به نام خدا

دوره کیفیت نرم افزار جاواکاپ

مهدی نیک نژاد

موضوع جلسه : **Refactoring**

(جلسه بعد از Junit)

فهرست مطالب

مقدمه: **2**

تعاریف : **2**

فرآیند Refactoring : **4**

اشکالات ساختاری(Bad Smell) : **5**

بو های بد و تکنیک های بازآرایی : **6**

چندین نکته : **12**

تمرین های عملی در مورد تکنیک های بازآرایی : **12**

مطالب تکمیلی : **22**

مقدمه:

هدف : ارایه راهکار هایی برای افزایش کیفیت نرم افزار (بهبود ساختار در مورد تمیزی و قابل فهم بودن بیشتر)

یکی از این کار ها Refactoring یا بازآرایی هست .

در ضمن از IDE ها هم برای اتومانیک انجام دادن code refactoring استفاده میکنیم .

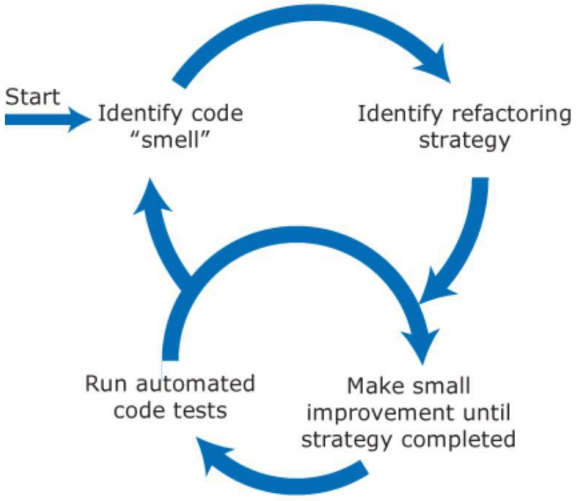
تعاریف :

- Refactoring :
  - یک فرآیند منظم برای بازسازی ساختار برنامه با هدف بهبود کیفیت کد بدون ایجاد تغییر در رفتار برنامه
  - **تغییر در ساختار داخلی نرم افزار**
  - فایده :
    - کد راحت تر فهمیده شود (خوانایی) – کاهش پیچیدگی
    - تغییر آن کم هزینه تر باشد
    - افزایش قابلیت نگهداری (maintain) نرم افزار – امکان تغییر یا گسترش برنامه
    - کاهش احتمال ایجاد باگ در آن

- **کارهایی که Refactoring نیستند :**
  - تغییر در رفتار برنامه
  - ایجاد امکانات و feature های جدید
  - رفع باگ ها – معمولا وقتی refactoring انجام میدهیم که نرم افزار به درستی کار میکند و فقط میخواهیم ساختار داخلی را بهبود دهیم
- **کارهایی که Refactoring انجام میدهد :**
  - بهبود ساختار داخلی برنامه
  - اجرای فرآیند منظم برای تمیز کردن کد
  - بهبود طراحی برنامه بعد از نوشتن کد
    - فرآیند های Agile

XP Practices : Refactoring

- Design refactoring به مهندس نرم‌افزار این امکان را می‌دهد که بدون تغییر کارکرد یا رفتار بیرونی ساختار داخلی طراحی یا کد را بهبود بخشد. به این ترتیب refactoring برای بهبود کارایی، خوانایی و عملکرد طراحی یا کد مورد استفاده قرار می‌گیرد.
- Code refactoring تمام برنامه نویسان باید به طور مداوم روی کد، به محض اینکه امکان بهبود آن وجود داشت refactoring را انجام دهند. (حتی اگر نیازی نباشد.)
- مثال‌هایی از refactoring عبارتند از: سازمان‌دهی مجدد سلسله‌مراتب کلاس‌ها برای حذف duplicate code، تغییر نام صفات و متدها، جای‌گذاری فراخوانی متدها به جای خطوط کد برنامه و ...



پس با این وجود کار هایی که یک برنامه نویس دارد انجام میدهد :

- کد نویسی برای business code
- کد نویسی برای test code
- Refactoring

فرآیند Refactoring :

- در هر مرحله ، یک اشکال ساختاری در متن برنامه را پیدا میکنیم
  - مثلا : متدی که زیادی طولانی شده است و حجم زیادی دارد
  - توجه : منظور از اشکال ، باگ نیست بلکه اشکال در ساختار نرم افزار است
    - برای اینکه تمایزی بین bug و اشکال ساختاری قایل شویم یک نامگذاری انجام شده است .
    - اشکالات ساختاری : Code smell یا Bad Smell یا **بوی بد**
    - هر بوی بد ، با تکنیکی از بازآرایی برطرف میشود

مثال :

```
Scanner s = new Scanner(System.in);

System.out.println("Rectangle Info.");
System.out.print("Enter the width: ");
int a1 = s.nextInt();
System.out.print("Enter the length: ");
int a2 = s.nextInt();

System.out.println("Rectangle Info.");
System.out.print("Enter the width: ");
int b1 = s.nextInt();
System.out.print("Enter the length: ");
int b2 = s.nextInt();

int x = a1*a2;
int y = b1*b2;

if(x == y)
        System.out.println("Equal");
```

کاری که این نرم افزار انجام میدهد این است که طول و عرض 2 تا مستطیل را میگیرد و مساحت انها را محاسبه میکند و در آخر هم چک میکند که مساحت های برابری دارند یا خیر .

اشکالات ساختاری(Bad Smell) :

- نام گذاری ها صریح نیستند مثلا .... ,a1,a2,b1 . میشد از نام های length1 , width1 استفاده کرد . **(تکنیک تغییر نام)**

- مفهوم مستطیل یا Rectangle در دل این مساله نهفته بود ، لذا بهتر است که آنرا یک کلاس معرفی کنیم
(تکنیک استخراج کلاس)

- موضوع محاسبه مساحت را داشتیم و میشود یک متد برای آن ایجاد کرد به نام area() . همانطور که دیدیم در صورتی که این مفهوم را متد تعریف نکنیم ، مدام کد تکراری داریم . پس بهتر است آنرا فاکتور گیری کنیم و یکبار تعریف کنیم و چند جا بار ها استفاده کنیم. **(تکنیک استخراج متد)**
- گرفتن اطلاعات از کاربر چند بار تکرار شده بود . میتوانیم داخل همان کلاس Rectangle ، یک متد برای همین کار در نظر بگیریم به نام readRectangle(Scanner scanner) که البته حس میکنم غلطه این کار . (تکنیک استخراج متد)

کد Refactor و بازیابی شده :

```
Scanner scanner = new Scanner(System.in);

Rectangle rectangle1 = readRectangle(scanner);
Rectangle rectangle2 = readRectangle(scanner);

int area1 = rectangle1.area();
int area2 = rectangle2.area();

if(area1 == area2)
    System.out.println("Equal");
```

توجه :

```
class Rectangle{
    ...
    public int area(){
        return length * width;
    }
}
```

```
private static Rectangle readRectangle(Scanner scanner) {
    int width;
    int length;
    System.out.println("Rectangle Info.");
    System.out.print("Enter the width: ");
    width = scanner.nextInt();
    System.out.print("Enter the length: ");
    length = scanner.nextInt();
    Rectangle rectangle2 = new Rectangle(length, width);
    return rectangle2;
}
```

مرور مثال :

- کد درست کار میکرد و فقط ساختار داخلی آنرا بهبود دادیم .
- در هر مرحله یک bad smell پیدا کردیم و آنرا با تکنیک ها برطرف کردیم .
- پس :
  - Refactoring = Find Bad smell + Figure it out with techniques

بو های بد و تکنیک های بازآرایی :

- بوی بد :
  - علامتی که نشان از یک مشکل عمیق در برنامه دارد
  - خطایی در ساختار برنامه است که ظاهرا تا الان مشکلی ایجاد نکرده است
  - باگ نیست ولی در آینده ، روند توسعه را پرهزینه میکند
  - توسط تکنیک های بازآرایی قابل رفع است
- انواع بوی بد :
  - **کد تکراری (duplicated code) :**
    - قطعه کد یکسان یا خیلی مشابه که بیش از یکجا تکرار شده است
    - مشکلات :

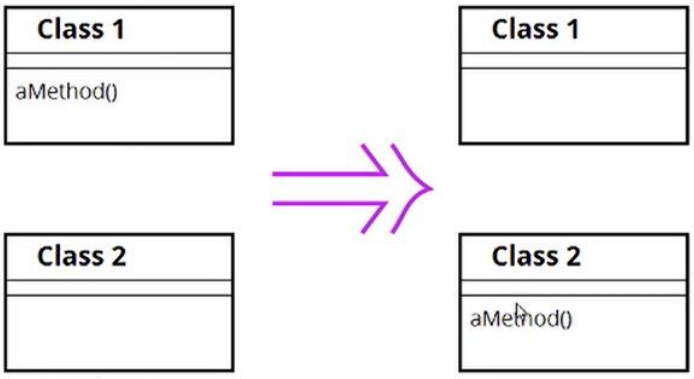
- تعییر در این بخش مستلزم تغییر همه تکرار ها است – پس از copy-paste استفاده نکنید !
- راه حل ها و تکنیک های بازآرایی :
  - استخراج متد
  - استخراج کلاس
  - استخراج متغیر

- **متد طولانی (long method)**
  - به سختی پیدا و مشخص میشوند - چون قانونی برای اینکه متد با چند خط طولانی هست یا خیر نداریم!
  - تغیر آنها سخت است
  - یک متد خوب ، کار منسجم و مستقل انجام میدهد نه چند کار مختلف !
    - انسجام بالا (High cohesion)
    - استقلال کم (low coupling)
  - راه حل ها و تکنیک های بازآرایی :
    - استخراج متد – بخش های مختلف این متد را متد میکنیم و فقط در اینجا صدا میکنیم .
- **کلاس طولانی :**
  - راه حل ها و تکنیک های بازآرایی :
    - استخراج کلاس
- **تعداد زیاد پارامتر های یک متد :**
  - راه حل ها و تکنیک های بازآرایی :
    - فراخوانی متد به جای پاس شدن مقدار پارامتر
    - تبدیل مجموعه پارامتر ها به یک شی
- **Feature Envy (حسادت به داشته های دیگران)**
  - کلاسی ، بیش از حد از متد های یک کلاس دیگر استفاده کند
  - راه حل ها و تکنیک های بازآرایی :
    - تکنیک انتقال متد [1]
      - میتوانیم آن متد را به همین کلاس منتقل کنیم چون داره بیش از حد از آن استفاده میکنه پس حتما در طراحی اشتباهی شده و این متد متعلق به این کلاس هست !
- **تغییر واگرا (Divergent Change) :**
  - اگر برای هر نوع تغییر مجبور شویم که بخش های مختلفی از یک کلاس را تغییر دهیم .
- **جراحی مجروح تفنگ ساچمه ای (Shotgun Surgery) :**
  - اگر برای هر نوع تغییر ، مجبور باشیم بخش های کوچکی از کلاس های مختلف را تغییر دهیم
  - وجه تسمیه : تعداد زیادی خرده گلوله به یک نفر برخورد کرده است
- **کلاس تنبل (Lazy class)**
  - کلاسی که آنقدر کوچک است که کلاس مستقل به حساب نمی آید و باید با کلاس های دیگر ترکیب شود

[1] **رجوع کنید به بحث "انتقال" در صفحه بعد**

- **data clumps**[2]
  - گروهی از داده ها که در کنار هم استفاده میشوند . مثلا : نام – نام خانوادگی – شناسه – رمز
  - *من حقیقتا نمیدانم مشکل این چی هست ؟ و کجایش bad smell است ؟*
- **و ....**

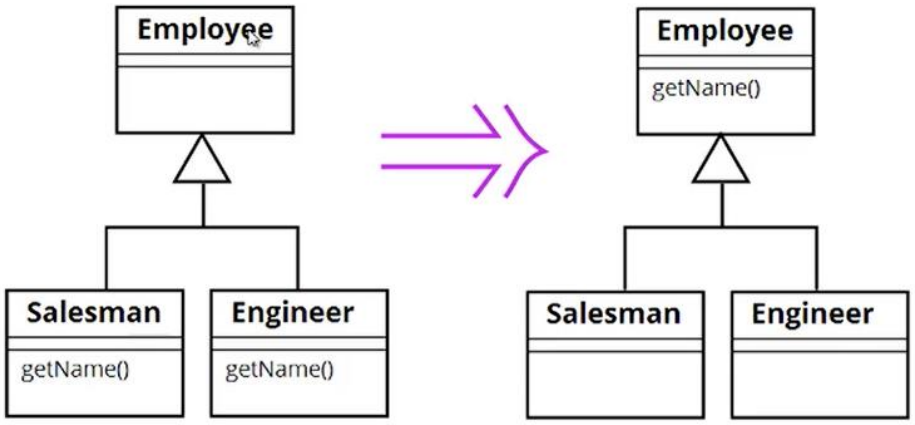
- تکنیک های بازآرایی :
  - **تغییر نام**
    - برای کلاس ، متد و متغیر
    - همه جاهایی که این استفاده شده باید replace شود نامش
    - ابزار های اتوماتیک در IDE ها برای این مورد هست : Refactor -> Rename
  - **استخراج متد (Extract method)**
    - از یک تعداد خط در برنامه ، یک متد در بیاوریم
    - در جاهایی که یک تکه کد دارد مدام تکرار میشود ، کاربرد دارد
  - **متد در خط (inline method)**
    - برعکس استخراج متد
    - فرضا متدی که ساختیم بسیار کوتاه و بدیهی است ، می آییم محتوای متد را داخل برنامه می آوریم و آن متد را حذف میکنیم
  - **استخراج کلاس**
  - **کلاس در خط**
  - **انتقال**
    - برای کلاس ، متد و متغیر
    - یک کلاس به نام class1 داریم و دارای متد aMethod() هست و به مرور زمان میفهمیم که در مورد class2 ، Bad smell داریم و class2 دارد بیش از حد از aMethod() استفاده میکند و دارای بوی بد Feature Envy هست پس بهتر است که این متد را به class2 انتقال دهیم .



[2] **رجوع کنید به بحث "ارایه شی پارامتر" در سه صفحه بعد**

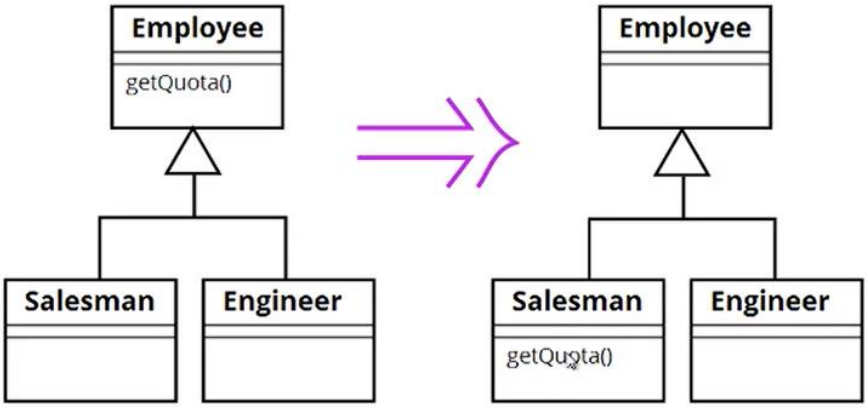
- **بالا کشیدن متد (pull up method)**

مثال : در اینجا یک سلسله مراتب را میبینیم و 2 تا کلاس Salesman و Engineer جفتشان ، متد getName() را دارند هر 2 هم از Employee ارث بری میکنند پس بهتر است که متد getName() را بالا بیاوریم و به Employee بدهیم . این طوری همه زیر کلاس های Employee ، هم این متد را دارند .



- **پایین آوردن متد**
  - برعکس pull up
  - مثال:

کلاس پدر، یک متد دارد که فقط یکی از فرزندانش به آن احتیاج دارد پس بهتر است آنرا فقط برای آن کلاس در نظر بگیریم .



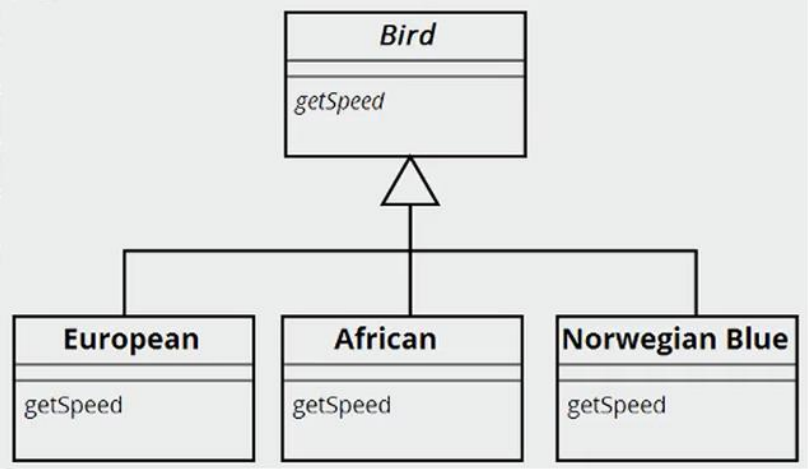
- **استخراج مقدار ثابت**
  - وقتی که یک ثابت در متن برنامه داریم ، خوب است که برای آن یک متغیر final در نظر بگیریم و استخراج شوند . مثلا وقتی در یک for ، مینویسیم از i=2 تا i=10 . این 2 و 10 چی هستند باید در متن برنامه مشخص شده باشند .

- باید به مقادیر ثابت اسم بدهیم و برایشان کامنت بنویسیم و یا مستند بنویسیم و کلا توضیحشان دهیم و منظورمان را بنویسیم . مثلا : pi = 3.14 را بنویسیم نه اینکه a=3.14

- **تغییر امضای متد (Change method signature)**
  - امضای متد شامل نام و پارامتر های متد میشود
  - *کاربردش کجا هست ؟*
  - کم و زیاد کردن پارامتر
  - تغییر دادن نوع برگشتی
  - تغییر دادن سطح دسترسی
  - تغییر خطاها Exception
- **تبدیل شروط به چندریختی و ساختار سلسله مراتبی**
  - طبق مثالی که میبینیم یک switch – case داریم که به ازای هر شرطی یک رفتاری داریم . در این متد getSpeed() ، یک switch-case داریم و در getWeight() هم دوباره همین switch-case را داریم و همینطور در getLength() و .... لذا بهتر است که از polymorphism استفاده کنیم .

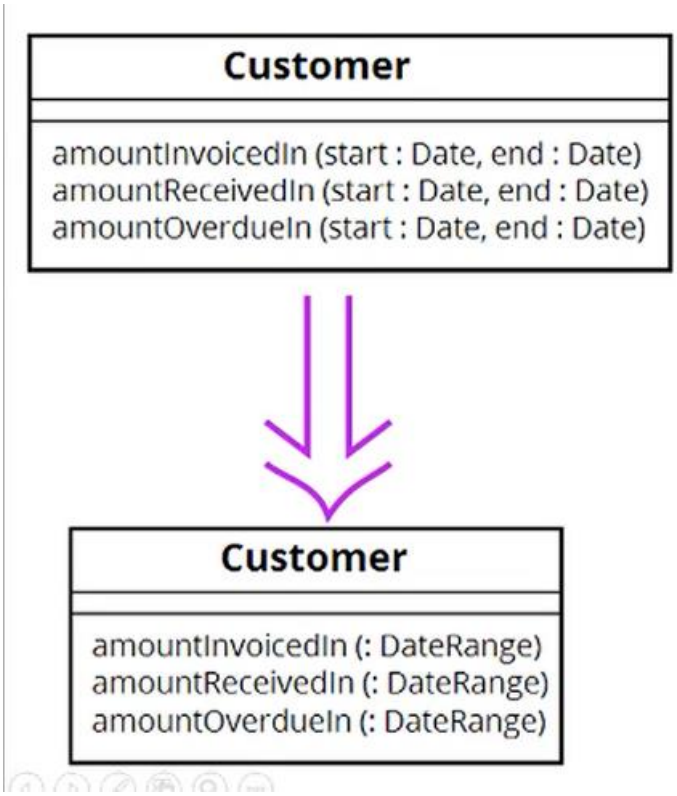
```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
}
```

یعنی بهتر است که ساختار زیر را داشته باشیم :



یعنی متد getSpeed() در European یک جور پیاده سازی شود و آنهای دیگر جور های متفاوت !
پس بهتر است که این شروط از دل کلاس بیرون برود و هر متد خودش نخواهد تصمیم بگیرد که نوع آبجکت را تشخیص دهد و متناسب با آن کاری انجام دهد و بلکه بهتر است از polymorphism استفاده کنیم.

- **تکنیک ارایه شی پارامتر**
  - وقتی که تعدادی پارامتر همیشه با هم استفاده شوند ، میتوانیم آنها را به عنوان داده جدید در نظر بگیریم .مثلا در اینجا یک بازه از تاریخ داریم . تاریخ شروع پایان .و میتوانیم آبجکت DateRange داشته باشیم .



چندین نکته :

- به سایت http://refactoring.com/catalog/ سر بزنید
- استعاره 2 کلاه :

در هنگام برنامه نویسی پیشنهاد میشود که برای خودمان نقش (کلاه) فرض کنیم یعنی زمان خودمان را به 2 بخش مجزا تقسیم کنیم :

  - تولید برنامه
  - بازآرایی

توجه کنید که باید وقت جداگانه ای برای بازآرایی کنار بگذاریم یعنی در هنگام تولید برنامه ، نباید درگیر بازآرایی شویم و در هنگام بازآرایی نباید امکانات جدید ایجاد کنیم . و باید دفعات زیاد و به سرعت نقش عوض کنیم بین این 2 حالت . اما هر نقش باید به طور مستقل ایفا شود .

- IDE ها امکاناتی برای Refactoring دارند که خودکار و سریع هستند البته خودکار بودن این طور نیست که از IDE بخواهیم تا کد ما را بازآرایی کند بلکه باید خودمان دانش بازآرایی و شناخت bad smell ها را داشته باشیم . البته باز هم شناسایی bad smell ، ابزار مخصوص هم دارد .

تمرین های عملی در مورد تکنیک های بازآرایی :

مثال 1:

در این مثال ، ابتدا کدی که زدیم را با کلاه برنامه نویس مینویسیم و بعد کلاه را عوض میکنیم و به نقشی برای بازآرایی(کلاه بازآرایی) میرویم و شروع میکنیم به پیدا کردن bad smell ها .

کد :

```
import java.util.*;
public class AnimalFarm {
	public static void main(String[] args) {
		Scanner s = new Scanner(System.in);

		System.out.println("Enter type and name of the animal");
		String t = s.next();
		String n = s.next();

		String voice = null;
		switch(t.toLowerCase()) {
		case "cat":
			voice = "Meew";
			break;
		case "dog":
			voice = "Hop";
			break;
		}
		System.out.println(voice);

		if ("cat".equalsIgnoreCase(t)) {
			tanbali();
		}else if ("dog".equalsIgnoreCase(t)) {
			hefazat();
		}

	}

	private static void hefazat() {
		System.out.println("...Hefazat");
	}

	private static void tanbali() {
```

```
                System.out.println("...Tanbali");
        }

}
```

این برنامه ساده را اجرا میکنیم و میبینیم که مشکلی ندارد و اجرا میشود .

Bad smell ها :

- اسم های کوتاه و ناواضح :
  - مثلا t و n
    - راه حل : راست کلیک + Refactor + Rename
    - پس میتوانیم به جای t ، از type و به جای n ، از name ، استفاده کنیم .
  - در ضمن میتوانیم ، متد های finglish که نوشته ایم را هم به صورت انگلیسی بنویسیم
    - پس میشود به جای tanbali() ، نوشت laze()
- متغیر های type و name دارند در کنار هم مفهوم حیوان را بازنمایی میکنند
  - راه حل : میتوانیم کلاس Animal را ایجاد کنیم با متغیر های type و name .
    و همچنین با alt + shift + S میتوانیم getter,setter,constructor ایجاد کنیم .
    حالا باید تغییراتی در متن برنامه اصلی ایجاد کنیم .
- مقدار های ثابت و تکراری وجود دارند مثلا "cat" یا "dog" و ....
  - علتی که هست این است که احتمال اشتباه نوشتن یکی از این ثابت ها وجود دارد ولی اگر یک متغیر برای آن داشته باشیم بهتر است .
  - راه حل : استخراج متغیر با استفاده از : Refactor + Extract Constant و در نتیجه مثلا `private static final String CAT = "cat";` ساخته میشود .
- بخشی که دارد صدا (voice) را هندل میکند ، میتواند خودش مستقل باشد و هست .
  - راه حل : پس بهتر است که این بخش را متد کنیم . با استفاده از Refactor + Extract method
  - و خواهیم داشت :

```
String voice = voiceOf(animal);

....

private static String voiceOf(Animal animal) {
          String voice = null;
          switch(animal.getType().toLowerCase()) {
          case CAT:
                voice = "Meew";
                break;
          case DOG:
                voice = "Hop";
                break;
          }
          return voice;
}
```

- متد هایی که extract کردیم مثل voiceOf() و jobOf() ، مفاهیمی هستند که در داخل کلاس Aimal معنا دارند .یعنی رفتار های یک حیوان هستند .
  - راه حل : انتقال این 2 تا متد به کلاس Animal
  - اما در خلال این انتقال، اتفاقاتی می افتد در کلاس Animal و مشکلاتی پیش می آید که باید آنها را رفع کنیم :
    - چون داریم منتقل میکنیم دیگه لازم نیست که private static باشند و public میکنیم .
    - یک اشکال این است که متغیر های CAT و DOG ، دیگه قابل استفاده نیستند و همچنین متد های laze() و protect() . پس اینها را هم منتقل میکنیم .
  - حالا خود متد main() هم به مشکلاتی میخورد که باید رفع گردد :
    - قبلا میگفتیم voiceOf(animal) اما الان باید بگیم animal.voiceOf() چون که شی ساخته شده از کلاس Animal ، یعنی animal هست که این متد را دیگه دارد . و باید پارامتر هایش را هم حذف کنیم و به جای animal ها در کلاس Animal ، this بگذاریم :

```
String voice = animal.voiceOf();
System.out.println(voice);

animal.jobOf();
```

```
public void jobOf() {
    if (CAT.equalsIgnoreCase(this.getType())) {
        laze();
    }else if (DOG.equalsIgnoreCase(this.getType())) {
        protect();
    }
}

public String voiceOf() {
    String voice = null;
    switch(this.getType().toLowerCase()) {
    case CAT:
        voice = "Meew";
        break;
    case DOG:
        voice = "Hop";
        break;
    }
    return voice;
}
```

- توجه : خیلی خوب است که قدم به قدم با refactoring ، برنامه را اجرا کنیم ، ببینیم که درست اجرا میشود یا نه . البته بعدا میبینیم که باید unit test داشته باشیم و آنها مدام اجرا شوند تا مطمین شویم رفتار نرم افزار همچنان درست میباشد و با refactoring خراب نشده اند .
- قدم بعدی استفاده کردن از polymorphism هست . اگر به متد های jobOf و voiceOf نگاه کنیم ، میبینیم که هی داریم میگیم اگه cat بود این کار رو بکن و اگه dog بود این کارو بکنه . پس معلومه که طراحی مون خوب نبوده و ساختار سلسله مراتبی درستی نداریم.
    - راه حل : بهتر است کلاس های Dog و Cat ، را هم داشته باشیم . به این شکل :

```
class Dog extends Animal{

   public Dog(String type, String name) {
          super(type, name);
          // TODO Auto-generated constructor stub
   }
}

class Cat extends Animal{
   public Cat(String type, String name) {
          super(type, name);
          // TODO Auto-generated constructor stub
   }
}
```

و حالا باید تغییراتی بدهیم در جایی که داریم از کاربر ، اطلاعات حیوان را میگیریم . به این شکل :

اما همانطور که میبینیم مجبور هستیم که متغیر های CAT, DOG را دوباره به کلاس AnimalFarm بیاوریم .

```java
public class AnimalFarm {
    private static final String DOG = "dog";
    private static final String CAT = "cat";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter type and name of the animal");
        String type = scanner.next();
        String name = scanner.next();
        Animal animal = null;
        switch(type.toLowerCase()) {
        case CAT:
            animal = new Cat(type, name);
            break;
        case DOG:
            animal = new Dog(type, name);
            break;
        }

        String voice = animal.voiceOf();
        System.out.println(voice);

        animal.jobOf();

    }
```

- حالا میتوانیم کل فرآیند گرفتن اطلاعات حیوان را متد کنیم

```java
private static Animal readAnimalInfo() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter type and name of the animal");
        String type = scanner.next();
        String name = scanner.next();
        Animal animal = null;
        switch(type.toLowerCase()) {
        case CAT:
            animal = new Cat(type, name);
            break;
        case DOG:
            animal = new Dog(type, name);
            break;
        }
        return animal;
}
```

- حالا باید فرآیند `polymorphism` را تکمیل کنیم و متد های `voiceOf` و `jobOf` را برای هر کدام از کلاس های `Cat` و `Dog` ، پیاده سازی کنیم :

در ضمن میدانیم که کلاس `Animal` ، باید کلاس `abstract` باشد و انتزاعی .

مثلا متد `jobOf` را باید به زیر کلاس ها ببریم و باید آنرا `abstract` تعریف کنیم. و همچنین `voiceOf`

در ضمن متد های `laze` و `protect` هم باید منتقل کنیم هر کدام را به کلاس خودشان . ولی قبلش `protected` شان کردیم. چرا ؟ `protected` چی بود ؟

این طوری دیگه اون `switch-case` و `if-else` ها را نداریم و از شرشان خلاص شدیم . آنها را `comment` کردم در کد

در آخر هم بنا به منطق تشخیص دادم که نام متد `jobOf` به `job` تغییر پیدا کند و همچنین `voiceOf` به `voice`

و در نهایت به این شکل درآمد :

```java
import java.util.*;

class Dog extends Animal{

	public Dog(String type, String name) {
		super(type, name);
	}

	protected void protect() {
		System.out.println("...Hefazat");
	}

	@Override
	public void job() {
		protect();
	}

	@Override
	public String voice() {
		return "Hop";
	}

}

class Cat extends Animal{
	public Cat(String type, String name) {
		super(type, name);
	}

	protected void laze() {
		System.out.println("...Tanbali");
	}

	@Override
	public void job() {
		laze();
	}

	@Override
	public String voice() {
		return "Meew";
	}

}


abstract class Animal{
	private String type;
	private String name;

	private static final String DOG = "dog";
	private static final String CAT = "cat";
```

```
	public Animal(String type, String name) {
		this.type = type;
		this.name = name;
	}

	public String getType() {
		return type;
	}

	public void setType(String type) {
		this.type = type;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}




	public abstract void job();
//	{
//		if (CAT.equalsIgnoreCase(this.getType())) {
//			laze();
//		}else if (DOG.equalsIgnoreCase(this.getType())) {
//			protect();
//		}
//	}

	public abstract String voice();
//	{
//		String voice = null;
//		switch(this.getType().toLowerCase()) {
//		case CAT:
//			voice = "Meew";
//			break;
//		case DOG:
//			voice = "Hop";
//			break;
//		}
//		return voice;
//	}


}
public class AnimalFarm {
	private static final String DOG = "dog";
	private static final String CAT = "cat";

	public static void main(String[] args) {
		Animal animal = readAnimalInfo();
```

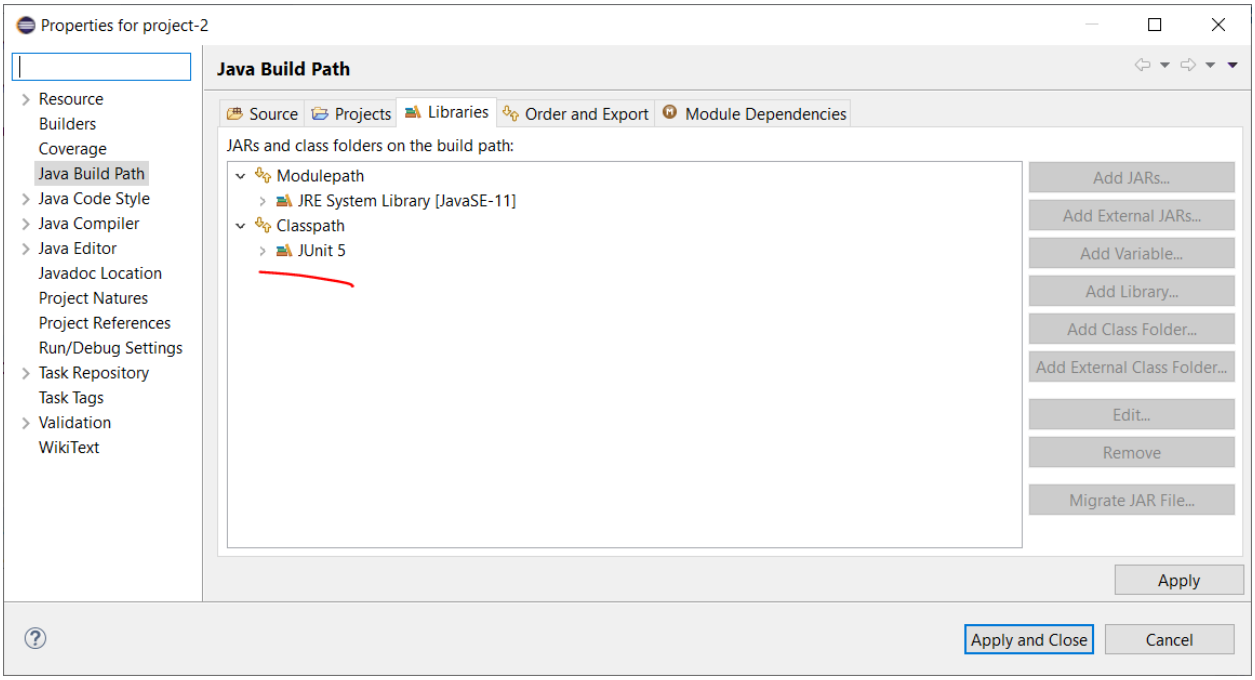
```
		String voice = animal.voice();
		System.out.println(voice);

		animal.job();

	}

	private static Animal readAnimalInfo() {
		Scanner scanner = new Scanner(System.in);

		System.out.println("Enter type and name of the animal");
		String type = scanner.next();
		String name = scanner.next();
		Animal animal = null;
		switch(type.toLowerCase()) {
		case CAT:
			animal = new Cat(type, name);
			break;
		case DOG:
			animal = new Dog(type, name);
			break;
		}
		return animal;
	}

}
```

مثال 2:

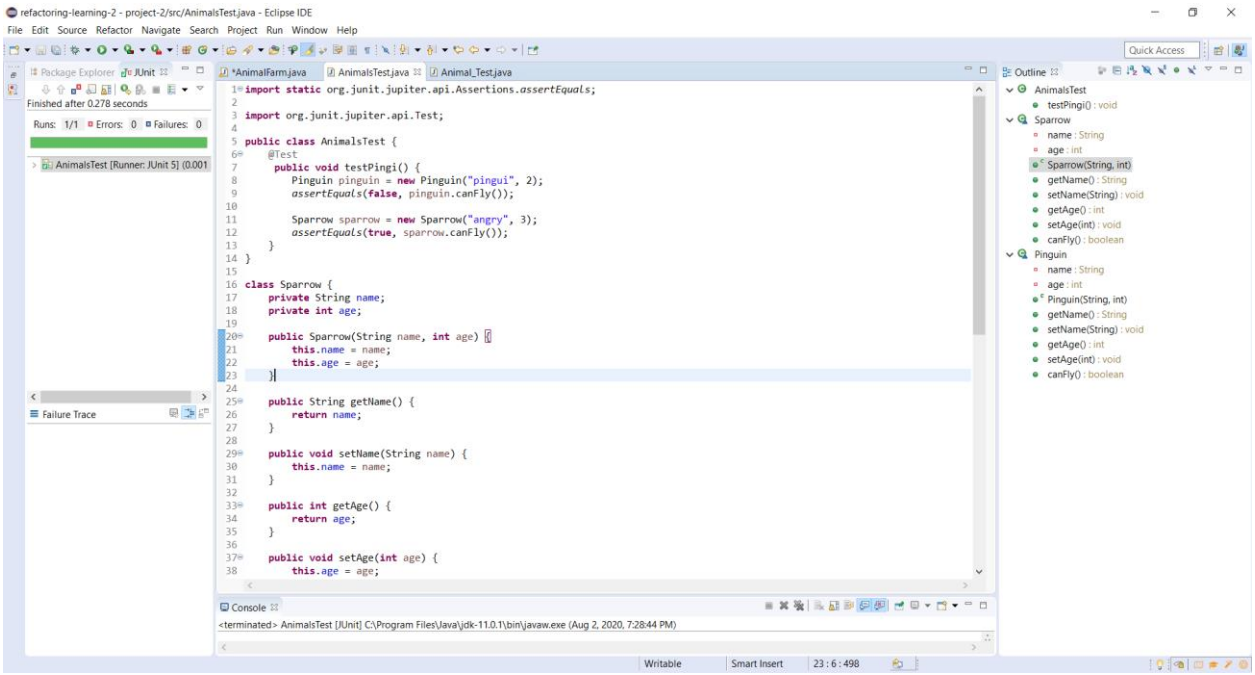
تاثیر `unit test` و `Refactoring` :

توجه : برای اینکه کد ها را در eclipse ، فرمت کنیم و منظم کنیم : Ctrl + shift + F

توجه کنید که Junit اینجا باشد:



در اینجا یک کلاس داریم برای تست کردن و یک متد هم برای تست کیس داریم و همچینن ، 2 تا کلاس هم داریم برای گنجشک و پنگوین .




```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class AnimalsTest {
	@Test
	 public void testPingi() {
		Pinguin pinguin = new Pinguin("pingui", 2);
		assertEquals(false, pinguin.canFly());

		Sparrow sparrow = new Sparrow("angry", 3);
		assertEquals(true, sparrow.canFly());
	}
}

class Sparrow {
	private String name;
	private int age;

	public Sparrow(String name, int age) {
		this.name = name;
		this.age = age;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
```

```
		this.name = name;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public boolean canFly() {
		return true;
	}
}

class Pinguin {
	private String name;
	private int age;

	public Pinguin(String name, int age) {
		this.name = name;
		this.age = age;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public boolean canFly() {
		return false;
	}
}
```

همانطور که میبینیم تست ها پاس میشوند . پس کد صحیح هست و حالا میخواهیم آنرا بازیابی کنیم :

- میخواهیم از محتوای کلاس Sparrow ، یک superclass در بیاوریم. این کار را با Refactor + Extract Superclass انجام میدهیم . و بعد برای Pinguin هم همین کار را انجام میدهیم . و میبینیم که تست ها پاس نمیشوند این دفعه یعنی یکی شان میشه یک نه . چون نیومدیم در canFly() مشخص کنیم که اگه Bird عه ما پنگوین بود false برگررونه و اگه گنجشک بود ، true برگردونه . حالا برای کلاس Pinguin ، باید متد canFly() را Override کنیم.

```
class Pinguin extends Bird{
   public Pinguin(String name, int age) {
          this.name = name;
          this.age = age;
   }

   @Override
   public boolean canFly() {
          return false;
   }
}
```

مطالب تکمیلی :

- **زمان مناسب برای بازآرایی :**
  - در این باره قانون 3 گانه داریم :
    - وقتی امکان جدید (new feature) به برنامه اضافه میکنیم
    - وقتی یک bug را برطرف میکنیم – باید ببینیم که حل این مشکل ، مشکل جدیدی ایجاد نکرده باشد
    - وقتی داریم کد ها را مرور میکنیم یعنی code review انجام میدهیم .

      مثلا در همان جلسات technical review ؟

      توجه : فرآیند code review کار مهمی در متدلوژی Agile هست

      توجه : code review به صورت اتوماتیک هم انجام میشود بعضا در مورد پیدا کردن bad smell ها
- **تاثیر بازآرایی در کارایی :**
  - انتقاد ها به بازآرایی :
    - بازآرایی ، performance و کارایی برنامه را کاهش میدهد
      - تعداد متغیر ها ، کلاس ها و فراخوانی ها ی جدید و گرفته شدن فضای حافظه و ....

      اما در واقع توجه داریم که بعضی تکنیک های بازآرایی ، کارایی را افزایش هم میدهند . منظور این هست که اگر extract method داریم ، inline method هم داریم .

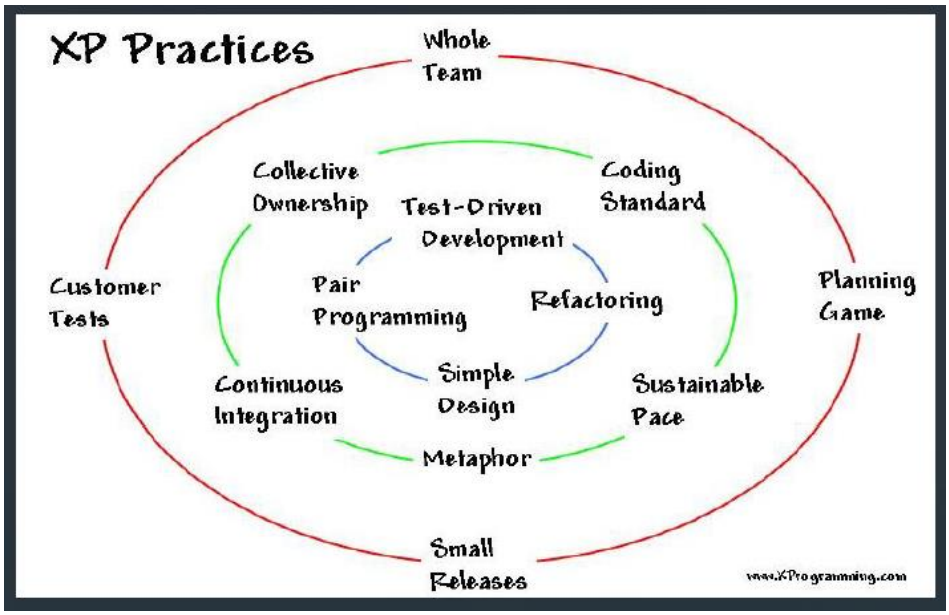
      و همچنین ، تاثیر تکنیک های بازآرایی روی کارایی تاثیر چندانی ندارد .
  - فایده بازآرایی :
    - ساختار کد قابل بهبود میشود – هدف از بازآرایی ، نگهداری کد بود و یکی از جنبه های maintain و نگهداری برنامه همین قابل بهبود بودن هست .
- *توصیه : ابتدا نرم افزار قابل بهبود بنویسید و بعد در صورت لزوم ، آنرا بهبود دهید برای کارایی بالاتر*
- **ریسک بازآرایی :**
  - بازآرایی تا حدی ریسک است چرا که داریم ، برنامه ای که صحیح بوده را تغییر میدهیم و ممکن است این فرآیند باعث ایجاد bug شود .
  - چه طور مدیر پروژه را متقاعد کنیم که باید این ریسک را انجام دهیم ؟
    - **مارتین فاولر :** اگر مدیر شما یک فرد فنی نیست ، لازم نیست تا از او برای بازآرایی اجازه بگیرید چون بازآرایی بخشی از کار شما هست و زمانی که صرف بازآرایی میکنید ، تولید آینده برنامه را تسریع میکند و زمان بیهوده ای نیست .
  - نحوه مهار کردن خطر های بازآرایی :
    - انجام تکنیک های refactoring به صورت سیستماتیک

- استفاده از ابزار ها و امکانات IDE ها
- انجام فدم های کوچک – در هر مرحله تست کنیم که برنامه همچنان صحیح باشد
- استفاده از unit testing
- توجه : در انجام بازآرایی ، شجاع باشید . چرا که :
  - 5 ارزش در متدلوژی XP :
    - تعامل (ارتباطات)
    - سادگی
    - بازخورد (فیدبک از مشتری)
    - شجاعت
    - احترام

XP Values

Value	توضیح
Customer involvement	از طریق همراهی مداوم مشتری با تیم تولید انجام می پذیرد. یک نماینده مشتری در تولید شرکت داده می شود که از طریق آن نیازمندی‌ها شناسایی و تست‌های پذیرش تعریف می‌گردند.
Incremental delivery	از طریق انتشارهای کوچک و مکرر پشتیبانی می‌شود. کارکردهایی که در هر increment سیستم قرار می‌گیرند بر مبنای user story ها و سناریوهایی می‌باشد که نیازمندی‌های کاربر را در بر می‌گیرند.
People not process	از طریق pair programming، collective code ownership، sustainable pace پشتیبانی می گردد.
Embrace change	از طریق انتشار های منظم سیستم به مشتری، test first development ، refactoring و یکپارچه سازی مداوم کارکردهای جدید پشتیبانی می شود.
Maintain simplicity	از طریق refactoring کد که موجب بهبود کیفیت آن می شود و هم چنین با طراحی ساده که تغییرات آینده غیر ضروری را پیش‌بینی نمی‌کند پشتیبانی می‌شود.

- **جایگاه بازآرایی در متدلوژی های چابک (Agile) :**
  - گام های XP :



- whole team : تمام مشارکت کنندگان در پروژه XP ، اعضای تیم هستند یعنی هم تیم توسعه و هم نماینده مشتری
- Release planning : در این مرحله تیم توسعه مشخص میکند که بعد از چند iteration ، release بدهیم و همچنین در هر iteration ، چه چیزی تحویل دهیم – user story , value , split story
- Iteration planning : برنامه ریزی کردن وظایف تیم توسعه برای مدت زمان مشخص
- small release : در هر iteration ، مجموعه ای حداقلی از useful functionalities که ارزش کسب و کاری فراهم میکند ، تولید میشود.
- Acceptance customer test : تست های پذیرشی که مشتری انها را انجام میدهد .همچنین توجه کنید که تست های پذیرش ، تست های black boxهستند .
- Simple design : طراحی ساده بر مبنای قانون keep it simple – اگر لازم باشد تا طراحی بهبود داشت باشد بعدا روی آن بازارایی را انجام میدهیم .
- Pair programming : برنامه نویسی 2 نفره
- Test first development : نوشتن تست پیش از آغاز کد نویسی
- Collective code ownership : تمام افراد در تمام بخش های کد سیستم کار میکنند و همگی مالک کد هستند نه یک نفر و در نتیجه هر فرد میتواند هر بخش از کد را که لازم باشد ، تغییر دهد .
- Coding standard : کد زدن کل تیم بر اساس یک سری استاندارد
- Methaphor : روش نامگذاری مشترک برای کلاس ها و متد ها
- Sustainable pace : اعضای تیم XP ، نباید بیش از 40 ساعت در هفته کار کنند
- Continious integration : تیم XP ، سیستم را همیشه یکپارچه نگه میدارد و به محض اینکه کار یکtask ، به پایان رسید ،آنرا با کل سیستم یکپارچه میکند

- بازآرایی جایگاه بالایی در متدلوژی XP دارد .
- در این روش XP ، خیلی سریع به محصول میرسیم و کمتر روی فاز های طراحی اولیه گیر میکنیم .
- Refactoring , unit testing , code review, pair programming و ..... همگی با هم باعث بهبود کد میشوند .
- متدلوژی چابک ، تغییر را میپذیرند – تغییر در طراحی و نیازمندی ها و ...

- **مخالفان بازآرایی :**
  - عقب بودن پروژه از زمانبندی !!! 😐
  - هدر رفتن زمان برای انجام بازآرایی !!! 😐
  - وظیفه برنامه نویس ، بازآرایی نیست !!! 😐

توجه : بازآرایی ، یک گام به عقب رفتن نیست بلکه مثل دورخیز میماند

- **بازآرایی برای انطباق با الگو های طراحی :**
  - الگوی طراحی (design pattern) : در مورد مسایل طراحی نرم افزار ، راه حل ارایه میکنند
  - anti pattern یا پادالگو : روش های رایج ولی غلط
  - تفاوت **"بوی بد"** با **"پادالگو"** :
    - پادالگو ها مفاهیم سطح بالاتری هستند و طراحی را مدنظر دارند اما بوی بدها خود کد را بیشتر مد نظر دارند
    - پادالگو ها معمولا توسط ابزار های خودکار تشخیص داده نمیشوند اما ابزار هایی برای تشخیص بوی بد داریم
    - تکنیک های بازارایی معمولا برای پادالگو ها خودکار نیستند اما IDE ها این تکنیک ها را برای بو های بد خودکار انجام میدهند
    - تشخیص محل استفاده از design pattern ها سطح بالا است و نیاز به تجربه برنامه نویس دارد
    - اما این 2 مفهوم گاها یکی هستند :
      - Large class
        - یک bad smell هست
        - راه حل : Refactoring
      - God object
        - یک پادالگو هست
        - این شی کارهای زیادی میتواند انجام دهد
        - راه حل : Design pattern
- در اینجا مفهوم **refactoring to patterns** پیش می آید یعنی اجرای تکنیک های refactoring برای رعایت شدن الگو های طراحی در برنامه (باز آرایی به منظور رعایت الگو های طراحی)
  - در فرآیند تولید نرم افزار خیلی سریع پیش میرویم (مخصوصا در چابک) لذا امکان دارد که الگو های طراحی رعایت نشوند
  - و گاهی از عمد از over-engineering پرهیز میکنیم و سعی میکنیم طراحی برنامه زیاد منطبق بر الگو ها نباشد

- **دردسر های بازآرایی :**

بازآرایی مشکلاتی میتواند داشته باشد و هزینه هایی را میتواند در بر داشته باشد (خطر ها و ریسک های بازآرایی) . لذا باید تبعات بازآرایی بررسی شوند و تمهیداتی برایشان اندیشیده شود

  - موضوع interface :
    - وقتی interface ها را refactor میکنیم ، ممکن است مشکلاتی رخ دهد
    - اینترفیس ها واسط هایی هستند که همه به آن وابسته هستند و در حال استفاده کردن از آن هستند
    - در این حالت مجبور میشویم که یک نسحه قدیمی را از واسط حذف کنیم و انرا deprecated در نظر بگیریم .
  - موضوع پایگاه داده ها :
    - بازآرایی دیتابیس ها کار بسیار پر هزینه ای هست
    - بازآرایی دیتابیس ، یعنی تغییر دادن طراحی جداول و schema ی دیتابیس ها
    - بسیاری از برنامه ها به شدت به دیتابیس وابسته هستند
      - مهاجرت داده های قدیمی به ساختار جدید هزینه دارد
  - موضوع بازآرایی کردن در تصمیمات سطح بالا مثل طراحی :
    - بازارایی اینها سخت هست
    - مثل انتخاب تکنولوژی

- **تحلیل استاتیک کد :**
  - یکی مورد دیگری که به ما در پیدا کردن بوی بد کمک میکند ، استفاده کردن از ابزار ها است که این کار را به صورت خودکار انجام میدهند
    - مثلا : findbugs,sonarqube , ...

در پایان گفته شد که تعدادی نمونه کد همراه فیلم وجود دارد برای انجام بازآرایی روی انها به همراه جواب پیشنهادیشان . حتما انها را پیدا و دانلود کن و انجام بده