کدنامه ۴

برنامه‌سازی پیشرفته بهار ۱۴۰۱ | دانشکده مهندسی کامپیوتر دانشگاه صنعتی شریف

اصول SOLID

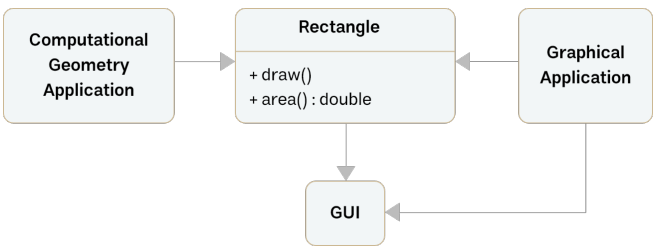
عرفان مجیبی و آرش یادگاری

اکثر ماها وقتی ترم ۲، شروع به یادگرفتن جاوا کردیم، این احساس رو داشتیم که چقدر راحت‌تر از C میشه باهاش کد زد و خب زندگی چقدر راحت شده. درسته؟
نه شاید، تمرین ۲ که مربوط به مسائل پیاده سازی بود در واقع اولین برخورد ما با کد های حجیم نسبت به کدهایی هست که قبلا زدیم. رویارویی با این مسائل برامون این شکلی آغاز میشد که از خودمون می‌پرسیدیم از کجا باید شروع کنم؟ چه کلاس‌هایی رو چجوری باید پیاده‌سازی کنم و ... . بعدش هم بدون جواب چندان مشخصی سراغ کد زدن می‌رفتیم و در آخر هم نتیجه کار ،اغلب کد بزرگی میشد که پیدا کردن باگ‌های آن به شدت سخت بود و یک تغییر کوچیک همه جای برنامه رو تحت تاثیر قرار می‌داد. جالبه بدونید که اتفاقات بالا فقط برای من و شما نیفتاده بلکه برای خیلی از برنامه نویس‌ها رخ داده. برای رفع این مشکل یه سری اصل وضع شدن که ما با رعایت اونها بتونیم کد تمیزی داشته باشیم . عجله نکنید! به زودی متوجه منظورمون میشید. SOLID یکی از این مجموعه اصوله. این قانون از ۵ اصل زیر تشکیل شده:

- **Single Responsibility Principle**
- **Open Close Principle**
- **Liskov Substitution Principle**
- **Interface Segregation Principle**
- **Dependency inversion principle**

Single Responsibility Principle (SRP)

این اصل به ما میگه که برای تغییر یک کلاس شما تنها باید یک دلیل داشته باشید. این جمله ممکنه کمی عجیب بنظر برسه. بیایید با یک مثال براتون توضیح بدم.
یادتون هست که وقتی برنامه‌نویسی شئ گرا رو یاد گرفتیم شروع به توصیف یک شیء کردیم و براش صفات و یه سری متد تعیین کردیم؟ فرض کنید برنامه‌ای داریم که قابلیت رسم مستطیل و محاسبه مساحت مستطیل رو داره. در نگاه اول ایجاد کلاسی به اسم مستطیل با ویژگی‌های زیر منطقی بنظر میرسه.



اما این طراحی به ظاهر منطقی، اصلی که گفتیم رو نقض میکنه؟ چرا؟ چون که کلاس مستطیل داره دو مسئولیت کاملا جدا رو انجام میده. به شما حق میدم که کمی بنظرتون عجیب برسه که چنین طراحی‌ای بد تلقی میشه پس بیاید باهم دلیلش رو بررسی کنیم.
سپردن مسئولیت‌های زیاد به یک کلاس باعث حجیم شدن یک کلاس و در نتیجه تبدیلش به God object میشه. همچنین کار با شیء های این کلاس زمان‌بر میشن. در مثال بالا در کلاس مستطیل باید کامپوننت‌های گرافیکی هم ایمپورت بشن که یک مشکل بزرگه (این یه اشتباهیه که ممکنه شما هم توی فاز دو پروژه‌تون مرتکب بشید! :)).
یه مثال دیگه ببینیم. این بار می‌خوایم به کمک یک interface رفتارهای یک دستگاه سانترال رو تعیین کنیم . در نگاه اول interface به این صورت خواهد بود.

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

حالا بیاید با دید SRP بهش نگاه کنیم. همونطور که میبینید، این interface به طور کلی دو مسئولیت رو انجام میده. مسئولیت اولش جابجایی اطلاعاته (send and recv) و مسئولیت دوم برقراری ارتباط (dial and hangup). پس SRP نقض شده است و بهتره این interface به دو inteface مجزا شکسته بشه.

دیدیم که SRP یک مفهوم بسیار ساده و در عین حال پیچیده‌تر در پیاده سازیه. ما معمولا به طور ناخودآگاه مسئولیت‌ها رو باهم ترکیب می‌کنیم؛ ولی حواسمون باشه که طراحی برنامه در واقع همین جداسازی مسئولیت‌هاست.

Open for extension – Close for modification (OCP)

بیاید با طرح یک داستان این اصل رو بررسی کنیم.
عده‌ای دانشجو یک ایده فوق العاده به ذهنشون میرسه و تصمیم میگیرن یک استارت آپ درست کنن. با صد تا مکافات بالاخره موفق میشن شرکت خودشون رو بسازن. همه چیز خوب پیش میره، همه از این ایده استقبال می کنن و با این شرکت نو شروع به همکاری میکنن. مدتی میگذره و چند تا اتفاق رخ میده. اولیش اینه که در سیستم یک باگ پیدا میشه که باید سریعا رفع شه و دومی اینکه یه شرکت چینی شروع به کپی برداری از این برنامه کرده و باعث شده که عده ای از مشتری‌ها از دست برن. اگه شما جای این دانشجوها بودید چیکار میکردید؟ احتمالا در مورد مشکل اول سریعا به عده‌ای از تیم میگفتین که برن و باگ رو پیدا کنن و در مورد مشکل دوم هم تصمیم میگرفتین که برای بردن رقابت شروع به اضافه کردن قابلیت‌های جدید بکنین. مدتی میگذره و نتایج هر دو تیم توسعه دهنده و رفع کننده ایراد مشخص میشه. تیم توسعه دهنده قابلیت بسیار جذابی رو پیاده کردن ولی مشکل اینجاست که باید قسمت بزرگی از کدهای برنامه شرکت عوض بشن تا بشه این قابلیت رو اضافه کرد و مشکل باگ هم پیدا شده ولی مشخص شده که برای تغییر این باگ برنامه باید دچار تغییرات اساسی بشه. پس از مدتی شرکت مشتری هاش رو از دست میده و ورشکسته میشه...
جالبه بدونید دو موردی که در داستانمون بهش اشاره کردیم سناریوهای کاملا واقعی هستند! مشکل اصلی این تیم دانشجویی باانگیزه این بود که در اولین قدم به دو مسئله فکر نکردند. اونها در مرحله نوشتن برنامه اولیه، بستر توسعه برنامشون رو فراهم نکردن و همینطور به علت وابستگی قسمت‌های مختلف برنامه‌شون بهم، ایرادیابی و رفعش کار سختی شده بود.
حالا که این داستان غم‌انگیز رو خوندید حتما به این فکر می‌کنید که برای حل این مشکل‌ها باید چیکار کرد؟ کلید حل مسئله در اصل دوم یعنی OCP هست.

OCP بـه مـا میگـه کـدی کـه زده میشـه بایـد بـه شـکلی باشـه کـه اگـر روزی تصمیـم گرفتـه شـد کـه قابلیتـی بـه برنامـه اضافـه بشـه بسـترش کامـلا فراهـم باشـه. همچنیـن تغییـری کـه بـه هـر علتـی در یـک کلاس رخ میـده هیـچ کلاس دیگـه‌ای رو نبایـد تغییـر بـده. بیاییـد بـرای درک بهتـر یـک مثـال رو بـا هـم بررسـی کنیـم.

بـه مثـال SRP برگردیـم. بـه برنامـهٔ شـما شـکل مثلـث اضافـه شـده اسـت.قراره کـه برنامـه شـما بتوانـد مجمـوع مسـاحت اشـکال ترسـیم شـده رو بدسـت بیـاره. نظرتـون در مـورد پیاده‌سـازی زیـر چیـه؟

```
public class Rectangle {
   int width;
   int height;

   public int getRectangleArea() {
       return width * height;
   }
}

public class Triangle {
   int height;
   int base;

   public int getTriangleArea() {
       return height * base / 2;
   }
}

public class ArithmeticUnit {
   ArrayList<Triangle> triangles;
   ArrayList<Rectangle> rectangles;

   public int getAreaSum() {
       int sum = 0;
       for (Triangle triangle : triangles) {
           sum += triangle.getTriangleArea();
       }
       for (Rectangle rectangle : rectangles) {
           sum += rectangle.getRectangleArea();
       }
       return sum;
   }
}
```

بـه متـد محاسـبه کننـده مسـاحت دقـت کنیـد. حـالا فـرض کینـد عـلاوه بـر مثلـث بایـد شـکلای دیگـه‌ای هـم اضافـه بشـه. مشـخصه کـه بـا هـر شـکل اضافـه ایـن کلاس دچـار تغییـرات زیـادی میشـه. بـرای رفـع ایـن مشـکل بنظـر شـما چـه کاری میشـه انجـام داد؟ کافیـه یـک interface shape بسـازیم و کـدو بـه صـورت زیـر تغییـر بدیـم:

```
public interface Shape {
   int getArea();
}
public class Rectangle implements Shape {
   int width;
   int height;

   public int getArea() {
       return width * height;
   }
}
public class Triangle implements Shape {
   int height;
   int base;

   public int getArea() {
       return height * base / 2;
   }

}
public class ArithmeticUnit {
   ArrayList<Shape> shapes;

   public int getAreaSum() {
       int sum = 0;
       for (Shape shape : shapes) {
           sum += shape.getArea();
       }
       return sum;
   }
}
```

همانطـور کـه میبینیـد اضافـه کـردن ۱۰۰هـا شـکل دیگـه هـم تـوی کلاس محاسـبه کننـده مسـاحت تغییـری ایجـاد نخواهـد کـرد. همچنیـن اگـه تـوی بدسـت آوردن مسـاحت یـک شـکل تغییراتـی ایجـاد بشـه، باعـث عـوض شـدن ایـن کلاس نخواهـد شـد.

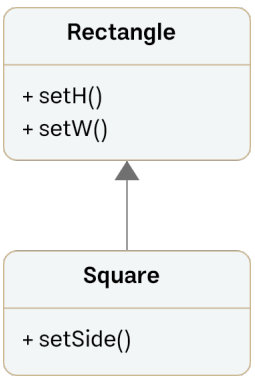
OCP رو میشـه قلـب برنامه‌نویسـی شـیءگرا دونسـت. رعایـت ایـن اصـل بـه مـا انعطـاف بسـیار زیـادی چـه در تغییـر و چـه درتوسـعه برنامه‌هامـون میـده؛ هرچنـد کـه ممکنـه کمـی پیـاده سـازی ایـن اصـل در برنامه‌مـون سـختی‌هایی هـم داشـته باشـه.

Liskov Substitution Principle (LSP)

اصـل مهـم بعـدی، میگـه اگـه یـک کلاس مثـل A زیـر مجموعـه یـک کلاس مثـل B باشـه (از B ارث بـرده باشـه یـا در مـورد اینترفیس‌هـا، اون رو ایمپلمنـت کـرده باشـه) بایـد بتونیـم B رو بـا A جایگزیـن کنیـم؛ بـدون اینکـه تـوی رونـد برنامـه مشـکلی پیـش بیـاد! بـه عبارتـی رفتـار کلاس پـدر بایـد در فرزنـد تعریـف شـده باشـد و بتونیـم پـدر رو بـا فرزنـد جایگزیـن کنیـم.

برای درک بهتر بیاید یه مثال رو بررسی کنیم.

فـرض کنیـد یـه کلاس مسـتطیل داریـم و خـب همه‌مـون هـم میدونیـم، مربـع یـه مسـتطیله. پـس بنظـر طراحـی‌ای کـه مربـع فرزنـد مسـتطیل باشـه، طراحـی خوبیـه!



اما این طراحی یه مشکلی داره! چه مشکلی؟ به تست زیر دقت کنید:

```
Rectangle r = new Rectangle();
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

خب، الان تست پاس میشه، اما تست زیر چطور؟

```
Rectangle r = new Square();
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

تست پاس نمیشه چون مستطیل رو نمیشه با مربع جایگزین کرد، در واقع رفتار تابع setH و setW مطابق انتظارمون نیست!
دیدیم که اصل نه چندان پیچیده Liskov Substitution لزوم امکان جایگزینی کلاس پدر با فرزانداش رو بیان میکنه و اگه این اصل رعایت نشه ممکنه با رفتار پیش‌بینی نشده و اشتباه مواجه بشیم!

Interface Segregation Principle (ISP)

همونطور که از اسمش هم پیداست، این اصل میگه اینترفیس‌های بزرگتر باید به اینترفیس‌های کوچیکتر تقسیم بشن. اینجوری مطمئن میشیم که کلاس‌هایی که اونا رو ایمپلمنت میکنن فقط به بخش‌هایی بستگی دارن که بهش نیاز دارن.
یه مثال برای فهم بهتر ببینیم:

```
public interface MediaPlayer {
    public void playAudio();
    public void playVideo();
}
public class VlcMediaPlayer implements Medi-
aPlayer {

    @Override
    public void playAudio() {
        System.out.println(«Playing audio»);
    }
    @Override
    public void playVideo() {
        System.out.println(«Playing video»);
    }
}
public class WinampMediaPlayer implements Me-
diaPlayer {

     // Play video is not supported in Winamp
player

    public void playVideo() {
        throw new VideoUnsupportedException();
    }


    @Override
    public void playAudio() {
        System.out.println(«Playing audio»);
    }
}
```

خب اینجا WinampMediaPlayer که ویدئو رو ساپورت نمیکنه، چرا باید تابعشو پیدا سازی کنه؟ طراحی بهتر و درست تر بر اساس Interface Segregation این شکلیه که اینترفیس MediaPlayer رو تفکیک کنیم. یعنی به صورت زیر:

```
public interface AudioMediaPlayer {
    public void playAudio();
}
public interface VideoMediaPlayer {
    public void playVideo();
}
public class VlcMediaPlayer implements VideoMe-
diaPlayer, AudioMediaPlayer {

 . . .
}
public class WinampMediaPlayer implements Au-
dioMediaPlayer {

    @Override
    public void playAudio() {
        System.out.println(«Playing audio»);
    }
}
```

طبق ISP متوجه شدیم که اینترفیس‌ها باید به نحوی تعریف بشن که هر کلاس صرفا توابع مورد نیازش رو پیاده سازی کنه و مجبور به پیاده‌سازی تابع‌هایی که نیازی بهشون نداره نباشه!

Dependency Inversion Principle (DIP)

ایده کلی این اصل اینه که ماژول‌های سطح بالا، که منطق پیچیده‌ای رو پیاده میکنن، باید به راحتی قابل استفاده مجدد باشن و تحت تأثیر تغییرات در ماژول‌های سطح پایین قرار نگیرن. برای این کار لازمه از abstracion استفاده کنیم، طوری که ماژول‌های سطح بالا از ماژول‌های سطح پایین جدا بشن.
چه تعریف پیچیده‌ای! بذارید یا یه مثال قضیه رو روشن‌تر کنیم.
فرض کنید برای یک فروشگاه کتاب یک کلاس Book و Shelf بصورت زیر تعریف شده:

```
public class Book {

    void seeReviews() {...}
    void readSample() {...}
}

public class Shelf {

     Book book;

     void addBook(Book book) {...}
     void customizeShelf() {...}
}
```

همه چیز خوب به نظر میرسه، اما ازاونجایی که کلاس Shelf (سطح بالا) به Book (سطح پایین) بستگی داره، کد بالا DIP رو نقض می کنه. این موضوع کی مشخص میشه؟ وقتی که بعد از توسعه فروشگاه بخواد یه محصول جدید، مثلا DVD بفروشه! میخوایم یک کلاس DVD اضافه کنیم و از Shelf برای DVD هم استفاده کنیم:

```
public class DVD {
```

```
     void seeReviews() {...}
     void watchSample() {...}
}
```

حالا چون داخل Shelf از Book استفاده کردیم، نمیتونیم از اون برای DVD هم استفاده کنیم! پس چاره چیه؟ بیاید بر اساس DIP سعی کنیم با ایجاد یک abstraction وابستگی مستقیم بین Book و Shelf رو از بین ببریم:

```
public interface Product {
    void seeReviews();
    void getSample();
}
```

حالا واضحه Book و DVD هر دو Product هستند:

```
public class Book implements Product {

    @Override
    public void seeReviews() { ...}

    @Override
    public void getSample() {...}
}

public class DVD implements Product {

    @Override
    public void seeReviews() {...}

    @Override
    public void getSample() {...}
}
```

و در نهایت کلاس Shelf که برای هر Productای قابل استفاده خواهد بود:

```
public class Shelf {

    Product product;

    void addProduct(Product product) {...}
    void customizeShelf() {...}
}
```

دیدید استفاده از کلاس سطح بالای Shelf چجوری به دلیل وابستگی به کلاس Book، محدود شده بود؟ و چطور تونستیم با ایجاد ابسترکشن و یک اینترفیس اونو قابل استفاده مجدد کنیم؟ این همون چیزیه که Dependency Inversion دنبالشه!

Unit Test

آرش یادگاری

مطمئنم شما هم مثل ما لحظاتی که تست‌های سوال پاس نمیشن رو تجربه کردید. یه سوال ازتون داریم، تو این مواقع چیکار می‌کنید (به غیر از آه و نفرین)؟
احتمالا شروع به تست کردن برنامه‌تون می‌کنید. سعی می‌کنید تستی بنویسید که اشتباه شما رو نشون بده.
تا حالا به این فکر کردید که شرکت‌های بزرگ چجوری برنامه‌هاشون رو تست می کنن؟ آیا مثل ما برنامه‌شون رو ران می‌کنن و سعی می‌کنن تستی پیدا کنن که برنامشون جواب درستی نمیده بهش؟ اگر بخوایم واقعیت رو بگیم ، هم آره و هم نه.
تست کردن برنامه‌ها به دو صورت انجام میشه. اولیش تست برنامه به صورت دستیه تا از عملکرد نهایی محصول اطمینان حاصل بشه و دومیش نوشتن تست‌هایی به شکل کده.
در این قسمت قصد داریم تا شمارو با یکی از انواع تست نویسی یعنی unit test آشنا کنیم.

در تعریف کلی، unit test یعنی تست کردن کوچک‌ترین واحدهای برنامه. ممکنه اولش فکر کنید که کوچک‌ترین واحدهای برنامه ما همون متدها هستند ولی باید بگم که اینطور نیست! در واقع منظور از این واحدها ، همون واحدهایی هستند که درSRP معرفی کردیم. واحدهایی که تنها یک مسئولیت دارند. بنظر شما چرا unit test نوشتن مهمه؟

اگه بخوایم از دید صعنت بهش نگاه کنیم باید بگیم که باعث میشه هزینه کلی برنامه کمتر آب بخوره. آزمایش‌ها نشون دادند برنامه‌نویس‌هایی که همراه نوشتن برنامه‌شون یونیت تست می‌نویسند در نهایت مدت کمتری رو برای تست نهایی برنامه و در مجموع زمان کمتری برای نوشتن اون برنامه صرف میکنن. اگه باور نمیکنید یه نگاهی به این جدول بندازید.

Stage	Team without tests	Team with tests
Implementation (coding)	7 days	14 days
Integration	7 days	2 days
Testing and bug fixing	Testing, 3 days Fixing, 3 days Testing, 3 days Fixing, 2 days Testing, 1 day Total: 12 days	Testing, 3 days Fixing, 1 day Testing, 1 day Fixing, 1 days Testing, 1 day Total: 7 days
Overall release time	26 days	23 days
Bugs found in production	71	11

یک دلیل بسیار مهم دیگه که باعث میشه نوشتن یونیت تست اهمیت زیادی پیدا کنه اینه که کد خروجی ما با کیفیت‌تر و دارای code smellهای کمتری میشه. به عبارتی زمانی میتونیم برای یونیت‌های برنامه‌مون تست بنویسیم که قسمت‌های برنامه ما قواعدی مثل جداسازی وابستگی‌ها یا داشتن تنها یک وظیفه رو رعایت کرده باشن. در آخر یک نکته خیلی مهم، نوشتن تست باعث میشه که کد شما خوانا باشه.

خب حالا که با یونیت تست و فایده‌هاش آشنا شدید ، دیگه وقتشه که با هم اولین تست رو بنویسیم.

میخوایم عملیات جمع و تفریق زیر رو تست کنیم.

```
public class Calculator {

   public int add(int numberOne, int numberTwo)
{
       return numberOne + numberTwo;
   }
}
```

برای اینکار اول از همه یک پروژه با فریم ورک maven بسازید و این دیپندسی رو به فایل pom.xml اضافه کنید.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

حال به پوشه test و سپس java بروید و یک کلاس جدید باز کنید و اسمشو testCalculator بذارید. وظیفه این کلاس تست کردن calculator هست و پیاده‌سازیش به شکل زیر است.

```
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

   @Test
   public void testAddition() {
       Calculator calculator = new Calculator();
        assertEquals(calculator.add(1, 2), 3);
   }
}
```

تبریک میگم شما موفق شدین اولین تستتون رو بنویسید. کافیه تست رو ران کنید و نیتجه رو بررسی کنید. قبول داریم که ممکنه بنظر برسه که نوشتن تست برای توابع بسیار ساده و بدیهی کاربرد داره و نمیشه باهاش تست‌های مهمی نوشت ولی نگران نباشید! به زودی در کارگاه یاد میگیرید که به چه صورتی برای هر کجای برنامه‌تون تست بنویسید.

اول تست، بعد کد!

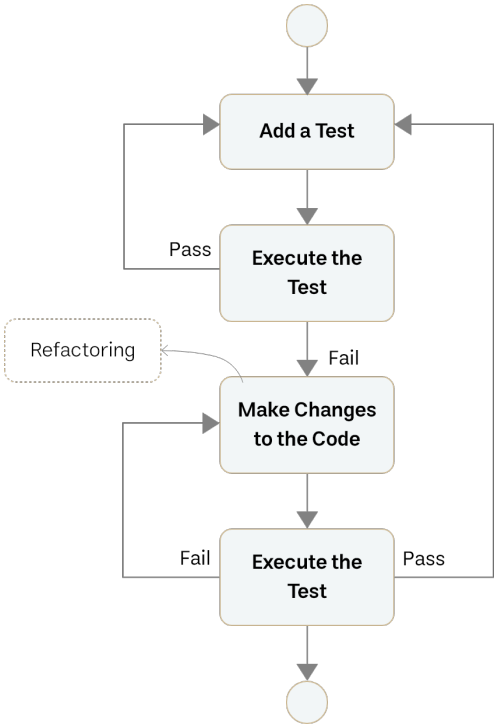
امیرحسین رازلیقی

همین‌طور که می‌دونین، دنیای برنامه‌نویسی دائما در حال تغییر و بهبوده! یکی از مهم‌ترین تغییرات جدید توی مفاهیم برنامه‌نویسی، مفهومیه به اسم TDD (یا همون Test Driven Development). همین‌طور که از اسمش مشخصه ، این Paradigm برنامه نویسی ، میگه که برنامه نویس ها قبل از اینکه بخوان شروع به نوشتن یک پروژه بکنن، باید تست‌هاش رو بنویسن! می‌دونم ممکنه در دید اول

خیلی عجیب به نظر بیاد، ولی بذارید بیشتر توضیح بدم. آقای Kent Beck (که خالق این paradigm در برنامه‌نویسی هست)، در یکی از مصاحبه‌هاش توضیح میده و میگه که وقتی ما (یعنی برنامه‌نویس ها!) در ابتدای پروژه هستیم و در حال بررسی نیازمندی‌های پروژه هستیم، دیدمون نسبت به پروژه خیلی باز تره. اگه وقتی توی همین وضع هستیم، بیایم و با توجه به نیازمندی‌های پروژه، براش تست بنویسیم، تست‌هامون خیلی بهتر و بامعناتر هستن! مثلا فرض کنین ما می‌خوایم یه پروژه‌ی مدیریت فروشگاه بزنیم. اول می‌بینیم که این App ما باید در نهایت چه کارهایی بکنه. برای مثال، باید بتونه با بررسی Database، چک بکنه که مقدار موجودی یک محصول چند تاست، یا بتونه قیمت هر محصول رو بهمون اعلام کنه. حالا که این نیازمندی‌ها رو درآوردیم، همین ابتدا میایم و برای پروژه‌مون تست می‌نویسیم (یه تست می‌نویسیم که یک محصول رو از اپ درخواست کنه و تعداد موجودیش رو بگیره، یا یه تست دیگه که قیمت یک محصول رو از اپ ما بپرسه و غیره)

خب، حالا که برای هر چیزی که فکرشو میکردیم، تست نوشتیم (!) بیاین و برنامه‌ی خودمون رو run کنیم. چه اتفاقی میوفته؟ مشخصا تمام تست‌هامون fail میشن! چون هنوز هیچ کدی ننوشتیم! حالا از اینجا به بعد، وظیفه‌ی ما بعنوان developer اینه که برنامه‌مون (شامل توابع، کلاس‌ها و ماژول‌های مختلف) رو بنویسیم تا تک‌تک این تست‌ها پاس بشن!

این دقیقا برعکس رویه‌ی سنتی تست‌نویسی هست (که اول کد رو می‌نوشتیم و بعد تست می‌کردیم که ببینیم باگ‌هاش کجان!). برای همین بعضی جاها به این رویه، Test First هم میگن . در نهایت هم یه فلوچارت مفهومی از کل فرآیند TDD قرار دادیم که بهتر درکش کنین . اگر کنجکاو شدین ، پیشنهاد می‌کنیم کتاب TDD By Examples (نوشته ی خالق این paradigm برنامه نویسی) رو حتما مطالعه کنین! البته آقای Kent Beck نظریات جنجالی دیگری هم علاوه بر این یک مورد دارن که یکی از اونها ، Test-Commit-Revert هست! اگر دوست داشتین در این مورد کمی سرچ کنین که باهم در موردش بحث کنیم (در حال حاضر یکی از بحث برانگیزترین paradigm های برنامه‌نویسیه!).



مستندسازی کد

عرفان مجیبی

مستندسازی کد

به مجموعه متن‌هایی که همراه با کد نرم افزار ارائه می‌شود تا توضیح دهد که کد شما چه کاری انجام می‌دهد، چرا اینگونه نوشته شده است، و چگونه می‌توان از آن استفاده کرد، مستندات کد(Code Documentation) گفته می‌شود.

مستند کردن کد ممکن است در ابتدا کاری عجیب و بی‌فایده به نظر برسد اما توجه کنید درک و استفاده از کد شما را برای سایر افراد ساده‌تر می‌کند. اگر کد شما به خوبی مستند نشده باشد و سایرین نتوانند نحوه کار کردن آن را درک کنند، کد شما عملا به درد نخواهد خورد! همچنین مستند کردن کد کمک می‌کند وقتی برنامه‌نویس به کدهای قدیمی خودش مراجعه می‌کند، ساده‌تر آنها را درک کند.

مستندنویسی در جاوا

شما برای مستند کردن کدتان می‌توانید هرگونه که دوست داشتید عمل کنید! مثلا کارکرد بخش‌های کوچک کدتان را با کامنت‌هایی توضیح دهید! اما این کار به نظر چندان تمیز و منسجم نیست. پس چه باید کرد؟

معمولا زبان‌های مختلف قراردادهایی برای مستند کردن کد لحاظ کرده اند. خوشبختانه جاوا ابزار javadoc را برای مستندسازی تامین کرده است که در کنار JDK بر روی کامپیوتر شما نصب شده است. به کمک javadoc، شما می‌توانید مستندات مربوط به کدتان را به صورت خروجی HTML دریافت کنید. البته لازم است قالب مشخصی را برای این هدف رعایت کنید که در ادامه به شرح آن می‌پردازیم.

قالب javadoc

کامنت‌های javadoc، کامنت‌های چند خطه هستند که با /** شروع می‌شوند. این کامنت‌ها ممکن است بالای هر کلاس، متد یا فیلدی که می‌خواهیم مستندسازی کنیم، قرار گیرند و معمولاً از دو بخش تشکیل شده‌اند:

۱- توصیفی کلی از آنچه در مورد آن مستند می‌نویسیم.

۲- تگ‌های مشخصی(با «@» مشخص شده‌اند) که اطلاعات دقیق‌تری به ما می‌دهند.

برای درک بهتر به مثال زیر توجه کنید:

```
  /**
   * Adds two integers. This is
   * the simplest form of a class method, just to
   * show the usage of various javadoc tags.
   * @param numA the first parameter to add method
   * @param numB the second parameter to add method
   * @return addition of numA and numB.
   */
 public int add(int numA, int numB) {
      return numA + numB;
}
```

تگ‌های قابل استفاده به دو دسته Block tag و Inline tag تقسیم می‌شوند. Block tag ها دارای محدودیت برای توصیف اهداف و بخش‌های مشخصی هستند، اما از Inline tag ها در هر بخش از توضیحات و کامنت‌ها می‌توان استفاده کرد. مثلا {@code} یک Inline tag است که بخش مد نظر شما در خروجی مستند را به عنوان کد در نظر می‌گیرد.

در ادامه به معرفی Block tagهای مهم می‌پردازیم:

@author

احتمالا بتوانید حدس بزنید که برای معرفی نویسنده یا نویسنده‌های یک کلاس، پکیج یا اینترفیس استفاده می‌شود.

@version

برای ذکر ورژن فعلی کد کلاس، اینترفیس، یا پکیج از آن استفاده می کنیم .

@since

نشان می‌دهد یک متغیر، متد یا کانستراکتور، کلاس یا اینترفیس، و یا پکیج در چه ورژنی آخرین تغییرات را داشته یا به کد اضافه شده ا ست .

@see

عنوان «همچنین ببینید» را با لینک یا ورودی متنی که به یک مرجع

اشـاره دارد اضافـه می‌کنـد.

@param

برای توصیف پارامترهای ورودی متود یا کانستراکتور استفاده می‌شود.

@return

برای توصیف مقدار بازگشتی یک متد مورد استفاده قرار می‌گیرد.

@exception

بـرای بیـان نـوع اسـتثناء(یا اسـتثناءهایی) کـه یـک متـد پرتـاب می‌کنـد و توضیحـات مربـوط بـه آن، کاربـرد دارد.

به مثال زیر توجه کنید تا کاربرد چند مورد از تگ‌های بالا و همچنین نحوه صحیح مستند کردن را ببینید:

```
/**
 * Returns the character at the specified index. An index
 * ranges from {@code 0} to {@code length() - 1}.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfBoundsException
 *              if the index is not in the range {@code 0}
 *              to {@code length()-1}.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
    ...
}
```

برای کسب اطلاعات بیشتر درمورد تگ‌ها می‌توانید به اینجا مراجعه کنید.

همچنیـن بـرای دیـدن مثال‌هـای بیشـتر، می‌توانیـد IDE خـود را بـاز کنیـد و هـر کلاسـی از کتابخانه‌هـای built-in جـاوا کـه دوسـت داشـتید را بـاز کـرده، و داکیومنتیشـن آن را بخوانیـد!

خروجی گرفتن از javadoc

همانطـور کـه اشـاره کردیـم، امـکان جذابـی کـه javadoc بـرای مـا فراهـم می‌کنـد خروجـی دادن فایـل html مسـتندات کـد اسـت! امـا چطـور؟ در ادامـه بـه ایـن موضـوع می‌پردازیـم.

ابتـدا توجـه کنیـد فایل‌هایـی کـه javadoc آنهـا را پـردازش و خروجـی متناظـر بـا آنهـا را بـرای شـما تولیـد می‌کنـد، شـامل سـه دسـته زیرنـد:

- فایل‌هـای کـد: عمومـا شـامل کلاس‌هـا و اینترفیس‌هـا هسـتند کـه مسـتندات را نیـز در کنـار خـود دارنـد.
- فایل‌هـای مسـتندات پکیـج: بـرای نوشـتن مسـتندات یـک پکیـج شـما می‌توانیـد یـک فایـل package-info.java داخـل پکیـج خـود ایجـاد کنیـد و مسـتندات مربـوط بـه آن پکیـج را داخـل آن، بـا فرمتـی کـه توضیـح دادیـم، بنویسـید. راه دیگـری نیـز وجـود دارد کـه ایجـاد یـک فایـل package.html در داخـل پکیـج و نوشـتن مسـتندات اسـت، کـه چنـدان توصیـه نمی‌شـود.
- فایـل overview: هـر برنامـه یـا مجموعـه‌ای از پکیج‌هایـی کـه مسـتند می‌کنیـد بهتـر اسـت یـک توضیـح و معرفـی کلـی نیـز داشـته باشـند. کافیسـت یـک فایـل html در مسـیر پروژه‌تـان ایجـاد کنیـد و آن را بـه javadoc معرفـی کنیـد. javadoc در تولیـد خروجـی نهایـی، آن را مـد نظـر قـرار خواهـد داد!

خـب حـالا بـرای تولیـد خروجـی می‌توانیـد از کامنـد javadoc اسـتفاده کنیـد:

```
javadoc src/*
```

ایـن دسـتور فایل‌هـا و پکیج‌هـای داخـل src را بررسـی کـرده و خروجـی را تولیـد میکنـد! دسـتور javadoc آپشـن‌هایی نیـز دارد مثـلا می‌توانیـد مسـیر خروجـی را بـه صـورت زیـر بـرای آن مشـخص کنیـد:

```
javadoc -d docs src/*
```

بـرای آشـنایی بیشـتر بـا ایـن آپشـن‌ها می‌توانیـد بـه اینجـا مراجعـه کنیـد.

اگـر هـم تمایلـی بـه اسـتفاده از کامنـد ندارید، معمـولا IDEهـا امـکان تولیـد javadoc را فراهـم می‌کننـد. مثـلا در IntelliJ می‌توانیـد از زبانـه Tools بـا انتخـاب گزینـه Generate JavaDoc ایـن کار را انجـام دهیـد.

حـالا کـه بـا مسـتند کـردن کـد آشـنا شـدید، توصیـه می‌کنیـم از ایـن بـه بعـد تـلاش کنیـد هـر کـدی کـه می‌زنیـد را مسـتند کنیـد! توجـه کنیـد ایـن کار بـرای یـک برنامه‌نویـس خـوب، بـه نوعـی بخشـی از فراینـد کـد زدن محسـوب می‌شـود. پـس بهتـر اسـت همـراه بـا کـد زدن و تعریـف هـر کلاس و متـد و ...، مسـتندات مربـوط بـه آن را نیـز بنویسـید و آن را بـه زمـان دیگـر نیندازیـد!

> If you Automate a Mess, you get an Automated Mess.
>
> – *Aristotle*

کدنامه

شماره: ٤
تاریخ انتشار: ۳۱ فروردین ۱۴۰۱
نویسندگان: عرفان مجیبی، آرش یادگاری و امیرحسین رازلیقی
طراحی: سجاد سلطانیان و حسام الدین سلیمانی

دستیاران آموزشی دروس مبانی برنامه‌سازی و برنامه‌سازی پیشرفته، طی نیمسال‌های گذشته، مطالب متعدد درسی را در قالب فایل‌های گوناگون منتشر می‌کردند. از سال گذشته، برآن شدیم تا با تشکیل «کدنامه»، تمامی مطالب آموزشی را در این قالب تدوین کنیم تا هم به سرعت بتوان آنها را مطالعه کرد و هم بتوانیم به این بهانه، محتوایر کاربردی‌تری را در اختیار دانشجویان قرار دهیم. توجه شود که «کدنامه»، به هیچ عنوان، یک نشریه نیست و زمان عرضه مشخصی نیز ندارد، بلکه تنها قالبی جدید برای عرضه همان مطالب و محتوای آموزشی است و در آن تنها به مباحث درسی مخصوص درس برنامه‌سازی پیشرفته نیم‌سال جاری پرداخته می‌شود.

مطالعه مطالب «کدنامه»، برای انجام بهتر پروژه و تمرین‌های درس برنامه‌سازی پیشرفته، اکیدا توصیه می‌شود.