

جزوه‌ی درس

طراحی

سیستم‌های شیء گرا

نسخه ۱

مهندس مهرداد



جزوه‌ی درس

# طراحی سیستم‌های شیء گرا

گروه مهندسی برق و کامپیوتر  
دانشکده فنی - دانشگاه تهران

مدرس:

مهندس یوسف مهرداد

به کوشش:

نعیم اصفهانی

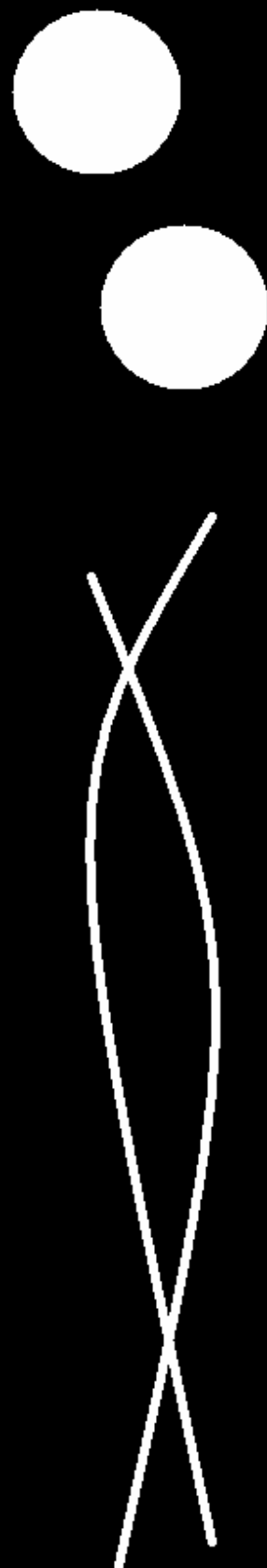
حسین حجت

میثاق باقریان

با تشکر از:

صادق علی‌اکبری

بهار ۱۳۸۴



## فهرست:

۳	۱. شیء گرایبی
۳	Abstraction ۱-۱
۳	Encapsulation ۲-۱
۳	Modularity ۳-۱
۴	Hierarchy ۴-۱
۴	Polymorphism ۵-۱
۵	۲. برنامه سازی به صورت Test-Driven
۹	۳. اصول طراحی کلاسها
۹	۳-۱ اصل تک مسؤلیتی (Single Responsibility Principle - SRP)
۱۳	۳-۱-۲ State Design Pattern
۲۴	۳-۲ اصل باز و بسته (Open-Closed Principle - OCP)
۲۸	۳-۲-۱ شیء ساختگی (Mock Object)
۳۰	۳-۳ اصل جایگزینی لیسکوف (Liskov's Substitution Principle - LSP)
۳۲	۳-۳-۱ Sub-Classing vs. Sub-Typing
۳۲	۳-۳-۲ مفهوم طراحی بر پایه قرارداد (Design by Contract - DbC)
۳۵	۳-۴ اصل وابستگی معکوس (Dependency Inversion Principle - DIP)
۴۰	۳-۵ اصل تفکیک واسطها (Interface Segregation Principle - ISP)
۴۲	۳-۵-۱ Facade Pattern vs. Adapter Pattern
۴۴	۴. رویه‌ی تولید یک سیستم نرم‌افزاری
۴۴	۴-۱ تا رسیدن به مرحله‌ی "طراحی" باید چه کار کرد؟
۴۴	۴-۱-۱ مدل سازی کسب و کار
۴۴	۴-۱-۲ نیازمندی‌ها (Requirements)
۴۵	۴-۱-۲-۱ مدیریت نیازمندی‌ها (Requirements Management)
۴۵	۴-۱-۲-۲ قانون Pareto (80/20) چیست؟
۴۶	۴-۱-۲-۳ ذی‌النفع در یک پروژه چیست؟
۴۶	۴-۱-۲-۴ Vision چیست؟
۴۷	۴-۱-۳ حالا "معماری" ...
۴۷	۴-۲ ترتیب روال تولید سیستم نرم‌افزاری در RUP
۴۷	۴-۲-۱ Disciplineها
۴۸	۴-۲-۲ فازها

## تعریف اصل‌های طراحی کلاس‌ها:

۹	اصل تک‌مسئولیتی
۲۴	اصل باز و بسته
۳۰	اصل جایگزینی لیسکوف
۳۷	اصل وابستگی معکوس
۴۰	اصل تفکیک واسط‌ها

## ۱. شیء گرایی

*"People are like stained glass windows: they sparkle and shine when the sun is out, but when the darkness sets in their true beauty is revealed only if there is a light within."*

Elizabeth Kubler-Ross

مدلهایی که قبل از شیء گرایی وجود داشتند عموماً مبتنی بر مدل‌های ریاضی بودند. در طراحی شیء‌گرا تلاش بر آن بود که طراحی بیشتر به مدل واقعی جهان نزدیک باشد. یک شیء در این مدل تجریدی از یک موجود در جهان واقع است. یک شیء دارای سه مشخصه‌ی زیر است:

- state: خصوصیات و مقادیری که شیء می‌تواند اختیار کند.
- behavior: چگونگی عملکرد یک شیء یا واکنش آن از لحاظ تغییر وضعیت‌هایش<sup>۱</sup> نسبت پیغام‌هایی که با دیگر شیء‌ها رد و بدل می‌کند.
- identity: شناسه‌ای که شیء را از همسان‌ها جدا می‌کند.

ساختار و رفتار اشیاء مشابه در یک کلاس مشترک تعریف می‌شود.

پنج اصل اساسی شیء‌گرایی عبارتند از:

Encapsulation (۲)

Abstraction (۱)

Hierarchy (۴)

Modularity (۳)

Polymorphism (۵)

### ۱-۱ Abstraction

به حذف کردن جزئیات غیر لازم از سیستم و برجسته کردن موارد مهم اصطلاحاً تجرد یا abstraction می‌گویند.

### ۱-۲ Encapsulation

لفاف‌بندی یا encapsulation به کاربر این امکان را می‌دهد که از توانایی‌های شیء مورد نظر استفاده کند بدون آن که نیاز به دانستن نحوه‌ی پیاده‌سازی آن داشته باشد. در لفاف‌بندی، کاربر تنها به آن چیزی دسترسی دارد که به آن نیازمند است. همچنین جدا کردن interface از implementation این مزیت را دارد که در آینده می‌توان به راحتی پیاده‌سازی را تغییر داد.

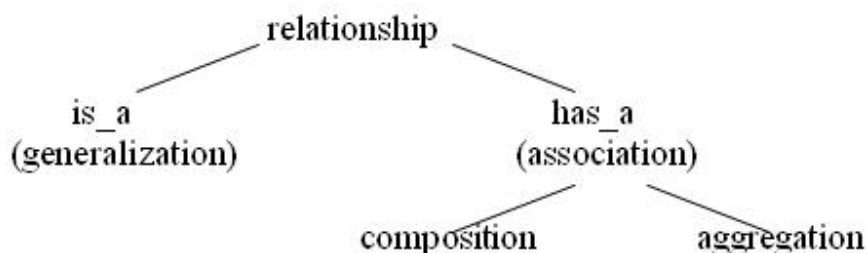
### ۱-۳ Modularity

پیمانانه‌بندی یا modularity فرآیند تقسیم کردن یک سیستم پیچیده به اجزاء کوچک‌تر است که راحت‌تر قابل مدیریت کردن باشند. هدف اصلی در پیمانانه‌بندی مبارزه با پیچیدگی<sup>۲</sup> است.

<sup>۱</sup> State changes

## Hierarchy ۴-۱

سلسله مراتب یا hierarchy طبقه بندی ماژول‌ها در ساختارهایی مشابه درخت است. برای ارتباط برقرار کردن بین ماژول‌ها از روش‌های مختلفی استفاده می‌شود.



تفاوت روابط composition و aggregation در وابستگی زمانی است. هر دو رابطه نشان دهنده‌ی جزء - کل و یا شامل بودن هستند، ولی composition قوی‌تر است. در رابطه‌ی composition این کل است که حتماً بایستی جزء را از بین ببرد. به عنوان مثال، چرخ اتومبیل با خود آن رابطه‌ی aggregation دارد. در این مثال چرخ می‌تواند بدون وجود اتومبیل وجود داشته باشد.

رابطه‌ی شعبه با شرکت رابطه‌ی composition است. یک شعبه بدون وجود شرکت نمی‌تواند وجود داشته باشد، و با از بین رفتن شرکت بایستی شعبه‌های آن نیز از بین بروند.

◀ رابطه‌ی انسان و قلبش از چه نوعی است؟ (aggregation/composition)

بستگی به حوزه‌ی کاربرد دارد؛ اگر در کاربردی قلب بدون انسان معنا نداشته باشد و با از بین رفتن یک نمونه انسان قلبش نیز از بین رود این رابطه از نوع composition می‌باشد. ولی اگر معنا داشته باشد (پیوند قلب) از نوع aggregation است.

## Polymorphism ۵-۱

چندریختی یا polymorphism توانایی پاسخ دادن به پیغام یکسان توسط انجام کارهای متفاوت از هم است. در حالت خاص، می‌توان چندریختی را توانایی تعریف دوباره‌ی متدها در کلاس‌های مشتق شده دانست. برای مثال، با داشتن کلاس Shape، چندریختی به برنامه نویس این امکان را می‌دهد که متدهای مختلف area را برای هر کلاس مشتق شده (مثل دایره، مثلث، مستطیل) بنویسد. صرف نظر از اینکه شیء دقیقاً چه شکلی است، فرستادن پیغام area به شیء متد صحیح را صدا می‌زند و رفتار مورد انتظار انجام داده می‌شود.

## ۲. برنامه سازی به صورت Test-Driven

*"To wash your hands before performing surgery."*  
Anonymous

برنامه سازی مبتنی بر آزمون<sup>۳</sup> از یک چرخه ی شش مرحله ای تشکیل می شود:

۱. مقداری برنامه ی تست بنویسید.
۲. تست ها را کامپایل کرده اشکالات کامپایلری را مشخص کنید.
۳. با اضافه کردن مقادیری کد، اشکالات کامپایلری را برطرف کنید.
۴. تست ها را اجرا کنید، تا اشکالات زمان اجرا مشخص شوند.
۵. با تغییر دادن کد، اشکالات زمان اجرا را رفع کنید.
۶. کد نوشته شده را بازبینی کنید، و در آن از قوانین refactoring استفاده کنید.

مثال - می خواهیم یک کلاس Person بنویسیم. این کلاس حاوی اسم و نام خانوادگی شخص است. در نوشتن کلاس Person از برنامه سازی مبتنی بر آزمون استفاده خواهیم کرد.  
قدم اول: برای کلاس Person مقداری برنامه ی تست می نویسیم. این برنامه در کالبد JUNIT<sup>۴</sup> نوشته شده است.

```
import junit.framework.TestCase;

public class TestPerson extends TestCase
{
    public void testConstructor()
    {
        Person P = new Person( "ali", "ahmadi");
        assertEquals( "ali", P.getFName());
        assertEquals( "ahmadi", P.getLName());
    }
}

import junit.framework.TestSuite;
import junit.framework.Test;

public class AllTests
{
    public static void main( String args[])
    {
        junit.textui.TestRunner.run( AllTests.suite());
    }
    public static Test suite()
    {
        TestSuite suite = new TestSuite("mytest");
        suite.addTestSuite( TestPerson.class);
        return suite;
    }
}
```

<sup>3</sup> test-driven

<sup>4</sup> <http://www.junit.org>



قدم دوم: تست را کامپایل می کنیم تا اشکالات کامپایلری مشخص شوند. در کد فوق به دلیل وجود نداشتن کلاس Person، کامپایلر اشکال می گیرد.  
قدم سوم: برای رفع نمودن اشکال، کلاس Person را ایجاد می کنیم.

```
public class Person
{
    public Person( String name, String family) {}
    public String getFName()
    {
        return null;
    }
    public String getLName()
    {
        return null;
    }
}
```

قدم چهارم: با رفع شدن اشکالات مربوط به کامپایلر، تست ها را اجرا می کنیم. در این مرحله ادعاهای (assertions) نوشته شده در تست رد می شوند.

قدم پنجم: برای پذیرفته شدن ادعاهای موجود در تست، کد کلاس Person را کامل می کنیم.

```
public class Person
{
    String name, family;
    public Person( String name, String family)
    {
        setFName( name);
        setLName( family);
    }
    public void setFName( String name)
    {
        this.name = name;
    }
    public void setLName( String family)
    {
        this.family = family;
    }
    public String getFName()
    {
        return name;
    }
    public String getLName()
    {
        return family;
    }
}
```

قدم ششم: کد را refactor می کنیم. همان طور که می دانید refactoring روشی منظم برای سازمان دادن بدنه ی برنامه است. در تمامی روش های refactor ساختار داخلی کد تغییر می کند اما رفتار بیرونی آن بدون تغییر باقی می ماند. بعد از انجام refactor باید تمام Test-case های قبلی روی کد جدید با موفقیت اعمال شوند.

در اینجا یکی از روش های معمول refactoring را برای کد فوق بررسی می کنیم. این روش استخراج متد extract method نام دارد. استخراج متد روشی برای ساختن روایی جدید از یک قطعه کد موجود است. برای این کار قسمتی از برنامه را انتخاب کرده آن را در یک روال جدید قرار می دهیم. سپس به جای قسمت انتخاب شده روال جدید را صدا می زنیم. برای مثال، فرض کنید که کلاس فوق یک تابع برای بازگرداندن میزان بدهکاری شخص دارد.

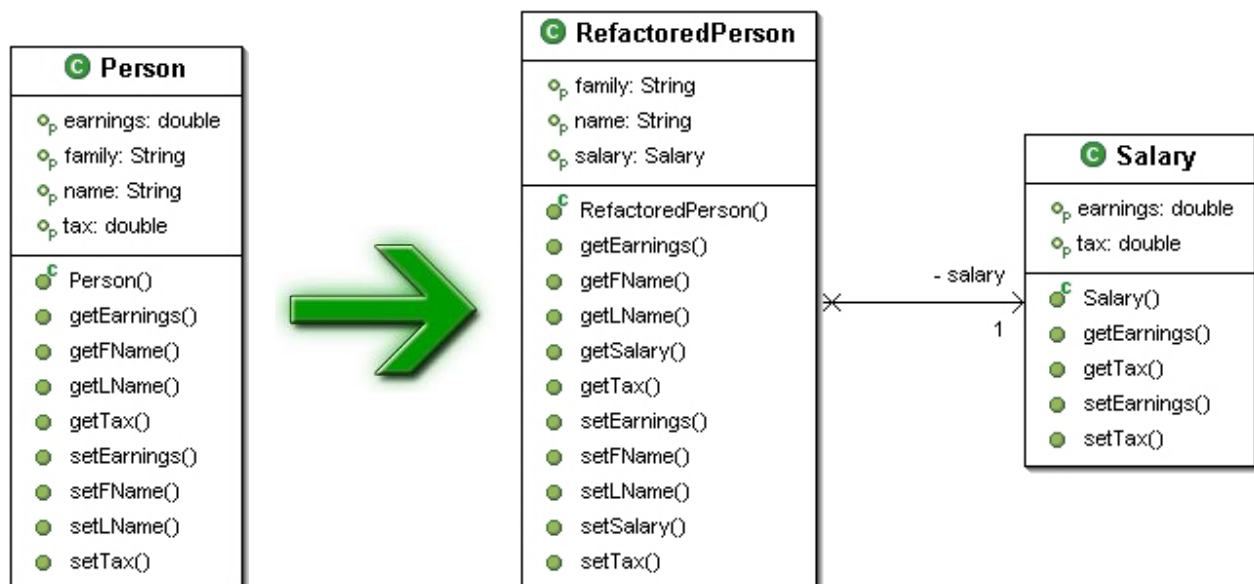
```
void printOwing()
{
    printBanner();

    //print details
    System.out.println ("name: " + name);
    System.out.println ("amount " + getOutstanding());
}
```

این متد را به صورت زیر refactor می کنیم:

```
void printOwing()
{
    printBanner();
    printDetails(getOutstanding());
}
void printDetails (double outstanding)
{
    System.out.println ("name: " + name);
    System.out.println ("amount " + outstanding);
}
```

در ادامه سعی می کنیم که به کلاس نوشته شده در بالا، حقوق پایه و مالیات را اضافه کنیم. یک روش برای این کار، اضافه کردن فیلدهایی در کلاس Person برای حقوق و مالیات است. کلاس Person جدید به نوبه خود قابل refactor شدن است. بدین منظور از روش استخراج کلاس (extract class)، استفاده می کنیم. در استخراج کلاس، از کلاس موجود یک کلاس جدید ساخته می شود و متدها و فیلدهای مرتبط از کلاس قدیمی به کلاس جدید انتقال داده می - شوند. به شکل زیر توجه کنید.



دقت کنید که در انجام refactoring واسط کلاس تغییری نکرده است.

این طراحی در لفاف بندی ممکن است دچار اشکال شود. برای مثال، کاربری به صورت زیر حقوق یک شخص را به دست می آورد:

```
RefactoredPerson Ali = new RefactoredPerson( "Ali", "Akhavan");  
Salary aliSalary = Ali.getSalary();
```

کاربر می تواند بدون اطلاع Person مقدار حقوق شخص را تغییر دهد.

```
aliSalary.setEarnings( 1000000 );
```

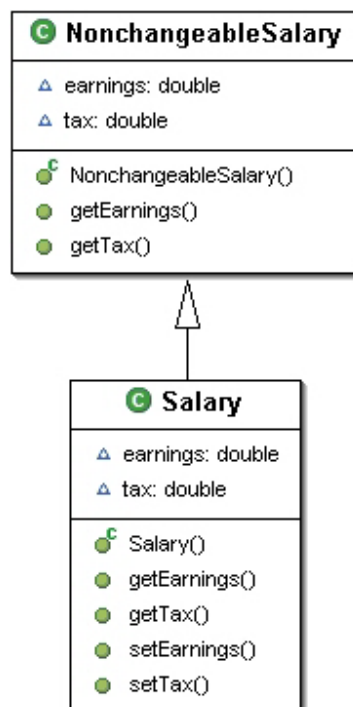
به عبارت دیگر، کاربر نبایستی حق تغییر Salary و خراب کردن مقادیر آنرا داشته باشد. برای حل این مشکل روش های مختلفی موجود هستند، که در زیر به چند تا از آنها اشاره می شود:  
روش اول: یک کپی از Salary به کاربر داده می شود تا تغییرات کاربر روی حقوق شخص تأثیری نگذارد.

روش دوم: از object cloning استفاده می کنیم.

```
public class Salary implements Cloneable
```

هر چند که این روش جلوی تغییرات ناخواسته را می گیرد اما برای کاربر دچار ابهام می کند چون هر چقدر به حقوق دست می زند تغییرات او منعکس نمی شوند.

روش سوم: استفاده از یک پدر که فقط دارای getter باشد.



روش چهارم: به جای قرار دادن یک پدر غیر قابل تغییر، از یک واسطه<sup>5</sup> غیر قابل تغییر استفاده می کنیم.

<sup>5</sup> interface

## ۳. اصول طراحی کلاس‌ها

### ۱-۳ اصل تک‌مسئولیتی (Single Responsibility Principle - SRP)

المسؤول حر حتى يعد ، ومسترق المسؤول حتى ينجز.  
امام حسن(ع)

#### تعریف :

"هرگز نباید بیش از یک دلیل برای تغییر یک کلاس وجود داشته باشد."  
به این دلیل به این اصل تک‌مسئولیتی می‌گویند که هر مسئولیت پایه و اساس یک تغییر می‌باشد.

**تذکر:** همان‌طور که اشاره شد به هر عامل تغییر در یک کلاس مسئولیت می‌گویند ولی توجه داشته باشید که بحث هم‌زمانی رخ داده‌ها در اینجا بسیار مهم است. یعنی اگر دو یا چند عامل توأم در یک کلاس اتفاق بیفتند می‌توان آن‌ها را یک مسئولیت برای کلاس در نظر گرفت و هیچ مغایرتی با اصل فوق ندارد.

**مثال:** واسط مودم<sup>۶</sup> زیر را در نظر بگیرید. به نظر می‌رسد که طراحی این واسط هیچ مشکلی ندارد و قابل قبول است، زیرا هر ۴ متد تعریف شده منطقی‌متعلق به یک مودم می‌باشند.

ولی در این واسط برای یک مودم ۲ نوع مسئولیت وجود دارد:

```
public interface Modem {  
    public void dial(String pno);  
    public void hangup();  
    public void send(char c);  
    public char recv();  
}
```

۱. برقراری ارتباط :

dial ○

hangup ○

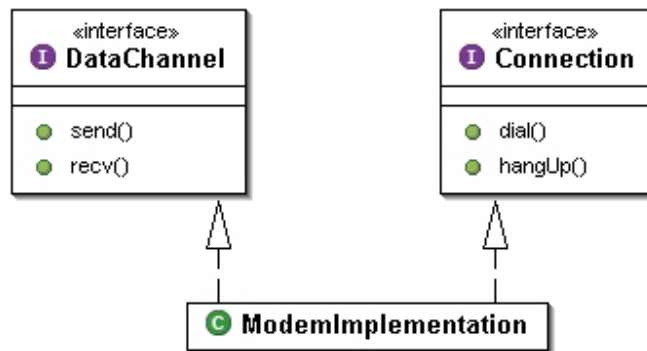
۲. تبادل داده :

send ○

recv ○

بنابراین طبق اصل SRP این طراحی باید همانند نمودار UML زیر تصحیح شود. چون همان‌طور که در بالا مشاهده می‌کنید این کلاس منطقی ۲ مسئولیت دارد که یکی مربوط به برقراری ارتباط و دیگری مربوط به تبادل داده می‌باشد که لزوماً نیز هم‌زمان نمی‌باشند.

<sup>6</sup> Interface Modem



**مثال:** فرض کنید که تعدادی شخص (از نوع کلاس Person) در پایگاه داده‌ای ذخیره شده اند و می‌خواهیم آن‌ها را بر حسب تعداد فرزندان با دو رنگ متفاوت نمایش دهیم به قسمی که آن‌هایی که بیش از دو فرزند دارند با یک رنگ و بقیه با رنگ دیگر نمایش داده شوند. جای این دو کار در کجای معماری سه لایه قرار دارد؟

**پاسخ:** با استفاده از تکنیکی موسوم به Duplication به سؤال فوق پاسخ می‌دهیم. در این تکنیک ابتدا باید جواب این پرسش را بدانیم که اگر مکانیزم یکی از این کارها تغییر کند آیا لزوماً دیگری نیز باید تغییر می‌کند؟ مسلماً در این مثال این‌گونه نیست. زیرا مثلاً اگر واسط کاربری بخواهد از swing به SWT تغییر کند و یا اصلاً بخواهیم به سیستم‌عامل وابسته شویم و از خود Windows استفاده کنیم، لزومی ندارد که شرط رنگ‌آمیزی دچار تغییر شود. پس این کارها دو مسؤولیت جداگانه بوده و باید در دو لایه متفاوت قرار بگیرند. مثلاً شرط رنگ‌آمیزی باید در لایه‌ی Business Logic و نمایش لیست در لایه‌ی Presentation قرار بگیرد.

**مثال:** در سیستم انبار و حسابداری نیز مسؤولیت ارتباط حسابداری با انبار یک مسؤولیت جداست که باید کلاس دیگری نقش این ارتباط را به عنوان یک Gateway بازی کند.



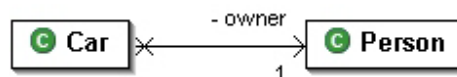
**تذکر مهم:** جداکردن بیش از حد مسؤولیت‌ها و وسواس شدن در این قضیه به هیچ عنوان درست نیست و نباید دچار افراط شد.

Keep It Simple Stupid (KISS)

## Null Object Design Pattern ۱-۱-۳

**مثال:** کلاس‌های Car و Person را در طراحی زیر در نظر بگیرید. در این طراحی فرض شده است که هر ماشینی حداکثر یک شخص به عنوان صاحب ماشین دارد که نمونه‌ای از همان کلاس Person می‌باشد. کد مربوط به پیاده‌سازی کلاس Car در زیر آمده است.

```
public class Car {
    private Person owner;
    //...
    public Person getOwner() {
        return owner;
    }
    //...
}
```



حال فرض کنید می‌خواهیم به شکل زیر از این کلاس Car استفاده نماییم:

```
private Car car;
//...
if (car.getOwner() == null)
    ownerFirstName = "";
else
    ownerFirstName = car.getOwner().getFirstName();
//...
if (car.getOwner() == null)
    ownerLastName = "";
else
    ownerLastName = car.getOwner().getLastName();
//...
```

**نکته:** هر جا دیدید که زیاد از if-then-else استفاده می‌کنید، می‌تواند نشانه این باشد که به نوعی شیء‌گرایی را نقض می‌کنید.

**راه‌حل:** برای حل این مشکل به الگویی اشاره می‌کنیم با نام "شیء تهی" که بسیار خوب است که در کدهایمان در چنین شرایطی از این الگو تبعیت کنیم.

ابتدا راه حل مبتنی بر این الگو را مشاهده می‌کنیم، سپس تعریف آکادمیک الگوی شیء تهی را ارائه می‌کنیم:

### Real Object

```
public class Person {
    private String firstName;
    private String lastName;
    //...
    public boolean isNull() {
        return false;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    //...
}
```

### Null Object

```
public class NullPerson extends Person {
    private String firstName;
    private String lastName;
    //...
    public boolean isNull() {
        return true;
    }
    public String getFirstName() {
        return "";
    }
    public String getLastName() {
        return "";
    }
    //...
}
```

<sup>7</sup> Null Object Design Pattern

**نکته ۱:** از آن جایی که فقط یک NullPerson وجود خواهد داشت می توان این کلاس را Singleton نمود.

**نکته ۲:** مشکلی که وجود دارد این است که ممکن است چند برخورد مختلف با تهی (null) بودن کلاس های مختلف داشته باشیم. در این صورت کافی است یک واسط تهی<sup>۸</sup> تعریف کنیم که هر کلاسی که بخواهد تهی باشد کافی است این واسط را پیاده سازی نماید. به عنوان مثال در این صورت پیاده سازی NullPerson این گونه خواهد بود:

```
public class NullPerson extends Person implements Null {  
    //...  
}
```

حال می خواهیم ببینیم در صورت استفاده از این الگو کلاس Car چه تغییری می کند؟ آیا بالاخره از شر if-then-else ها نجات پیدا کردیم یا نه؟

```
public class Car {  
    private Person owner = new NullPerson();  
    //...  
    public Person getOwner() {  
        return owner;  
    }  
}
```

**تعریف:** شیء تهی، شیء ایست که تمامی وظایف شیء حقیقی<sup>۹</sup> را بصورت "هیچ"<sup>۱۰</sup> پیاده سازی می کند.

◀ چه وقت باید از الگوی شیء تهی استفاده نمود؟

۱. استفاده کننده تفاوت بین شیئی که کاری انجام می دهد و شیئی که کاری انجام نمی دهد را نخواهد در نظر بگیرد.

۲. وقتی که استفاده کننده بخواهد از رفتار "هیچ" استفاده مجدد نماید.

۳. اشیائی که با یک شیء مرتبط هستند، امکان تهی<sup>۱۱</sup> بودن داشته باشند.

◀ مزایا؟

۱. افزایش قابلیت استفاده مجدد<sup>۱۲</sup>

۲. افزایش قابلیت چندریختی<sup>۱۳</sup>

۳. لفاف بندی رفتار هیچ<sup>۱۴</sup>

۴. ساده تر شدن متن برنامه های استفاده کنندگان

◀ معایب؟

۱. اضافه شدن تعدادی کلاس اضافه بر سازمان که در صورت افراطی شدن در این قضیه (استفاده از الگوی شیء تهی)، باعث افزایش پیچیدگی می شود.

۲. تعدد احتمالی اشیاء تهی در ارتباط با یک کلاس خاص که پیچیدگی را دوچندان خواهد نمود.

<sup>8</sup> Null Interface

<sup>9</sup> Real Object

<sup>10</sup> Do Nothing

<sup>11</sup> null

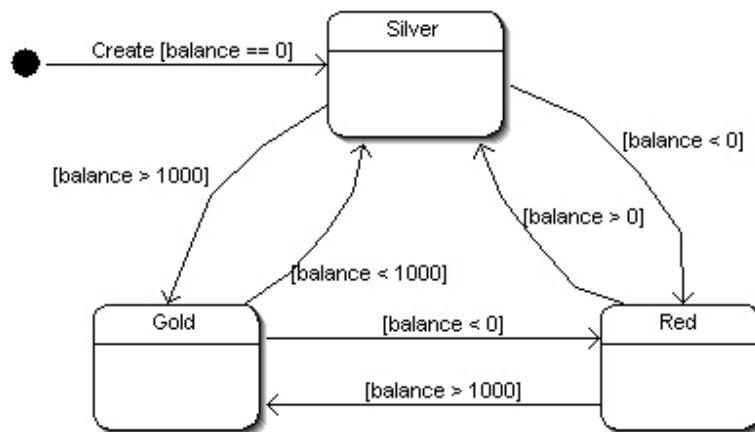
<sup>12</sup> Reusability

<sup>13</sup> Polymorphism

<sup>14</sup> Encapsulation of do nothing behavior

**مثال:** فرض کنید در یک بانک حساب جاری اشخاص منطقیاً بتوانند در یکی از سه وضعیت قرمز، نقره‌ای و طلایی قرار بگیرد. ابتدای کار که حساب افتتاح می‌شود در وضعیت نقره‌ای قرار دارد و اعتبار آن صفر است. حال اگر میزان اعتبار این حساب به ۱۰۰۰ دلار برسد و یا از آن بالاتر رود حساب وارد وضعیت طلایی شده و از این پس اگر صاحب حساب فرضاً بخواهد  $n$  دلار به اعتبار حسابش اضافه کند به عنوان پاداش ۱،۰۵ برابر  $n$  دلار به اعتبار حسابش افزوده می‌شود که به این ۰،۰۵ سود (interest) حساب‌های طلایی گفته می‌شود. از آن سو اگر صاحب حساب آن قدر از حسابش برداشت کند که اعتبارش از صفر کمتر شود، آن-گاه حساب وارد وضعیت قرمز شده و از این پس اگر مجدداً صاحب حساب بخواهد از حسابش برداشت کند نه تنها مبلغی برگردانده نمی‌شود بلکه مبلغی معدل ۱۵ دلار نیز به عنوان حق ارائه سرویس (Service Fee) به میزان بدهکاریش افزوده می‌شود.

◀ می‌خواهیم وضعیت متغیر این حساب‌ها را طراحی و پیاده‌سازی کنیم.



**راه حل الف:** در این راه حل ساده و ابتدایی یک سری ثابت<sup>۱۵</sup> را به همراه یک پرچم<sup>۱۶</sup> که مشخص کننده وضعیت کنونی یک حساب می‌باشد را در کلاسی موسوم به Account در نظر می‌گیریم. برای نوشتن کلاس Account ابتدا تست آن را باید نوشت:

```

public class TestAccount{
    //...
    public TestAccount(){
        Account acc = new Account();
        acc.deposit(10);
        acc.withdraw(50);
        //...
    }
    //...
}
  
```

حال خود کلاس Account را در طراحی الف خواهیم دید:

<sup>15</sup> Constant

<sup>16</sup> Flag



```


public class Account {

    final static String RED_STATE = "RED-STATE";
    final static String SILVER_STATE = "SILVER-STATE";
    final static String GOLD_STATE = "GOLD-STATE";


    private double serviceFee;
    private double interest;

    private String currentState;
    private double balance;

    public Account() {
        setBalance(0);
        currentState = SILVER_STATE;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
    //...
    public void deposit (double amount){
        if (currentState.equals(RED_STATE))
            //...
        else if (currentState.equals(SILVER_STATE))
            //...
        else if (currentState.equals(GOLD_STATE))
            //...

        
        if ( -100 < balance && balance < 0 )
            currentState = RED_STATE;
        else if ( 0 <= balance && balance < 1000)
            currentState = SILVER_STATE;
        else
            currentState = GOLD_STATE;
    }

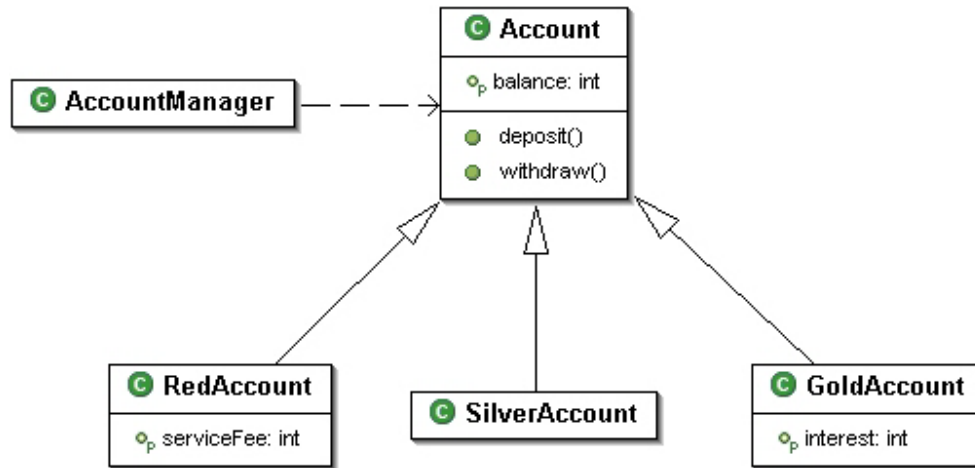
    public void withdraw (double amount){
        if (currentState.equals(RED_STATE))
            //...
        else if (currentState.equals(SILVER_STATE))
            //...
        else if (currentState.equals(GOLD_STATE))
            //...

        
        if ( -100 < balance && balance < 0 )
            currentState = RED_STATE;
        else if ( 0 <= balance && balance < 1000)
            currentState = SILVER_STATE;
        else
            currentState = GOLD_STATE;
    }
    //...
}

```

همان طور که اشاره شد هر جا از if-then-else متوالی استفاده شده به احتمال زیاد یکی از اصول طراحی کلاسها در حال نقض شدن می باشد. حتی اگر نخواهیم به ترتیبی این شرطها را حذف کنیم باید قسمت های مشخص شده در انتهای متدهای deposit و withdraw را با تکنیک Extract Method به یک متد private مانند stateChangeCheck() تبدیل کنیم.

راه حل ب: نمودار UML این راه حل در شکل زیر آمده است. این نمودار بسیار شبیه نمودار راه حل ج است. برای دریافتن تفاوت اصلی راه حل های ب و ج توصیه می شود که حتماً کد پیاده سازی طراحی های ب و ج را به دقت مطالعه کنید. مقایسه این دو طراحی فهم مناسبی از الگوی طراحی حالت<sup>17</sup> را نتیجه خواهد داد.



```

public abstract class Account {

    // Fields:
    protected AccountManager accountManager;
    protected double lowerLimit;
    protected double upperLimit;

    // Setters and Getters
    public double getLowerLimit() {
        return lowerLimit;
    }
    public void setLowerLimit(double lowerLimit) {
        this.lowerLimit = lowerLimit;
    }
    public double getUpperLimit() {
        return upperLimit;
    }
    public void setUpperLimit(double upperLimit) {
        this.upperLimit = upperLimit;
    }

    // Abstract Methods:
    public abstract void Initialize();
}

```

```

public class RedAccount extends Account {

    // Fields:
    private double serviceFee;

    // Constructors:
    public RedAccount(AccountManager accountManager) {
        this.accountManager = accountManager;
        Initialize();
    }
}

```

<sup>17</sup> State Design Pattern

```

    }

    // Setters and Getters:
    public double getServiceFee() {
        return serviceFee;
    }
    public void setServiceFee(double serviceFee) {
        this.serviceFee = serviceFee;
    }

    // Methods:
    public void Initialize() {
        // These are some typical initializations
        lowerLimit = -100.0;
        upperLimit = 0.0;
        serviceFee = 15.00;
    }
    public String toString(){
        return "Red Account";
    }
}



---


public class SilverAccount extends Account {

    // Constructors:
    public SilverAccount(AccountManager accountManager) {
        this.accountManager = accountManager;
        Initialize();
    }

    // Methods:
    public void Initialize() {
        // These are some typical initializations
        lowerLimit = 0.0;
        upperLimit = 1000.0;
    }
    public String toString(){
        return "Silver Account";
    }
}



---


public class GoldAccount extends Account {

    // Fields:
    private double interest;

    // Constructors:
    public GoldAccount(AccountManager accountManager) {
        this.accountManager = accountManager;
        Initialize();
    }

    // Setters and Getters:
    public double getInterest() {
        return interest;
    }
    public void setInterest(double interest) {
        this.interest = interest;
    }

    // Methods:
    public void Initialize() {

```

```

        // These are some typical initializations
        interest = 0.05;
        lowerLimit = 1000.0;
        upperLimit = 10000000.0;
    }
    public String toString(){
        return "Gold Account";
    }
}

public class AccountManager {

    // Fields:
    private String owner;
    private double balance;
    private Account currentAccount;

    // Constructors
    public AccountManager( String owner ) {
        // New accounts are 'Silver' by default
        this.owner = owner;
        balance = 0.0;
        currentAccount = new SilverAccount(this);
    }

    // Setters and Getters:
    public String getOwner() {
        return owner;
    }
    public Account getCurrentAccount() {
        return currentAccount;
    }
    public void setCurrentAccount(Account currentAccount) {
        this.currentAccount = currentAccount;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }

    // Methods:
    public void Deposit(double amount) {
        if (currentAccount instanceof RedAccount){
            balance += amount;
            if (balance > currentAccount.getUpperLimit())
                currentAccount = new SilverAccount(this);
        }else if (currentAccount instanceof SilverAccount){
            balance += amount;
            if (balance > currentAccount.getUpperLimit())
                currentAccount = new GoldAccount(this);
        }else if (currentAccount instanceof GoldAccount)
            balance += ((GoldAccount)currentAccount).getInterest() * amount;

        System.out.println(" Deposited $ " + amount);
        System.out.println(" Balance = $ " + balance);
        System.out.println(" Status = " + currentAccount);
    }
    public void Withdraw(double amount) {
        if (currentAccount instanceof RedAccount){
            balance -= ((RedAccount)currentAccount).getServiceFee();

```

```

        System.out.println(" * No funds available to withdraw!");
    }
    else if (currentAccount instanceof SilverAccount){
        balance -= amount;
        if (balance < currentAccount.getLowerLimit())
            currentAccount = new RedAccount(this);
    }
    else if (currentAccount instanceof GoldAccount){
        balance -= amount;
        if (balance < 0.0)
            currentAccount = new RedAccount(this);
        else if (balance < currentAccount.getLowerLimit())
            currentAccount = new SilverAccount(this);
    }
    System.out.println(" Withdrew $ " + amount);
    System.out.println(" Balance = $ " + balance);
    System.out.println(" Status = " + currentAccount);
}
public String toString(){
    return owner + "'s Account Manager";
}
}

public class Main {
    public static void main(String[] args) {
        // Open a new account
        AccountManager accountManager = new AccountManager( "Misagh Bagherian" );

        // Apply financial transactions
        accountManager.Deposit( 500.0 );
        accountManager.Deposit( 300.0 );
        accountManager.Deposit( 550.0 );
        accountManager.Withdraw( 2000.00 );
        accountManager.Withdraw( 1100.00 );
    }
}

```

---

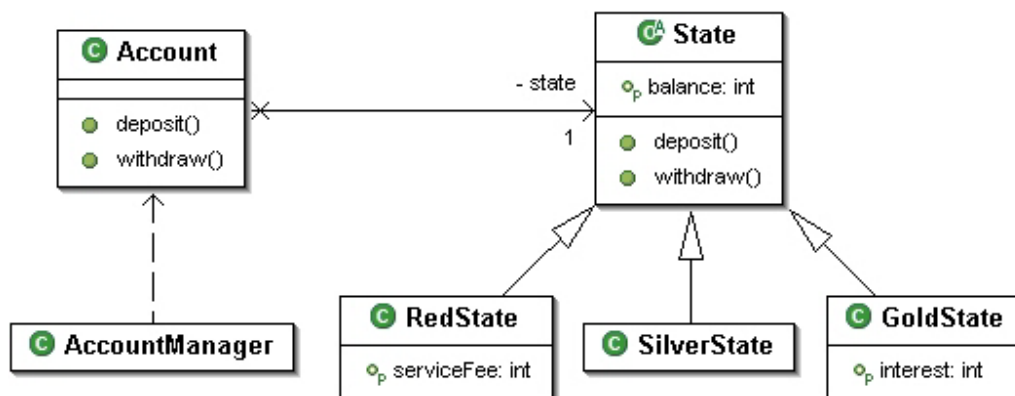
**Standard Output:**

```

Deposited $ 500.0
Balance = $ 500.0
Status = Silver Account
Deposited $ 300.0
Balance = $ 800.0
Status = Silver Account
Deposited $ 550.0
Balance = $ 1350.0
Status = Gold Account
Withdrew $ 2000.0
Balance = $ -650.0
Status = Red Account
* No funds available to withdraw!
Withdrew $ 1100.0
Balance = $ -665.0
Status = Red Account

```

این راه حل مبتنی است بر الگوی حالت. نمودار UML و کد پیاده‌سازی این طراحی در زیر آمده است:



```

public abstract class State {

    // Fields:
    protected Account account;
    protected double balance;
    protected double lowerLimit;
    protected double upperLimit;

    // Setters and Getters:
    public Account getAccount() {
        return account;
    }
    public void setAccount(Account account) {
        this.account = account;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }

    // Abstract Methods:
    public abstract void Initialize();
    public abstract void Deposit( double amount );
    public abstract void Withdraw( double amount );
    public abstract void StateChangeCheck();
}
  
```

```

public class RedState extends State {

    //Fields:
    private double serviceFee;

    // Constructors:
    public RedState(
        double balance, Account account ) {
        this.balance = balance;
        this.account = account;
        Initialize();
    }
    public RedState(State state) {
  
```

```

        this.balance = state.getBalance();
        this.account = state.getAccount();
        Initialize();
    }

    //Methods:
    public void Initialize() {
        // These are some typical initializations
        lowerLimit = -100.0;
        upperLimit = 0.0;
        serviceFee = 15.00;
    }
    public void Deposit(double amount) {
        balance += amount;
        StateChangeCheck();
    }
    public void Withdraw(double amount) {
        balance -= serviceFee;
        System.out.println(" * No funds available to withdraw!");
    }
    public void StateChangeCheck() {
        if( balance > upperLimit )
            account.setState(new SilverState( this ));
    }
    public String toString() {
        return " Red State ";
    }
}

```

---

```

public class SilverState extends State {

    // Constructors
    public SilverState(
        double balance, Account account ) {
        this.balance = balance;
        this.account = account;
        Initialize();
    }
    public SilverState( State state ) {
        this.balance = state.getBalance();
        this.account = state.getAccount();
        Initialize();
    }

    // Methods:
    public void Initialize() {
        // These are some typical initializations
        lowerLimit = 0.0;
        upperLimit = 1000.0;
    }
    public void Deposit(double amount) {
        balance += amount;
        StateChangeCheck();
    }
    public void Withdraw(double amount) {
        balance -= amount;
        StateChangeCheck();
    }
    public void StateChangeCheck() {
        if( balance < lowerLimit )
            account.setState(new RedState( this ));
        else if( balance > upperLimit )

```

```

        account.setState(new GoldState( this ));
    }
    public String toString() {
        return " Silver State ";
    }
}

```

---

```

public class GoldState extends State {

    // Fields:
    private double interest;

    // Constructors
    public GoldState(
        double balance, Account account ) {
        this.balance = balance;
        this.account = account;
        Initialize();
    }
    public GoldState( State state ) {
        this.balance = state.getBalance();
        this.account = state.getAccount();
        Initialize();
    }

    //Methods:
    public void Initialize() {
        // These are some typical initializations
        interest = 0.05;
        lowerLimit = 1000.0;
        upperLimit = 10000000.0;
    }
    public void Deposit(double amount) {
        balance += interest * amount;
        StateChangeCheck();
    }
    public void Withdraw(double amount) {
        balance -= amount;
        StateChangeCheck();
    }
    public void StateChangeCheck() {
        if( balance < 0.0 )
            account.setState(new RedState( this ));
        else if( balance < lowerLimit )
            account.setState( new SilverState( this ));
    }
    public String toString() {
        return " Gold State ";
    }
}

```

---

```

public class Account {

    // Fields
    private State state;
    private String owner;

    // Constructors
    public Account( String owner ) {
        // New accounts are 'Silver' by default
        this.owner = owner;
        state = new SilverState( 0.0, this );
    }
}

```



```

    }

    // Getters and Setters
    public double getBalance() {
        return state.getBalance();
    }
    public State getState() {
        return state;
    }
    public void setState(State state) {
        this.state = state;
    }
    public String getOwner() {
        return owner;
    }

    // Methods
    public void Deposit( double amount ) {
        state.Deposit( amount );
        System.out.println(" Deposited $ " + amount);
        System.out.println(" Balance = $ " + state.getBalance());
        System.out.println(" Status = " + state);
    }
    public void Withdraw( double amount ) {
        state.Withdraw( amount );
        System.out.println(" Withdrew $ " + amount);
        System.out.println(" Balance = $ " + state.getBalance());
        System.out.println(" Status = " + state);
    }
    public String toString(){
        return owner + "'s Account";
    }
}



---


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

        // Open a new account
        Account account = new Account( "Misagh Bagherian" );

        // Apply financial transactions
        account.Deposit( 500.0 );
        account.Deposit( 300.0 );
        account.Deposit( 550.0 );
        account.Withdraw( 2000.00 );
        account.Withdraw( 1100.00 );
    }
}

```

---

**Standard Output** (Compare with that of part B)

```

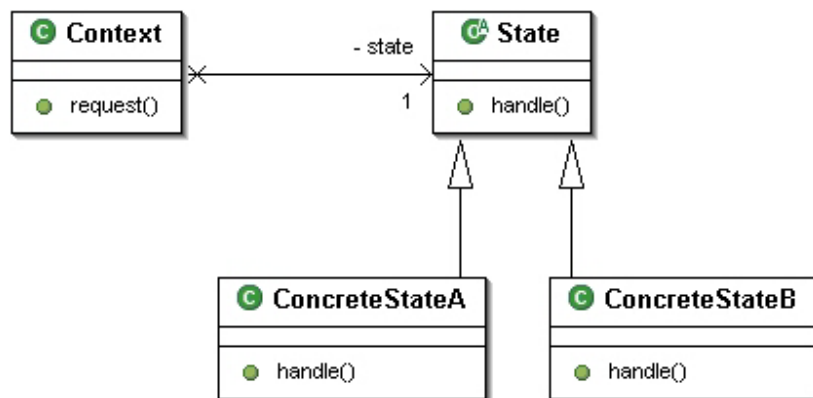
Deposited $ 500.0
Balance = $ 500.0
Status = Silver State
Deposited $ 300.0
Balance = $ 800.0
Status = Silver State
Deposited $ 550.0
Balance = $ 1350.0
Status = Gold State
Withdrew $ 2000.0
Balance = $ -650.0

```

```
Status = Red State
* No funds available to withdraw!
Withdrew $ 1100.0
Balance = $ -665.0
Status = Red State
```

حتماً تفاوت راه‌حل‌های ب و ج را متوجه شده‌اید. در راه‌حل ب تمام منطق حالات و انتقال از یک حالت به حالت دیگر در AccountManager آمده است در حالی که در راه‌حل ج که مبتنی بر الگوی حالت است این منطق در خود کلاس‌های مربوط به حالات آمده است، یعنی RedState، SilverState و GoldState. در راه‌حل‌های مبتنی بر الگوی حالت دانش مورد نیاز برای هر تغییر حالت را در داخل خود آن کلاس حالت قرار می‌دهیم نه در کلاس حالت دیگر و یا کلاسی دیگر مانند AccountManager مثال فوق.

نتیجه: نمودار UML در حالت کلی برای الگوی حالت به این گونه است. در ادامه نمودار UML یک مثال دیگر نیز آمده است.



## ۲-۳ اصل باز و بسته (Open Closed Principle - OCP)

*"You must be the change you wish to see in the world."*  
Mahatma Gandhi

هدف ساختن نرم‌افزار با عمر بیشتر است که طبق تعریف آقای جیکابسون دارای قابلیت تحمل تغییر بیشتر و توانایی توسعه باشد.

تعریف:

"المان‌های نرم‌افزاری (کلاس‌ها، ماژول‌ها، توابع و...) باید برای توسعه<sup>۱۸</sup> باز و نسبت به تغییر<sup>۱۹</sup> بسته باشند."

در تعریف فوق منظور از باز بودن این است که بتوان به راحتی نرم‌افزار را توسعه داد. منظور از بسته بودن این است که در روند توسعه کدهای نوشته شده‌ی قبلی تغییر نکنند.

**مثال:** فرض کنید که یک برنامه کاربردی قرار است که یک لیست از دایره‌ها با شعاع‌های مختلف را دریافت کرده و آن‌ها را روی صفحه نمایش رسم نماید. در زیر ۲ نمونه راه حل برای این مسأله ارائه شده است که اولی ناقض OCP و دومی منطبق بر آن می‌باشد.

راه حل ۱:

```
public class Circle {
    double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    //...
    public void draw(){
        // Draws a circle
    }
    //...
}
```

```
public class DrawShapeList {
    //...
    public void draw(Circle[] list){
        for ( int i = 0;
            i < list.length; i++) {
            list[i].draw();
        }
    }
    //...
}
```

این راه حل می‌تواند ناقض OCP باشد. زیرا طراحی فوق نسبت به تغییر بسته نمی‌باشد. فرض کنید که قرار باشد روزی این برنامه کاربردی یک لیست از مربع‌ها را رسم نماید، و یا این که در حالت کلی قرار باشد که کلاً یک لیست از شکل‌های مختلف را دریافت کرده و رسم نماید. طراحی فوق این امکان را نمی‌دهد که بتولن بدون تغییر در متن برنامه کاربردی این تغییر را اعمال نمود. به عبارتی دیگر بدون تغییر در متن برنامه کاربردی، طراحی فوق نسبت به توسعه باز نمی‌باشد.

<sup>18</sup> Extension

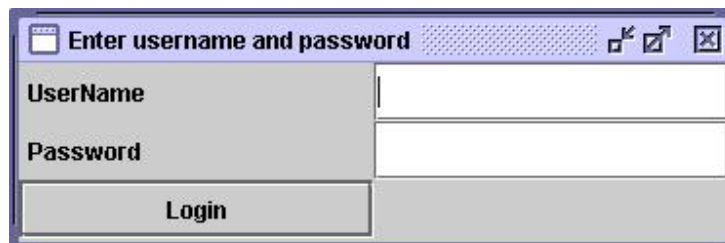
<sup>19</sup> Modification

<pre>public class Circle implements Drawable{     private double radius;     public Circle(double radius) {         this.radius = radius;     }     public void draw() {         //Draws the circle     }     //... }</pre>	<pre>public class Square implements Drawable{     private double sideLen;     public Square(double sideLen) {         this. sideLen = sideLen;     }     public void draw() {         //Draws the square     }     //... }</pre>
<pre>public interface Drawable {     //...     public void draw();     //... }</pre>	<pre>public class DrawShapeList {     public void draw(Drawable[] list){         for (int i = 0;             i &lt; list.length; i++) {             list[i].draw();         }     }     //... }</pre>

**مثال:** مورد کاربرد ورود به سیستم<sup>۲۰</sup> را در نظر بگیرید.

• جریان اصلی

۱. سیستم فرم ورود به سیستم را به کاربر نشان می‌دهد.
۲. کاربر نام کاربری و رمز عبور خود را وارد می‌نماید.
۳. سیستم اجازه ورود را به کاربر داده و فرم اصلی را به کاربر نشان می‌دهد.



مساله را به صورت زیر حل می‌کنیم.

```
public class LoginForm implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        DBAuthenticator authenticator = new DBAuthenticator();
        if( authenticator.authenticate( userName.getText(), password.getText() ) )
            System.out.println( "Authenticvated" );
        else
        {
            //...
        }
        // ...
    }
}
```

<sup>20</sup> Login

```

public class DBAuthenticator
{
    public boolean authenticate( String usrName, String password)
    {
        boolean result = false;
        try
        {
            Connection con = DriverManager.getConnection("Database",
                "myLogin", "myPassword");
            // authenticate the user
        } catch( SQLException e)
        {
            System.out.println("Unable to connect");
        }
        return result;
    }
}

```

این پیاده سازی اشکال دارد. اگر روش احراز هویت<sup>۲۱</sup> تغییر کند، بایستی کلاس DBAuthenticator را برداشته و کلاس جدیدی را جایگزین نماییم. مثلاً برای روش Active Directory بایستی کلاس ActiveDirAuthenticator اضافه شود و به همین ترتیب.

طراحی فوق نسبت به تغییرات « بسته » است، زیرا کلیه اعمال احراز هویت در روال پاسخ دهی به عمل انجام می شود. همچنین این طراحی نسبت به توسعه « باز » نیست، زیرا نمی توان به راحتی یک کلاس ActiveDirAuthenticator اضافه کرد. برای رفع مشکل LoginForm بایستی به یک تجرید وابسته شود. تجرید می تواند از نوع واسط و یا کلاس مجرد باشد.

```

public interface AuthenticatorInterface
{
    boolean authenticate( String usrName, String password);
}

```

با استفاده از این تجرید، کاربر می تواند به شیوهی زیر یک کلاس احراز هویت بسازد:

```
AuthenticatorInterface auth= new DBAuthenticator();
```

همچنین به وسیلهی الگوی کارخانهی اشیاء<sup>۲۲</sup> می توانیم پیاده سازی کلاس Authenticator را پنهان کنیم. یک کارخانهی اشیاء تولید کننده ای برای یک سری شیء است. به کارخانهی اشیاء اطلاعاتی در زمینهی نحوه ی ساختن شیء داده می شود و کارخانهی اشیاء یک نمونه از شیء را باز می گرداند. به عبارت دیگر دانش ساختن شیء به عهده ی کارخانه است و کاربر اطلاعی از آن ندارد.

یک روش برای پیاده سازی کارخانهی اشیاء استفاده از یک فایل جانبی XML است. به عنوان مثال، این فایل می تواند به صورت زیر باشد:

```

<Authenticator>
    <assembly name = "Test"/>
    <class name = "ActiveDirectoryBaseAuthenticator"/>
</Authenticator>

```

<sup>21</sup> authentication

<sup>22</sup> Object Factory

در سطر اول، آدرس class file ی که در آن پیاده سازی کلاس احراز هویت قرار داده شده آمده است. برنامه‌ی کارخانه‌ی اشیاء در زمان اجرا، به کمک reflection ابتدا پیاده سازی را از آدرس assembly پیدا می کند و سپس اسم داده شده در فیلد class name را برآن قرار می دهد. آن گاه شیء جدید تولید شده را باز می گرداند.

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

import java.io.FileReader;

public class XMLObjectFactory extends DefaultHandler
{
    /**
     * The path for object class file
     */
    String assembly;
    /**
     * The class name
     */
    String className;

    /**
     * a SAX method
     * @param namespace
     * @param localName
     * @param qName
     * @param atts
     */
    public void startElement(String namespace, String localName,
                             String qName, Attributes atts)
    {
        if ( localName.compareTo( "assembly") == 0)
        {
            assembly = atts.getValue(0);
            System.out.println( "Assembly = " + assembly);
        }
        else if ( localName.compareTo( "class") == 0)
        {
            className = atts.getValue(0);
            System.out.println( "Class = " + className);
        }
    }
    public XMLObjectFactory ()
    {
        super();
    }

    public Object create()
    {
        Class classForInstantiation;
        Object result = null;
        try {
            classForInstantiation = Class.forName( assembly );
            result = classForInstantiation.newInstance();
            System.out.println("The class of is " +
(classForInstantiation.getConstructors()[0]));

        } catch (ClassNotFoundException e) {
            e.printStackTrace(); //To change body of catch statement use File | Settings
| File Templates.
        } catch (IllegalAccessException e) {
```

```

        e.printStackTrace(); //To change body of catch statement use File | Settings
| File Templates.
    } catch (InstantiationException e) {
        e.printStackTrace(); //To change body of catch statement use File | Settings
| File Templates.
    }
    return result;
}

public static void main (String args[])
    throws Exception
{
    System.setProperty("org.xml.sax.driver", "org.apache.xerces.parsers.SAXParser");
    XMLReader xr = XMLReaderFactory.createXMLReader();
    XMLObjectFactory handler = new XMLObjectFactory();
    xr.setContentHandler(handler);
    xr.setErrorHandler(handler);

    FileReader r = new FileReader( "obj.xml");
    xr.parse(new InputSource(r));
}
}

```

در حالت خاص ممکن است شیء مورد نظر به یک شیء دیگر هم وابسته باشد. در این حالت می توان کد فوق را به صورتی تغییر داد که علاوه بر ساختن شیء ، اشیاء وابسته ی آن را نیز بسازد و پس از اتصال آنها به شیء نتیجه را بازگرداند.

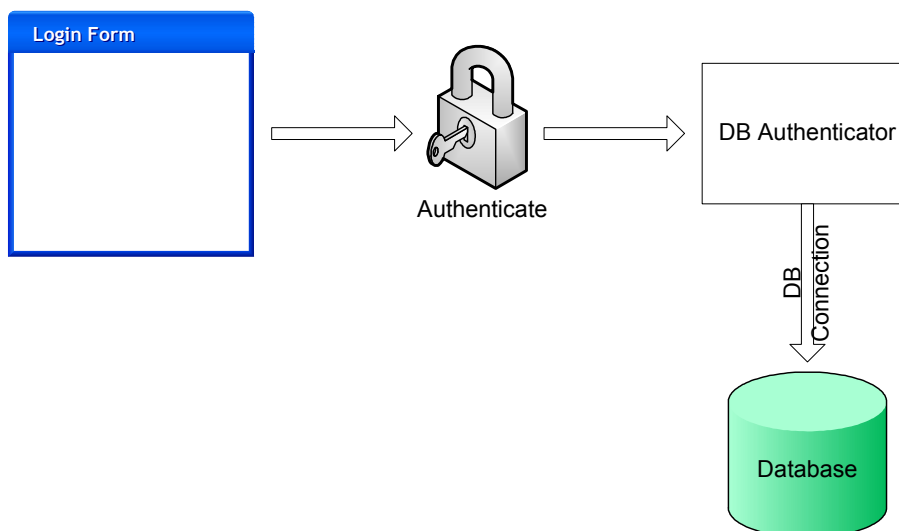
```

<Authenticator>
  <assembly name = "Test"/>
  <class name = "ActiveDirectoryBaseAuthenticator"/>
  <dependent name = "DBConnection"/>
</Authenticator>

```

### ۳-۲-۱ شیء ساختگی (Mock Object)

در قسمت قبلی چنین برنامه ای طراحی کردیم:



یکی از مشکلاتی که با آن مواجه هستیم، تست کردن فرم است. برای تست کردن فرم بایستی پایگاه داده ایجاد شود، ارتباط با آن برقرار شود، مکانیزم احراز هویت اعمال شود تا پس از آن بتوان فرم را تست کرد.

آیا می توان ترتیبی اتخاذ کرد که فرم مستقیماً تست شود؟ به عبارت دیگر می خواهیم unit testing انجام دهیم، و بدون وجود اجزاء خارجی یک جزء سیستم را تست کنیم. روشی که معمولاً در این مواقع استفاده می شود اشیاء ساختگی است. این اشیاء مصنوعی هستند، و تمامی رفتار شیء اصلی را شبیه سازی می کنند.

توجه کنید که Null Object با Mock Object تفاوت دارد. از شیء تهی در خود برنامه استفاده می شود، ولی مورد استفاده شیء ساختگی در تست کردن برنامه است. همچنین یک شیء ساختگی اصولاً مقادیر واقعی بر می گرداند، در حالی که شیء تهی وظیفه دارد رفتار «هیچ» را شبیه سازی کند.

برای ساختن شیء ساختگی معمولاً از دو روش استفاده می شود:

✓ یک شیء به صورت استاتیک می سازیم که یک سری رفتار ثابت را تولید کند.

```
public boolean authenticate( String userName, String password)
{
    if ( userName.compareTo("Hossein") == 0 && password.compareTo("Hojjat") == 0)
        return true;
    return false;
}
```

✓ به صورت پویا یک شیء ساختگی باز می گردانیم. برای این کار می توان از الگوی کارخانهی اشیاء استفاده کرد، به این صورت که خصوصیات شیء ساختگی مورد نظر را به کارخانه می دهیم و آن شیء ساختگی دلخواه را برمی گرداند.

◀ چه وقت باید از اصل باز و بسته<sup>۲۳</sup> استفاده نمود؟

در شروع طراحی نمی توان جاهایی را پیدا کرد که ocp را نقض کند. برای رفع این مساله جمله

**«اگر بار اول مرا فریب دادی، شرم بر تو! ولی اگر بار دیگر مرا فریب دادی، شرم بر من!!»**

را در ذهن می سپاریم. اگر دفعهی اول نیازی پیش آمد و ما تغییراتی اعمال کردیم (با هر قدر زحمت و دردسر) اشکالی ندارد. ولی اگر بار دوم نیازی مشابه نیاز اول پیدا شد و مجبور شدیم باز هم برای اضافه کردن نیاز زحمت بکشیم کار ما مشکل داشته است چون طراحی را طوری عوض نکردیم که این طور تغییرات را تحمل کند..

به محض وقوع تغییر و توسعه اول، باید طراحی را به گونه ای تغییر داد که به سمت OCP برویم. در نتیجه هنگام وقوع توسعه ها و تغییرات بعدی دیگر نباید مشکل خاصی پیش بیاید، وگرنه شرم بر من ☺.



### ۳-۳ اصل جایگزینی لیسکوف (Liskov's Substitution Principle - LSP)

وفا کنیم و ملامت کشیم و خوش باشیم که در طریقت ما کافر است رنجیدن حافظ

- ? آیا وراثتی که من انجام داده‌ام درست است و اگر نه می‌تواند بهتر باشد؟
- ? براساس چه قواعدی می‌توان استفاده از وراثت را کنترل نمود؟
- ? مشخصه‌ی یک سلسله مراتب وراثت<sup>۲۴</sup> خوب چیست؟
- ? چه عواملی باعث ایجاد سلسله مراتبی مغایر با OCP می‌گردند؟

#### تعریف:

▪ "متد هایی که از اشاره‌گر یا ارجاع به کلاس پایه استفاده می‌کنند، باید بتوانند همچنین از تمام نمونه‌های کلاس‌های مشتق شده استفاده کنند بدون این که در مورد این کلاس‌های مشتق شده چیزی بدانند."

البته خانم باربارا لیسکف در ابتدا این اصل را این گونه معرفی کرده است:

▪ اگر برای هر شیئی O1 از نوع S یک شیئی O2 از نوع T وجود داشته باشد، به طوری که برای تمامی برنامه‌های P که بر حسب T نوشته شده‌اند، اگر O1 با O2 جایگزین شود، رفتار P بدون تغییر باقی بماند، آنگاه S زیرنوعی از T است.

مثال: در تکه برنامه‌های زیر می‌توان مورد نقض این قانون را دید:

<pre>package shapes;  public class Rectangle {     protected int _height;     protected int _width;      public Rectangle(int height, int width) {         _height = height;         _width = width;     }      public int getHeight() {         return _height;     }      public void setHeight(int height) {         _height = height;     }      public int getWidth() {         return _width;     }      public void setWidth(int width) {         _width = width;     } }</pre>	<pre>package shapes;  public class Square extends Rectangle {     public Square(int height) {         super(height, height);     }      public void setHeight(int height) {         super.setHeight(height);         super.setWidth(height);     }      public void setWidth(int width) {         super.setWidth(width);         super.setHeight(width);     } }</pre>
--	--

<sup>24</sup> Inheritance Hierarchy

```

package shapes;

import junit.framework.TestCase;

public class TestLSP extends TestCase {
    public void testExtent() {
        Rectangle r = new Rectangle(10, 11);
        // Rectangle r = new Square(10);

        //Lots of lines of code
        r.setHeight(20);
        r.setWidth(12);
        assertEquals(240, r.getHeight() *
            r.getWidth());
    }

    public void testSetters() {
        Rectangle r = new Rectangle(10, 11);
        // Rectangle r = new Square(10);

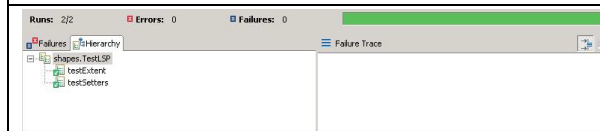
        g(r);
        f(r);

        assertEquals(r.getHeight(), 20);
        assertEquals(r.getWidth(), 30);
    }

    private void g(Rectangle r) {
        r.setWidth(30);
    }

    private void f(Rectangle r) {
        r.setHeight(20);
    }
}

```



```

package shapes;

import junit.framework.TestCase;

public class TestLSP extends TestCase {
    public void testExtent() {
        // Rectangle r = new Rectangle(10,
        11);
        Rectangle r = new Square(10);

        //Lots of lines of code
        r.setHeight(20);
        r.setWidth(12);

        assertEquals(240, r.getHeight() *
            r.getWidth());
    }

    public void testSetters() {
        // Rectangle r = new Rectangle(10,
        11);
        Rectangle r = new Square(10);

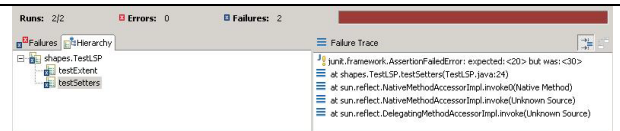
        g(r);
        f(r);

        assertEquals(r.getHeight(), 20);
        assertEquals(r.getWidth(), 30);
    }

    private void g(Rectangle r) {
        r.setWidth(30);
    }

    private void f(Rectangle r) {
        r.setHeight(20);
    }
}

```



حالا با این مشکلات چه کنیم؟ اصل لیسکوف خیلی کلی است و ما نمی‌توانیم همه‌ی موارد را بیازماییم؛ بنابراین بدنبال نشانه‌هایی مبنی بر نقض این قانون می‌گردیم.

موارد نقض (احتمالی) LSP:

- ۱- تغییر در کلاس پایه در پی افزودن یک اشتقاق<sup>۲۵</sup>
- ۲- استفاده از شرط<sup>۲۶</sup> برای تعیین نوع اشیاء
- ۳- تنزل<sup>۲۷</sup> متدها در کلاس‌های مشتق شده (بدنه‌ی متدها در کلاس مشتق خالی باشد)
- ۴- پرتاب استثناء<sup>۲۸</sup> در اشتقاق

انواع وراثت:

- single یا multiple
- اجباری و اختیاری
- ارث‌بری فقط ساختار یا رفتار علاوه بر آن

<sup>25</sup> Derivation

<sup>26</sup> if/else

<sup>27</sup> Degenerate

<sup>28</sup> Throwing Exception

بررسی اینکه آیا اصولا Multiple Inheritance بد است:

- باید دید که آیا زبان مورد نظر از آن پشتیبانی می‌کند.
- اگر نمی‌کند هم فقط باید در Design Model از آن پرهیز کنیم و نه در Analysis Model یا Domain Model

### ۳-۳-۱ Sub-Classing vs. Sub-Typing

بحث Sub-Classing طبق همان تعریفی است که ما برای وراثت داشتیم: ارتباط بین پدر و فرزند که خصوصیات و رفتار پدر به فرزند انتقال پیدا می‌کند اما بحث Sub-Typing با تعریف لیسکوف آزموده می‌شود.

- ممکن است یک ارث‌بری Sub-Classing باشد ولی Sub-Typing نباشد و این توسط آزمون لیسکوف معلوم می‌شود.
- نقض اصل لیسکوف معمولا معادل نقض OCP است.

### ۳-۳-۲ مفهوم طراحی بر پایه‌ی قرارداد (Design by Contract - DbC)

این روش از روش‌هایی است که برای تعیین صحت نرم‌افزار استفاده می‌شود.

بحث شیئی‌گرایی بر اساس ارتباط شیئی‌هاست و این ارتباطات به صورت بی‌قاعده و غیرقابل کنترل هستند. در DbC می‌گوییم که این روابط را با توجه به قراردادهایی کنترل کنیم و هر کلاس شرایطی داشته باشد.

- پیش‌شرط<sup>۲۹</sup>

تعهدات صدا کننده هستند و باعث می‌شوند کار عامل (صدا شونده) راحت‌تر شود

- پس‌شرط<sup>۳۰</sup>

تعهداتی هستند که صدا شونده در صورت رعایت تعهدات توسط صدا کننده آن‌ها را تضمین می‌کند و باعث می‌شوند صدا کننده راحت‌تر کار را تحویل گیرد.

- بدنه<sup>۳۱</sup>

یک سری از شرایط<sup>۳۲</sup> که در طول عمر یک کلاس باید برقرار باشند. اگر این شرایط فقط در قالب متدهای عمومی<sup>۳۳</sup> آزموده شوند کافی است. چون متدهای دیگر نیز در قالب همین متدها فراخوانی می‌شوند.

ارتباط DbC با LSP:

اصل DbC بیشتر به ما کمک می‌کند تا تغییر در کلاس پایه در پی افزودن یک اشتقاق را درک کنیم:

- پیش‌شرط‌های تعریف شده برای یک متد در کلاس مشتق شده، باید ضعیف‌تر و یا برابر با پیش‌شرط‌های تعریف شده برای آن متد در کلاس پایه باشند.

<sup>29</sup> Precondition

<sup>30</sup> Postcondition

<sup>31</sup> Body

<sup>32</sup> invariant

<sup>33</sup> Public

- پس شرط‌های تعریف شده برای یک متد در کلاس مشتق شده، باید قوی‌تر و یا برابر با پس شرط‌های تعریف شده برای آن متد در کلاس پایه باشند.

برای پس شرط‌های `setHeight` در مثال قبلی داریم:

Rectangle	Square
<code>_width = oldWidth</code> <code>_height = height</code>	<code>_height = height</code> <code>_width = height</code>

می بینیم که شرط "`_width = oldWidth`" در کلاس `Square` حذف شده است پس این متد پس شرط ضعیف‌تری در کلاس مشتق شده دارد. باید توجه داشت که این رابطه معکوس پذیر نیست؛ یعنی لزوماً اگر پس شرطی از یک کلاس از دیگری ضعیف‌تر است دیگری قوی‌تر از کلاس ما نمی‌شود.

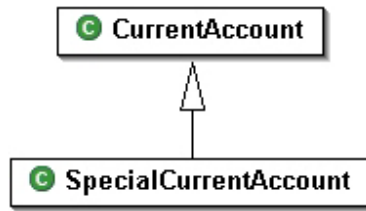
**مثال:** در یک سیستم بانکی ۲ نوع حساب جاری عادی (`CurrentAccount`) و جاری ویژه (`SpecialCurrentAccount`) را در نظر بگیرید. فرق یک حساب ویژه با حساب عادی تنها در این است که مکانیزم سوددهی به حساب ویژه اندکی متفاوت بوده و دیگر این که حساب ویژه در زمانی زودتر از یک موعد مقرر نمیتواند بسته گردد. چیزی که در وهله اول به نظر می‌رسد این است که حساب ویژه یک حالت خاص از حساب معمولی می‌باشد. پس طراحی زیر درست به نظر می‌رسد. (ولی ...)

```
public class CurrentAccount {
    protected int balance;
    protected int period;

    public CurrentAccount(
        int balance, int period) {
        this.balance = balance;
        this.period = period;
    }
    public boolean openAccount(
        int balance) {
        this.balance = balance;
        return true;
    }
    public boolean closeAccount() {
        if(balance >0)
            return true;
        else
            return false;
    }
    public int getBalance() {
        return this.balance;
    }
    public void setBalance(
        int balance) {
        this.balance = balance;
    }
    public int getPeriod() {
        return this.period;
    }
}

    public void setPeriod(int period){
        this.period = period;
    }
}
public class SpecialCurrentAccount extends
CurrentAccount {
    private int defaultPeriod;

    public SpecialCurrentAccount(
        int balance, int period) {
        super(balance, period);
    }
    public boolean closeAccount() {
        //The precondition is stronger
        //than previous method
        if( balance>0 &&
            period>defaultPeriod)
            return true;
        else
            return false;
    }
    public int getDefaultPeriod() {
        return this.defaultPeriod;
    }
    public void setDefaultPeriod(
        int defaultPeriod) {
        this.defaultPeriod =
defaultPeriod;
    }
}
```



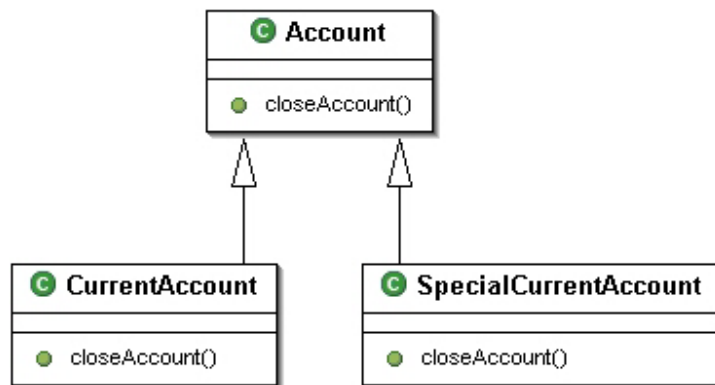
حال فرض کنید که متد زیر برای بستن یک حساب وجود دارد:

```

public void closeAnAccount(
    CurrentAccount ac) {
    System.out.println("Account close result: " + ac.closeAccount() );
}
  
```

مشخص است که این گونه نیست که متد فوق به ازای هر نمونه از CurrentAccount و SpecialCurrentAccount دقیقاً عین هم عمل کرده و نتیجه یکسانی برگردانند. ممکن است یک نمونه CurrentAccount قابل بسته شدن باشد در حالی که یک نمونه از SpecialCurrentAccount قابل بسته شدن نباشد و این نکته کاملاً بدیهی است. طبق طراحی بر اساس قرارداد چون پیش شرط متد closeAccount() در کلاس مشتق شده قوی تر شده، پس LSP نقض می گردد.

طراحی مناسب و منطبق بر LSP مطابق نمودار زیر می باشد:



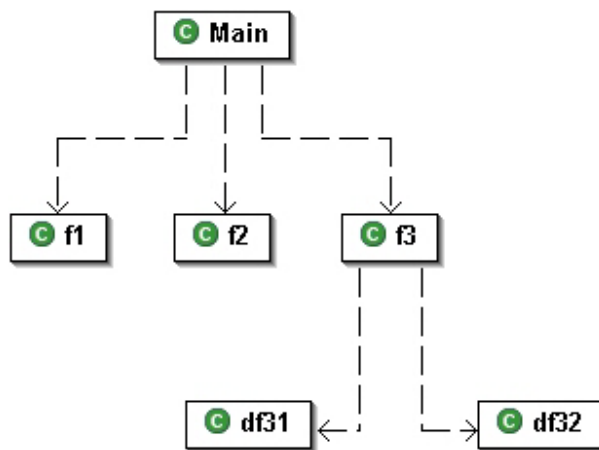
### ۳-۴ اصل وابستگی معکوس (Dependency Inversion Principle - DIP)

*"Don't believe the world owes you a living; the world owes you nothing it was here first."*

Mark Twain

در تقسیم بندی<sup>۳۴</sup> نگاه ما معمولا بالا به پایین است و سطوح بالاتر از اهمیت بالاتری برخوردارند؛ این سطوح برای انجام کار خود به سطوح پایین تر نیاز دارند.

- بالا: Abstraction
- پایین: Detail(Implementation)

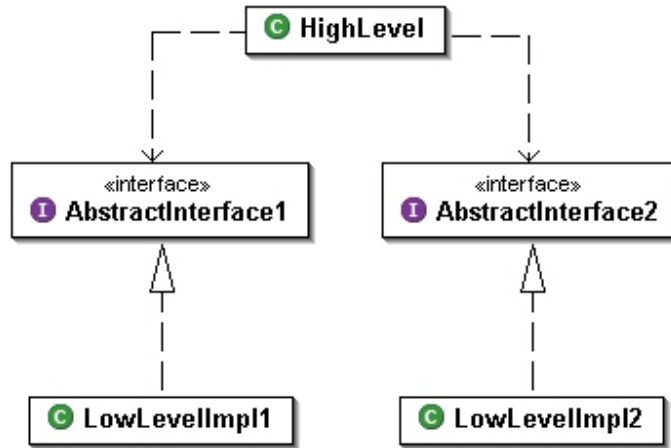


مشاهده می شود که وابستگی ها به گونه ای هستند که اگر پایینی ها عوض شوند روی بالایی ها تأثیر می گذارند و ما می خواهیم این وابستگی را برعکس کنیم چون لایه های بالاتر که از اهمیت بالاتری برخوردارند نباید وابسته به لایه های پایین تر که اهمیت کمتری دارند باشند.

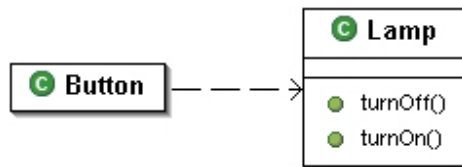
این کار با قرار دادن لایه ای میانی بین دو لایه انجام می شود و عملا لایه های بالایی و پایینی به لایه ای جدیدی که بین آن ها قرار می گیرد وابسته می شوند. این لایه ای میانی را تجرید<sup>۳۵</sup> می نامیم.

<sup>34</sup> Decomposition

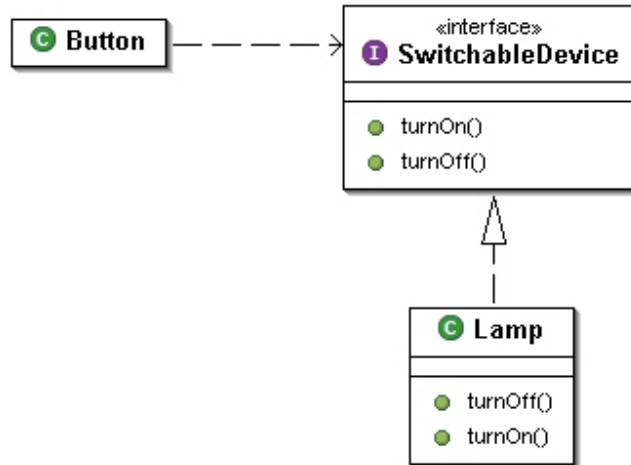
<sup>35</sup> Abstraction



برای مثال کلاس کلیدی داریم که در حال حاضر یک لامپ را کنترل می‌کند:



حال اگر قرار باشد شیئی که این کلید کنترلش می‌کند تغییر کند بهتر است طراحی را به این صورت عوض کنیم:



هرکجا که احساس کردیم احتمال تغییر در سرویس دهنده زیاد باید DIP را اعمال کنیم. در این مثال هم احتمال داده می‌شد که لامپ تغییر کند.

## تعریف:

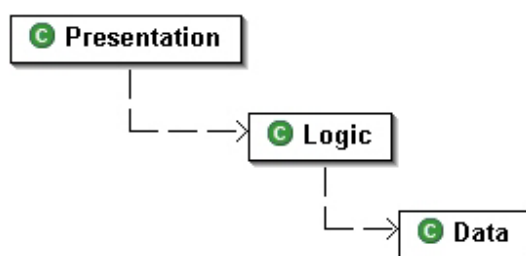
- ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند و هر دوی آن‌ها باید به تجربی‌ها وابسته باشند.
- تجربی‌ها نباید به جزئیات وابسته باشند، بلکه جزئیات باید به تجربی‌ها وابسته باشند.

## :Abstract coupling

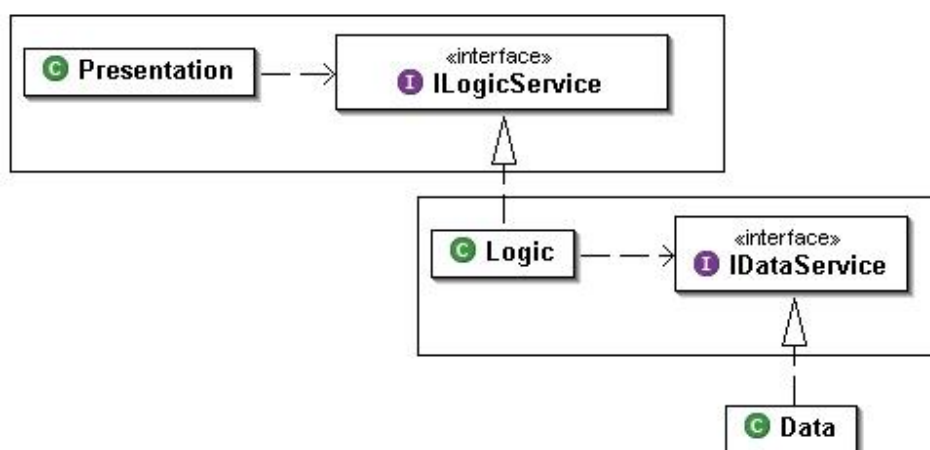
در مواقعی که احتمال تغییر می‌رود به abstract ها وابسته باش نه به concrete ها در نتیجه‌ی این خواهیم داشت:

- هیچ متغیری از نوع یک کلاس واقعی تعریف نخواهد شد.
- هیچ کلاسی نباید از یک کلاس واقعی ارث بری کند.

برای مثال در معماری سه لایه ممکن است طراحی اولیه به این صورت باشد(توجه داشته باشیم این کلاس‌ها عملاً مربوط به یک کار خاص هستند مثلاً یک دانش‌آموز و این که نام آن‌ها کلی است برای جدا شدن از غالب یک مسأله‌ی خاص است):



این طراحی ناشی از دید لایه‌ای<sup>۳۶</sup> به سیستم است. دید دیگری که متصور است دید کالبدی<sup>۳۷</sup> است(نام گذاری و محل قرار گرفتن این واسط‌ها مسأله‌ای مورد بحث است):

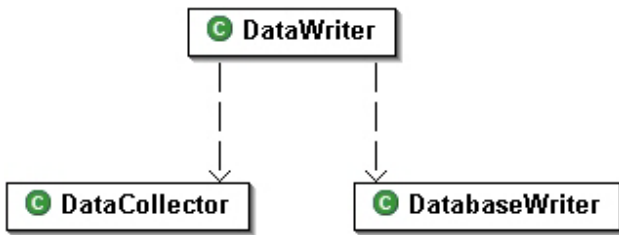


<sup>36</sup> Layering View

<sup>37</sup> Frame Work View



**مثال:** طراحی مقابل را در نظر بگیرید.



در این مثال وظیفه کلاس Data Collector این است که که اطلاعات را از یک منبع اطلاعاتی تهیه کند و این داده‌ها را در یک فرمت استاندارد و به شکل یک object به کلاس DataWriter تحویل دهد. آن‌گاه کلاس DatabaseWriter این object را به کلاس DatabaseWriter تحویل می‌دهد تا این کلاس داده‌ها را در یک پایگاه داده ذخیره کند.

```
public class DataCollector {
    public void collectData(String source) {

        //collect the data from source, let us say String data
        //initialise a DataWriter object
        DataWriter writer = new DataWriter();

        //ask the writer object to write the data
        writer.writeData(the collected data);
    }
}

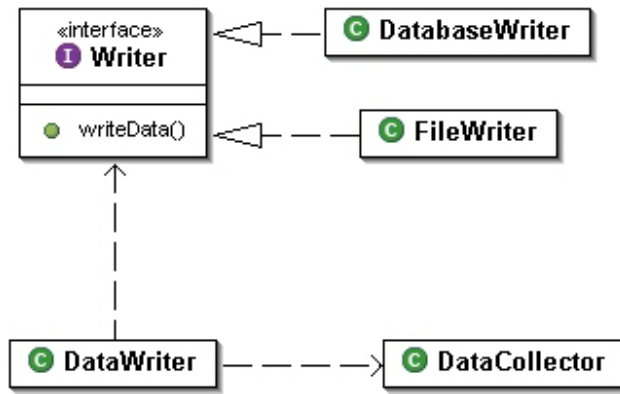
public class DataWriter {

    public void writeData(the data) {
        //write the data to the database
        DatabaseWriter dw = new DatabaseWriter();
        dw.writeToDB(params);
    }
}

public class DatabaseWriter {
    public void writeToDB(params) {
        //write physically to the database
    }
}
```

مشکل این طراحی این است که ناقض DIP می‌باشد زیرا ماژول سطح بالای DataWriter به جزئیات ماژول سطح پائین DatabaseWriter وابسته می‌باشد و اگر قرار باشد که اطلاعات در یک فایل ساده نوشته شوند و کلاسی با عنوان FileWriter عهده‌دار این وظیفه باشد، آن‌گاه کلاس DataWriter باید تغییر کند که این امر هم ناقض DIP است و هم ناقض OCP.

طراحی زیر می‌تواند یک راه‌حل مناسب و منطبق بر DIP برای مشکل فوق باشد.



### ۳-۵ اصل تفکیک واسطها (Interface Segregation Principle - ISP)

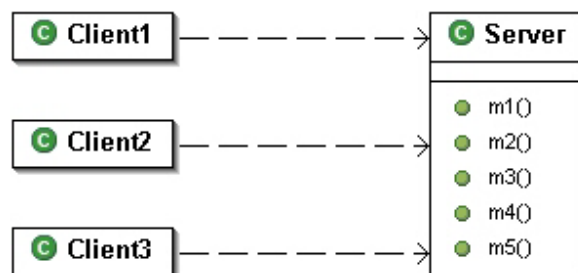
غم زمانه خورم یا فراق یار کشم

به طاقتی که ندارم کدام بار کشم

سعدی

این اصل آخرین اصل از اصول طراحی کلاسهاست و بحث آن در مورد شیءهایی است که سرویسهای مختلفی را می توان از آنها گرفت. این شیءها که در جاهای مختلف نقشهای مختلفی را دارند را Architectural Entity یا Core Entity می نامیم.

یک راه برخورد با این شیءها این است که به هر سرویس گیرنده خودش را بفرستیم. در عمل ما همه ی سرویسها را در اختیار همه قرار داده ایم در حالی که ممکن است همه ی سرویسها مورد نیازشان نباشد.



تعریف:

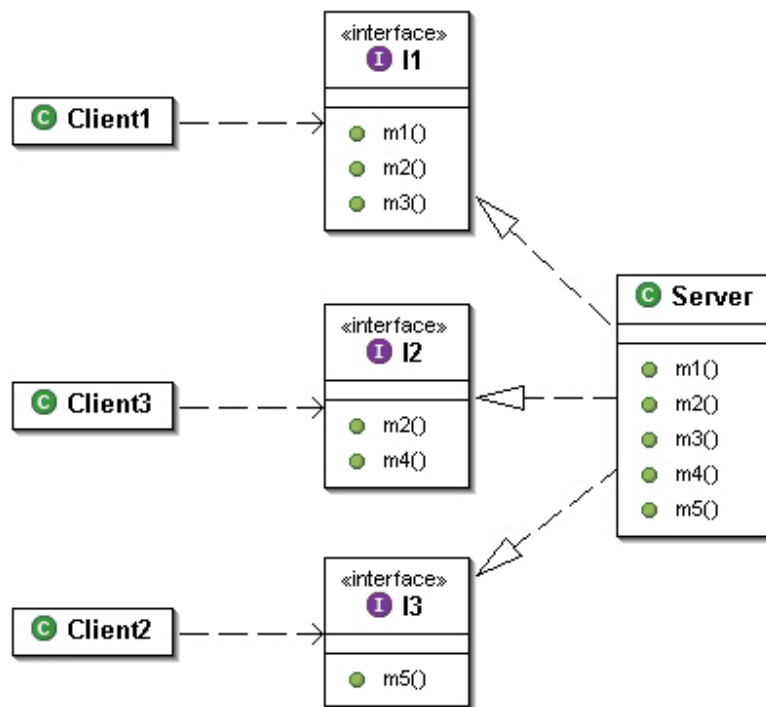
سرویس گیرندگان نباید به متدهایی که به آنها نیاز ندارند وابسته شوند.

روشهای اعمال ISP:

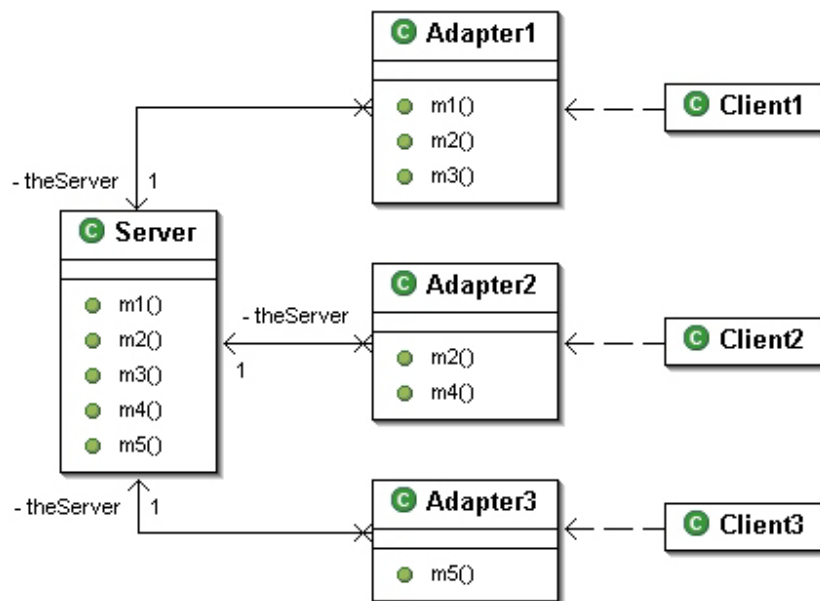
۱. به ازای هر سرویس یک واسط<sup>۳۸</sup> تعریف کنیم که سرویس دهنده آن را پیاده سازی کند.
۲. این روش مثل روش قبلی است ولی از کلاس مجرد<sup>۳۹</sup> استفاده می شود. در این حالت باید قابلیت زبان از نظر وراثت چندگانه را در نظر گرفت.

<sup>38</sup> Interface

<sup>39</sup> Abstract Class

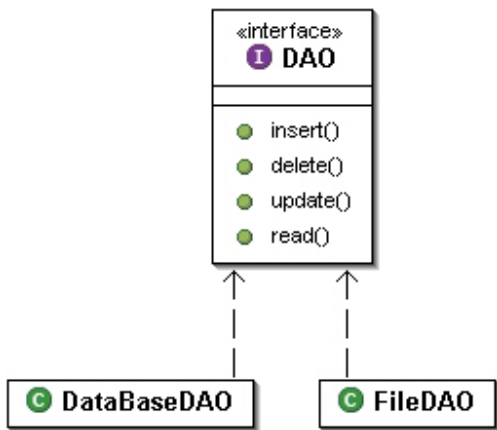


برای برآوردن اهداف فوق می‌توان از الگوی انطباق دهنده<sup>40</sup> استفاده کرد و مسأله را با Delegation حل کرد.



<sup>40</sup> Adapter Pattern

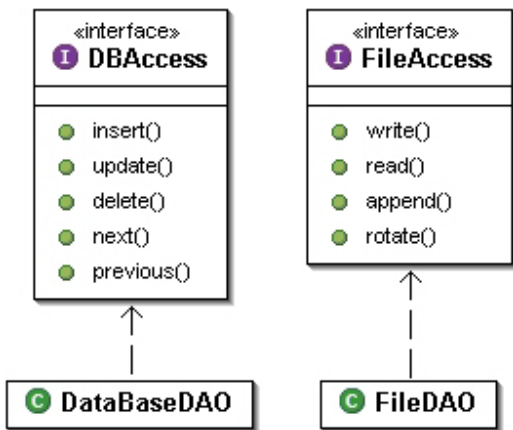
**مثال:** فرض کنید که در برنامه‌ای به یک سری اشیاء تحت عنوان "اشیاء دسترسی به داده" (Data Access Objects) نیاز داشته باشیم که قرار است واسطی برای منابع داده‌ای مختلف مانند فایل و پایگاه داده ارائه دهند. طراحی مقابل را ملاحظه کنید.



حال این سؤال پیش می‌آید که اگر یک منبع داده‌ای "فقط خواندنی" باشد و منطقاً احتیاجی به متدهای insert() و update() نداشته باشد، تکلیف چیست؟

آیا خالی گذاشتن بدنه پیاده‌سازی متدهای فوق عاقلانه است؟

**نتیجه:** این پیاده‌سازی با ISP در تناقض است.

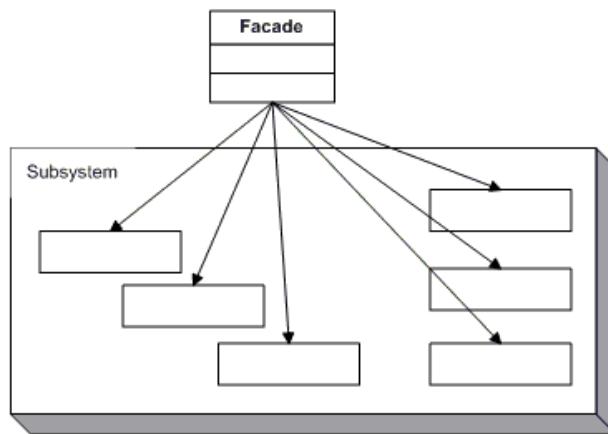


مشکل دیگر این است که فرضاً بخواهیم به منابع داده‌ای از نوع فایل متد جدیدی مانند append() اضافه کنیم. اگر طبق طراحی فوق فقط یک واسط داشته باشیم در چنین شرایطی حتماً باید واسط مذکور تغییر کرده و نتایجاً تمام انواع منابع داده‌ای دیگر که پیاده‌ساز این واسط می‌باشند، باید بدنه این متدها را به نحو مناسب پیاده‌سازی نمایند. بر مبنای ISP طراحی مقابل راه‌حل این مشکل می‌باشد.

### ۳-۵-۱ Facade Pattern vs. Adapter Pattern

همان طور که دیدیم الگوی انطباق دهنده واسط یک کلاس را به واسط دیگری که کاربر انتظار دارد تبدیل می‌کند. به عبارت دیگر واسط کلاس را بر نیاز کاربر منطبق می‌سازد. بی مورد نیست که این الگو را در مقابل الگوی دريچه<sup>۴۱</sup> مورد بررسی قرار دهیم. الگوی دريچه دسترسی به یک سری اشیاء را با فراهم کردن یک شیء واحد که به آن دريچه می‌گوییم راحت تر می‌سازد. دريچه یک واسط سطح بالاتر است که استفاده از زیرسیستم را ساده تر می‌کند. دريچه توسط delegation درخواست های داده شده را به زیرسیستم ها انتقال می‌دهد.

<sup>41</sup> Facade



دریچه عموماً در برابر یک سری از اشیاء قرار دارد، حال آنکه الگوی انطباق یک کلاس را برای کاربران خود آداپته می کند. خصوصیات مختلف الگوهای دریچه و انطباق در جدول زیر مورد مقایسه قرار گرفته اند:

دریچه	انطباق	سوال
√	√	آیا کلاس هایی از قبل وجود دارند؟
-	√	آیا از قبل واسطی وجود دارد که می خواهیم برای آن طراحی کنیم؟
-	√	آیا بحث چند ریختی مطرح می شود؟
√	-	آیا واسط ساده تری نیاز است؟

## ۴. رویه‌ی تولید یک سیستم نرم‌افزاری

*"Whoever fights monsters should see to it that in the process he doesn't become a monster."*

Friedrich Nietzsche

حال می‌خواهیم ببینیم که برای حل یک مسأله و تولید یک سیستم نرم‌افزاری باید چه گام‌هایی و با چه اصولی برداشته شود؟ مثلاً فرض کنید قرار است سیستم نرم‌افزاری ثبت نام دانشجویان نوشته شود. چه باید کرد؟ از کجا باید شروع کرد؟ واژه‌های زیر در این روال چه نقشی دارند؟

Use Case  
Inception

Prototype  
Elaboration



Design  
Test

Construction  
Architecture

بحث روش انجام کار در روال ساخت سیستم‌های نرم‌افزاری از این جهت مهم است که فرآیند ساخت یک نرم‌افزار با خط تولید یک شرکت تولیدی متفاوت است و برای پروژه‌های مختلف این روش می‌تواند متفاوت باشد.

### ۱-۴ تا رسیدن به مرحله‌ی "طراحی" باید چه کار کرد؟

#### ۱-۱-۴ مدل‌سازی کسب و کار<sup>۴۲</sup>

در این مرحله هدف، شناخت وضع موجود سازمان و مشخص شدن مجموعه workflow‌های جاری می‌باشد. در واقع در این مرحله می‌خواهیم روال‌هایی که کاندیدای خوبی برای خودکارسازی<sup>۴۳</sup> هستند را شناسایی کنیم. در UML مفاهیم Business Actor، Business Worker، Business Entity و ... مربوط به حیطه Business Modeling می‌باشند.

روال‌های دستی + روال‌های مکانیزه ← روال‌های جدید سازمان

#### ۲-۱-۴ نیازمندی‌ها (Requirements)

تعریف: شرایط<sup>۴۴</sup> و قابلیت‌هایی<sup>۴۵</sup> که نرم‌افزار باید بتواند آن‌ها را پوشش دهد.

<sup>42</sup> Business Modeling

<sup>43</sup> Automation

<sup>44</sup> Conditions

<sup>45</sup> Capabilities

#### ۴-۱-۲-۱ مدیریت نیازمندی‌ها (Requirements Management)

**تعریف:** مدیریت نیازمندی‌ها عبارتست از یک روش سیستماتیک جهت استخراج، سازمان‌دهی و مستندسازی نیازمندی‌های یک سیستم نرم‌افزاری به اضافه ثبت قراردادهای مابین کارفرما و گروه تولید سیستم نرم‌افزاری برای مدیریت چگونگی تغییر نیازمندی‌ها.

**نکته ۱:** مدل‌سازی موارد کاربرد<sup>۴۶</sup> جزئی از مدیریت نیازمندی‌ها می‌باشد.

**نکته ۲:** نهایتاً نتیجه کار در این قسمت راه‌حل ما (به عنوان مجری) در جهت رفع معضلات موجود می‌باشد.

◀ اولین قدم در این مرحله (مدیریت نیازمندی‌ها) چیست؟

پاسخ: دریافتن مشکل

- از طریق برگزاری مصاحبه‌های کتبی و شفاهی ← بررسی دقیق وضع موجود
- هدف: رسیدن به پاسخ این سؤال که کارفرما چرا می‌خواهد سیستم قدیمی خود را به یک سیستم جدید تبدیل کند؟

#### ۴-۱-۲-۲ قانون Pareto (80/20) چیست؟

این قانون پس از قرن ۱۹ میلادی توسط Vilfredo Pareto نام‌گذاری شد و بعدها در سال ۱۹۵۰ میلادی توسط JM Juran به طور رسمی ارائه شد. این قانون می‌گوید که بیشتر تأثیرات ناشی از موارد اندکی هستند. مثلاً ۲۰ درصد موارد ممکن ۸۰ درصد تغییرات را نتیجه دهند. همانطور که متوجه شدید این قانون به قانون ۸۰/۲۰ نیز معروف است.

◀ این قانون چه ربطی به مدیریت نیازمندی‌ها دارد؟

همانطور که اشاره شد اولین گام در این مرحله دریافتن مشکل است و این قانون در تحلیل شرایط موجود و ارائه راه‌حل به ما کمک خواهد کرد. به عنوان نمونه مثال زیر را در نظر بگیرید. فرض کنید ۴ کار A، B، C و D با درآمدها و هزینه‌های زیر موجودند. برای بهبود این وضعیت با استفاده از قانون Pareto کدام کار(ها) باید حذف شود؟

کار	درآمد	هزینه
A	٪۴۵	٪۵۰
B	٪۴۰	٪۲۰
C	٪۱۰	٪۲۰
D	٪۵	٪۱۰

<sup>46</sup> use case modeling



قانون Pareto می‌گوید که :

کار C باید حذف شود چون جزء ۸۰ درصد هزینه‌هاست ولی در ۸۰ درصد درآمدها نیست.

#### ۳-۲-۱-۴ ذی‌النفع<sup>۴۷</sup> در یک پروژه کیست ؟

**تعریف:** ذی‌النفع در یک پروژه کسی است که :

- به نحوی هزینه تولید نرم‌افزار را می‌پردازد،
- به طور مستقیم یا غیر مستقیم از نرم‌افزار استفاده می‌کند،
- و یا به گونه‌ای تحت تأثیر حاصل خروجی نرم‌افزار می‌باشد.

**نکته:** کاربران نیز جزء ذی‌النفع‌های پروژه می‌باشند.

#### ۴-۲-۱-۴ Vision چیست ؟

بررسی وضع مشکلات، ذی‌النفع‌ها، کاربران، محیط، ویژگی‌ها<sup>۴۸</sup> و محدودیت‌ها<sup>۴۹</sup> دورنمایی (چشم‌اندازی) از سیستم را به ما خواهد داد که به آن Vision و به سندی که بر مبنای آن تنظیم می‌شود **Vision Document (سند چشم‌انداز)** می‌گویند.

**توجه:** از این پس مرجع تمام افراد تیم تولید پروژه برای ادامه کار سند چشم‌انداز خواهد بود.

**نکته:** معمولاً قرارداد بر مبنای سند چشم‌انداز بسته می‌شود ولی حالتی نیز متصور است که قرارداد بر مبنای درخواست پیشنهاد<sup>۵۰</sup> ساده بسته شده و از آن به بعد سند چشم‌انداز مبنای حرکت به سمت هدف می‌شود.

#### اما بعد ...

پس از بیان سیستم به صورت سطح بالا، نوبت به توصیف سیستم در سطوح پایین‌تر می‌رسد. در این سطح به توصیف دقیق موارد کاربرد پرداخته می‌شود. توجه کنید که مجموعه موارد کاربرد راه‌حل ما را نتیجه می‌دهد که از طریق پیش‌نمایش<sup>۵۱</sup> به کاربر(ان) نشان داده خواهد شد.

◀ اما تکلیف نیازمندی‌های غیر وظیفه<sup>۵۲</sup> مانند کارایی<sup>۵۳</sup>، امنیت<sup>۵۴</sup>، هم‌روندی<sup>۵۵</sup> و ... چیست ؟

<sup>47</sup> Stakeholder

<sup>48</sup> Features

<sup>49</sup> Constraints

<sup>50</sup> Request For Proposal

<sup>51</sup> prototype

<sup>52</sup> Non Functional

<sup>53</sup> Efficiency

<sup>54</sup> Security

<sup>55</sup> Concurrency

- آن‌هایی که فقط مخصوص یک مورد کاربرد می‌باشند در قسمت "نیازمندی‌های خاص"<sup>۵۶</sup> در انتهای توضیحات مربوط به مورد کاربرد<sup>۵۷</sup> آورده می‌شوند مانند زمان پاسخ<sup>۵۸</sup> یک مورد کاربرد خاص.
- آن‌هایی که مربوط به تمام موارد کاربرد می‌باشند نیز در سندی جداگانه موسوم به "مشخصات تکمیلی"<sup>۵۹</sup> به آن‌ها پرداخته می‌شود. مانند امنیت در سیستم نرم‌افزاری.

#### ۳-۱-۴ حالا "معماری" ...

اکنون باید معماری سیستم مشخص شود :

- componentهای مورد استفاده در سیستم و چگونگی ارتباط آن‌ها با یکدیگر
- تکنولوژی مورد استفاده
- ساختار
- پیاده‌سازی بخشی از موارد کاربرد پر مخاطره‌تر
- مکانیزم چگونگی برآورده نمودن نیازمندی‌های غیروظیفه مهم مانند امنیت و هم‌روندی

خروجی این قسمت سیستمی کوچک و قابل اجراست<sup>۶۰</sup> که به آن "برش عمودی"<sup>۶۱</sup> سیستم نیز می‌گویند.

**نکته:** معمار سیستم نرم‌افزاری برای معماری خود یک سری راه‌بردها و رهنمون‌هایی<sup>۶۲</sup> می‌سازد تا طراحان<sup>۶۳</sup> بر اساس آن کار کنند. همچنین در حین انجام کار و بزرگ‌شدن پروژه از طریق اعمال یک سری تست‌های از پیش تعریف شده بر آورده شدن این راه‌بردها نظارت می‌کند.

#### ۲-۴ ترتیب روال تولید سیستم نرم‌افزاری در RUP

##### ۱-۲-۴ Disciplineها

۱. Business Modeling
۲. Requirement Management
۳. Analysis & Design
۴. Implementation
۵. Test

<sup>56</sup> Special Requirements

<sup>57</sup> Use case description

<sup>58</sup> Response Time

<sup>59</sup> Supplementary Specifications

<sup>60</sup> Executable Prototype

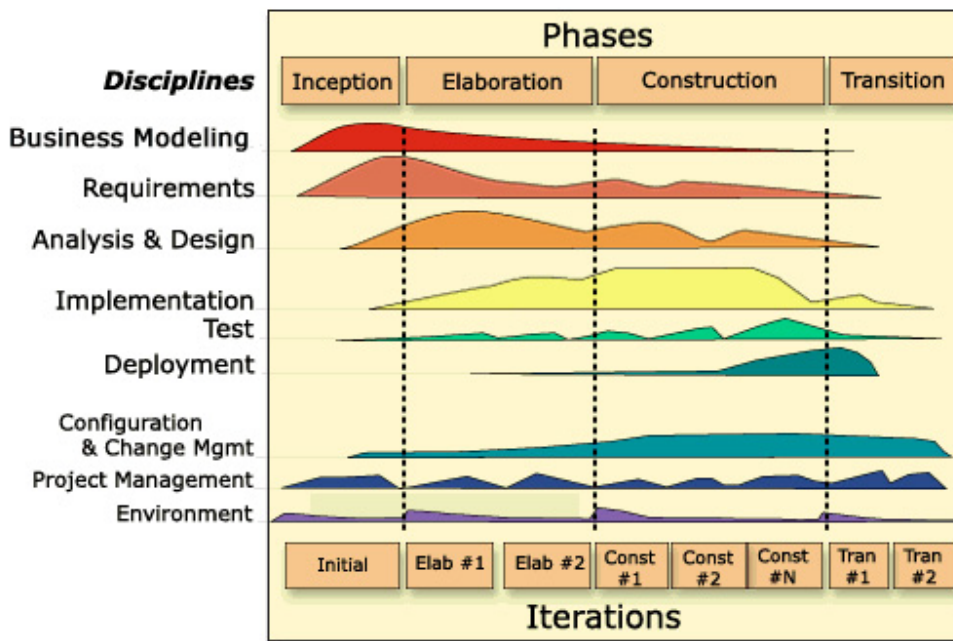
<sup>61</sup> Vertical Slice

<sup>62</sup> Guidelines

<sup>63</sup> Designers

- ۶. Deployment
- ۷. Configuration & Change Management
- ۸. Project Management
- ۹. Environment

در روالی مانند RUP موارد فوق محور عمودی را تشکیل می‌دهند که مستقل از زمان بوده و کارهایی که باید در هر مرحله زمانی<sup>۶۴</sup> انجام شوند را نشان می‌دهند. حال اگر پارامتر زمان را هم که از ۴ مرحله زیر تشکیل شده است، به عنوان محور افقی اضافه نمائیم نمودار زیر حاصل خواهد شد که مبنای اصلی حرکت در RUP خواهد بود.



- ۱. Inception
- ۲. Elaboration
- ۳. Construction
- ۴. Transition

ارتفاع تپه‌های شنی در این شکل نشان دهنده میزان و حجم کار یک Discipline در فازهای مختلف می‌باشد.

۴-۲-۲ فازها

#### ◀ هدف از مرحله Inception چیست؟

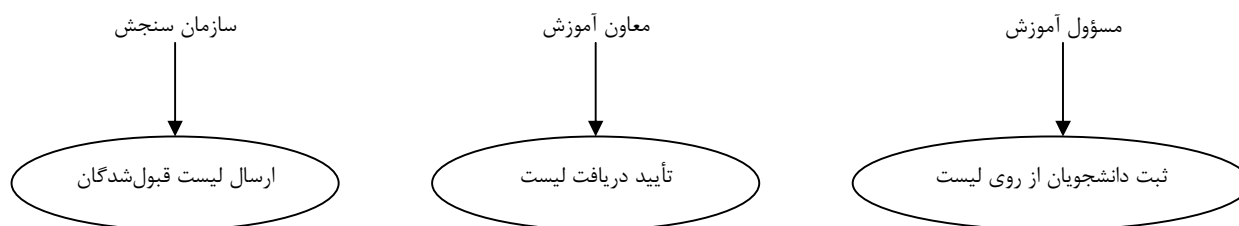
در این مرحله هدف تعیین حوزه و قلمرو<sup>۶۵</sup> پروژه می‌باشد. به عنوان نمونه مثال ارسال لیست قبول شدگان از سازمان سنجش به دانشگاه را در نظر بگیرید.

**ریسک:** در این مرحله ریسک‌های مربوط به تعریف و اهداف پروژه حل می‌شود.

<sup>64</sup> Phase

<sup>65</sup> Scope

راه حل ۱:



راه حل ۲:

این راه حل می گوید که به سازمان سنجش واسط ندهیم و مثلاً اطلاعات را از طریق لوح فشرده تحویل گرفته و به سیستم آموزشی اضافه نمائیم.

حال این سؤال به وجود می آید که راه حل ۱ مناسب است یا راه حل ۲؟ پاسخ به این سؤال حوزه و قلمرو پروژه را تعیین می کند و باید در این مرحله (Inception) به آن پاسخ داده شود.

#### ◀ هدف از مرحله Elaboration چیست؟

محصول اصلی فاز Elaboration ، معماری نرم افزار<sup>۶۶</sup> است. همان طور که اشاره شد در مرحله اول (Inception) راه حلی برای اجرای پروژه پیشنهاد می شود. در این مرحله (Elaboration) با بررسی معماری کلی سیستم و پیاده سازی موارد کاربرد پرمخاطره تر سعی می کنیم به این سؤال پاسخ دهیم که آیا راه حل ارائه شده شدنی هست یا خیر؟  
ریسک: در این مرحله ریسک های فنی<sup>۶۷</sup> پروژه مرتفع می شوند.

#### ◀ هدف از مرحله Construction چیست؟

در پایان فاز construction ، نسخه بتای<sup>۶۸</sup> نرم افزار حاصل می شود. در این مرحله تاکید کار بر موازی کاری است. به این معنی که موارد کاربرد بین گروه ها تقسیم می شود. این گروه ها به صورت موازی عمل می کنند.  
ریسک: در این مرحله ریسک های مربوط به ساخت سیستم نرم افزاری حل می شوند.

#### ◀ هدف از مرحله Transition چیست؟

در فاز Transition محصول بتا به سایت مشتری انتقال داده می شود.  
ریسک: در این مرحله ریسک های انتقال محصول رفع می شوند.

<sup>66</sup> architecture

<sup>67</sup> Technical Risks

<sup>68</sup> β version

**نکته:** نسخه‌ی بتا به محصولی اطلاق می‌گردد که آماده‌ی تحویل به مشتری است و حداقل ۹۰ تا ۹۵ درصد کارکرد سیستم واقعی و مطلوب را دارا می‌باشد.

#### ۳-۲-۴ جمع‌بندی

هر فاز دارای نُه discipline است. هر discipline یک semi-waterfall است، زیرا دارای بخشهایی مثل تحلیل و طراحی، پیاده‌سازی و تست است. RUP در این موارد با SSADM (فرایند آبشاری) متفاوت است:

- وجود معماری
- Use-Case Driven بودن
- Iterative بودن

تضمین کیفیت<sup>۶۹</sup> در واژگان RUP، Process Engineering خوانده می‌شود و مسؤول آن Process Engineer نامیده می‌شود. وظیفه این نقش، تعریف یک سری استاندارد و Test-Case و نظارت بر صحت و انطباق نرم‌افزار با اینهاست. وظایف مدیر پروژه برنامه‌ریزی<sup>۷۰</sup>، کنترل اجرای برنامه و نظارت بر انجام آن است.

---

<sup>69</sup> quality assurance

<sup>70</sup> plan