီမှာ functions တွေအများကြီးရှိတယ်။ စာစီစာရိုက် software တစ်ခုမှာ သာမန်အားဖြင့် ပါဝင်လေ့ရှိတဲ့ “save, edit, save as” အစရှိတဲ့ ဟာတွေ တခုချင်းစီကို function တစ်ခုချင်းစီအဖြစ် အသေးစိတ် ခွဲခြားကြည့်ပါ။ ပုံမှန် သုံးနေတဲ့ အချိန်မှာ သတိမထားမိပေမယ့် သေချာသတိထားပြီး ရေတွက်ကြည့်ရင် software အသေးလေးတစ်ခုမှာ တောင် functions တွေက ရာချီပါနေတတ်တယ်။

အဲ့လို function တွေရာထောင်ချီ လာတဲ့ software မျိုးကို စနစ်တကျ တည်ဆောက်နိုင်ဖို့ Command Pattern ကို အသုံးပြုလေ့ရှိတယ်။ Command Pattern ရဲ့ အဓိပ္ပါယ်ဖွင့်ဆိုချက်ကတော့ function ကို တိုက်ရိုက် မခေါ်ဘဲ object တစ်ခုအနေနဲ့ encapsulate လုပ်ထားပြီးမှ အဲ့ object ကနေတဆင့်ခေါ်တယ်။ ပြီးရင် အဲ့ function နဲ့ သက်ဆိုင်တဲ့ undo ကို ပါထောက်ပံ့ပေးခြင်း လို့ ဖွင့်ဆိုထားတယ်။

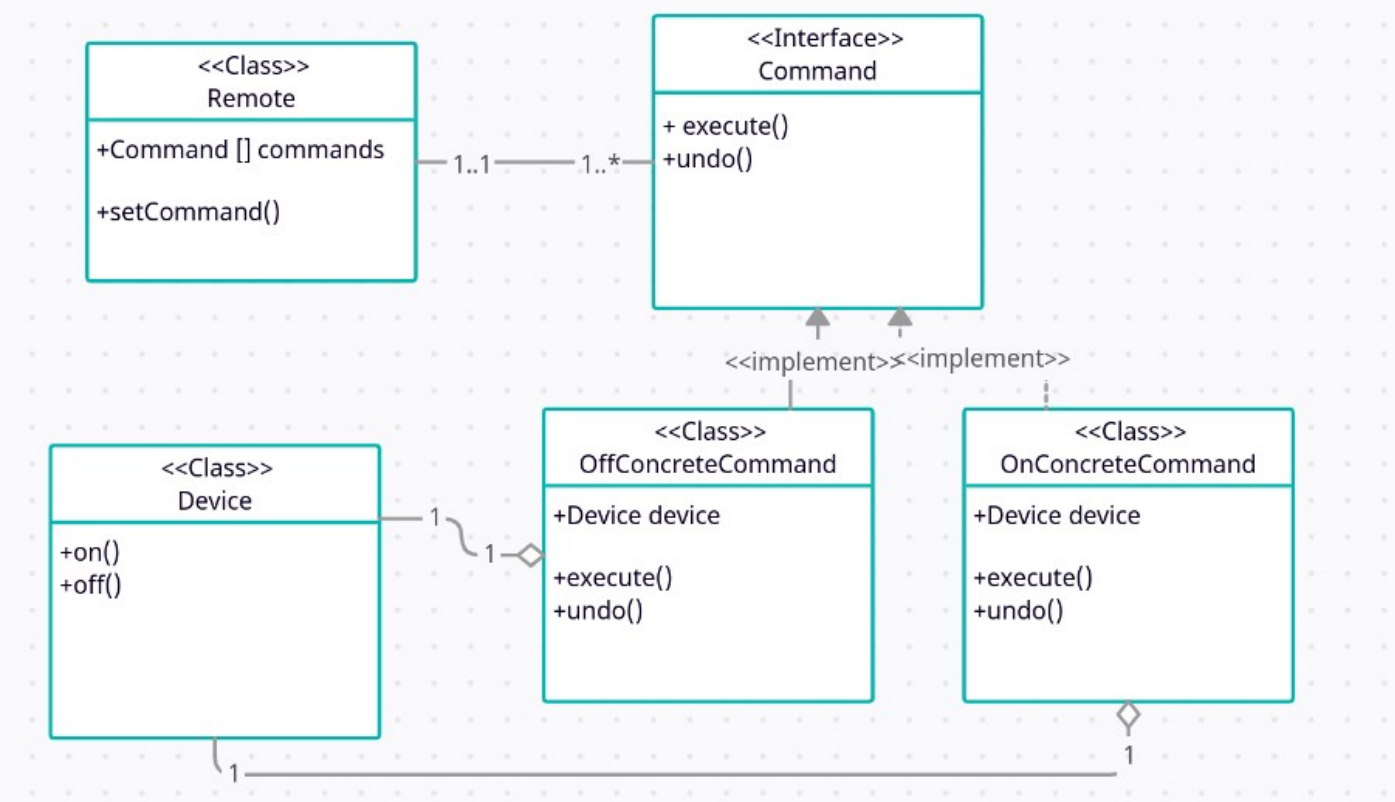
ထပ်ပြီး အကျဲ့ချဲ့ရှင်းရလျှင် function ကို request လုပ်မယ့် object နဲ့ function ကို တကယ် လုပ်ဆောင် ပေးမယ့် object ကြားက အပြန်အလှန် မှီခိုမှုကို လျော့ချစေချင်တဲ့ အတွက် နှစ်ခုကြားမှာ interface တစ်ခုခံထားပြီး အဲ့ကနေတဆင့် အလုပ်လုပ်သွားတဲ့ ပုံစံမျိုးပါ။ နောက်ထပ် undo ဆိုတာ က ဥပမာ function ရဲ့လုပ်ဆောင်ချက် က ထည့်လိုက်တဲ့ ကိန်းဂဏန်းကို နှစ်ဆတိုးစေတာ (၂ နဲ့မြှောက်တာ) ဆိုရင် အဲ့ function ရဲ့ undo က တစ်ဆကိုပြန်လျော့စေတာ (၂ နဲ့စားတာ) ဖြစ်တယ်။ အဲ့လို မျိုး function စရေးကတည်းက သက်ဆိုင်ရာ undo ကိုပါတစ်ခါတည်း ထည့်ရေးသွားတာမျိုးကို ဆိုလိုတယ်။

Example

ထပ်ပြီးသေချာသဘောပေါက်စေဖို့ universal remote control api လေးကို ဆက်ရှင်းပြပါမယ်။

ပုံမှန် ဆိုရင်တော့ remote တစ်ခုမှာ ပါဝင်တဲ့ key က fixed အသေဖြစ်တယ်။ အသေမို့လို့ code က ထပ်ပြင်စရာမလိုဘူး။ universal ကျတော့ code တွေက သက်ဆိုက်ရာ brand တွေရဲ့ remote အားလုံးရဲ့ key အားလုံးကို ထည့်ရေးဖို့လိုတယ်။ အဲ့အတွက် အသေတခါတည်းရေးဖို့ ဆိုတာ မဖြစ်နိုင်ဘူး။ တကယ်လို့ အသေတခါတည်းရေးလိုက်ရင် နောက်ထွက်လာမယ့် ပစ္စည်းအသစ်တွေ အတွက် remote အသစ်တွေကို ထပ်ထည့်ဖို့က မဖြစ်နိုင်တော့ဘူး။ အဲ့တော့ api (application programming interface) တစ်ခုလိုလာပြီ။ api ကို သုံးမယ့်သူက ပစ္စည်းအသစ် ထုတ်တဲ့ company ဖြစ်တယ်။ ပစ္စည်းအသစ် ထုတ်တိုင်း မှာ သက်ဆိုင်ရာ remote ကို api ကနေတဆင့် universal remote မှာ ထပ်ဖြည့်သွားမယ့်ပုံစံမျိုးဖြစ်တယ်။

ရေရှည် ထပ်ခါထပ်ခါ ပြုပြင်ရမှာ ဖြစ်တဲ့ အတွက် “open close principal” အရ “open for extension and close for modification” ဖြစ်အောင် functions တွေကို command pattern သုံးပြီး တည်ဆောက်ထားတဲ့ class diagram ကို တဖက် စာမျက်နှာမှာ ဖော်ပြပေးထားပါတယ်။



အပေါ် က class diagram ကတော့ ရိုးရှင်းပါတယ်။ Remote Class ထဲမှာ ကျတော်တို့ commands တွေ တနည်းဆိုရလျှင် keys တွေကို array အနေနဲ့ ထားထားမယ်။ နောက် setCommand() method ကနေ တဆင့် array ကို initialize လုပ်မယ်။ Command pattern အရ function execute လုပ်ဖို့ရယ် undo လုပ်ဖို့ရယ် အတွက် execute နဲ့ undo ဆိုတဲ့ implement မလုပ်ရသေးတဲ့ method နှစ်ခုပါတဲ့ command interface ရှိမယ်။ တဖက်မှာ တော့ ကျတော်တို့ company တွေရဲ့ ထုတ်ကုန်တွေတစ်ခုချင်းစီက Device Class အနေနဲ့ သီးသန့် ရှိနေမယ်။ အဲ့ Device တစ်ခုချင်းစီမှာ ရှိတဲ့ function တစ်ခုချင်းဆီအတွက် သူနဲ့ သက်ဆိုင်ရာ Command Interface ကို implements လုပ်ထားတဲ့ ConcreteCommand Class ရှိရမယ်။ တကယ်လို့ ထုတ်ကုန်အသစ်တိုးရင် တခြားcode ကို ထိစရာမလိုဘဲ နောက်ထပ် Device Class တစ်ခုနဲ့ device ရဲ့ function တစ်ခုချင်းဆီအတွက် သူနဲ့ သက်ဆိုင်ရာ Command Interface ကို implements လုပ်ထားတဲ့ ConcreteCommand Class အသစ်တွေကို ထပ်တိုးရေးရုံပါဘဲ။ code အပြည့်အစုံကို တဖက်စာမျက်နှာမှာ ဖော်ပြပေးထားပါတယ်။

```
public class Device {

    public void On() {
        System.out.println("Device On!");

    }

    public void Off() {
        System.out.println("Device Off!");

    }
}


public interface Command {

    public void execute();

    public void undo();
}


public class OnConcreteCommand implements Command {

    Device device;

    @Override
    public void execute() {
        device.On();
    }

    @Override
    public void undo() {
        device.Off();
    }

}
```

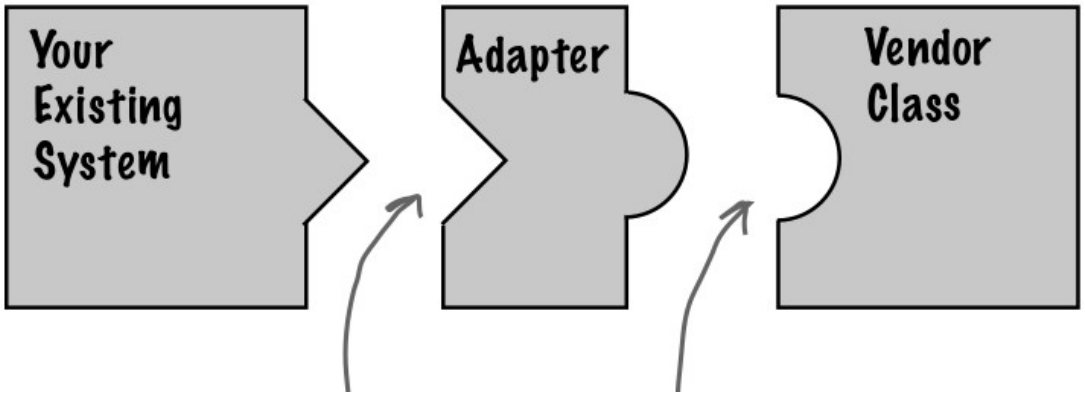
```
public class OffConcreteCommand implements Command {

    Device device;

    @Override
    public void execute() {
        device.Off();
    }

    @Override
    public void undo() {
        device.On();
    }

}

public class Remote {

    Command [] commands;


    public void setCommands(Command[] commands) {
        this.commands = commands;
    }

}


public class CommandPattern {

    public static void main(String[] args) {

        OnConcreteCommand onCommand = new OnConcreteCommand();
        OffConcreteCommand offCommand = new
         OffConcreteCommand();
        Command[] commands = {onCommand, offCommand};
        Remote newRemote = new Remote();
        newRemote.setCommands(commands);
    }

}
```

Adapter Pattern ဆိုသည်မှာ .....

Adapter ဆိုတဲ့ စကားလုံးကို လျှပ်စစ်ဆိုင်ရာ ပစ္စည်းတွေမှာ ကြားဖူးကြမှာပါ။ ကျတော်တို့ အိမ်သုံးလျှပ်စစ်ပစ္စည်းတွေရဲ့ မီးကြိုးplug က ၃ pin ဖြစ်ပြီး အိမ်မှာရှိတဲ့ အထိုင်(plug) ပေါက်မှာ က ၂ pin ပေါက်ဘဲ ပါတာမျိုးဆို ရင် သူတို့နှစ်ခုကြားမှာ AC Power Adapter အတိုကောက် Adapter လေးကိုကြားခံထားပြီး သုံးရပါတယ်။

အောက်မှာ ပုံလေးဖော်ပြပေးထားပါတယ်။

ကျတော်တို့ design pattern မှာလည်း အလားတူပါဘဲ အဓိက ကတော့ သူများ api တွေနဲ့ ချိတ်ဆက်ရတဲ့ အပိုင်းမှာ အသုံးများပါတယ်။ api ဘက်ကပြောင်းလဲမှုတွေရှိလာလို့ နဂိုရှိရေးထားတဲ့ code နဲ့ မကိုက်ညီတဲ့ အခါမှာ ရှိနေတဲ့ ကိုယ့် code ကို မပြင်ဘဲ ကြားမှာ ချိတ်ဆက်ဖို့ code အသစ် ကို ထပ်ဖြည့်တဲ့ ပုံစံမျိုးကို Adapter Pattern လို့ ခေါ်ပါတယ်။



အပေါ်ကပုံစံမျိုး နဂိုရှိရေးထားတဲ့ code နဲ့ api က update လုပ်လိုက်တဲ့ code က အံဝင်ခွင်ကျ မဖြစ်တော့တဲ့ အခါ နဂိုရှိရေးထားတဲ့ code အပိုင်းကို မပြင်ဘဲနဲ့ သူတို့ နှစ်ခု ကြားထဲမှာ ကြားခံအဖြစ် code အသစ်ထပ်ဖြည့် ပြီး နှစ်ခုကို ချိတ်ဆက်ထားတဲ့ ပုံစံမျိုး ကို Adapter Pattern လို့ခေါ်ပါတယ်။

အဲ့လို ထပ်ဖြည့်တဲ့ နေရာမှာ ဖြည့်ပုံဖြည့်နည်း အပေါ်အခြေခံပြီး Object Adapter နဲ့ Class Adapter ဆိုပြီး Adapter Pattern နှစ်မျိုးထပ်ကွဲပါသေးတယ်။ Object Adapter ကတော့ interface implementation and composition နည်း ကို အသုံးပြုပြီး Class Adapter ကတော့ multiple inheritance နည်း ကို အသုံးပြုပါတယ်။ Java က multiple inheritance ကို support မပေးတာကြောင့် Class Adapter ရဲ့ code example ကိုတော့ ရှင်းပြပေးလို့မရနိုင်ပါဘူး။

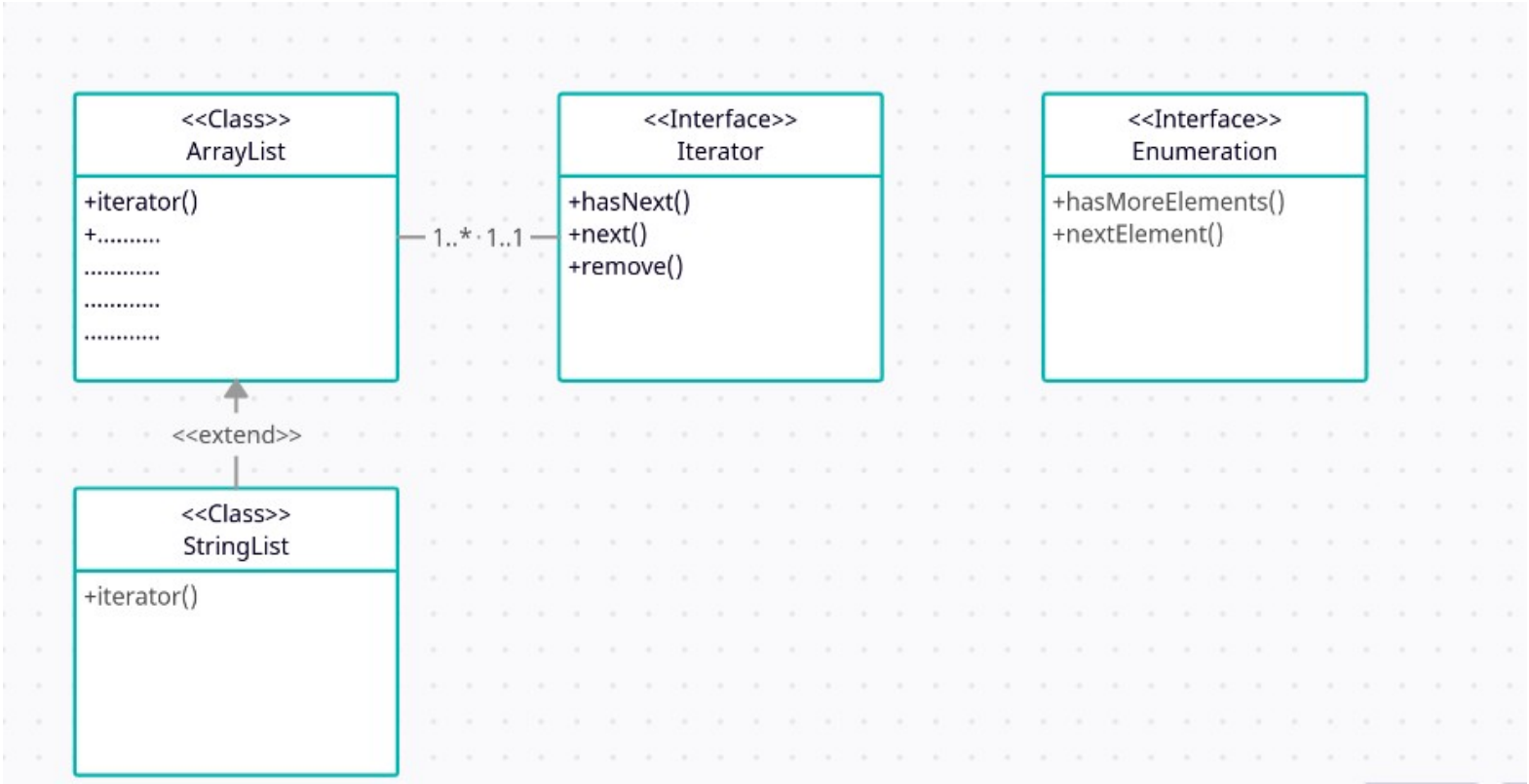
Example

Object Adapter ကို စရှင်းပါမယ်။ နားလည်လွယ်အောင် java build-in ပါတဲ့ class တွေနဲ့ဘဲ ရှင်းပါမယ်။ အစောပိုင်း java version တွေမှာကျတော်တို့ Collections တွေကို loop လုပ်ပြီး elements တွေကို ထုတ်ပြတဲ့ method တွေကို java.util.Enumeration interface မှာရေးထားပါတယ်။ နောက်ပိုင်း အသစ် java version တွေမှာတော့ java.util.Iterator interface ဆိုပြီး interface အသစ်တစ်ခု သုံးပြီးရေးထားပါတယ်။

ပြဿနာက အရင် develop လုပ်ထားတဲ့ code တွေက နဂိုရဲ့ Enumertion interface ကို သုံးထားပြီး နောက်ထပ်အသစ် code တွေမှာကျတော့ Iterator interface ကို သုံးရင် standard မကျဖြစ်မှာကြောင့် နဂိုရဲ့ ရှိနေတဲ့ code ကော အသစ် code ကိုပါ Iterator တစ်ခုတည်း ကိုဘဲ သုံးပြီးရေးရပါမယ်။ ပုံမှန်ဆိုရင် တော့ Enumeration မှာ ကိုယ်ပြင် ထားတဲ့ code တွေအားလုံး အတွက် အစအဆုံးနီးပါး အကုန်လုံး ကို Iterator မှာပါ ပြန်ပြင်ရမှာပါ။

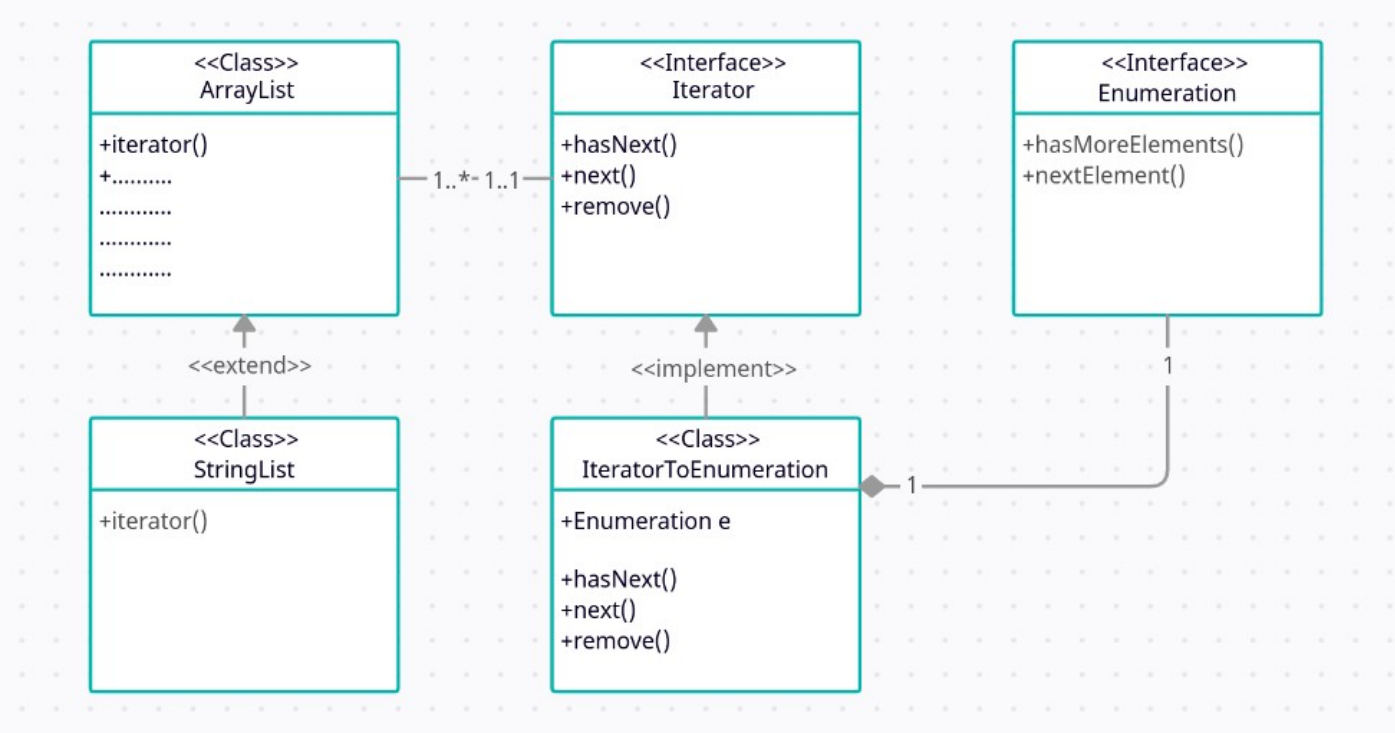
ဆိုပေမယ့် တကယ်လို့ အရင် code တွေမှာ Iterator ကို ခေါ်တဲ့အခါ Enumeration ကိုဘဲ ပြန်သုံးတဲ့ ပုံစံမျိုး ရေးနိုင်ရင်တော့ Iterator မှာ ဘာမှ ထပ်ရေးဖို့ မလိုတော့ ဘူး။ Iterator ကိုခေါ်တဲ့ အချိန်မှာ ကြားထဲက Adapter class လေးက Enumerator ကို ပြန်ခေါ်သုံးပေးနိုင်မယ့် ပုံစံမျိုးလေး နဲ့ ပြဿနာကို ဖြေရှင်းပါမယ်။

မဖြေရှင်းရသေးခင် အရင် ရှိနေတဲ့ ပုံစံကို class diagram နှင့် အောက်မှာ ဖော်ပြ ပေးထားပါတယ်။



ArrayList ကို extend လုပ်ထားတဲ့ StringList class ရှိမယ်။ တကယ်တော့ ArrayList နဲ့ Iterator ကတိုက်ရိုက်ကြီးချိတ်ဆက် ထားတာမဟုတ်ဘူး။ နားလည်ရ လွယ်အောင် တိုက်ရိုက်ချိတ်ပြထားခြင်းဖြစ်ပါတယ်။ နောက် ဘာနဲ့မှ အချိတ်အဆက်မ ရှိတဲ့ Enumeration interface ရှိမယ်။

လိုချင်တဲ့ ပုံစံက StringList ရဲ့ Iterator ကိုခေါ်သုံးတဲ့ အခါ Iterator အစား Enumeration က ဝင်ပြီးအလုပ်လုပ်တဲ့ ပုံစံမျိုးဖြစ်ပါတယ်။ Adapter pattern ကို သုံး ပြီး လိုချင်တဲ့ ပုံစံရအောင်ပြင်ထား တဲ့ class diagram ကို အောက်မှာ ဖော်ပြပေးထားပါ တယ်။



IteratorToEnumeration ဆိုတဲ့ Adapter class လေးထပ်ထည့်တယ်။ အဲ့ class ထဲမှာ ကိုယ်တကယ်သုံးမယ့် interface ရဲ့ object တစ်ခုကို field variable အဖြစ်ထည့် ထားမယ်။ နောက် ပြောင်းချင်တဲ့ interface ကို implements လုပ်လိုက်ရုံပါဘဲ။

အသေးစိတ် ပိုမြင်သာအောင် code အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါ တယ်။ Interface တွေရဲ့ code ကတော့ method name တွေသာဖြစ်တဲ့ အတွက် မ ဖော်ပြတော့ပါဘူး။ ArrayList ကလည်း java build-in class ဖြစ်တဲ့အတွက် ထပ်မံ မ ဖော်ပြတော့ပါဘူး။

```
public class StringList <String> extends ArrayList <String> {

    @Override
        public Iterator<String> iterator() {

            Object[] elementData =StringList.this.toArray();

            List data = Arrays.asList(elementData);

            Enumeration e = Collections.enumeration(data);

            return new IteratorToEnumeration(e);

    }

}
```

ArrayList ကို inheritance လုပ်ထားပြီး iterator method ကို override လုပ်ထားပါတယ်။ ArrayList ရဲ့ Enumeration object ကို ရဖို့အတွက် List object တစ်ခုကို ထည့်ပေးရပါမယ်။ လိုအပ်တဲ့ List object ကို ရဖို့အတွက် toArray() method ကိုသုံးပြီး Object array အရင်ယူပြီး နောက် Object array ကို List အဖြစ်ထပ်ပြောင်းပါတယ်။ List object ရပြီးတဲ့နောက် Collections.enumeration() method မှာ List object ကိုထည့်ပေးပြီး Enumeration object ကိုယူထားပါတယ်။ နောက်ဆုံးအဆင့်အနေနဲ့ IteratorToEnumeration object ကို enumeration object ကို အခြေခံပြီး ဆောက်ပြီး return ပြန်ပေးလိုက်ရုံပါဘဲ။ အဲ့တော့ iterator() method ကို ခေါ်လိုက်ရင် နဂိုရ် Iterator object အစား IteratorToEnumeration object ကို return ပြန်ပါလိမ့်မယ်။

```
public class IteratorToEnumeration implements Iterator {

    Enumeration e;

    public IteratorToEnumeration(Enumeration e) {
        this.e = e;
    }

    @Override
    public boolean hasNext() {
        return e.hasMoreElements();
    }

    @Override
    public Object next() {
        return e.nextElement();
    }

    @Override
    public void remove() {
        System.err.println("This operation is not
supported!");
    }

}
```

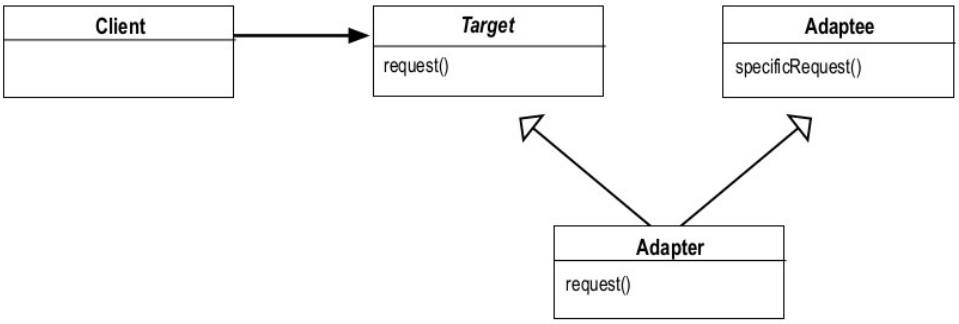
IteratorToEnumeration ဆိုတဲ့ နာမည်အတိုင်းဘဲ Iterator ကနေ Enumeration အဖြစ် ပြောင်းပေးမယ့် Adapter class ဘဲဖြစ်ပါတယ်။ ပြောင်းရမယ့် interface ဖြစ်တဲ့ Iterator ကို implements လုပ်ပြီး ပြောင်းချင်တဲ့ interface ဖြစ်တဲ့ Enumeration interface ရဲ့ object တစ်ခုကို field variable အဖြစ် ထားထားပါတယ်။ နဂိုရှိ Iterator interface ရဲ့ method တွေအကုန်လုံးကို override လုပ်ပြီး သက်ဆိုင်ရာ Enumeration object ရဲ့ method တွေကို ခေါ်သုံးလိုက်ပါတယ်။ Enumeration object မှာ မရှိတဲ့ remove method အတွက်တော့ error message လေးပြတဲ့ ပုံစံဘဲ ရေးထားပါတယ်။

```
public class Test {

    public static void main(String[] args) {

        ArrayList list = new StringList<>();

        String arr[] = {"orange", "banana"};

        list.addAll(Arrays.asList(arr));

        Iterator i = list.iterator();

        while (i.hasNext()) {

            System.out.println(i.next());

             i.remove();
        }


    }

}
```

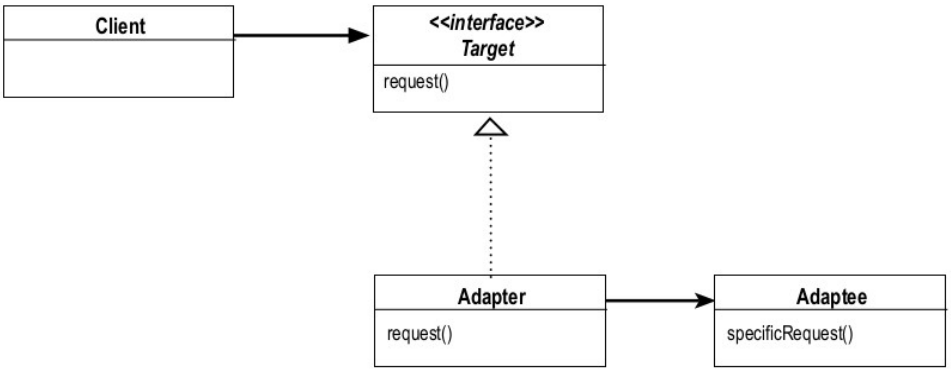
IteratorToEnumeration adapter တကယ်အလုပ်လုပ်လား သိချင်တဲ့ အတွက် အပေါ်က Test class လေးနဲ့စမ်းထားပါတယ်။ စစချင်း StringList object တစ်ခု ဆောက်ပြီး အဲ့ object ထဲကို data သွင်းတယ်။ ဒုတိယအဆင့် အနေနဲ့ အဲ့ object ရဲ့ Iterator object ကို ထပ်ယူပြီး အဲ့ Iterator object ကနေမှတဆင့် StringList ကို loop ပတ်ပြီး ထုတ်ပြတာတို့ remove လုပ်တာတို့ လုပ်ကြည့်ထားပါတယ်။ ထုတ်ပြတာကတော့ Enumeration ရဲ့ nextElement() method ကိုသုံးပြီး ထုတ်ပြပေး ပေမယ့် remove ကတော့ ကျတော်တို့ ရေးထားတဲ့ error message လေးဘဲ ပြမှာဖြစ်ပါတယ်။ အပေါ်က ရှင်းပြချက်ကတော့ object adapter pattern အကြောင်း အပြည့်အစုံဘဲ ဖြစ်ပါတယ်။

နောက်ထပ် class adapter pattern ကတော့ mutiple inheritance ဖြစ်တာကြောင့် code implementation တော့ ရှင်းပြနိုင်မှာ မဟုတ်ပါဘူး။ design ပုံလေးတော့ အောက်မှာ ဖော်ပြပေးထားပါတယ်။

Class Adapter



Object Adapter



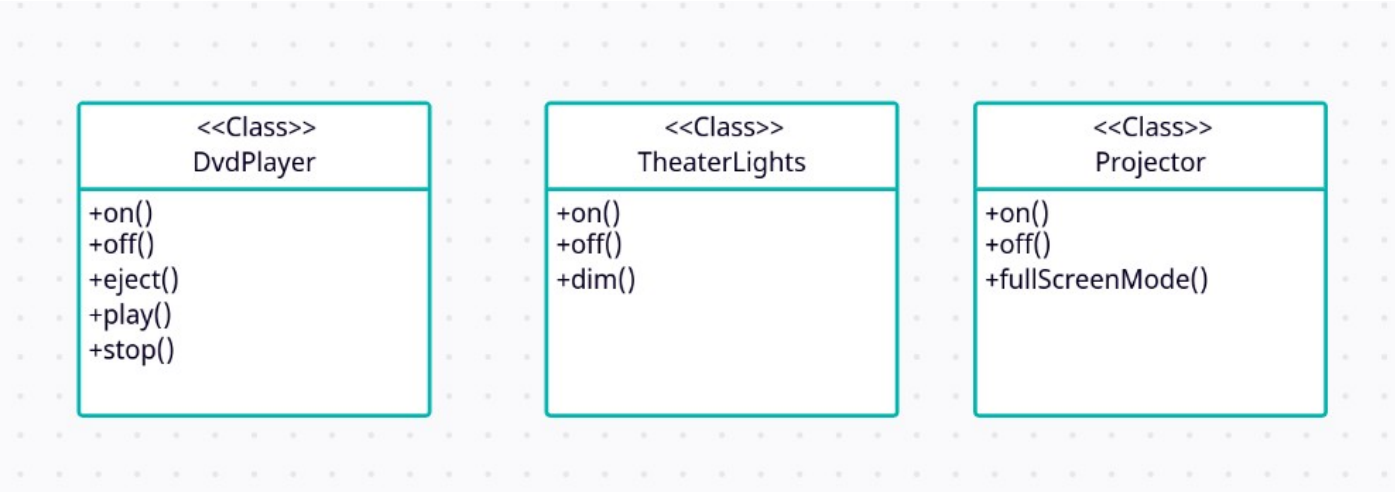
အပေါ်ကပုံမှာ နှိုင်းယှဉ်ပြထားသလို ပါဘဲ object adapter pattern ကတော့ ပြောင်းရမယ့် interface ကို implement လုပ်ပြီး ပြောင်းချင်တဲ့ class ကို composition အရ ထည့်သုံးပါတယ်။ class adapter pattern ကတော့ ပြောင်းရမယ့် class ကိုရော ပြောင်းချင်တဲ့ class ကိုပါ inherit လုပ်ထားပြီး ပြောင်းပေးတဲ့ ပုံစံမျိုးဖြစ်ပါတယ်။

Facade Pattern ဆိုသည်မှာ....

Facade ဆိုတာကို မြန်မာပြန်ရင် "မျက်နှာစာ" လို့အဓိပ္ပါယ်ရတယ်။ အဆောက်အဦးတွေ အိမ်တွေ ရဲ့ "မျက်နှာစာ" ကိုဆိုလိုပါတယ်။ ပုံမှန် အိမ်တွေရဲ့ မျက်နှာစာ ကို သတိထားကြည့်ကြည့်ပါ အိမ်အတွင်းပိုင်းနဲ့ နှိုင်းယှဉ်ရင် ပိုပြီး ရိုးရိုးရှင်းရှင်း နဲ့ ကြည့်ကောင်းအောင် ပြင်ဆင်ထားတတ်ကြပါတယ်။ Design Pattern အရ Facade Pattern ဆိုတာ ကလည်း အိမ်တွေရဲ့ မျက်နှာစာ ကဲ့သို့ပါဘဲ coding ကို အသုံးပြုရ လွယ်ကူအောင် ရိုးရှင်းအောင် ပြုလုပ်လိုက်တဲ့ ပုံစံကို ဆိုလိုပါတယ်။ Facade Pattern က တခြား patterns တွေနဲ့ နှိုင်းယှဉ်ရင် အတော်လေး နားလည်ရ လွယ်ကူပါတယ်။

Example

ရုပ်ရှင်ရုံတစ်ခု ရဲ့ program နဲ့ ရှင်းပြပါမယ်။ program ရဲ့ class diagram ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။



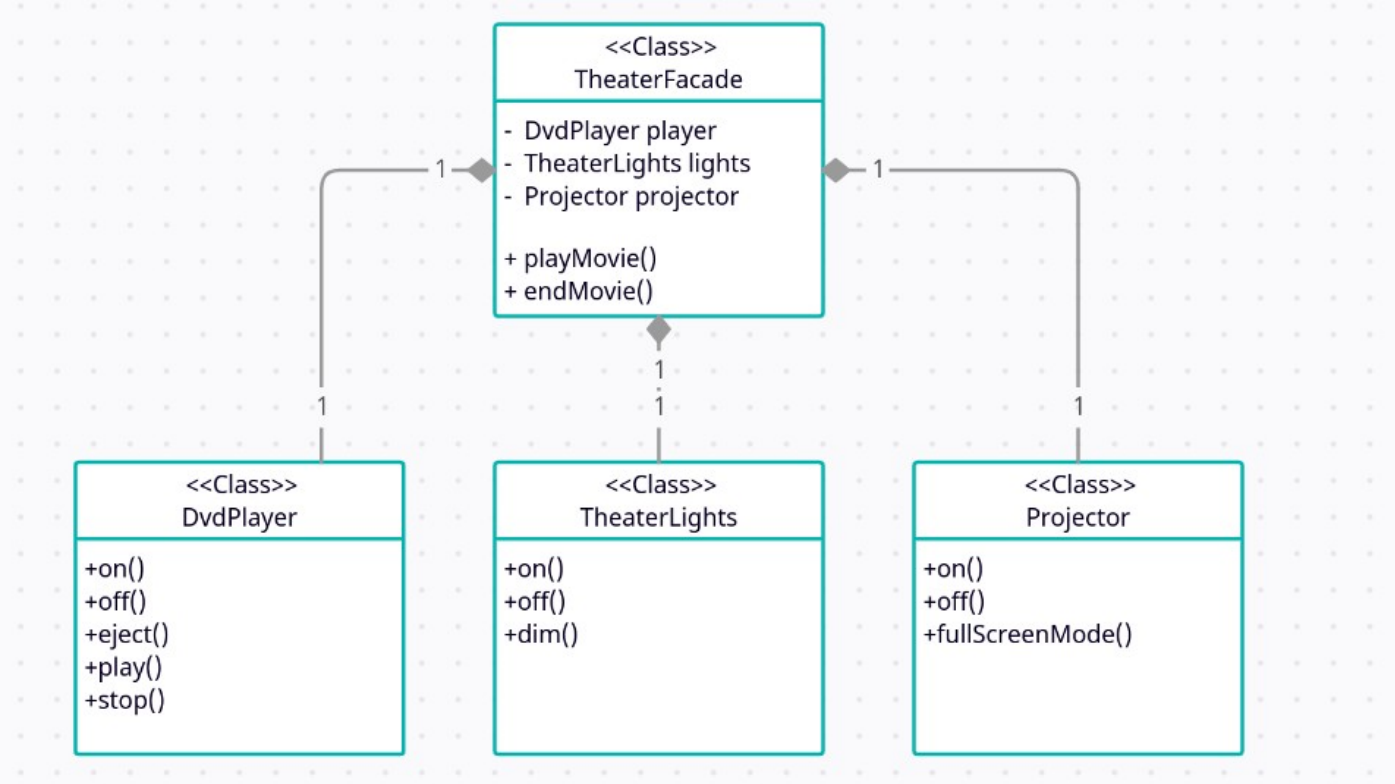
အမှန်တကယ် ကတော့ class တွေကအများကြီး ပါဝင်ပါတယ် ဆိုပေမယ့် ရှင်းပြရ လွယ်ကူအောင် လိုအပ်တဲ့ class တွေကို ဘဲ ဖော်ပြပေးထားပါတယ်။ အဓိက လိုအပ်တဲ့ dvd စက် ၊ မီး နဲ့ projector ကို control လုပ်တဲ့ class သုံးခုကို ဖော်ပြပေးထားပါတယ်။

အခုလက်ရှိ class design အရ ရုပ်ရှင်ရုံမှာ ရုပ်ရှင်တကား ပြဖို့အတွက် လိုအပ်တဲ့ code ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public class Test {

    public static void main(){

        DvdPlayer player = new DvdPlayer();
        TheaterLights lights = new TheaterLights();
        Projector projector = new Projector();
        lights.on();
        lights.dim();
        player.on();
        player.eject();
        player.play();
        projector.on();
        projector.fullScreenMode();

    }
}
```

အပေါ်က Test Class မှာ ဖော်ပြထားသလိုပါဘဲ ရုပ်ရှင်စဖွင့်တဲ့ အချိန်မှာ တစ်ဆင့် ချင်းဆီ လုပ်စရာ ရှိတာ တွေကို လုပ်သွားရမယ်။ တကယ်လို့ ပိတ်ချင်ပြီဆိုရင်လည်း အပေါ်ကလိုမျိုး ပိတ်တဲ့ အဆင့်တွေကို တစ်ဆင့်ချင်းစီ လုပ်ဆောင်သွားရမှာ ဖြစ်ပါတယ်။ အစအဆုံးရေးထားတဲ့ developer အတွက် ကတော့ အပေါ်ကလိုမျိုး class တစ်ခုချင်းစီရဲ့ method တွေကို အလွယ်တကူ ခေါ်သုံးနိုင်ပေမယ့် တခြား developer တစ်ယောက်အတွက် တော့ code တွေအကုန် အစအဆုံး ဖတ်ရမယ် ပြီးမှ ခေါ်သုံးနိုင်တဲ့ အဆင့်ကို ရောက်မယ်။ အဲ့အတွက် အလွယ်တကူ တခြားသူပါ သုံးလို့ရနိုင်အောင် facade pattern ကို သုံးပြီး class design ကိုပြန်ပြင်ဆွဲပါမယ်။

Facade Pattern အရ ဆွဲထားတဲ့ class design ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။



အပေါ်က class design မှာတော့ နဂိုရဲ့ class တွေကို အလွယ်တကူ သုံးနိုင်အောင် facade class တစ်ခု ထပ်ထည့်ထားပါတယ်။ တခြားခေါ်သုံးရမယ့် class တွေရဲ့ object တွေကို field variable အဖြစ် ထားထားပြီး လိုအပ်တဲ့ အချိန် ခေါ်သုံးတဲ့ ပုံစံမျိုးကို အသုံးပြုထားပါတယ်။ အားသာချက် အနေနဲ့ တခြား class တွေကို အသေးစိတ် သိစရာမလိုဘဲ facade class ကို သိရုံနဲ့ အလွယ်တကူ အသုံးပြုလို့ရသွားပါပြီ။

TheaterFacade class ရဲ့ code အနည်းငယ်ကို အောက်တွင် ဖော်ပြပေးထားပါတယ်။

```
public class TheaterFacade {

    DvdPlayer player;
    TheaterLights lights;
    Projector projector;

    public TheaterFacade(){

        this.player = new DvdPlayer();
        this.lights = new TheaterLights ();
        this.projector = new Projector();

    }


    public void playMovie(){

        lights.on();
        lights.dim();
        player.on();
        player.eject();
        player.play();
        projector.on();
        projector.fullScreenMode();

    }
}
```

တကယ်တော့ တခြား method တွေလည်းအများကြီး ရှိပေမယ့် သဘောတရားနားလည်ဖို့ ရှင်းတာဖြစ်တဲ့ အတွက်ကြောင့် အဓိကထား ရှင်းချင်တဲ့ playMovie() method လေးကို ဘဲ ဖော်ပြပေးလိုက်ပါတယ်။ တခြား class သုံးခုလုံးကို အစီအစဉ်တကျ ထိန်းပြီး လုပ်ဆောင်ပေးသွားတဲ့ playMovie() method ကြောင့် နဂိုရှိခေါ်သုံးရခက်နေတဲ့ အပိုင်း ကို အောက်မှာ ဖော်ပြထားသလို အလွယ်တကူ ခေါ်သုံးလို့ရသွားပါပြီ။

```
public class Test {

    public static void main(){

        TheaterFacade theater = new TheaterFacade();
        theater.playMovie();
    }
}
```

အခုကဲ့သို့ class တွေအများကြီးနဲ့ ခေါ်သုံးရခက် နေတဲ့အပိုင်းကို အလွယ်သုံးလို့ ရအောင် class တစ်ခု ထပ်ဖြည့် ပြီး ရိုးရှင်းအောင် လုပ်ဆောင်တဲ့ pattern ကို facade pattern လို့ခေါ်ပါတယ်။

Template Method Pattern ဆိုသည်မှာ...

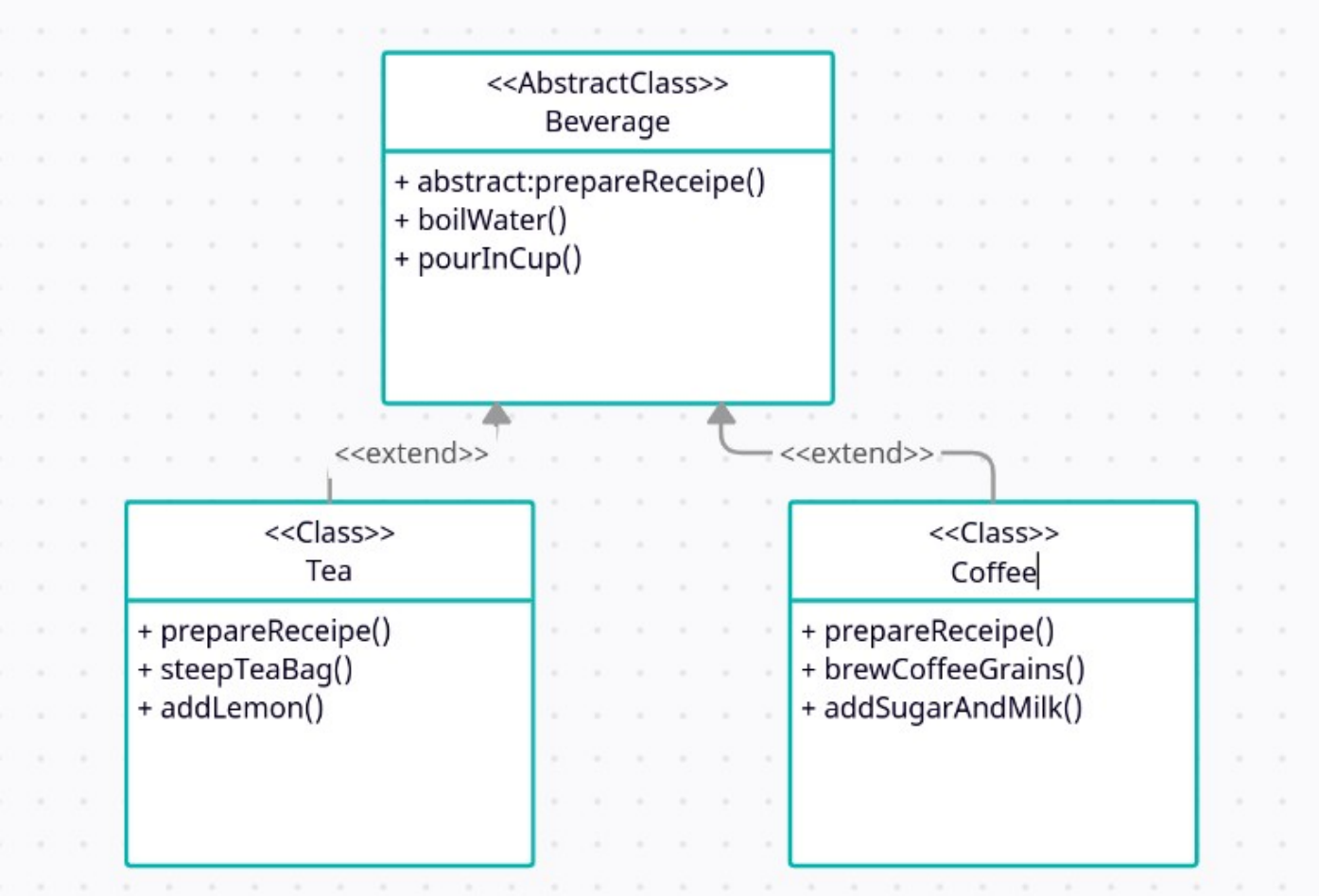
Template ဆိုတာကို တိုက်ရိုက် မြန်မာလို ပြန်ဆိုရင် "ပုံစံခွက်" လို့ အဓိပ္ပါယ်ရပါတယ်။ ကြေးတို့ သံတို့ ကို ကိုယ်လိုချင်တဲ့ ပုံစံထွက်အောင် အရည်ကြို ပြီး သွန်းလောင်း တဲ့ အခါမှာ သုံးတဲ့ "ပုံစံခွက်" ကိုဆိုလိုပါတယ်။ Template Method Pattern ဆိုတာကလည်း ထိုနည်းတူပါဘဲ method တစ်ခုကို template အနေနဲ့ ထား ထားပြီး အဲ့ method ထဲမှာ ရှိနေတဲ့ processing အစီအဥ်အလိုက်ကို implementation မရှိရင်တောင် အစီအစဥ် (order) ကို အပြင်ဘက်ကနေ ပြောင်းလို့မရအောင် ပြုလုပ် ထားတဲ့ ပုံစံမျိုးကို Template Method Pattern လို့ခေါ်ပါတယ်။ Template Method Pattern ကို Framework တွေမှာ အသုံးများပါတယ်။ သေချာနားလည်အောင် example လေးနဲ့ ထပ်ရှင်းပါမယ်။

Example

Example အနေနဲ့ လက်ဖက်ရည်ဖျော်တဲ့ process နဲ့ ကော်ဖီဖျော်တဲ့ process နှစ်ခု နဲ့ ရှင်းပါမယ်။ သီးခြား class နှစ်ခု ဖြစ်တဲ့ အတွက် code ကို ဘဲ အဓိကထားရှင်း ပါမယ်။

```
public class Coffee {
    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }
    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }
    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}

public class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
    public void boilWater() {
        System.out.println("Boiling water");
    }
    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }
    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
    public void addLemon() {
        System.out.println("Adding Lemon");
    }

}
```

အပေါ်က class နှစ်ခုမှာ တူညီနေတဲ့ method တွေ (boilWater() နဲ့ pourInCup()) ကို ပြန်သုံးထားတာ ကို တွေ့ပါလိမ့်မယ်။ အဲ့လို ဖြစ်နေတာမျိုးကို code duplication ဖြစ်တယ်လို့ ခေါ်ပြီး ဖြေရှင်းရမယ့် ပြဿနာတစ်ခု ဖြစ်တယ်။ လတ်တလော ဘုံတူညီတဲ့ method တွေကို class တစ်ခု ခွဲထုတ်ပြီး ဖြေရှင်းပါမယ်။

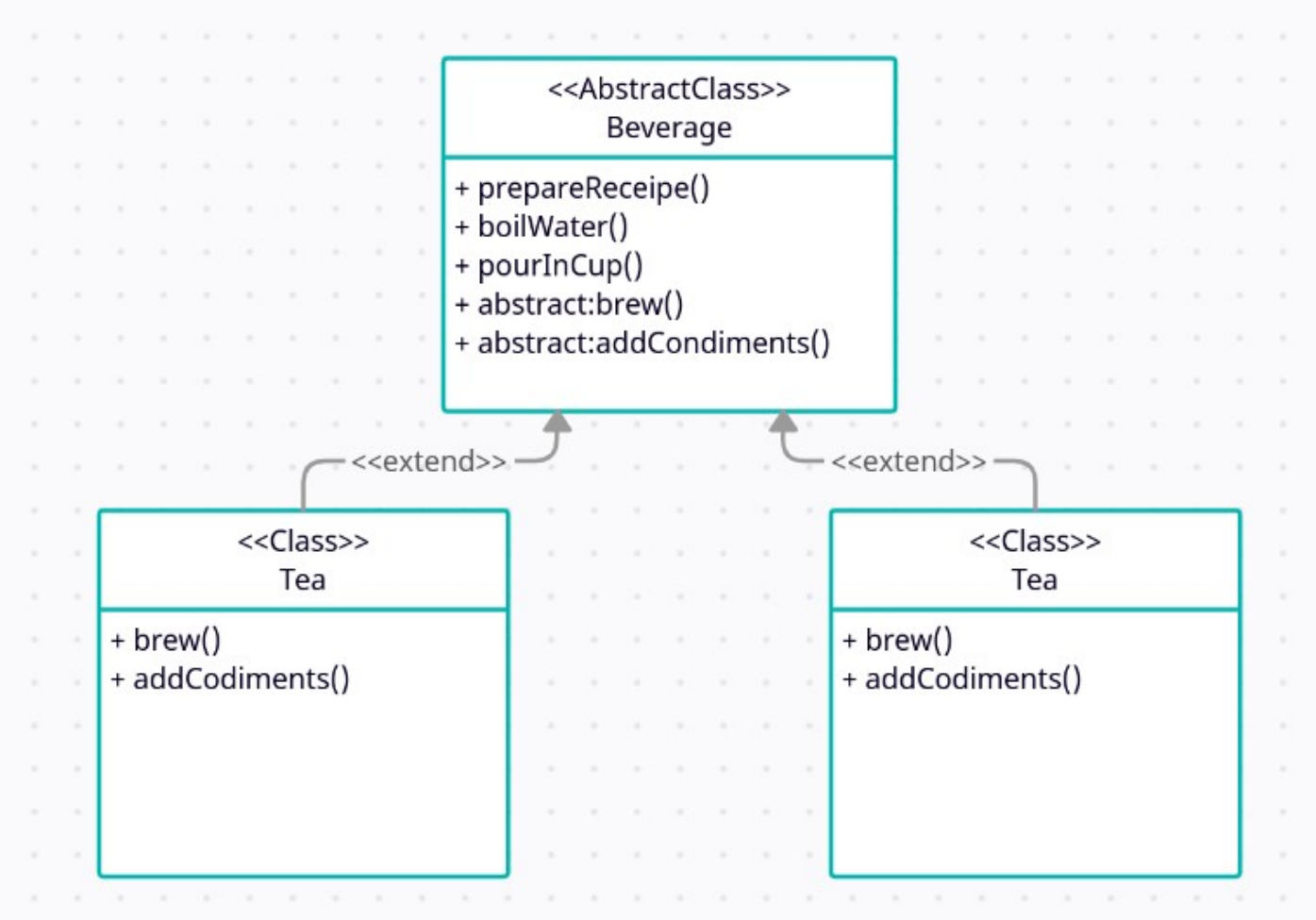


တူညီနေတဲ့ method တွေကို abstract class ထဲမှာ ရေးထားပြီး မတူညီတဲ့ prepareReceipe() method ကိုတော့ abstract အဖြစ်ထားထားပြီး child class ထဲမှာ သီးသန့်ပြန်ရေးတဲ့ ပုံစံမျိုးသုံးထားပါတယ်။

နောက်ထပ် တူညီနေတာကို ထပ်ကြည့်မယ်ဆိုရင် prepareReceipe() method တစ်ခုဘဲ ကျန်တယ်။ ဆိုပေမယ့် အပေါ်ဖက် coding ကိုပြန်ကြည့်ကြည့်ပါ။ Tea နဲ့ Coffe မှာ prepareReceipe() method ထဲမှာ သုံးထားတဲ့ method တွေက မတူပါဘူး။

```
public class Coffee {
    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
}

 public class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
}
```

အပေါ်က highlight ပြထားတဲ့ code တွေမှာ သုံးထားတဲ့ method တွေ မတူပေမယ့် processing အစီအစဉ် တူနေတာကို သတိထားကြည့်ပါ။ တကယ်လို့ အပေါ်က လိုမျိုး အစီအစဉ် ကတူညီနေတယ်ဆိုရင် Template Method Pattern ကို သုံးပြီး Design ပိုင်းကို ပိုကောင်းအောင် လုပ်လို့ရပါသေးတယ်။ Template Method Pattern ကို သုံးပြီး ပြင်ထားတဲ့ class diagram ကိုအောက်မှာ ဖော်ပြပေးထားပါတယ်။



နဂိုရ် abstact method ဖြစ်တဲ့ prepareReceipe() method ကို template အနေနဲ့ အသေရေးထားလိုက်ပြီး ဖျော်ချင်တဲ့ အရာမူတည် ပြီး ထည့်ရမယ့် ပစ္စည်း ပြောင်းလဲသွားမယ့် အဆင့်နှစ်ခုကို ဘဲ abstract methods တွေအနေနဲ့ brew(), addCondiments() ဆိုပြီး ရေးထားတာကို တွေ့ပါလိမ့်မယ်။ အခုလိုမျိုး ရေးထားတဲ့ ပုံစံကို Template Method Pattern လို့ခေါ်ပါတယ်။ ထပ်ပြီးပို့ရှင်းအောင် Beverage class ရဲ့ prepareReceipe() method code ကိုအောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public abstract class Beverage {
    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();

    }
}
```

အပေါ်မှာသုံးထားတဲ့ brew() နဲ့ addCondiments() method တွေက Beverage class ထဲမှာ တော့ abstract method အနေနဲ့ ဘဲရှိနေမှာ ဖြစ်ပြီး child class ကျမှ implementation code က ရှိမှာဖြစ်ပါတယ်။ အခုလိုမျိုး processing steps တူညီ ပြီး implementation မတူညီ တဲ့အရာတွေကို Template method တစ်ခု အဖြစ် စုစည်းပြီး code duplication ကို လျော့ချပြီး အပြင်ဘက်ကနေ order ကို ပြင်လို့ မရအောင် လုပ်တဲ့ Design Pattern ကို Template method pattern လို့ခေါ်ပါတယ်။

Template Method pattern ကို java.util.Arrays မှာလည်းသုံးထားပါတယ်။ သုံးထားတဲ့ method ကတော့ Arrays.sort() မှာသုံးထားခြင်းဖြစ်ပါတယ်။ Arrays.sort() မှာ parameter အနေနဲ့ sort လုပ်မယ့် array ကို ထည့်ပေးရပါတယ်။ ထူးခြားချက်က Arrays.sort() ကို သုံးချင်ရင် သုံးချင်တဲ့ array ရဲ့ class က Comparable interface ကို implements လုပ်ပြီး compareTo() method ရဲ့ implementation code ကို ရေးပေးရပါတယ်။ Arrays.sort() method မှာ အဲ့ ကိုယ် ရေးထားတဲ့ compareTo() method ကိုဘဲ သုံးပြီး sorting လုပ်သွားပါတယ်။ objects တွေက အမျိုးမျိုး ဖြစ်နိုင်တဲ့အတွက် သူတို့ကို Compare လုပ်ပုံကို စိတ်ကြိုက် ကိုယ့် ဘာသာ ရေးနိုင်အောင် ပြုလုပ်ပေးထားခြင်းဖြစ်ပါတယ်။

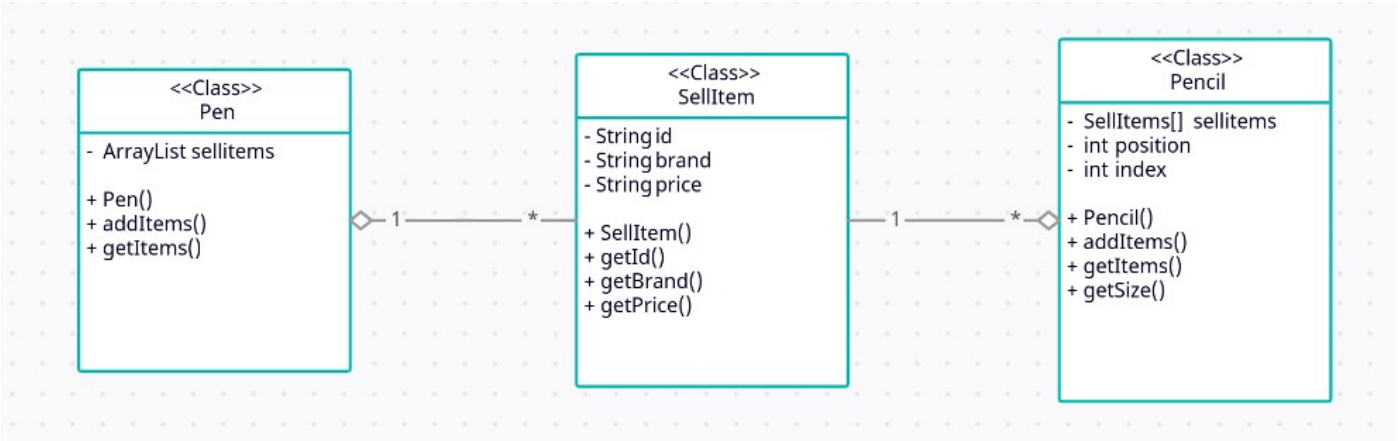
Processing procedure ကိုဘဲ အသေ (fix) လုပ်ထားပြီး implementation code ကို အရှင် (variable) လုပ်ပေးထားခြင်း ဖြစ်ပါတယ်။ အဲ့ကြောင့် Template method pattern ကို framework တွေမှာ အသုံးများပါတယ်။

Iterator Pattern ဆိုသည်မှာ...

Iterate ဆိုတာ က arrays တွေ collections တွေ တနည်းပြောရရင် အစုအဖွဲ့နဲ့ သိမ်းဆည်းထားတဲ့ အရာတွေ ကို အစုအဖွဲ့ထဲကနေ တခုချင်းစီ ထုတ်တာကို ဆိုလိုပါ တယ်။ Iterator pattern ဆိုတာက Iterate လုပ်တဲ့ နေရာမှာ အသုံးပြုတဲ့ pattern ဖြစ် ပါတယ်။ Iterator pattern ကို သုံးခြင်းအားဖြင့် arrays တွေမှာ data သိမ်းထားတဲ့ ပုံစံ ကို အသေးစိတ် သိစရာမလိုဘဲ အဲ့ arrays ထဲက elements တွေကို ထုတ်ပြနိုင်ပါ တယ်။ အသေးစိတ်သေချာ နားလည်အောင် example လေးနဲ့ ထပ်ရှင်းပါမယ်။

Example

Stationary store စာရေးကိရိယာဆိုင် တစ်ခုရဲ့ data တွေ store လုပ်ထားပုံ လေးနဲ့ ရှင်းပါမယ်။ နားလည်လွယ်အောင် pen နဲ့ pencil နှစ်မျိုး ရဲ့ မတူညီတဲ့ သိမ်းဆည်းပုံ နှစ်ခုကို အဓိကထားရှင်းပါမယ်။



အပေါ်က class design မှာ Pen ကော Pencil ကောနှစ်ခုလုံးက Sell Items ဆို တဲ့ class ရဲ့ objects တွေကို စုစည်းပြီး arrays ပုံစံ သိမ်းထားပေးတဲ့ class တွေဘဲ ဖြစ် ပါတယ်။ Pen ကတော့ ArrayList ပုံစံကို သုံးပြီး object တွေကို သိမ်းထားတယ်။ Pencil ကတော့ Object Array ပုံစံကို သုံးပြီး object တွေကို သိမ်းထားတယ်။ အဲ့လိုမျိုး မတူညီတဲ့ ပုံစံနဲ့ သိမ်းထားတော့ ပြန်ထုတ်သုံးခါနီးလည်း တမျိုးစီ ပြန်ထုတ်ရမယ်။ အဲ့

လိုမျိုး ပြန်ထုတ်သုံးပြထားတဲ့ ပုံစံကို အောက်က code မှာ Test class အနေနဲ့ ဖော်ပြပေးထားပါတယ်။

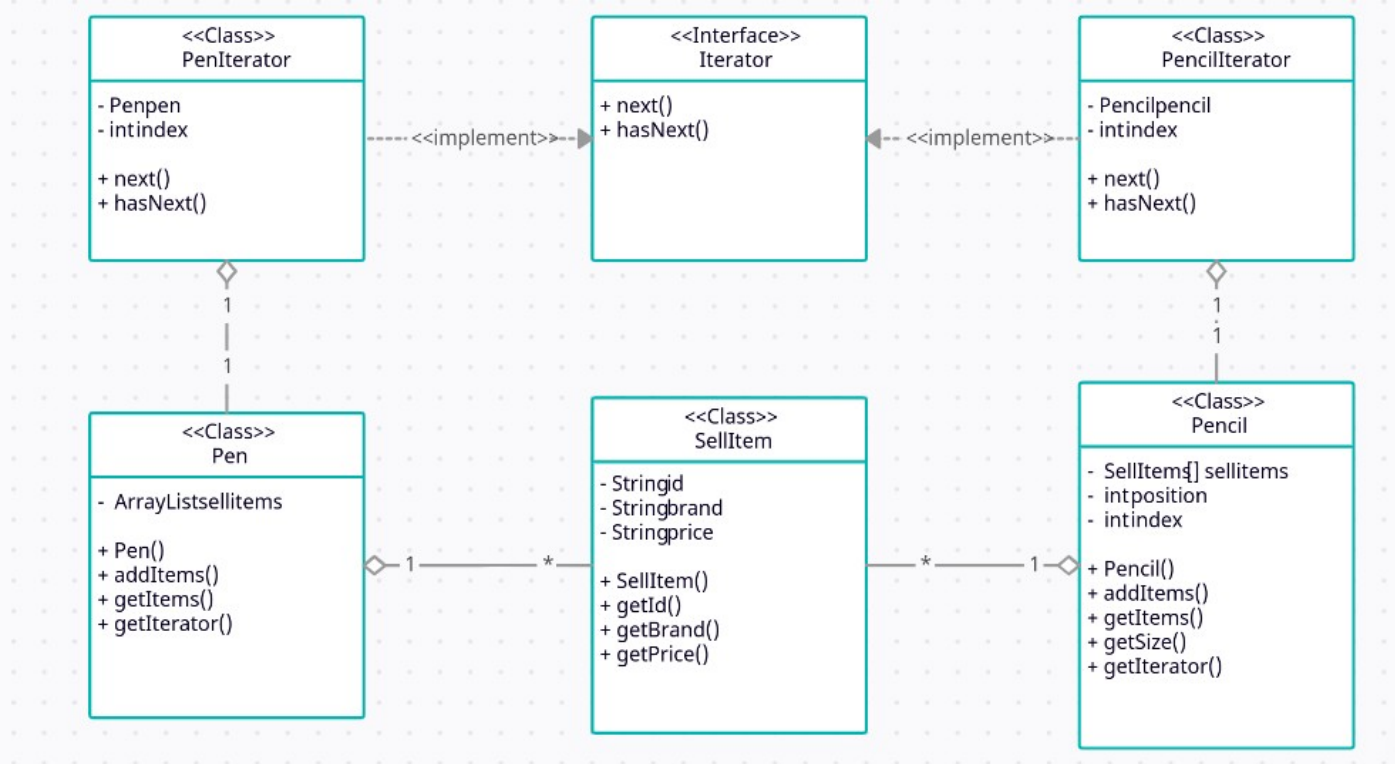
Code အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public class Pencil {
   private int position = 0;

   private SellItem [] sellitems;

    public Pencil() {
       sellitems = new SellItem [100];
    }

    void addItems(String id, String brand, String quantity) {
        SellItem item = new SellItem(id, brand, quantity);
        sellitems [position]= item;
        position = position + 1;
    }

    SellItem [] getItems() {
        return sellitems;
    }

    int getSize(){
        return position;
    }
}
```

```
public class Pen {

    private ArrayList sellitems;

    public Pen() {
        sellitems = new ArrayList();
    }
    void addItems(String id, String brand, String quantity) {
        SellItem item = new SellItem(id, brand, quantity);
        sellitems.add(item);
    }

    ArrayList getItems() {
        return sellitems;
    }

}

public class SellItem {
    private String id;
    private String brand;
    private String price;

    public SellItem(String id, String brand, String price) {
        this.id = id;
        this.brand = brand;
        this.price = price;
    }

    public String getId() {
        return id;
    }

    public String getBrand() {
        return brand;
    }

    public String getPrice() {
        return price;
    }
}
```

```
public class Test {

    public static void main(String[] args) {
        Pen pen = new Pen();
        pen.addItems("001", "Pilot", "1000");
        pen.addItems("002", "Rainbow", "100");
        Pencil pencil = new Pencil();
        pencil.addItems("001", "2B", "50");
        pencil.addItems("002", "Boss", "100");

        ArrayList penlist = pen.getItems();
        SellItem[] pencillist = pencil.getItems();

        for (int i = 0; i < penlist.size(); i++) {
            SellItem item = (SellItem) penlist.get(i);
            System.out.println(item.getBrand());
        }
        for (int i = 0; i < pencil.getSize(); i++) {
            SellItem item = pencillist[i];
            System.out.println(item.getBrand());
        }
    }
}
```

အပေါ်မှာ highlight ပြထားတဲ့ for loop အပိုင်းနှစ်ခု လုံးကို တူညီအောင် ပြုလုပ်ဖို့ အတွက် ဆိုရင် Iterator Pattern ကို သုံးရပါမယ်။ Iterator Pattern ကို သုံးပြီး ဖြေရှင်းထားတဲ့ class design ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။



အပေါ်က design မှာ Iterator interface ကို ဘုံအဖြစ်ထားထားပါတယ်။ pen class အတွက် PenIterator class ကိုသီးသန့်ရေးထားတယ်။ Pencil class အတွက် PencilIterator class ကိုသီးသန့်ရေးထားပါတယ်။ နှစ်ခုစလုံးကို ဘုံ Iterator interface ကို implements လုပ်ခြင်းဖြင့် Iterate process တွေကို တူညီအောင် လုပ်လို့ရပါတယ်။ ထိုကဲ့သို့ ဖြေရှင်းတဲ့ပုံစံကို Iterator Pattern လို့ခေါ်ပါတယ်။ ပိုပြီးနားလည်အောင် code အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public class SellItem {

    private String id;
    private String brand;
    private String price;

    public SellItem(String id, String brand, String price) {
        this.id = id;
        this.brand = brand;
        this.price = price;
    }

    public String getId() {
        return id;
    }

    public String getBrand() {
        return brand;
    }

    public String getPrice() {
        return price;
    }

}

public interface Iterator {

    public Object next();

    public boolean hasNext();
}
```

```
public class Pen {

    private ArrayList sellitems;

    public Pen() {
        sellitems = new ArrayList();
    }

    void addItems(String id, String brand, String quantity) {
        SellItem item = new SellItem(id, brand, quantity);
        sellitems.add(item);
    }

    ArrayList getItems() {
        return sellitems;
    }

    Iterator getIterator() {
        return new PenIterator(this);
    }

}
public class PenIterator implements Iterator {

    private Pen pen;
    private int index;

    public PenIterator(Pen pen) {
        this.pen = pen;
        this.index = 0;
    }

    @Override
    public Object next() {
        ArrayList sellItems = pen.getItems();
        return sellItems.get(index++);
    }

    @Override
    public boolean hasNext() {
        ArrayList sellItems = pen.getItems();
        return index < sellItems.size();
    }

}
```

```
public class Pencil {

    private int position = 0;
    SellItem[] sellitems;

    public Pencil() {
        sellitems = new SellItem[100];
    }

    void addItems(String id, String brand, String quantity) {
        SellItem item = new SellItem(id, brand, quantity);
        sellitems[position++] = item;
    }

    SellItem[] getItems() {
        return sellitems;
    }

    int getSize() {
        return position;
    }

    Iterator getIterator() {
        return new PencilIterator(this);
    }

}
```

```
public class PencilIterator implements Iterator {
    private int index;
    private Pencil pencil;

    public PencilIterator(Pencil pencil) {
        this.pencil = pencil;
        index = 0;
    }

    @Override
    public Object next() {
        SellItem[] sellitems = pencil.getItems();
        return sellitems[index++];
    }

    @Override
    public boolean hasNext() {
        int size = pencil.getSize();
        return index < size;
    }
}

public class Test {
    public static void main(String[] args) {
        Pen pen = new Pen();
        pen.addItems("001", "Pilot", "1000");
        pen.addItems("002", "Rainbow", "100");

        Pencil pencil = new Pencil();
        pencil.addItems("001", "2B", "50");
        pencil.addItems("002", "Boss", "100");

        Iterator peniterator = pen.getIterator();
        Iterator penciliterator = pencil.getIterator();

        while (peniterator.hasNext()) {
            SellItem item = (SellItem) peniterator.next();
            System.out.println(item.getBrand());
        }
        while (penciliterator.hasNext()) {
            SellItem item = (SellItem) penciliterator.next();
            System.out.println(item.getBrand());
        }
    }
}
```

PenIterator နဲ့ PencilIterator တို့ရဲ့ Iterator interface ကို implements လုပ်ထားတဲ့ next(), hasNext() methods တွေက တစ်ခုနဲ့ တစ်ခု မတူညီတာကို သတိထားကြည့်ပါ။ ဆိုပေမယ့် Test class မှာ နှစ်ခုစလုံးကို Iterator အဖြစ် ပြန်ခေါ်သုံးတဲ့နေရာမှာ ပုံစံတူစွာ ပြန်ခေါ်သုံးနိုင်အောင် လုပ်ထားတာကို လည်း သတိထားကြည့်ပါ။ အခုလိုမျိုး object တွေကို store လုပ်ပုံ ချင်း မတူညီပေမယ့် ပြန်ပြီး iterate လုပ်တဲ့အခါ ပုံစံတူတူ လုပ်နိုင်အောင် ပြုလုပ်ပေးတဲ့ pattern မျိုးကို Iterator Pattern လို့ခေါ်ပါတယ်။ java မှာလည်း collection တွေကို iterate လုပ်နိုင်ဖို့ ရေးထားတဲ့ java.util.Iterator class ကလည်း Iterator Pattern နဲ့ဘဲ တည်ဆောက်ထားတာ ဖြစ်ပါတယ်။

တလက်စတည်း code example ရှိနေတဲ့ အတွက် သတိထားသင့်တဲ့ အရာတစ်ခု ကို ထပ်ရှင်းပါမယ်။ PenIterator နဲ့ PencilIterator တို့ရဲ့ method တွေမှာ

```
ArrayList sellItems = pen.getItems();
return sellItems.get(index++);
```

ဆိုတဲ့ နှစ်ကြောင်းအစား တစ်ကြောင်းတည်း အောက်ကလိုမျိုး လည်း ရေးလို့ရပါတယ်။

```
return pen.getItems().get(index++);
```

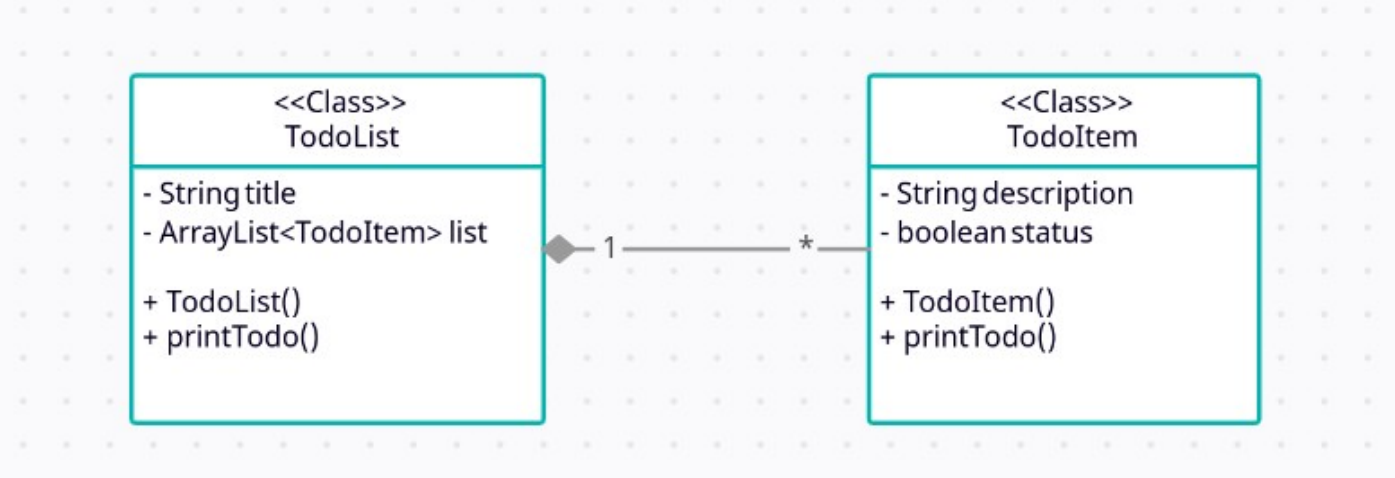
ရေးလို့ရတယ် ဆိုပေမယ့် မရေးသင့်ပါဘူး။ method တစ်ခုကနေ return ပြန်တဲ့ object ကနေတဆင့် အဲ့ object ရဲ့ method ကို တခါ ပြန်ခေါ်တာမျိုးက ရုတ်တရက် ဖတ်ရခက်ပါတယ်။ အများနားလည်လွယ်အောင် return object ကို အရင် variable တစ်ခုနဲ့ လက်ခံပြီး ထို variable မှတဆင့် အဲ့ object ရဲ့ method ကို တခါ ပြန်ခေါ်တာမျိုးက ကြည့်ရရှင်းပါတယ်။ တခါတလေ ကိုယ်ရေးထားတာ ကိုယ်တိုင်တောင် ဖတ်လို့မရတာမျိုး ဖြစ်နိုင်တဲ့ အတွက် အပေါ်က လို တကြောင်းတည်း ရေးထုံးကို တတ်နိုင်သမျှ ရှောင်ပါ။ Design Priciple အနေနဲ့ အဲ့တာကို Principle of Least Knowledge လို့ခေါ်ပါတယ်။

Composite Pattern ဆိုသည်မှာ...

Class တစ်ခုသည် သူ့ရဲ့ object ကိုတည်ဆောက်တဲ့ အခါမှာ တခြား class တွေ ရဲ့ object ကိုပါရောပြီး တည်ဆောက်ရတဲ့ class မျိုးကို Composite class လို့ခေါ်ပါ တယ်။ Composite pattern ဆိုတာကလည်း Composite class ပေါ် အခြေခံပြီး ပေါ်လာတဲ့ design pattern တစ်ခုပါဘဲ။ object တွေကို အထက်အောက် ချိတ်ဆက် ထားတဲ့ ပုံစံမျိုး (တနည်းပြောရမယ်ဆိုရင် tree structure ပုံစံမျိုး) အပေါ်ဘက်မှာ (parent) object တစ်ခုဘဲရှိမယ် သူ့အောက်မှာ (children) objects အများကြီးရှိမယ်။ အဲ့လိုပုံစံမျိုးမှာ parent ကိုကော children ကိုပါ တပုံစံတည်း ပုံစံတူအဖြစ် သတ်မှတ် ထားပြီး အသုံးပြုနိုင်အောင် လုပ်ပေးတဲ့ pattern ကို Composite pattern လို့ ခေါ်ပါ တယ်။

Example

Todo List program တစ်ခုကို နမူနာအနေနဲ့ ထားရှင်းပါမယ်။ စစချင်း အနေနဲ့ ကျတော်တို့ မှာ Todo တွေကို အသေးစိတ်ဖော်ပြထားတဲ့ class တစ်ခုနဲ့ အဲ့ Todo တွေ ကို စုစည်းပြီးမှ တည်ဆောက်ထားတဲ့ composite class တစ်ခု ရှိပါမယ်။ အဲ့နှစ်ခုရဲ့ class diagram နဲ့ code ကိုဖော်ပြပေးထားပါတယ်။

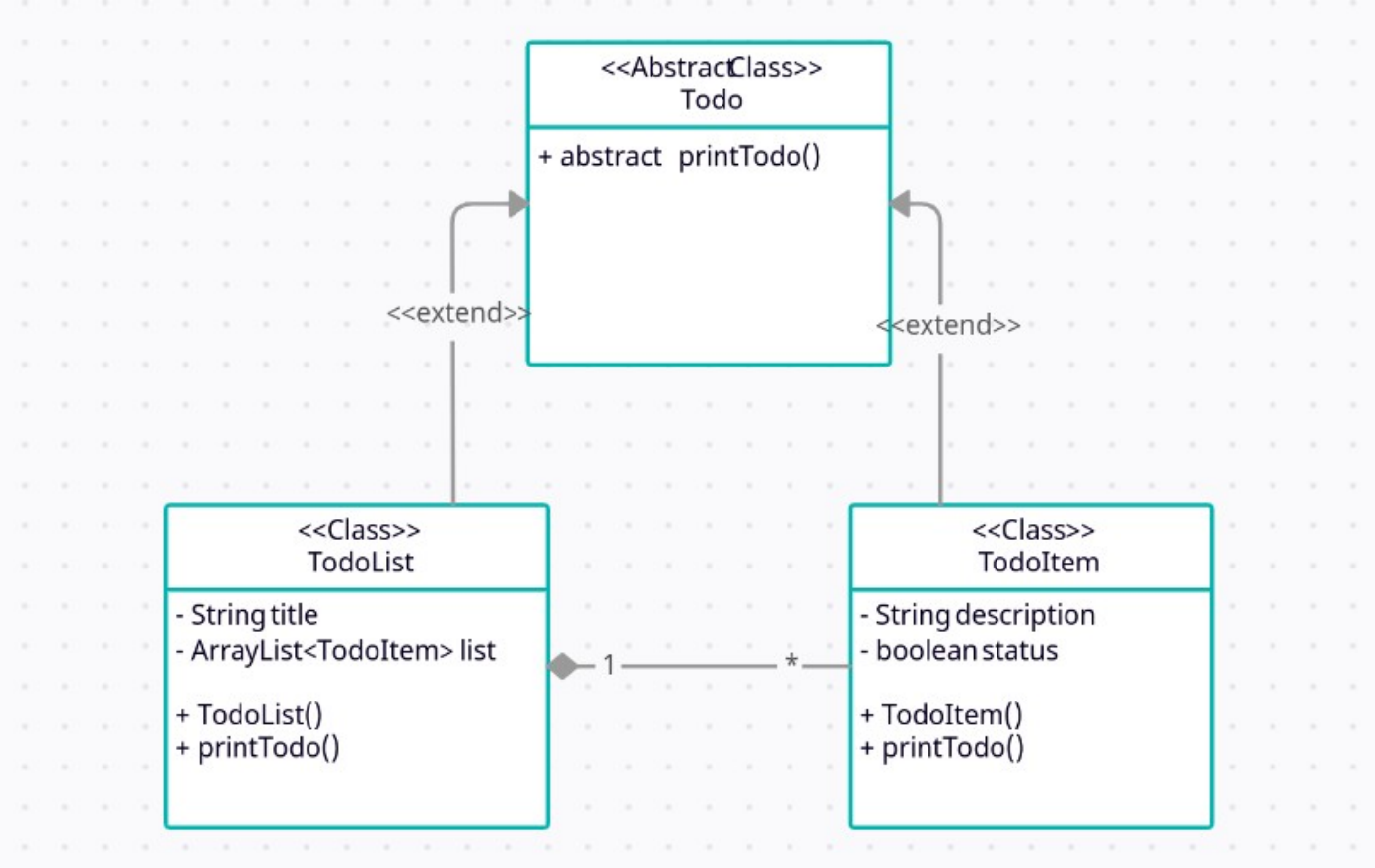


အပေါ်က class diagram မှာ TodoList က composite class ဖြစ်ပါတယ်။

```
public class TodoItem {

    private String description;
    private boolean status;

    public TodoItem(String description, boolean status) {
        this.description = description;
        this.status = status;
    }

    void printTodo() {
        String finished = "finished";
        if (!status) {
            finished = "not finished yet";
        }
        System.out.println(description + " " + finished);
    }

}

public class TodoList {

    private String title;
    private ArrayList<TodoItem> list = new ArrayList<>();

    public TodoList(String title, ArrayList<TodoItem> list) {
        this.title = title;
        this.list = list;
    }

    public void printTodo() {
        System.out.println(title);
        for (TodoItem todo : list) {
            todo.printTodo();
        }
    }
}
```

```
public class Test {

    public static void main(String[] args) {

        TodoItem task1 = new TodoItem("Read Book", true);
        TodoItem task2 = new TodoItem("Clean Room", true);
        ArrayList<TodoItem> list = new ArrayList<>();
        list.add(task1);
        list.add(task2);
        TodoList mainTask = new TodoList("Today works", list);
        mainTask.printTodo();


    }

}
```

အပေါ်က highlight ပြထားတဲ့ code မှာ TodoItem objects တွေကို Arraylist ထဲ ထည့်ပြီး အဲ့ list ကိုမှ တစ်ခါ Todolist object ဆောက်တဲ့အခါ ပြန်သုံးပြီး တည်ဆောက်ထားတာကို သေချာကြည့်ကြည့်ပါ။ နောက်ပြင်ချင်တဲ့ ပုံစံမျိုးက Read Book မှာ ဘာစာအုပ်တွေ ဖတ်လဲဆိုတာ အသေးစိတ် item အနေနဲ့ ထပ်ဖြည့်ချင်ပါတယ်။ အဲ့လိုဖြည့်လိုက်ရင် Todolist object ဖြစ်သွားပြီး list မှာ add လုပ်လို့မရပါဘူး။ list ထဲ add လုပ်တဲ့နေရာမှာ todoitem တင်မဟုတ်ဘဲ todoList တစ်ခုကို ပါ add လုပ်နိုင်တဲ့ ပုံစံမျိုး ကိုလိုချင်ပါတယ်။ list ဘဲ ဖြစ်ဖြစ် item ဘဲဖြစ်ဖြစ် တပုံစံတည်း add လုပ်နိုင်ရမယ်။ ပြန်ခေါ်သုံးတဲ့ အခါလည်း အတူတူပုံစံမျိုး သုံးနိုင်ရမယ်။ အဲ့လို ပုံစံမျိုး ဖြစ်အောင် လုပ်ဖို့ဆိုရင် composite pattern ကို သုံးရပါမယ်။ composite pattern အရ ဆွဲထားတဲ့ class design ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။



အပေါ်က design မှာ ရိုးရှင်းစွာ abstract class တစ်ခု ထပ်ဖြည့်ပြီး composite pattern ဖြစ်အောင်လုပ်ထားပါတယ်။ သေချာနားလည်သွားအောင် code အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public abstract class Todo {

   abstract void printTodo();

}
```

```
public class TodoItem extends Todo {
    private String description ;
    private boolean status ;

    public TodoItem(String description, boolean status) {
        this.description = description;
        this.status = status;
    }


    void printTodo (){
        String finished = "finished";
        if (!status) finished = "not finished yet";
        System.out.println(description + " "+finished);
    }

}

public class TodoList extends Todo {

    private String title;
    private ArrayList<Todo> list = new ArrayList<>();

    public TodoList(String title, ArrayList<Todo> list) {
        this.title = title;
        this.list = list;
    }

    public void printTodo() {
        System.out.println(title);
        for (Todo todo : list) {

            todo.printTodo();
        }
    }

}
```

```
public class Test {

    public static void main(String[] args) {
        Todo task1 = new TodoItem("Science book", true);
        Todo task2 = new TodoItem("Java book", true);
        Todo task3 = new TodoItem("Clean Room", true);

        ArrayList<Todo> mainlist = new ArrayList<>();
        ArrayList<Todo> booklist = new ArrayList<>();

        booklist.add(task1);
        booklist.add(task2);
        Todo booktask = new TodoList("Read Book", booklist);
        mainlist.add(task3);
        mainlist.add(booktask);
        Todo maintodo = new TodoList("Daily Routine",
         mainlist);
        maintodo.printTodo();

    }

}
```

အပေါ်မှာ highlight ပြထားတဲ့ lines တွေကိုသတိထားကြည့်ပါ။ Todo ကို ဘုံ data type အဖြစ်ထားလိုက်တဲ့ အတွက် mainlist ထဲမှာ todoList ကော todoItem ပါ ပုံစံတူတူ ထည့်လို့ရသွားပါတယ်။ ပြန်ခေါ်တဲ့နေရာမှာလည်း todoList ကော todoItem အတွက်ပါ printTodo() method တစ်ခုတည်းကိုဘဲ ပုံစံတူ သုံးလို့ရသွားပါတယ်။ အခုကဲ့သို့ ပုံစံမျိုးကို composite pattern လို့ခေါ်ပါတယ်။

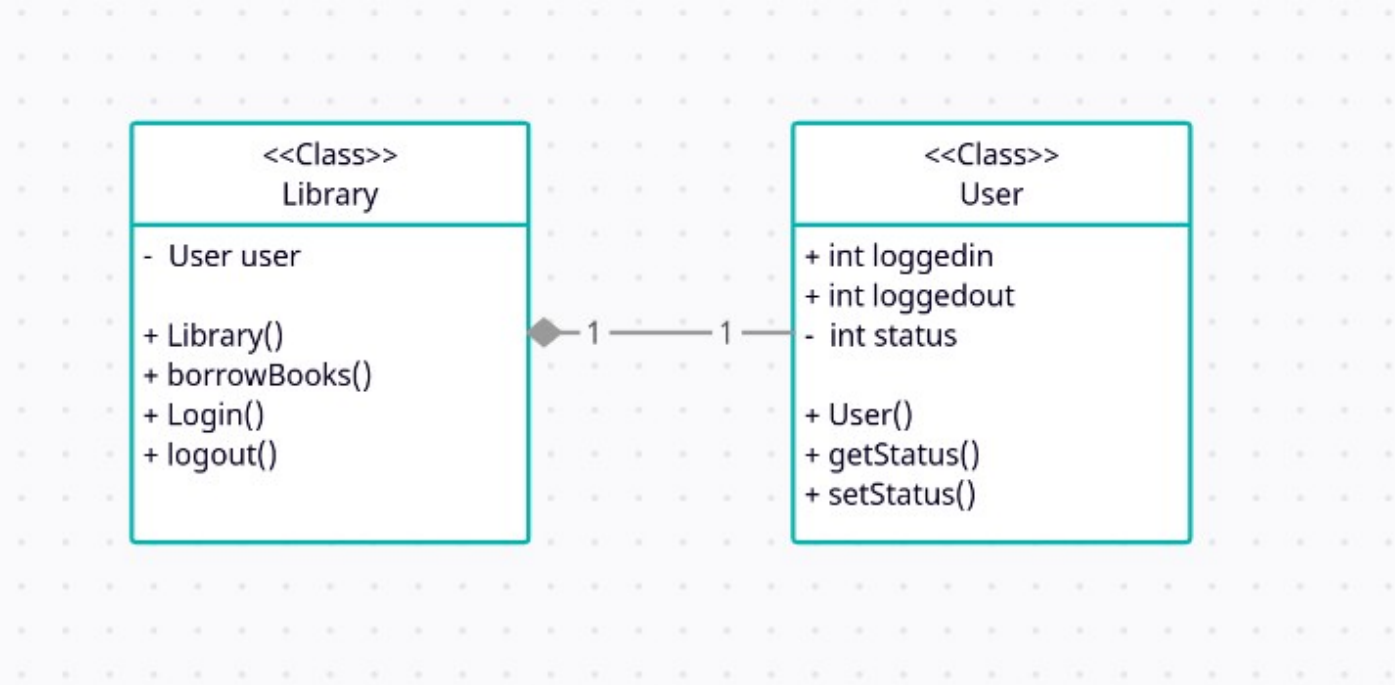
State Pattern ဆိုသည်မှာ...

State ဆိုတာကို တိုက်ရိုက်ဘာသာပြန်ဆိုရင် "အခြေအနေ" လို့ အဓိပ္ပါယ်ရတယ်။ Programming မှာ သုံးတဲ့ UML Diagrams တွေထဲက State Diagram နဲ့ State pattern နဲ့ က ဆက်စပ်မှုရှိပါတယ်။ State Diagram ကတော့ အခြေအနေတစ်ခုကနေနောက်တစ်ခုကို ပြောင်းတာကို ဖော်ပြတဲ့ နေရာမှာ အသုံးပြုလေ့ရှိပါတယ်။ Program တစ်ခုမှာ visitor ကနေ member အဖြစ်ပြောင်းလဲတဲ့ နေရာမျိုးကို ဆိုလိုပါတယ်။ အဲ့လိုမျိုး အခြေအနေအမျိုးမျိုး ရှိနေပြီး အခြေအနေတစ်ရပ်ကနေ အခြားတစ်ရပ် ကိုပြောင်းတဲ့ အခါမျိုးမှာ program ရဲ့ လုပ်ဆောင်မှုတွေပါ လိုက်ပြီး ပြောင်းလဲတဲ့ ပုံစံမျိုးကိုရေးသားတဲ့ အခါမှာ State pattern ကို သုံးလေ့ရှိပါတယ်။

State pattern ကို ထပ်ပြီး အတိအကျ အဓိပ္ပါယ်ဖွင့်ဆိုရလျှင် object တစ်ခုရဲ့ field variable တစ်ခုကို အပြောင်းအလဲ လုပ်ရုံဖြင့် အဲ့ object ရဲ့ functions တွေက နဂိုရ် ပုံစံကနေ လုံးဝကွဲထွက်ပြီး တခြား object တစ်ခုကဲ့သို့ ပြောင်းလဲ သွားအောင် ပြုလုပ်ထားသော ပုံစံမျိုးကို ဆိုလိုပါသည်။

Example

ပိုမိုရှင်းလင်းစွာ နားလည်အောင် library program တစ်ခုကို example အနေနဲ့ ထပ်ရှင်းပါမယ်။ ပုံမှန်အားဖြင့် library တွေက member ဝင်ထားမှသာလျှင် စာအုပ်ငှားခွင့်ရှိပါမယ်။ စာအုပ်အငှား process လေးကိုဘဲ ဥပမာအနေနဲ့ ထားရှင်းပါမယ်။ Class diagram ကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။



အပေါ်က class diagram မှာတော့ ကြည့်ရတာ လွယ်ကူအောင် လိုအပ်တဲ့ class နှစ်ခုကိုဘဲ ဖော်ပြပေးထားပါတယ်။ Library class မှာတော့ User object တစ်ခုကို field variable အဖြစ်ထားထားပါတယ်။ အဲ့ user object ရဲ့ အခြေအနေ logged in ဖြစ်လား logged out ဖြစ်လားပေါ်မူတည်ပြီး borrowbooks() method ရဲ့ အလုပ်လုပ်ပုံ ပြောင်းလဲပါမယ်။ စစချင်း State Pattern မသုံးထားတဲ့ အတွက် if else statement တွေနဲ့ စစ်ပြီး အလုပ်လုပ်ပါတယ်။ code အပြည့်အစုံကို အောက်မှာဖော်ပြပေးထားပါတယ်။

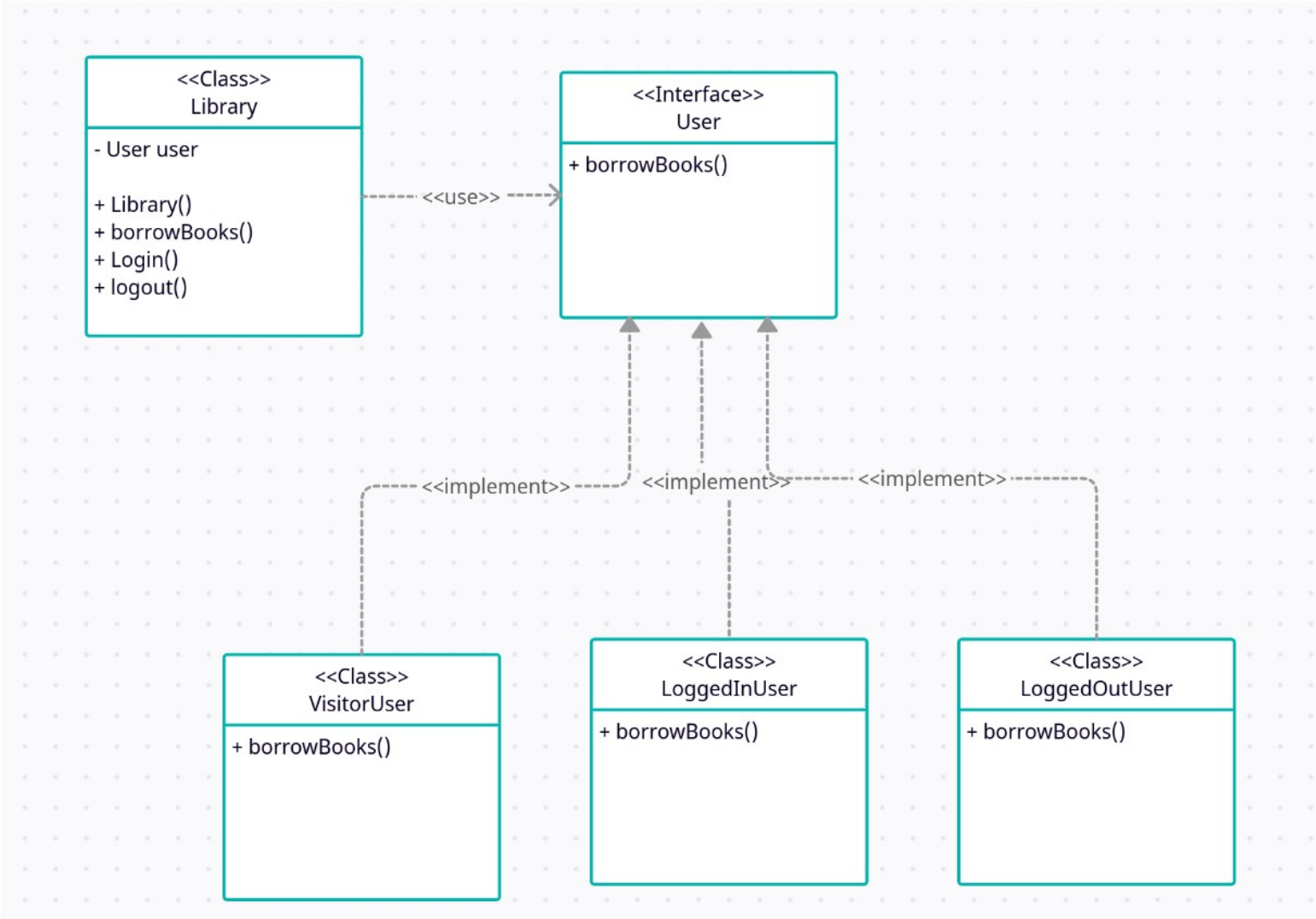
```
public class User {

    int loggedin  =  1;
    int loggedout = 0;
    private int status ;

    public User(){
        this.status = loggedout;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }


}

public class Library {

    private User user;

    public Library (){
       this.user =  new User();
    }


    public void borrowBooks(){

        if (this.user.getStatus() == this.user.loggedin){
            System.out.println("You can borrow books!");
        }
        else {
            System.out.println("Please logged in first to
             borrow books!");
        }
    }

    public void login(){
        this.user.setStatus(this.user.loggedin);
    }
```

```
    public void logout(){
        this.user.setStatus(this.user.loggedout);
    }


}

public class Test {

    public static void main(String[] args) {
        Library lib = new Library();
        lib.borrowBooks();
        lib.login();
        lib.borrowBooks();
    }
}
```

အပေါ်မှာ highlighted ပြထားတဲ့ code မှာ new Library() ဆိုပြီး object တည်ဆောက်တဲ့အခါ Library ရဲ့ constructor မှာ တစ်ခါ User object အသစ်တစ်ခု တည်ဆောက်ပါတယ်။ အဲ့ user object ထဲမှာ status ကို loggedout အဖြစ် အသေ ထည့်ပေးထားပါတယ်။ အဲ့အဆင့်တွေအားလုံး ကို highlight လုပ်ပေးထားပါတယ်။ နောက်ထပ် state pattern မသုံးထားဘဲ state ကို if else နဲ့ စစ်ထားတာကိုလည်း highlight လုပ်ပေးထားပါတယ်။ အဲ့အဆင့်တွေကို သေချာနားလည်အောင်ကြည့်ပါ။

ထပ်ပြီးပြင်ချင်တဲ့ ပုံစံမျိုးကတော့ စစချင်း visitor အနေနဲ့ ဝင်လို့ရရမယ် စာအုပ်တွေကို ငှားခွင့်မရှိရင်တောင် စာအုပ်ရှာခွင့်တို့ ဘာစာအုပ်တွေရှိလဲ ဆိုတာကို ကြည့်ခွင့်တော့ရှိမယ်။ အဲ့လိုပုံစံမျိုး နဂိုရှိနေတဲ့ loggedin loggedout နေရာမှာ visitor ဆိုတဲ့ state အသစ်တစ်ခု ထပ်ဖြည့်ချင်တယ်။ လက်ရှိ design မှာ ထပ်ဖြည့်မယ်ဆိုရင် user class ကော library class မှာပါ ပြင်ဆင်မှုတွေလုပ်ရတော့မယ်။ open-closed principle အရဆိုရင်တော့ နဂိုရှိပြီးသား class ကို မပြင်ဘဲ function အသစ်ထပ်တိုးနိုင်ရမယ်။ အဲ့လိုပုံစံမျိုး ဖြစ်ဖို့ဆိုရင် State pattern ကို သုံးမှဖြစ်မယ်။ နောက်ထပ် အခု အခြေအနေသုံးခု ဘဲရှိလို့ အဆင်ပြေပေမယ့် အခြေအနေ ရာချီ ရှိလာတဲ့အခါ ထိန်းသိမ်းရ ခက်လာပါလိမ့်မယ်။ if else တွေ ရာချီလာရင် တခုပြင်ရင်း တခြားတစ်ခုကို ထိခိုက်မိ

တာမျိုးတွေဖြစ်လာနိုင်ပါတယ်။ အဲ့တာတွေကို ကျော်လွှားနိုင်ဖို့ State pattern ကိုသုံးထားတဲ့ class diagram ကို အောက်မှာဖော်ပြပေးထားပါတယ်။



အပေါ်က design မှာ မတူညီတဲ့ user state တွေအတွက် သီးသန့် class တစ်ခုစီ အဖြစ် ခွဲထုတ်ထားတာကို တွေ့ရပါမယ်။ ထပ်ပြီး state ထပ်တိုးရင် class အသစ်ထပ်ဖြည့်ရုံပါဘဲ နဂိုရှိ class တွေရဲ့ code ကို ပြင်စရာမလိုပါဘူး။ ပိုပြီးရှင်းသွားအောင် code အပြည့်အစုံကို အောက်မှာဖော်ပြပေးထားပါတယ်။

```
public interface User {
    public void borrowBooks();

}

public class VisitorUser implements User{

    @Override
    public void borrowBooks() {
        System.out.println("You can't borrow books but you can
         view and search books!");
    }

}

public class LoggedInUser implements User {

    @Override
    public void borrowBooks() {
        System.out.println("You can borrow Books!");
    }

}

public class LoggedOutUser implements User {

    @Override
    public void borrowBooks() {
        System.out.println("Logged In again to borrow
         Books!");
    }

}
```

```
public class Library {

    private User user;

    public Library (){
        this.user = new VisitorUser();
    }


    public void borrowBooks(){
       user.borrowBooks();
    }

    public void login(){
        this.user = new LoggedInUser();
    }

    public void logout(){
        this.user = new LoggedOutUser();
    }
}

public class Test {
    public static void main(String[] args) {
        Library lib = new Library();
        lib.borrowBooks();
        lib.login();
        lib.borrowBooks();
        lib.logout();
        lib.borrowBooks();

    }
}
```

အပေါ်က code မှာ highlighted ပြထားတဲ့ အပိုင်းတွေကို သေချာကြည့်ပါ။ Library class မှာ interface ကို datatype အဖြစ်ထားပြီး object ကို အမျိုးမျိုး ပြောင်းသုံးသွားပါတယ်။ တစ်ခါ borrowBooks() method မှာ user object ရဲ့ method ကို အစားထိုးသုံးထားပါတယ်။ နောက်ထပ် Test class မှာ lib ရဲ့ state ကို အမျိုးမျိုး ပြောင်းခြင်းအားဖြင့် ခေါ်တဲ့ method တူညီသော်လည်း result မတူတာကို တွေ့ရပါမယ်။

Proxy Pattern ဆိုသည်မှာ...

'Proxy' ဆိုတာကို တိုက်ရိုက် ဘာသာပြန်ရင် 'ကြားခံ' လို့အဓိပ္ပါယ်ရပါတယ်။ 'Proxy Pattern' ဆိုတာလည်း အလားတူပါဘဲ အရာတစ်ခုကို ကြားခံ အဖြစ်ထားထားပြီး အဲ့အရာကနေ တဆင့် တခြားတစ်ခုကို လှမ်းခေါ်သုံးတဲ့ ပုံစံမျိုးကို ဆိုလိုပါသည်။ Proxy Pattern မှာလည်း အသုံးပြုပုံကို လိုက်ပြီး နောက်ထပ် သုံးမျိုး ထပ်ကွဲပါတယ်။ ကြားခံထားတာခြင်း တူပေမယ့် အသုံးပြုရတဲ့ ရည်ရွယ်ချက်ကို Virtual Proxy, Protection Proxy နဲ့ Remote Proxy ဆိုပြီး သုံးမျိုး ထပ်ကွဲပါတယ်။

Virtual Proxy ဆိုတာကတော့ file size ကြီးခြင်း processing အချိန် ယူရခြင်း အစရှိတဲ့ object မျိုးကို တည်ဆောက်တဲ့နေရာမှာ တိုက်ရိုက် မတည်ဆောက်ဘဲ ကြားခံအဖြစ် class တစ်ခုကို ထားထားပြီး အဲ့ class ကနေမှတဆင့် object တည်ဆောက်တဲ့ ပုံစံမျိုးကို ဆိုလိုပါသည်။

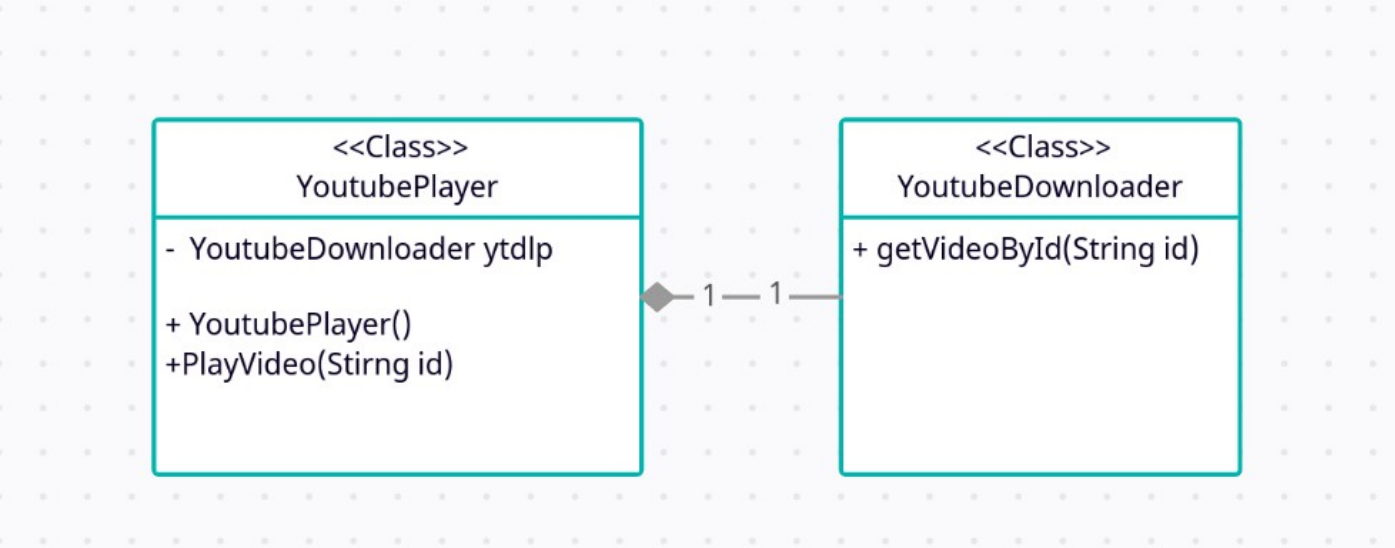
Protection Proxy ဆိုတာကတော့ access control လုပ်ချင်တဲ့ အခါမျိုးမှာ သုံးပါသည်။ access control လုပ်လိုတဲ့ object ကို တိုက်ရိုက် တည်ဆောက်ခွင့် မပြုဘဲ ကြားခံ class တစ်ခု ထားထားပါတယ်။ အဲ့ကနေမှတဆင့် authenticated ဖြစ်မဖြစ် စစ်ဆေးမယ် ပြီးမှ object တည်ဆောက်ခွင့်ပြု တဲ့ပုံစံမျိုးကို ဆိုလိုပါသည်။

Remote Proxy ဆိုတာကတော့ အဝေးတစ်နေရာမှာ ရှိတဲ့ object တစ်ခုကို လှမ်းပြီး access လုပ်တဲ့နေရာမှာ ကြားခံ အဖြစ် class တစ်ခုကို ထားတဲ့ ပုံစံမျိုးကို ဆိုလိုပါသည်။ java ရဲ့ RMI (Remote Method Invocation) သည် remote proxy ကို သုံးထားခြင်း ဖြစ်ပါသည်။ java ရဲ့ rmi ကိုသုံးရင် local network (intranet) အတွင်းက object ကို လှမ်းပြီး access လုပ်လို့ရပါတယ်။ rmi ကို သုံးချင်ရင် အရင်ဆုံး rmic command ကိုသုံးပြီး ကိုယ်သုံးချင်တဲ့ class ရဲ့ stub class တစ်ခုကို ဖန်တီးရပါတယ်။ အဲ့ stub class ကို proxy အဖြစ် တခြား computer ထားထားပြီး အဲ့ stub class ကနေ တဆင့် ကိုယ်တကယ်သုံးချင်တဲ့ class ကို လှမ်းပြီး သုံးနိုင်ပါတယ်။ အဲ့လိုပုံစံမျိုး ကို remote proxy လို့ခေါ်ပါတယ်။

ထပ်ပြီး အသေးစိတ်နားလည်အောင် example လေးကို virtual proxy နဲ့ ရှင်းပြပါမယ်။

Example

Youtube ကနေ video တွေကို download လုပ်ပြီး play လုပ်ပေးမယ့် program လေးကို example အနေနဲ့ထားပြီး ရှင်းပြပါမယ်။ class diagram ကိုအောက်မှာ ဖော်ပြပေးထားပါတယ်။

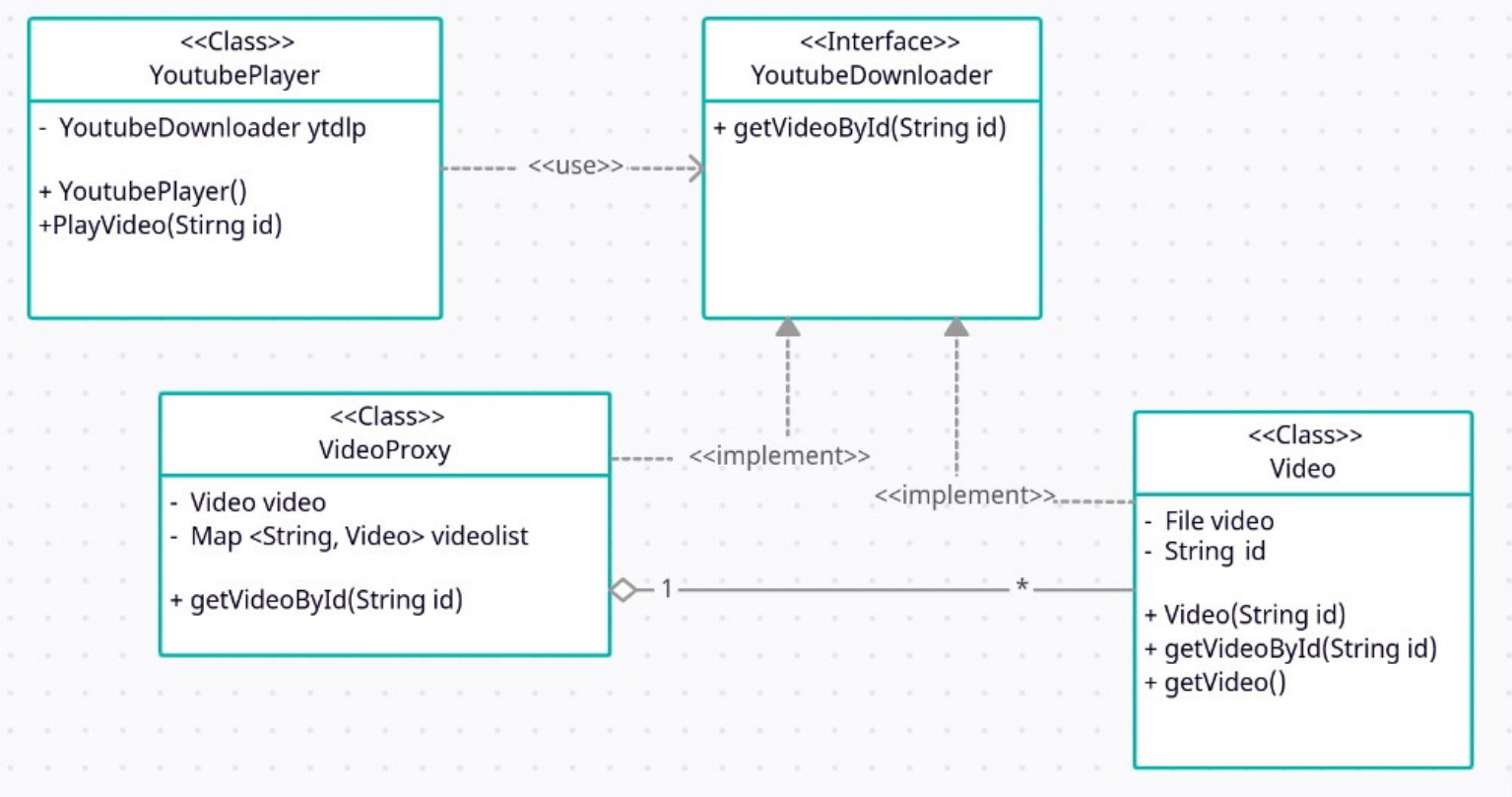


အပေါ်က class diagram မှာ player နဲ့ downloader ကို class နှစ်ခု ခွဲထားပြီး player မှာ downloader object ကို ခေါ်ပြီးသုံးထားပါတယ်။ code အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public class YoutubeDownloader {

    File getVideoById(String id){
    System.out.println("Video is downloading wait for a
     moment....");
    File f = new File("downloaded from youtube by "+id+".mp4");
    return f;
    }

}
```

```
public class YoutubePlayer {
    YoutubeDownloader ytdlp ;

    public YoutubePlayer() {
        ytdlp = new YoutubeDownloader();
    }

    public void PlayVideo(String id){

        File video =  ytdlp.getVideoById(id);
        System.out.println("Playing video "+id+".mp4");
        // other code for playing video -----
        //-----------------------------------
    }

}

public class Test {
     public static void main(String[] args) {
        YoutubePlayer yplay = new YoutubePlayer();
        yplay.PlayVideo("1abc");
        yplay.PlayVideo("1abc");

    }
}
```

အပေါ်က code ကို run ကြည့်ရင် နှစ်ခါလုံးမှာ video id အတူတူကို နောက်တစ်ခေါက်ပြန် download လုပ်ပြီးမှ play လုပ်တာကို တွေ့ရပါမယ်။ တကယ်တော့ id တူရင် တစ်ခါဘဲ download လုပ်ပြီး ဒုတိယအကြိမ်မှာ စောင့်စရာမလိုဘဲ တန်းပြီး play ပြတာမျိုး ဖြစ်သင့်ပါတယ်။ အဲ့လိုပုံစံမျိုး ဖြစ်ဖို့ proxy pattern ကို သုံးရပါမယ်။ proxy pattern သုံးပြီး ပြင်ထားတဲ့ class diagram ကိုအောက်မှာဖော်ပြပေးထားပါတယ်။



အပေါ်က class design မှာ youtubedownloader ကို interface အနေနဲ့ ထားထားပြီး proxy ကို ကော တကယ်သုံးမယ့် video class ကိုပါ interface ကို implements လုပ်ခိုင်းထားပါတယ်။ တကယ်ခေါ်သုံးတဲ့ အခါ interface ကိုဘဲ datatype အဖြစ်ထားပြီးခေါ်သုံးထားပါတယ်။ အခုလိုမျိုးပုံစံမျိုးကို proxy pattern လို့ ခေါ်ပါတယ်။ ထပ်ပြီးအသေးစိတ်နားလည်အောင် code အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

```
public interface YoutubeDownloader {
   public File getVideoById(String id);
}

public class Video implements YoutubeDownloader{

    private File video;
    private String id;

    public Video(String id) {
        this.id = id;
        video = new File("downloaded from youtube by
         "+id+".mp4");
    }

    @Override
    public File getVideoById(String id) {
        return this.video;
    }

    public File getVideo() {
        return video;
    }

}

public class VideoProxy implements YoutubeDownloader{
    Video video;
    Map <String, Video> videolist = new HashMap <String,
     Video> ();

    @Override
    public File getVideoById(String id) {
        if (videolist.containsKey(id)){
            System.out.println("video is already downloaded so
             there is no waiting.....");
            video = videolist.get(id);
        }
        else{
             System.out.println("Video is downloading wait for
             a moment....");
            video = new Video(id);
            File videotoplay = video.getVideo();
            videolist.put(id, video);
```

```
        }
        File videotoplay = video.getVideo();
        return videotoplay;
    }
}

public class YoutubePlayer {
    YoutubeDownloader ytdlp;

    public YoutubePlayer() {
       ytdlp = new VideoProxy();
    }

    public void PlayVideo(String id) {

        File video = ytdlp.getVideoById(id);
        System.out.println("Playing video " + id + ".mp4");
        // other code for playing video -----
        //-----------------------------------
    }
}

public class Test {

    public static void main(String[] args) {
        YoutubePlayer yplay = new YoutubePlayer();
        yplay.PlayVideo("1abc");
        yplay.PlayVideo("1abc");

    }
}
```

အပေါ် code မှာ highlight ပြထားတဲ့ အပိုင်းကို သေချာကြည့်ပါ video objects တွေကို Map နဲ့ သိမ်းထားပြီး ရှိပြီးသား object ဆိုနောက်တခေါက် object အသစ်မ ဆောက်ဘဲ Map ထဲကနေ ပြန်ထုတ်သုံးပြီး မရှိတဲ့ အခါမှသာ object အသစ်ဆောက်ပါ တယ်။ အခုလိုမျိုး processing အချိန်ကြာတဲ့ အပိုင်းကို proxy ခံပြီး လိုအပ်မှခေါ်သုံးတဲ့ ပုံစံမျိုးကို virtual proxy pattern လို့ခေါ်ပါတယ်။