Gentle Introduction to Object Oriented Programming

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

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

Evolution to OOP

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

ဖြစ်အောင်လုပ်မယ်။ ဒီလိုနည်းနဲ့.သွားပါတယ်။ နောက်ကျတော့ data abstraction ကိုထဲ့လာပါတယ်။

Data Abstraction

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

Beginning of OOP

Data abstraction ကြောင့် custom data type တွေ တော့ဖန်တီးလို့.ရပါပြီ။ ဒါပေမဲ့ ပြဿနာ တခုက သူ.ကိုပြင်မယ်ဆိုရင် ဆိုပါစို့. existing ADT(Abstract Data Type) တခုကိုပြင်မယ်ဆိုရင် သူ. source code ကိုပြင်ရမှာပါ။ ဒါကို destructive modification လို့.ခေါ်ပါတယ်။ အဲ့ဒါက အန္တရာယ်များပါတယ် ဘာလို့.လဲဆိုတော့ project တခုမှာ ADT တခုကို reference လုပ်ပြီးသုံးထားတဲ့ code တွေ အများကြီးရှိနိုင်လို့.ပါ။ ADT ရဲ့. source code ကိုမထိပဲပြင်နိုင်တဲ့ နည်းကတော့ OOP သုံးပြီး inheritance နဲ့. functionality အသစ်ကို ထပ်ထဲ့တာ existing method ကို modify လုပ်တာပါပဲ။ ဒါဆို ရှိပြီးသား code ကို မထိပဲနဲ့.

ပြင်လို.ရပါပြီ။ Software ဆိုတာ ရေးပြီးတာနဲ.ပြီးတာမဟုတ်ပဲ အမြဲ ပြောင်းလဲနေနိုင်ပါတယ် OOP ရဲ. feature တွေက ဒါတွေကို handle လုပ်နိုင်ပါလိမ့်မယ်။

Data Abstraction versus OOP

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

Encapsulation

Inheritance

Polymorphism

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

Pure and Impure OO Language

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

Encapsulation

Procedural language တွေမှာ data ကိုဘုံထားပြီး function တွေကနေဝိုင်းသုံးကြပါတယ်။ Project size ကြီးလာတာနဲ.အမျှ အဲ့ဒီ ဘုံသုံးထားတဲ့ variable(global variable) တွေကို ဘယ် module, ဘယ် function ကသုံးထားတယ်ဆိုတာ လိုက်ကြည့်ဖု.ိတော်တော်ခက်သွားပါပြီ။ PHP မှာဆို global variable သုံးပြီး page အများကြီး ဝိုင်းသုံးတာမျိုးပါ။ နောက်ပြီး ကိစ္စရှိလို. အဲ့ဒီ global variable ကိုပြင်မယ်ဆိုရင် သူ.ကို သုံးထားတဲ့ module တွေ function တွေကို ပါလိုက်ပြင်ရမှာပါ။ တခုခုကိုပြင်ရင် တခြား အပိုင်းတွေပါ ထိခိုက်ကုန်ပါတယ်။ ဒါကြောင့် OO language မှာ အဲ့တာကိုမဖြစ်အောင် ကာကြပါတယ်။ Program တွေကို modular ဖြစ်အောင်ရေးဖို.ကြိဆကြပါတယ်။ Modular ဖြစ်တယ်ဆိုတာ program module, function, contruct တခုဟာ သူ.ဟာသူနဲ.ရပ်တည်နိုင်ရပါမယ်။ သူများ function တွေကို အများကြီးသုံးနေရတာ dependency များနေတာ မဖြစ်ရပါဘူး။ Modular ဖြစ်မှ dependency နည်းမှပဲ ပြင်ရတာ လွယ်ကူမှာဖြစ်ပါတယ်။ ဒါကြောင့် global variable အစား

ကို. data ကိုပဲသုံးမယ် သူများက လာသုံးမရအောင် access ကို restrict လုပ်ထားမယ် သူ. data ကို သူ. method တွေကပဲသုံးမယ် တခြားသော module တွေက ဒီ module ကို သုံးချင်ရင် module ကပေးထားတဲ့ public interface(publicly accessible method) တွေ ကနေ သုံး။ ဒါဆိုရင် တခြား module တွေဟာ သူများရဲ. data ကိုသွားထိလို.မရတဲ့အတွက် သီးသန်.ဆန်သွားတဲ့အတွက် modularity ပိုကောင်းပါတယ်။ module တခုကိုပြင်မယ်ဆိုရင် public interface(public interface ကပဲသူများ module နဲ. dependence ဖြစ်ပါတယ်) ကိုမပြင်မချင်း လွတ်လွတ်လပ်လပ်ပြင်နိုင်မယ်။ ဒါကို encapsulation လို.သုံးပါတယ်။ Encapsulation ရဲ.အနှစ်သာရက ကို. data ကိုသူများက accidentally destroy မဖြစ်အောင် dependency မရှိအောင် ထားရတာပါ။ Encapsulation ကို data hidiing နဲ.တွဲသုံးပါတယ်။ Encapsulation နဲ. data hiding ဟာ တူသလိုလိုနဲ. ကွဲပါတယ် data hiding က ကို.data ကိုပြင်ပက လာသုံးလို.မရအောင်ကာထားတာမျိုးပါ။ Encapsulation ကတော့ data ကို ကို. function တွေနဲ. modularity ရအောင် ချိတ်သုံးတာကို ဆိုချင်တာပါ။ Java, C#, C++ မှာဆို private accessifier တွေသုံးပြီး encapsulation ရအောင်လုပ်ကြပါတယ်။ Member variable ကို Private ထားပြီး getter ,setter function တွေနဲ.သုံးကြပါတယ်။ အဲ့လိုသုံးတိုင်း encapsulation မရနိုင်ပါဘူး ။ အောက်က Java Program ကိုကြည့်ပါ။

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

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

```
<script>
  function getObject() {
    var data;

    function dataGetter() {
      return data;
    }

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

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

Encapsulation ထိန်းလို.ရပါတယ်။ Encapsulation ဆိုတာ class level ထိန်းမှမဟုတ်ပါဘူး package , module တွေဟာလဲ ပြင်ပ ကကောင်တွေတိုက်ရိုက်သုံးလို.မရအောင် ထိန်းပေးထားတဲ့ encapsulation construct တွေပါပဲ။ Encapsulation နဲ. ပတ်သတ်တဲ့ OO Principle တခုရှိပါတယ် အဲ့ဒါကတော့ Program to interface, not to implementation ပါ။ Program ရေးတဲ့အခါမှာ module တွေ class တွေဟာ သူများရဲ. implementation detail ကိုမသိသင့် မထိသင့်သလို. ကို.ဟာကိုလဲ expose မလုပ်သင့်ပါဘူး။ အပြင်ကသုံးမဲ့ ကိုနဲ.တကယ် interact ပေးလုပ်မဲ့ကောင်တွေကိုပဲ public interface အနေနဲ.ထားပေးရမယ်ဆိုလိုတာပါ။ ဒါမှ maintenance ကောင်းမှာပါ။Encapsulation နဲ.ပတ်သတ်တဲ့ တခြား OO principle တခုကတော့ Open Close Principle ပါပဲ။

A module should be open for extension but closed for modification.

Module တွေ classes တွေဟာ သူတို.ကို accidentally modified မလုပ်နိုင်အောင်ထားသင့်ပြီးတော့ extend လုပ်နိုင်အောင်တော့ ထားပေးရမယ်ဆိုတာပါ။ Encapsulation ရဲ. central theme ကိုက close for modification ပါပဲ။

Inheritance

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

ကိုပိုမိုလွယ်ကူစေပါတယ်။ Inheritance ရဲ. major usage က ၂ခုရှိပါတယ် design မှာသုံးလို.ရသလို implementation အတွက်လဲသုံးပါတယ်။ အဲ့ဒါတွေကတော့

Inheirtance as conceptual modelling

Inheritance as incremental program modification

OOP ရဲ.တခြား paradigm တွေထက် popular ဖြစ်ရတဲ့အကြောင်းရင်းက Object Orientation ဟာ implementation သက်သက်သာ မဟုတ်ပဲနဲ. modelling or design အတွက်ပါသုံးလို.ရလို.ဖြစ်ပါတယ်။ Real world problem တွေကို OO thinking နဲ. model ချတဲ့အခါမှာပိုမိုလွယ်ကူပါတယ်။ လူတွေဟာ အရာဝတ္ထုတွေ အကြောင်းအရာတွေကို ခွဲခြားစိတ်ဖြာတယ် (classification ဥပမာ လူဆိုတာ သတ္တဝါ တိရစ္ဆာန်ကလဲ သတ္တဝါ) ၊ generalization (ဥပမာ car engine ကလဲစက် motor engine ဆိုတာလဲ စက်)၊ grouping,composition အစရှိတာတွေနဲ.တွေးခေါ်ကြပါတယ်။ ခုနကပြောတဲ့ classification၊ generalization၊ specialization အစရှိတာတွေကို OO paradigm မှာ easiliy model လုပ်လို.ရပါတယ်။ classification ကို class contruct နဲ.လုပ်ကြပါတယ်။ Generalization နဲ. Specialization ကိုတော့ inheritance နဲ. model လုပ်ကြပါတယ်။ Grouping နဲ. composition ကိုတော့ class တွေမှာ တခြား classes တွေရဲ. reference variable တွေထဲ့သွင်းခြင်းအားဖြင့် သုံးကြပါတယ်။

Inheritance as conceptual modelling

Inheritance ကို Generalization, Specialization အတွက်သုံးပြီဆိုရင် ဒါဟာ inheritance as coneptual modelling ပါ။ Base class or parent class ကို generalized classes လို.ခေါ်ပြီးတော့ child class ကိုတော့ specialized class

လို.ခေါ်ပါတယ်။ Conceptual modelling လုပ်မယ်ဆိုရင် base class နဲ. drived class ဟာ taxonomy အရတူရမှာပါ။ ဥပမာ Teacher, Doctor နဲ. Human ဆိုတာဟာ taxonomy အရတူပါတယ်။ Hierarchical relationship ရှိတာကိုပြောချင်တာပါ။ Teacher, Doctor သည် is a kind of Human ပါပဲ။ Taxonomy အရ မတူဘူးဆိုရင် Generalization Specialization မလုပ်သင့်ပါဘူး ဥပမာ Bird နဲ. Aeroplane သည် ပျံတာချင်းတူပေမဲ့ သူတို.ကို Inheritance နဲ.ချိတ်လို.မရပါဘူး။ သူတို.က taxonomy အရမတူပါဘူး။ Inheritance လို conceptualization လုပ်ပေးနိုင်တဲ့နောက်တခုက တော့ Subtyping ပါ။Subtyping ဆိုတာ programming lanaugag ကနေပေးထားတဲ့ feature တွေကိုသုံးပြီး type တခု သည် အခြား type တခုရဲ. subtype ဖြစ်ပါတယ်ဆိုပြီးသုံးတာပါ။ Base type နေရာမှာ based type ကိုအခြေခံထားတဲ့ တခြား type ကိုအစားထိုးပြီးသုံးလို.ရပါတယ်။ Subtyping ကို type polymorphism လို.လဲသုံးကြပါသေးတယ် ။ Java, C# တို.မှာဆိုရင် interface construct ကိုထဲ့ပေးထားပါတယ်။ သူ.ကိုထဲ့ပေးထားရတဲ့အကြောင်းရင်းက conceptual modelling အရ taxonomy မတူတဲ့ကောင်တွေကို polymorphic လုပ်ချင်ရင်သုံးဖို.ပါ။ Java programmer တော်တော်များများ က abstract class နဲ. interface ဘာကွာလဲမေးလိုက်ရင် ရေရေရာရာကွဲပြားတဲ့အဖြေကို မဖြေနိုင်ကြပါဘူး။ အဓိက ကွာခြားချက်က taxonomy တူမယ်ဆိုရင် abstract class ကိုသုံးပြီး inheritance နဲ.ဖြေရှင်းမယ် မတူဘူးဆိုရင် subtyping ကို သုံးပြီး interface နဲ. design လုပ်ရမှာပါ။ ဥပမာ work ဆိုတဲ့ method ကို polymorphic ဖြစ်အောင်လုပ်ချင်တယ်ဆိုပါစို. Teacher မှာရော Doctor မှာရော လုပ်ချင်တာဆိုရင် သူတို. ၂ခုဟာ taxonomy အရတူတဲ့အတွက် abstract class ဒါမှမဟုတ် ရိုးရိုး class သုံးပြီး inheritance နဲ. model လုပ်ရမှာပါ။ Fly ဆိုတဲ့ method ကို Bird နဲ. Aeroplane အတွက် polymorphism လုပ်ချင်တယ်ဆိုရင် သူတို. ၂ ခုဟာ taxonomy မတူတဲ့အတွက် Flyable ဆိုတဲ့ interface တခုထားပြီး model လုပ်ရမှာပါ။ Static type language တွေမှာ polymorphic operation တွေလုပ်ဖို. subtype တွေဖြစ်မှ လုပ်လို.ရပါတယ်။ Subtype မဟုတ်ရင် method ရှိနေလဲ ခေါ်လို.မရပါဘူး။ Dynamic

language တွေမှာ duck typing ကိုပေးထားတဲ့အတွက် အဲ့လိုခေါ်လို.ရပါတယ်။ ဒါပေမဲ့ PHP လို dynamic language ဖြစ်တယ် duck typing ရတဲ့ language မှာ interface ကိုထဲ့ပေးထားတာဟာ conceptual modelling အရ subtyping ကိုသုံးစေချင်လို.ဖြစ်ပါတယ်။ နောက်တခုက normal class နဲ. abstract class ဘယ်လို usage ခွဲမလဲပေါ့ ။ Parent ကနေပဲ child က code တွေကို one direction ခေါ်ချင်ရင် hook အနေနဲ.ထားချင်ရင် abstract class ကိုသုံးပါတယ် မဟုတ်ရင် both direction ခေါ်ချင်ရင်တော့ normal class ကိုသုံးပြီး model လုပ်လို.ရပါတယ်။

Inheritance as incremental program modification

Inheritance as incremenatal program modification ဆိုတာ base class ရဲ. source code ကိုမပြင်ပဲနဲ. သူ.ကို implementation အရ ပြင်ချင်တဲ့အခါမှာ extend လုပ်ပြီးသုံးတာမျိုးပါ။ ဥပမာ Java မှာဆို window frame တခုဆောက်ချင်ရင် JFrame ကို extend လုပ်တာမျိုးပါပဲ။ ဒါကတော့ implementation အတွက်သုံးတာပါ။ တခါတလေမှာကိုပြင်ချင်တဲ့ source code ကိုမရနိုင်လို. binary class file ပဲရတဲ့အတွက် သူ. functionality ကို reuse လုပ်ချင်တာဖြစ်ဖြစ် modify လုပ်ချင်တာဖြစ်ဖြစ်သုံးတာမျိုးကိုလဲ ဒီနည်းထဲမှာအကျုံးဝင်ပါတယ်။ Modification ကိုပေးတဲ့နေရာမှာ language တခုနဲ.တခုမတူပါဘူး။ Inheritance breaks Encapsulation ဆိုတာလဲရှိပါတယ်။ ဘာကိုဆိုချင်တာလဲဆိုတော့ inheritance ကိုသုံးတာလွဲခဲ့ရင် သူဟာ parent classes ရဲ. function တွေကို တလွဲသုံးမိရင် encapsulation ဆိုတာကို ချိုးဖောက်တာဖြစ်နိုင်ပါတယ်။ သတိထားရမှာက parent classes ရဲ. encapsulation ကိုမထိအောင်သုံးရမှာပါ။ Favour inhertiance over composition ဆိုတာလဲရှိပါသေးတယ်။ တချို့က code reuse လုပ်ချင်ယုံသက်သက်နဲ. inheritance ကိုသုံးကြပါတယ် ။ taxonomy အရ မတူရင်သော်လည်းကောင်း modifcation or added functionality မထဲ့နိုင်ရင်သော်လည်းကောင်း အဲ့လိုသုံးတာမှားပါတယ်။ Code reuse လုပ်ချင်ရင် composition

ကိုသုံးပါ။ ဘာလို.လဲဆိုတော့ inheritance hierarchicy များလာတာနဲ့.အမျှ classes တွေဟာ dependency များလာပါတယ်။ Base class တခုကို ပြောင်းလိုက်တာနဲ့. အောက်က child classes တွေမှာပါ effect ဖြစ်နိုင်ပါတယ်။ Composition ဆိုတာက ကိုသုံးလို.တဲ့ class ကို reference variable သုံးပြီး ယူသုံးတာပါပဲ။

Single Inheritance vs Multiple Inheritance

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

Class Inheritance Versus Prototype Inheritance

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

Classical Inheritance

Java,C++,C# တို.လို language တွေမှာ support လုပ်တဲ့ inheritance ပုံစံကို classical inheritance လို.သုံးပါတယ်။ ဥပမာ အောက်က Java program ဆိုပါစို.

```
class Base
{
    int baseData;
}

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

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

Prototypical Inheritance

JavaScript မှာပေးထားတဲ့ inheritance model က prorotypical inheritance ပါ။

```
function Base()
{
   this.baseData = [];
}
function Child()
{
   this.childData  = "childdata";
}
Child.prototype = new Base();//set up inheritance
var c1 = new Child();
var c2 = new Child();
c1.baseData.push(100);
console.log(c2.baseData);
```

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

c1.baseData.push(100);

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

တကယ်လို.များ

c1.baseData = [200]; လို.ရေးလိုက်ရင် c1 မှာ သူ.ရဲ.ကိုယ်ပိုင် baseData ဆိုတဲ့ attribute ကို JS ကထဲ့ပေးတော့မှာပါ။ Attribute တွေကို JS မှာရှာတဲ့ပုံစံက read ဆိုရင် current object ကနေ ရှာပါတယ်။ နောက်မတွေ.ရင် သူ. prototype chain ကိုလိုက်ရှာပါတယ်။ တကယ်လို. write သာလုပ်မယ်ဆိုရင် parent မှာရှိနေလဲ သူ.current object မှာမရှိရင် attribute အသစ်အနေနဲ.ထဲ့ပေးမှာပါ။ ဒါကိုသတိထားစေချင်ပါတယ်။ Prototypical inheritance သည် share semantic ဖြစ်ပါတယ်။ Parent object ထဲကို attribute တွေ ထပ်ထဲ့တာ နှုတ်တာ အားလုံးသည် child အားလုံးမှာ effect ဖြစ်ပါတယ်။

Dynamic Inheritance

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

Polymorphism

Polymorphism ဆိုတာ Greek ဘာသာစကား Poly (များစွာသော) Morph(ပုံသဏ္ဌာန်ပြောင်းခြင်း) ဆိုတာကနေ ဆင်းသက်လာတာပါ။ Polymorphism is the ability to present the same interface for differing underlying forms (data types). တူညီတဲ့ interface (public method or method) ပေါ်မှာ contextual object သို.မဟုတ် data type အပေါ်မူတည်ပြီး ပြောင်းလဲနိုင်မှု.ကိုပြောတာပါ။ မြန်မာလို ရိုးရိုးရှင်းရှင်းပြောရမယ်ဆိုရင် Human လူဆိုရင် work ဆိုတဲ့ method ရှိပါတယ်။ Teacher, Doctor တွေဟာလဲ kind of Human ဖြစ်တဲ့အတွက် work ဆိုတဲ့ method ရှိပါတယ်။ ဒါပေမဲ့ Teacher က work ဆိုရင်တော့ စာသင်မှာဖြစ်ပြီးတော့ Doctor ရဲ. work ဆိုရင်တော့ ဆေးကုမှာပါ။ work ဆိုတဲ့ တူညီတဲ့ interface ပေါ်မှာ method call (message passing) လုပ်တာပါပဲ။ ဒါပေမဲ့ အခေါ်ခံရတဲ့ object အလိုက်မူတည်ပြီး work ရဲ. implementation ကွာမှာပါ။ အောက်က Java Code ကိုကြည့်ပါ။

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

    }
}
```

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

```
class Doctor extends Human
{
    public void work()
    {
        System.out.println("I give medical treatment");
    }
}
```

```
class Test
{
    public static void main(String args[])
    {
        Human h = new Teacher();
        h.work();
        h = new Doctor();
        h.work();
    }
}
```

Dynamic Polymorphism

အပေါ်က code မှာ Teacher ရော Doctor ၂ခုစလုံးက Human ကို extends လုပ်ပြီးတော့ work method ကို override လုပ်ထားပါတယ်။ အောက်က Test class ရဲ. main မှာ ပထမဆုံး Teacher ကိုဆောက်ပြီး Human အမျိုးအစားဖြစ်တဲ့ h ထဲကိုထဲ့ပါတယ်။ h သည် base class reference ဖြစ်တာကြောင့် child object (Teacher) ကိုထဲ့ခွင့်ရပါတယ်။ ဒါက sub typing သဘောအရခွင့်ပေးတာပါ။ Parent reference type ထဲကို child object တွေထဲ့ခွင့်ရှိပါတယ်။ ပြီးတော့ h.work() လို. method ကိုခေါ်ပါတယ်။ ဒါပေမဲ့ h.work() သည် Teacher object ရဲ. work ကိုလှမ်းခေါ်မှာဖြစ်ပါတယ်။ ဘာလို.လဲဆိုတော့ java method တွေက static မကြေငြာထားရင် auto virtual ဖြစ်လို.ပါ။ Human ရဲ. work ကိုခေါ်မဲ့အစား h ထဲမှာတကယ်ရှိတဲ့ object Teacher ကိုခေါ်ပါလိမ့်မယ်။ဒါဟာ polymorphism ပါပဲ။ နောက်တခါ h ထဲကို Doctor ထဲ့ထားပြီး h.work

လို.ခေါ်မယ်ဆိုရင်တော့ doctor class ရဲ. work ကိုခေါ်ပါလိမ့်မယ်။ ကျွန်တော်တို. ခေါ်တာ h.work ပါပဲ (same method) ပါ။ ဒါပေမဲ့ implementation (execute လုပ်မှာသည် Teacher ရဲ. work လား Doctor work လား)ကိုတော့ h ထဲမှာ ရောက်နေတဲ့ object ပေါ်မူတည်ပြီး ဆုံးဖြတ်မှာပါ။ ဒါကြောင့် Polymorphism သည် same method with different implementation လို.လဲဆိုကြပါတယ်။ တစ်ခုသတိထားရမှာက static language (Java,C#,C++)မှာ polymorphism သည် virtual method မှာပဲအလုပ်လုပ်ပါတယ်။ Java မှာတော့ဘာမှမရေးရပေမဲ့ C++ မှာ virtual လို.ရေးရပြီး pointer နဲ.သုံးရပါတယ်။ C# မှာတော့ parent class method မှာ virtual လို.ရေးရပြီး child class method မှာတော့ override ဆိုတဲ့ keyword ကိုထဲ့ပေးရပါတယ်။

ဘာကြောင့် polymorphism ကိုသုံးရတာလဲ သူ.ရဲ. ကောင်းကျိုးတွေက ဘာတွေလဲလို.မေးစရာရှိပါတယ်။ Polymorphism သုံးခြင်းအားဖြင့် hard code ရေးရတာ ဥပမာ object က Teacher ဆို teacher work ကိုခေါ်ပါဆိုပြီး လိုက်ရေးစရာမလိုပါဘူး။ ဒါကဘာကောင်းလဲဆိုတော့ နောင်တချိန်မှာ တခြား Engineer လို class တခု ထပ်ထည့်မယ်ဆိုရှိပြီးသား code တွေ ထိစရာမလိုပဲနဲ. ပြင်နိုင်ပါတယ်။ Extensibility ကောင်းတယ် လို.ပြောရမှာပါ။ ဒီပြင့်နေရာတွေလဲရှိပါသေးတယ်။ ဥပမာ ပြရရင် Procedural language နဲ.ယှဉ်ပေးရမှာပါ။ PHP စပေါ်ကာစက OOP ကို support မလုပ်သေးပါဘူး ။ အဲ့အချိန်မှာ database ကို connect လုပ်တဲ့ code တွေကို function တွေအနေနဲ.ရေးရပါတယ်။ ဥပမာ MySQL ကိုချိတ်မယ်ဆိုရင် mysql_connect ဆိုတဲ့ function ကိုသုံးရပါတယ်။ Oracle ကို connect လုပ်ချင်ရင်တော့ oci_connect ဆိုတဲ့ function ကိုသုံးရပါတယ်။ တခြား sql statment တွေ execute လုပ်ချင်ရင်လဲ သက်ဆိုင်ရာ mysql oracle function တွေလိုက်သုံးရပါတယ်။ တခြား datsbase server တွေဆိုလဲ သူတို. နဲ.ဆိုင်တဲ့ method တွေလိုက်မှတ်ရပါတယ်။ ဒါဟာ programmer တယောက်အတွက် မကောင်းလှပါဘူး။ နောက်ပိုင်းမှာ PHP မှာ OOP ရလာပြီး PDO (PHP Data Object) ဆိုပြီး Polymorphism,Inheritance သုံးပြီး database function တွေကိုပြောင်းရေးလိုက်ပါတယ်။ တကယ်တော့ mysql connect နဲ. oracle connect သည်

implementation ပဲကွာတာပါ ဒါကို OO thinking အရ connect ဆိုပြီး ဘုံထုတ် polymorphism အရ oracle PDO ဆိုရင် oracle connection code လှမ်းခေါ် MySQL PDO ဆိုရင် MySQL connection code ကိုလှမ်းခေါ်ပေးရုံပါပဲ။ ဒါဆို database တွေဟာ implementation တွေသာ ကွာချင်ကွာမယ် connection ချိတ်တာ sql statement တွေ run တာကို method တွေအနေနဲ့.ဘုံထုတ်လိုက်တော့ Programmer က MySQL ချိတ်နေတာလား Oracle ချိတ်နေတာလား ပြောစရာမလိုပဲသုံးလို.ရ.ပါပြီ။ Abstraction ပိုင်းအရကြည့်ရင် Programmer က အများကြီး လိုက်မှတ်စရာမလိုတော့ပါဘူး ။ PDO မှာ database server အားလုံးအတွက် လိုမဲ့ public interface တွေထားပြီး database တခုခြင်းဆီအတွက် different implemnetation ကို extends လုပ်ပြီးပေးလိုက်ယုံပါပဲ။ တကယ်လို. Programmer က MySQL ကနေ Oracle ကိုပြောင်းမယ်ဆိုရင် PDO မှာလွယ်ပါတယ် connection string ပြောင်းယုံပါပဲ။ နဂို procedural programming style အရသာဆိုရင် database code အကုန်ကိုပြောင်းရမှာပါ။ နောက်တခါ database server အသစ်တခု ထပ်ထွက်တယ်ဆိုပါစို့.ဒါဆိုရင်လဲ ရှိတဲ့ PDO class ကို inherit သုံးပြီး different implementation ပေးလိုက်ယုံပါပဲ။ ဒါဆို polymorphism ရဲ.အသုံးဝင်ပုံကိုသိလောက်ပါပြီ။ Java , C# တို.မှာလဲ ဒီလိုသဘောတရားသုံးပြီး JDBC , ADO.NET API တွေကိုဆောက်ထားတာပါ။ ခုကျွန်တော်ပြောသွားတဲ့ Polymorhpism အမျိုးအစားကို dynamic polymorphism လို.ခေါ်ပါတယ်။ ဘာလို.လဲဆိုတော့ ဘယ် method ရဲ. code (implementation) run မယ်ဆိုတာကို run time (dynamic) ရောက်မှ ဆုံးဖြတ်လို.ပါပဲ။

Subtyping vs Duck Typing

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

Static Polymorphism

Static polymorphism ဆိုတောကတော့ method overloading ကိုပြောချင်တာပါ။ method တွေကို နာမည်တူရမယ် ၊ return type (sub type ဆိုလဲရ) တူရမယ် ဒါပေမဲ့ protocol မတူပဲ ရေးရင် ဒါကို method overloading လို.ခေါ်ပါတယ်။ Method protocol ဆိုတာ ဒီနေရာမှာ method တခုရဲ. parameter အရေအတွက် ၊ parameter type ၊ parameter order အားလုံးကိုပြောတာပါ။ static polymorphism လို.ခေါ်ရခြင်းကတော့ method overloading မှာ ဘယ် metod ကိုခေါ်မယ်ဆိုတာကို

compile time မှာ ဆုံးဖြတ်လို.ပါ။ Dynamic language တွေဖြစ်တဲ့ Ruby, Python, JavaScript နဲ. PHP တို.မှာ method overloadig မရှိပါဘူး။

Parametric Polymorphism

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

SOLID Principle

ဒီတခေါက် ကတော့ OO Design မှာအခြေခံအကျဆုံး principle လို.ဆိုရမဲ့ SOLID ပါ။ SOLID ကိုလူသိများလာအောင် လုပ်ခဲ့တဲ့သူတော့ Uncle Bob လို.လူသိများတဲ့ Robert C Martin ပါ။ သူရေးခဲ့တဲ့ စာအုပ်တွေထဲက ထင်ရှားတဲ့တအုပ်ကတော့ Clean Code ပါ။ SOLID က principle တခုတည်းမဟုတ်ပဲ ၄ ခုကိုစုပေါင်းထားတာပါ။ အဲ့ဒါတွေက

SRP (Single Responsibility)

OCP (Open Close Principle)

LSP(Liskov Substitution Principle)

ISP(Interface Segregation Principle)

DI(Dependency Inversion)

တို.ပဲဖြစ်ပါတယ်။

(Single Responsiblity)

ပထမဆုံး principle က အရိုးရှင်းဆုံးနဲ့. သုံးရတာ လက်အဝင်ဆုံး principle ပါ။ သူ.ရဲ့. definition ကတော့ The Single Responsibility Principle (SRP) states that each software module should have one and only one reason to change. Class တစ်ခု သို.မဟုတ် module တခုဟာ အကြောင်းရင်းတခုကြောင့်ပဲ code ကို changes လုပ်ရမယ်။ ဒီနေရာမှာ class or module ရဲ့. responsibility ဆိုတာ သူလုပ်ရမဲ့ behaviour(set of method) ကိုဆိုချင်တာပါ။ ဆိုချင်တာက class, module တွေဟာ တူညီတဲ့ behaviour တွေကို စုထားတာပဲဖြစ်ရမယ် လို.ဆိုချင်တာပါ။ တူညီတဲ့ responsibility တွေကိုပဲစုထားရမယ် ဒါမှမဟုတ် responsibility တခုတည်းကိုပဲ အာရုံစိုက်ရမယ် လုပ်ရမယ်လို.ဆိုချင်တာပါ။ Robert C Matrin ရဲ့.ဥပမာ အတိုင်းပဲ ပြပါမယ်။ အောက်က class ကိုကြည့်ပါ။

```
public class Employee {
    public Money calculatePay();
    public void save();
     String reportHours();
}
```

ပြထားတဲ့ class မှာ responsiblitiy 3 ခုပါနေပါတယ်။ ဒီအတိုင်းကြည့်ရင် တခုတည်းထင်ရပေမဲ့ calculatePay ဆိုတာ accounting နဲ့.ဆိုင်တာပါ။ save ကတော့ database operation နဲ့.ဆိုင်တာပါ။ reportHours() ကတော့ HR နဲ့.ဆိုင်တာပါ။ ပြင်ပ realworld က အဲ့ဒီ responsibility တွေနဲ့. တိုက်ရိုက်သက်ဆိုင်တဲ့ entity (ဒီနေရာမှာ calculatePay သည် HR accounting, save သည် database, reportHours သည် HR, management) တွေသည် မတူပါဘူး။ ဒါကြောင့် Employee class သည် single responsiblity

ကိုချိုးဖောက်နေပါတယ်။ မြင်သာအောင်ပြောရမယ်ဆိုရင်။ Database ကိုပြောင်းမယ်ဆိုလဲ (ဥပမာ Employee Table structure ပြောင်းသွားလို.ဆိုပါတော့) Employee class ကိုပြင်ရမယ်။ Payment တွေပြောင်းသွားတယ်ဆိုရင်လဲ Employee class ကိုပြင်ရမယ် ဒါဆိုရင် reason of change သည် တခုထက်ပိုနေပါပြီ ဒါဆိုရင် single responsibility နဲ.မကိုက်တော့ပါဘူး။ Single Responsiblity နဲ.မကိုက်တော့ ဘာဖြစ်လဲဆိုတော့ Cohesion နည်းပါတယ်။ Cohesion နည်းတော့ သူ.ကို resuse လုပ်ဖို.အဆင်မပြေပါဘူး။ ဥပမာ တခြား module တခုကနေ Employee data ကိုပဲသုံးချင်တယ် pay တွက်တာ မလိုချင်ဘူးဆိုရင် Employee ကို reuse လုပ်ဖို.အဆင်မပြေပါဘူး။ နောက်တခုက responsibility တွေရောနေတဲ့အတွက် maintenance လုပ်ရတာခက်ပါတယ်။ အဆင်မပြေပါဘူး။ ဒါကြောင့် နောက်ပိုင်း JavaEE Framework တွေဖြစ်တဲ့ spring framework လိုကောင်မျိုးတွေမှာ layer ခွဲပြီးတော့ domain model ကိုသက်သက် (ဒီနေရာမှာ Employee နဲ.ဆိုင်တဲ့ အချက်အလက်ကိုပဲသိမ်းမယ်)။ နောက် service layer (ဒီနေရာမှာ pay calculation နဲ. reportHour ကိုလုပ်မယ်) နောက် DAO layer(ဒီနေရာမှာ Employee object ကို database မှာ save တာကိုလုပ်မယ်)ဒါမျိုးတွေခွဲလုပ်ကြပါတယ်။ အဲ့လိုခွဲလုပ်တော့ တခုခုကို ပြင်ချင်ရင် နောက်တခုကိုသွားမထိပါဘူး။ ဥပမာ view layer ကိုပြင်တာနဲ. database layer ကိုသွားထိစရာမလိုပါဘူး။ အဲ့ဒါမျိုးကိုကျတော့ seperation of concern လို.သုံးပါတယ်။ Concern ဆိုတာကတော့ system တစ်ခုကို feature တွေ အပေါ်မူတည်ပြီး ခွဲတာပါ။ Feature ဆိုတာကတော့ (domain, service, controller, view,database) အဲ့လို feature တွေ ပေါ်မူတည်ပြီး ခွဲပစ်တာပါ။

နောက်မြင်နိုင်တဲ့ Example တခုကတော့ begineer PHP example တွေမှာတွေ.ရတဲ့ code တွေပါပဲ။ Business logic နဲ. view (html markup, CSS etc) စတာတွေကို ရောရေးတာပါပဲ။ နောက်ပိုင်း framework တွေမှာတော့ တခုချင်းကို MVC သုံးပြီး seperation of concern နဲ.ခွဲထုတ်လိုက်ပါတယ်။ အကျိုးဆက်ကတော့ better maintenance ဖြစ်ပါတယ်။

နောက် VB program တွေမှာရေးလေ့ရှိပါတယ်။ Button တခုရဲ. OnClick event မှာ တခါတည်း database ကိုတန်းချိတ်တာရော business logic ပေါင်းရေးတာမျိုးရောပါ။ အဲ့လိုဆိုရင် single responsibility ကိုချိုးဖောက်ရာကျပါတယ်။ ဒါဆို maintenance လုပ်ရတာအဆင်မပြေပါဘူး။

Single Responsiblity နဲ.ပြောင်းပြန် anti pattern ကတော့ God Object ပါ ။ God Object ဆိုတာ object, class တစ်ခုမှာ ရှိသမျှ responsiblity တွေ method တွေ function တွေစုရေးထားတာကိုပြောတာပါ။ ဒါမျိုးက maintenance လုပ်ရတာ အလွန်ခက်တဲ့အတွက် မသုံးသင့်ပါဘူး။ SRP ဟာ class module တွေမှာပဲအသုံးဝင်တာမဟုတ်ပါဘူး method level, code block level မှာသုံးလဲရပါတယ်။ ဥပမာ method တခုဟာ responsibility တခုထက်ပိုနေပြီဆိုရင် ထပ်ခွဲထုတ် (refactoring နည်းနဲ. method အသစ်ပြန်ထပ်ထဲ့တာမျိုး)လုပ်ရမှာပါ။

OCP (Open Close Principle)

OCP ကတော့ open of extension ,close for modification ကို ရည်ညွှန်းတာပါ။ Class, module code တွေဟာ လွယ်လွယ်ကူကူ functionality တွေထပ်ထဲ့နိုင်ရမယ် (open for extension) ဒါပေမဲ့ သူတို.ကို code modification လုပ်စရာမလိုပဲနဲ. (close for modification) ဖြစ်ရမယ်လို.ဆိုချင်တာပါ။ ဥပမာနဲ.ပြပါမယ်။ ကျွန်တော်တို.က Shape တွေရဲ. sum area ကိုတွက်ပါမယ်။ Shape ကလောလောဆယ်မှာ Circle,Rectangle ဆိုပါစို.ဒါဆို pseudo code အနေနဲ.ဆိုဒီလိုရပါမယ်။

```
public long sumArea(List<Shape> shapes)
{
    long totalArea = 0;
    for(Shape s : shapes)
    {
        if(s instanceof Circle)
        {
            Circle c = (Circle)s;
            totalArea += c.area();
        }
        else if(s instanceof Rectangle)
        {
            Rectangle r = (Rectangle) s;
            totalArea += r.area();
        }
    }
    return totalArea;
}
```

အပေါ်ကပေးထားတဲ့ method မှာ shape တွေကို loop ပတ်ပြီး တခုချင်းစီကို စစ်ပါတယ် ။ Circle ဆို Circle, Rectangle ဆို Rectangle အနေနဲ့. typecast လုပ်ပြီး area တွေကိုပေါင်းပါတယ်။ အဲ့မှာ ဘာပြဿ နာရှိလဲဆိုတော့ close for extension ဖြစ်နေတာပါ။ ဆိုချင်တာက ကျွန်တော်က နောက်ထပ် Shape အသစ် area ကိုတွက်ချင်တယ် Cube ဆိုပါတော့ဒါဆိုရင် ခုနက method sumArea ရဲ့. for loop ထဲမှာ if တကြောင်းထဲ့ပြီး type cast လုပ် area ကိုခေါ် အဲ့လိုလုပ်ရမှာပါ။ ဘာလို.လဲဆိုတော့ OCP ကိုဖောက်ဖျက်လို.ပါ။ OCP အတိုင်း ရေးမယ်ဆိုရင် ဒီလိုရမှာပါ။

```
class Shape//base class
{
    long area();
}

class Circle extends Shape
{
    long area(){};
}

class Rectangle extends Shape
{
    long area(){};
}
```

```
public long sumArea(List<Shape> shapes)
{
    long totalArea = 0;
    for(Shape s : shapes)
    {
        if(s instanceof Circle)
        {
            Circle c = (Circle)s;
            totalArea += c.area();
        }
        else if(s instanceof Rectangle)
        {
            Rectangle r = (Rectangle) s;
            totalArea += r.area();
        }
    }
    return totalArea;
}
```

အပေါ်က AreaSummer class ဟာ OCP ကိုလိုက်နာပါတယ် သူ. loop ထဲမှာ polymorphism ကိုသုံးပြီး s.area ဆိုပြီးခေါ်ပါတယ်။ ဒါဆိုရင် ကျွန်တော်တို.က နောက်ထပ် shape တခုကိုထပ်ထဲ့မယ် သူ. area ကိုပေါင်းမယ်ဆို ရင် AreaSummer ရဲ. sum method ကိုပြင်စရာမလိုတော့ပါဘူး။ ဥပမာ Square class ကိုတွက်ချင်ရင် Sqaure class ကို base class Shape ကနေ extend လုပ်ပြီး area method ကို override လုပ်ပေးလိုက်ယုံပါပဲ။ ဒါဆိုရင် ရှိပြီးသားမူရင်း code ကိုလဲမထိပဲနဲ. ကိုလိုချင်တဲ့ functionality (extension) ကိုလဲ ထပ်ထဲ့နိုင်တဲ့အတွက် OCP နဲ.ကိုက်ညီပါပြီ။ OCP နဲ.ကိုက်အောင်ရေးမထားဘူးဆိုရင် code တွေဟာ တခုခု ထပ်ထဲ့မယ်ဆိုရင် လိုက် change နေရပါလိမ့်မယ်။

LSP: Liskov Substitution Principle

LSP ကိုအစပြုခဲ့တဲ့သူက MIT က Professor Barbara Liskov ဆိုတဲ့ဘွားတော်ပဲဖြစ်ပါတယ်။ ဘွားတော်က ACM Turing Award Winner လဲဖြစ်ပါတယ်။ ACM Turing Award ဆိုတာ Alan Turing ကို ဂုဏ်ပြုပြီး ပေးထားတဲ့ Computer Science မှာ အမြင့်ဆုံး ဆုပါပဲ။ Language Designer တွေ OS Designer တွေ Computer Science ကိုတနည်းတဖုံပြောင်းလဲပေးခဲ့တဲ့သူတွေပဲရကြပါတာပါ။ Dennis Ritchie (C language Designer) နဲ. Ken Thompson တို.လဲ ရဖူးပါတယ် UNIX ကို သူတို.နှစ်ယောက်လုပ်ခဲ့ကြလို.ပါ။ Liskov က CLU နဲ. Argus ဆိုတဲ့ Programming Languge ၂ခု ရဲ. designer ပါ။ အဲ့ language တွေက OOP ကိုပြောင်းလဲစေခဲ့ပါတယ်။ တခါတလေ ပြောနေကြတဲ့ ငါတို. langauge က feature ကို ဟိုဘက်က langauge က ခိုးသွားတယ်ဆိုတာမျိုး ကြားဖူးမှာပါ။ တကယ်ကရီစရာပါ modern language တော်တော်များများက သူတို.ကိုယ်ပိုင် concept ဆိုတာထက် ရှေးက langauge concept တွေအပေါ်ထပ်ကောင်းအောင် လုပ်ထားကြတာပါပဲ။

LSP အနှစ်ချုပ်ကတော့ အောက်ပါအတိုင်းပါပဲ။

Methods that use references to the base classes must be able to use the objects of the derived classes without knowing it

OO မှာ program တွေဟာ inheritance ကိုသုံးပြီး class heirarchy တွေဆောက်ကြပါတယ်။ အဲ့ဒီမှာ base class (parent class, super class, super type) ရဲ. reference ကိုသုံးထားတဲ့နေရာမှာ child class (derived class,subtype, sub class) ရဲ. object ကို child class object သည် ဘာ class ဆိုတာမျိုး သိစရာမလိုပဲ သုံးလို.ရ ရမယ်လို.ဆိုပါတယ်။ နည်းနည်းထပ်ရှင်းပါဦးမယ်။ parent reference သုံးထားတဲ့ method တွေမှာ child object တွေကို parent object တွေနေရာမှာ အစားထိုးသုံးလို.ရရမယ်။ အဲ့လိုသုံးလို.ရဖို. ဘယ်လို knowledge မှမလိုဘူး။ (ဒီနေရာ မှာ knowledge ဆိုတာ part1 ကပြထားတဲ့ open close principle example မှာပြထားတဲ့ instanceof operator နဲ.စစ်တာမျိုးကိုပါ)။ ဒါကို LSP Liskov Substitution principle လို.ပြောပါတယ်။ Subsitution ကဘာကိုဆိုလိုချင်တာလဲဆိုတော့ child object တွေသည် parent refernece သုံးတဲ့နေရာမှာ ဘာမှအပြောင်းအလဲမရှိပဲ သုံးနိုင်ရမယ်။ child class က object သည် parent class နေရာမှာ semantically equalivent အနေနဲ့သုံးနိုင်ရမယ်။ code သည် child object ကိုသုံးနေတာလား parent object ကိုသုံးနေတာလားဆိုတာကို သိစရာကိုမလိုဘူး ။ အစားထိုးနိုင်ရမယ် ဒါကိုပြောချင်တာပါ။ နကို parent reference ကိုသုံးထားတဲ့ code ကိုပြုပြင်စရာမလိုပဲ parent reference နေရာမှာ child object ကိုသုံးနိုင်ရမယ်။ ဒါမှသာလျှင် နောက်ထပ် functionality အသစ်ထဲ့တာလိုနေရာမျိုးမှာ မူရင်း code ကိုမထိပဲပြင်လို.ရမှာပါ။ extensible ဖြစ်အောင် ရယ် maintenance ကောင်းအောင်လို.ပြောရင် ရမှာပါ။ LSP သည် inheritance ကို နည်းမှန်လမ်းမှန်သုံးနိုင်အောင်

ပြထားတဲ့ principle လို.ပြောရမှာပါ။ LSP က OCP နဲ.ဆက်စပ်နေပါတယ်။ LSP ကိုချိုးဖောက်ပြီဆိုရင် OCP ကို ချိုးဖောက်သလိုလဲဖြစ်နိုင်ပါတယ်။

Inheritance hierarchy တွေချတဲ့နေရာမှာ child class (subtype) သည် parent class (super type) ရဲ. semantic အတိုင်း လိုက်နာရပါလိမ့်မယ်။ ဆိုချင်တာက child class မှာ override လုပ်ထားတဲ့ parent ရဲ. method တွေသည် paernt ရဲ. nature နဲ. semantically မတူဘူးဆိုရင် ဒါ LSP ကိုချိုးဖောက်တာပါပဲ။ ဒီ https://sanaulla.info/2011/11/28/so... link ကဥပမာလေးကိုပြင်ပြီး ပေးပါရစေ ရှင်းလွယ်လို.ပါ။

```
class Bird {
    public void fly(){}
    public void eat(){}
}

class Crow extends Bird {}

class Penguin extends Bird{
    fly(){
        throw new UnsupportedOperationException(); // violate LSP here
    }
}

public BirdTest{
    public static void main(String[] args){
        List<Bird> birdList = new ArrayList<Bird>();
        birdList.add(new Bird());
        birdList.add(new Crow());
        birdList.add(new Penguin());
        letTheBirdsFly ( birdList );
    }
    static void letTheBirdsFly ( List<Bird> birdList ){
        for ( Bird b : birdList ) {
            b.fly();
        }
    }
}
```

အပေါ်က example မှာ LSP ကိုချိုးဖောက်ထားတာကိုပြထားပါတယ်။ Bird ဆိုတဲ့ class ကနေ Crow ရယ် Penguin ရယ်က extend လုပ်ပါတယ်(subtyping)ပေါ့။ Bird မှာ fly ဆိုတာပါပါတယ်။ Crow ကတော့ ပျံနိုင်ပါတယ် ဒါဆို Bird fly ကိုသုံးရင် သူ. fly ကိုလဲသုံးလို.ရပါတယ်။ Penguin ကတော့မပျံနိုင်ပါဘူး ဒါကြောင့် သူ. method မှာ exception ကို လွှင့်ခိုင်းလိုက်ပါတယ်။ အဓိက Penguin သည် မပျံနိုင်ပါဘူး ဒါကြောင့် Bird ရဲ. fly သူ.မှာမရှိတဲ့အတွက် bird fly ကိုသုံးထားတဲ့ code တွေမှာ penguin သည် အလုပ်လုပ်နိုင်မှာမဟုတ်ပါဘူး။ ဒါဆိုရင် ကျွန်တော်တို.ရဲ. b.fly() code ကိုလိုက်ပြင်နေရတာမျိုးဖြစ်နေပါလိမ့်မယ်။ ဒါက LSP ကိုမလိုက်နာတဲ့အတွက် OCP ကိုပါ ထိသွားတာကိုမြင်နိုင်ပါလိမ့်မယ်။ LSP ကပေးချင်တဲ့ theme ကတော့ child class တွေကို extend လုပ်တဲ့အခါမှာ သူတို. method တွေဟာ parent method တွေနဲ. အစားထိုးသုံးလို.ရ ရပါလိမ့်မယ်ဆိုတာပါပဲ။

Interface Segregation principle (ISP)

ဒီ principle ကတော့ရှင်းပါတယ် မလိုအပ်ပဲနဲ. interface တွေမှာ method တွေ အများကြီးစုပြုံထားတာမလုပ်ရဘူးလို.ပြောချင်တာပါ။ အဲ့လို စုပြုံထားတော့ အဲ့ interface ကို implements လုပ်တဲ့ class တွေဟာ မလိုအပ်ပဲနဲ. တခြား method တွေကိုပါ လိုက်ရေးရတာမျိုးပါ။ အဲ့အစား interface လေးတွေ အများကြီး ထပ်ခွဲပြီး တခုချင်းဆီမှာ ဆိုင်တဲ့ method လေးတွေ ပဲထဲ့သင့်ပါတယ်။ ဒါကလဲ single responsibility (SRP) နဲ.ကိုက်ညီအောင် လုပ်လိုက်တဲ့သဘောပါပဲ။ Fat Interface မလုပ်ပဲနဲ. small interface လေးတွေပဲခွဲခိုင်းတာပါ။ client class တခုဟာ သူမသုံးတဲ့ သူ.အတွက်မလိုအပ်တဲ့ method တွေကို fat interface ကြောင့် လိုက် override လုပ်နေရရင် မကောင်းပါဘူး။

Dependency Inversion Principle (DIP)

သူ.ရဲ. main rule ၂ခုကတော့အောက်ပါအတိုင်းပါပဲ။

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

အပေါ်က ၂ကြောင်းကိုမရှင်းခင်မှာ Abstraction ဆိုတာကိုအရင်ရှင်းရပါလိမ့်မယ်။ Abstraction ဆိုတာ program module သို.မဟုတ် code တွေကို detail implementation ကိုသိစရာမလိုပဲ သုံးနိုင်အောင် အလုပ်လုပ်နိုင်အောင် လုပ်ပေးထားတာကို abstraction လို.ဆိုချင်တာပါ။ All implementation detail ကို client မမြင်ရအောင် abstract လုပ်ထားတာပါ။ အဲ့တော့ implementation ကပြောင်းမယ်ဆိုရင်လဲ client အနေနဲ. အဲ့ဒီ changes ကိုသိစရာမလိုတော့ဘူးပေါ့ဗျာ။

Depend on Abstraction

အဲ့တော့ depend on abstraction ကိုရှင်းရရင် class တွေ code တွေ module တွေဟာ concrete class (interface မဟုတ် abstract class မဟုတ်သော class များ) တွေ သုံးပြီး relation မချိတ်ပဲ မသုံးပဲ relationship အားလုံးကို abstract class သို.မဟုတ် interface ကိုသုံးပြီး model လုပ်တာကိုဆိုချင်တာပါ။ အဲ့တော့ ကျွန်တော်တို.က class တခုက တခြား class တခုကို သုံးဖို.လိုပြီဆိုရင် concrete class အစား abstract class သို.မဟုတ် interface ကိုသုံးတာက ပိုပြီး ကောင်းတယ်လို.ဆိုချင်တာပါ။ ဒါကို depened on abstraction ရဲ.သဘောပေါပဲ။ ဘာလို.လဲဆိုတော့ concrete class ကိုသုံးမယ်ဆိုရင် ကျွန်တော်တို.က သူ. class ရဲ. variable တွေ implementation detail တွေကို accidentally ယူသုံးမိနိုင်ပါတယ်။ interface ,abstract class တွေမှာတော့အဲ့လို မရှိနိုင်ပါဘူး။ နောက်ပြီး interface,abstract class သုံးမယ်ဆိုရင် သူ.တို.ရဲ. dervied class တွေ implementation ပြောင်းလဲ interface method တွေအပေါ်မှာပဲ မူတည်ပြီး သုံးထားတာဖြစ်တဲ့အတွက် code က changes လုပ်စရာမလိုပါဘူး။ဒါက depend on abstraction ရဲ. ကောင်းကျိုးပေါ့ဗျာ။ အဲ့တော့ module တခုနဲ.တခုကြားမှာ inteface တွေသုံးပြီးတော့ပဲ program dependency ကို create လုပ်သင့်တယ်လို.ဆိုလိုတာပါ။ ဥပမာ Java Spring Framework မှာဆို DAO, Service layer တွေမှာ interface ကိုသုံးထားခြင်းဟာ DI principleအတွက် depence upon abstraction ကိုလိုက်နာထားတာပါပဲ။

Dependency

class တခု module တခုက တခြား class တစ်ခုကို reference လုပ်ရပြီ ယူသုံးရပြီဆိုရင် ဒါဟာ dependency ပါပဲ။ အောက်က ဥပမာကိုကြည့်ပါ။

```
class Computer
{
    Mouse mouse;
    public setMouse(Mouse m)
    {
        this.mouse = m;
    }
}

class Mouse
{
    //mouse implementation
}
```

အပေါ်က code example မှာ Computer class ဟာ Mouse class ကို ယူသုံးထားပါတယ် ဒါက dependency ပါ။ ဒါဆိုရင် ဘာအားနည်းချက်ရှိလဲပေါ့ အဲ့ code မှာ Mouse သည် concrete class ဖြစ်ပါတယ် (ဒီနေရာမှာ Java class တွေက extend လုပ်ပြီး method တွေက virtual method ဖြစ်တာကြောင့် မသိသာပါဘူး C++ မှာဆို ရင်တော့သိသာပါပြီ)။ ဒါကြောင့် ကျွန်တော်တို.က တခြား Mouse တခုခုနဲ.အစားထိုးဖို. အဆင်မပြေနိုင်ပါဘူး ဘာလို.လဲဆိုတော့ကျွန်တော်တို.က Mouse class ရဲ. implementation အပေါ်မှာ directly depend ဖြစ်နေလို.ပါ။ ဒါဆိုကျွန်တော်တို.က concrete class Mouse အပေါ်မှာ တိုက်ရိုက်မမှီခိုပဲ အောက်ကလို ရေးလိုက်မယ်ဆိုရင် ဒါက Dependency Inversion ပါပဲ။

```
class Computer
{
    IMouse mouse;
    public setMouse(IMouse m)
    {
        this.mouse = m;
    }
}

interface IMouse
{
    //mouse methods
}
class WirelessMouse implements IMouse
{

}
class USBMouse implements IMouse
{

}
```

အပေါ်က code မှာ ပထမ ဥပမာနဲ့.မတူပဲ Computer class ဟာ concrete mouse class (WirelessMouse,USBMouse) တွေအပေါ်မှာ direct dependency မရှိတော့ပဲ interface IMouse အပေါ်မှာပဲ dependence ဖြစ်သွားပါပြီ။ Concrete class အပေါ်မှာ dependence ဖြစ်နေတာကို interface ကိုပြောင်းပြီး dependency ကို inverse လုပ်လိုက်တာပါ။ အကျိုးဆက်ကတော့ကျွန်တော်တို့.က Computer class က ကြိုက်တဲ့mouse implementation ကိုအစားထိုးသုံးလို.ရတာပါပဲ။ Dependency Inversion နဲ. Dependency Injection ကမတူပါဘူး ဒါပေမဲ့ ဆက်စပ်မှုတော့ရှိပါတယ်။ Dependency Inversion က Dependency တွေကိုရွေ.တာဖြစ်ပြီးတော့ Dependency Injection ကတော့ Dependence Object တွေကို Dependency Injection Container တခုသုံးပြီး system ကနေ dependency တွေကို ထဲ့ပေးတာကိုပြောတာပါ။

Compiler, Interpreter, JIT, AOT, Transpiler

Programming language implementation တွေမှာ အပေါ်က technique တခုခု ဒါမှမဟုတ် တခုထက်ပိုတာကိုသုံးကြပါတယ်။ တခုချင်းဆီမှာ အားသာချက် အားနည်းချက် language အပေါ်မူတည်ပြီး သင့်တော်တာ မသင့်တော်တာတွေရှိကြပါတယ်။ တချို့. langauge တွေမှာ တနည်းထက်ပိုသုံးကြတာလဲရှိပါတယ်။

Compiler

Compiler ဆိုတာကို အတိအကျ meaning ဖွင့်ရရင်တော့ source program (ဥပမာ C/C++/Java) တခုကို ယူပြီးတော့ executable (direct executable မဟုတ်ပဲ ခုနက source program ထက် low level representation လဲဖြစ်နိုင်တယ်) ထုတ်ပေးရင် အဲ့ဒီ ကောင်ကို compiler လို.သုံးပါတယ်။ ဘယ်လို language တွေအတွက်သင့်တော်လဲဆိုတော့ static typed programming language တွေပါ။ ဘာလို. static typed programming language တွေအတွက် compiler ကသင့်တော်သလဲဆိုတော့ runtime မှာ field, တွေ method implementation တွေ operator semantic တွေပြောင်းစရာမရှိလို.ပါ။ အဲ့တော့ တခါတည်း runtime မစခင်ကတည်းက code တွေကို executable အနေနဲ.ပြောင်းလိုက်လို.ရပါတယ်။ Dynamic language တွေလို runtime မှာ changes လိုတဲ့ကောင်တွေဖို.တော့ compiler က သိပ်အဆင်မပြေပါဘူး။ ဥပမာ + ဆိုပါစို. ဒါက static typed language တွေမှာ compile time မှာကတည်းက string concatenation လုပ်ရမှာလား arithmetic addition လုပ်ရမှာလားသိပါတယ်။ အဲ့အတွက် +

အတွက် machine code, bytecode ကိုတခါတည်းထုတ်ထားလို.ရပါတယ်။ Dynamic language တွေဖို.ကျတော့ အဲ့လို တခါတည်းထုတ်ထားလို.မရပါဘူး။ + ရဲ. semantics သည် operand type အပေါ်မူတည်ပြီးပြောင်းမှာပါ။ Operand type ကလဲ dynamic typed ဖြစ်တဲ့အတွက် runtime မှာပြောင်းမှာပါ အဲ့တော့ + အတွက် dynamic language တွေဖို. compiler သုံးထုတ်မယ်ဆိုရင် + ရဲ. operand တွေကိုစစ်၊ သူတို.ရဲ. type အပေါ်မူတည်ပြီး ဘာ operation လုပ်ရမယ်ဆိုတာကို ထုတ်ရပါလိမ့်မယ်။ အဲ့တော့ အလုပ်ရှုပ်ပါတယ်။ ဒါကြောင့် static typed programming langauge အများစုကို compiler သုံးထုတ်ပါတယ်။ Dyanmic typed programming language တွေကို compiler သုံးထုတ်တာရှားပါတယ်။ C/C++/C#/Java စတာတွေကို compiler သုံးပြီး native code, byte code, MSIL အစရှိတာတွေကိုထုတ်ပါတယ်။ C/C++ အတွက်ကတော့ exe native code တန်ထုတ်တော့ရှင်းပါတယ်။ Java/C# ကျတော့ bytecode, MSIL ထုတ်တော့ သူတို.ကိုပြန်ပြီးတော့ VM နဲ. interpretation/JIT လုပ်ရပါတယ်။

Interpreter

ခုနက Compiler တွေသည် static programming language မှာဆို သင့်တော်ပေမဲ့ dynamic programming language တွေမှာလို runtime ရောက်မှ ဘာလုပ်ရမလဲဆိုတော့ decision ချရမဲ့ dynamic language တွေဖို.တော့ အဆင်မပြေပါဘူး။ ဒါကြောင့် dynamic programming language တွေဖို. interpreter ကိုသုံးပါတယ်။ Interpreter လို.ပြောရင်း မူရင်း source program ကနေ တိုက်ရိုက် execute လုပ်တယ်လို.ထင်ကြပါတယ်။ တကယ်က modern programming language တွေမှာ အဲ့လိုမလုပ်ကြပါဘူး။ ဥပမာ Python ဒါမှမဟုတ် PHP source code တွေကို တိုက်ရိုက် execute မလုပ်ပဲ ဆိုင်ရာ bytecode တွေ အဖြင့် တဆင့် compile လုပ်ပါတယ်။ နောက် ရလာတဲ့ bytecode တွေကိုသာ switch based interpreter အနေနဲ. execute လုပ်ကြတာပါ။ ဥပမာ a+b ကိုဆိုရင်

```
LOAD a
LOAD b
ADD
```

ဒီလို byte code ၃ ခုရပါမယ်။ LOAD a ကို execute လုပ်ဖို.ဆိုရင် a ရဲ. တန်ဖိုး ကို memory ပေါ်ကနေ stack ပေါ်ကောက်တင်ရပါမယ်။ ဒီလိုနည်းနဲ. run သွားတာသည် interpreter ပါ။ Interpreter တွေ ကောင်းတာက သူတို.သည် runtime မှာ changes ဖြစ်နိုင်တဲ့ programming language တွေဖို. implement လုပ်ရတာလွယ်ပါတယ်။ အားနည်းတာက interpretation က နှေးပါတယ်။ ဘာလို.နှေးလဲဆိုတော့ ခုနက load ဆိုတာ ဘာလုပ်တယ်ဆိုတာသိရအောင် operator ကို switch statement သုံးပြီး decode လုပ်တယ်။ တကယ်က LOAD ဆိုတာ ကြည့်ရလွယ်အောင် ရေးထားတာ အထဲမှာတော့ 1 တို. 2 တို.လို. နံပါတ်တွေပေါ့။ အဲ့တော့အဆင့်တွေများတယ်။ဒါကြောင့်နောက်ပိုင်း ပိုမြန်အောင် JIT တွေသုံးတယ်။

JIT (Just In Time) Compilation

JIT ဆိုတာ VM, interpreter တွေမှာ interpretation က နှေးလို. bytecode တွေကို runtime မှာ machine code တန်းပြောင်းတဲ့ နည်းဖြစ်ပါတယ်။ ဥပမာ ခုနက LOAD A, အဲ့ကောင်တွေကို machine code အနေနဲ. တခါတည်း runtime မှာပြောင်းပစ်တာပါ။ တခါပြောင်းပြီးရင် သူက direct execute လုပ်လို.ရပြီ ဖြစ်တဲ့အတွက် interpreterထက် ပိုမြန်ပါတယ်။ ဒါပေမဲ့သူ.မှာကလဲ bytecode to machine code ကိုပြောင်းဖို.အတွက် JIT compiler ကိုသုံးရပါတယ်။ အဲ့တော့ အကုန်လုံးကို လိုက်ပြောင်းနေရင် အချိန်ကြာတဲ့အတွက် အသုံးများတဲ့နေရာလောက်ကိုပဲ ကွက်ပြောင်းတာမျိုးလုပ်ပါတယ်။ ဒါမျိုးကို hot spot

compilation လို.ဆိုပါတယ်။ hot spot ဆိုတာ ခနခန run တဲ့ code မျိုးကိုပဲ JIT machine code ပြောင်းပြစ်တာကို ဆိုလိုတာပါ။ JVM, C# CLR မှာဒီကောင်တွေကိုသုံးပါတယ်။

AOT (Ahead of Time)Compilation

ခုနက JIT မှာ runtime ရောက်မှ ပြောင်းရလို. အချိန်ကြာပါတယ်။ အဲ့လိုမဖြစ်ချင်လို. byte code to native code ကို program မ run ခင်ကတည်းက ပြောင်းရင် အဲ့တာကို AOT လို.သုံးပါတယ်။ ဥပမာ Flutter ပါ သူ.မှာ Dart ကနေ ဆိုင်ရာ native code ကို AOT သုံးပြီးပြောင်းပါတယ်။

Transpiler

သူကတော့ language တခုကနေ နောက်သူ့နဲ. abstraction level သိပ်မကွာတဲ့ language ကိုပြောင်းတာပါ။ ဥပမာ TypeScript compiler ကနေပြီးတော့ JavaScript code တွေပြောင်းတာ။ ES6 ကနေ ရိုးရိုး JS code တွေကို babel transpiler ကနေပြောင်းတာ အဲ့တာမျိုးကို transpilation လုပ်တယ်လို.ခေါ်ပါတယ်။