Clean Code

Chapter 3: Functions

author: Hamed Damirchi
hameddamirchi32@gmail.com
github.com/hamed98
linkedin.com/in/hamed-damirchi-ba4085178/

مقدمه

صفحه ۶۳ و ۶۴ پی دی اف: مثال تابع بد و خوب

کوچک بودن

- اولین قاعده یک تابع اینه که کوچیک باشه
- دومین قاعده هم اینه که کوچیکتر باشه!
- تجربه ۴۰ ساله نویسنده تو برنامه نویسی نشون میده که تابع هر چی کوچیکتر بهتر!
- دهه ۸۰ میگفتن یه تابع نباید از ارتفاع مانیتور بزرگتر باشه اما مانیتور های اون موقع کوچیک بود!
- الان: نباید از ۲۰ خط بیشتر شه
- متوسط ۶-۷ خط

تابع Listing-2 هم بزرگه حتی!

میتونه به شکل زیر ریفکتور شه:

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

اندازه بلوک ها:

بنابر چیزی که گفته شد توصیه میشه اندازه بلوک های داخل if, while, for یک خط باشه و اون خط هم یه فراخوانی تابع باشه.

این طوری علاوه بر این که اندازه تابع کوچیک میشه، مستند سازی هم بهتر میشه چون مجبوری واسه تابع اسم بزاری و این به مستند سازی کمک میکنه.

تابع نباید شامل کد های تو در تو باشه. همچنین ماکزیموم سطح ایندنت توصیه میشه بیشتر از یک یا دو سطح نباشه.

یک کار انجام دهید

Listing3-1 خیلی کارها انجام میدهد، اما Listing3-3 فقط یک کار انجام میده.

یک قاعده:

> ***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.***

اما مشکل این جاس که نمیشه راحت فهمید "یک کار" چیه؟

مثلا میشه گفتListing3-3 سه کار انجام میده:

۱- تشخیص تست بودن صفحه

۲- اگر بله، ستاپ رو انجام میده

۳-صفحه رو به صورت HTML رندر میکنه.

یک توضیح راجع به سطوح انتزاع

تو کتاب نیست این بخش، اما لازمه بدونیم

هر چه قدر سطح انتزاع بالاتر باشه، جزئیات کمتره و بیشتر با مفهوم سروکار داریم.

مثلا بالاترین سطح میشه کل برنامه، سطح بعدی میشه ماژول هاش، سطح بعدی مثلا تابع ها و کلاس ها، سطح بعدی داخل تابع ها، سطح بعدی ...

- نکته این جاست:
- کاری که تو بدنه تابع اتفاق میفته، باید دقیقا یک سطح پایین تر از سطح انتزاع اسم تابع باشه. اون موقع میشه گفت تابع یک کار انجام میده.
- به عبارتی نتونی تابع رو به دو بخش مجزا از هم تبدیل کنی.
- بنابراین یک راه واسه این که بفهمیم آیا تابع ما "یک کار" انجام میده یا نه، اینه که اگر بتونی از این تابع، یه تابع دیگه ای استخراج کنی که اسمش صرفا تغییر نام تابع فعلی نباشه، اون وقت یعنی این تابع بیشتر از یک کار داره انجام میده
- همچنین باید بتونی کار هر تابع رو تو یه پاراگراف مختصر توضیح بدی
- یه مثال دیگه از چند کار تو یه تابع: صفحه ۱۰۲ پی دی اف، تابع پرایم

یک سطح انتزاع در هر تابع

در یک تابع، همه اجزای آن تابع باید در یک سطح از انتزاع باشن.

مثلا در Listing3-1 یه تابع توشه getHTML() که سطح بالایی داره در حالی که سطح های پایین تر هم هستن تو تابع مثل parser.render() یا append(“\n”).

این کار باعث میشه که خواننده نتونه راحت تشخیص بده چه بخش هایی از تابع جزئیات هستن و چه بخش هایی، بخش اصلی.

خواندن کد از بالا به پایین(step-down rule)

ما میخوایم کد طوری باشه که بتونیم عین یه روایت از بالا به پایین بخونیم(یعنی از سطح بالای انتزاع بیاییم به سطح پایین انتزاع)

بنابراین وقتی از یه تابع میریم تو یه تابع دیگه، باید حتما یه لول اومده باشیم پایین تر

باید بتونیم برنامه رو به صورت چند TO paragraph بگیم. مثلا واسه مثال کتاب:

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.

To include the setups, we include the suite setup if this is a suite, then we include the regular setup.

To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.

To search the parent. . .

میبینیم که هر TO یعنی هر تابع، یک سطح عمیق تر میشه تو کد.

میدونیم که خیلی سخته واسه برنامه نویسا که بخوان از این قاعده "هر تابع یک سطح انتزاع" پیروی کنن.
اما خیلی مهمه که این کارو کنن و اگه از TO-paragraph استفاده کنن خیلی بهشون کمک میکنه
این قاعده کلید "هر تابع یک کار" است.

چگونه از switch استفاده کنیم؟

سوییچ ها طبیعتا قاعده "یک کار" را نقض میکنن

باید سوییچ ها رو با پولیمورفیسم هندل کنیم و از سوییچ تو یه تابع پابلیک استفاده نکنیم، بزاریمش تو یه تابع پرایوت تو یه کلاس.

کد روبرو خیلی چیزا رو نقض میکنه.

Listing 3-4
Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

- اول این که این کد بلنده و با اضافه شدن هر نوع جدید کارمند، طولانی تر هم میشه
- دوم این که بیشتر از یک کار انجام میده
- سوم این که قاعده Single Response Principle(SRP) رو نقض میکنه، چون بیش از یک دلیل واسه تغییر این تابع (کلاس) هست.(این قاعده مهمیه!)
- چهارم این که قاعده Open Closed Principle رو نقض میکنه، چون وقتی یه تایپ جدید اضافه شد باید تغییر کنه.https://7learn.com/programming/what-is-open-closed-principle
- اما مشکل اصلی این تابع اینه که خیلی تابع های دیگه هم میتونن باشن که چنین ساختاری دارن. مثلا isPayDay() یا deliverPay().

راه حل اینه که از دیزاین پترن فکتوری استفاده کنیم و سوییچ رو تو یه تابع پرایوت دفن کنیم و نذاریم کسی ببینه اونو.

در اصل باید این سوییچ ور موقع ایجاد کارمند داشته باشیم و از پولیمورفیسم استفاده کنیم:

Employee and Factory

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
-----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

از اسم توصیفی برای تابع استفاده کنید

در Listing3-7 این رعایت شده

حتی برای تابع های کوچیک یکی دو خطی هم یه اسم خوب که توصیف کننده باشه انتخاب کنید.

هر چه قدر تابع کوچیکتر باشه راحت تر میتونی اسم توصیفی واسش بزاری

یه اسم بلند واضح بهتر از یه اسم کوتاهیه که واضح نیست

یه اسم بلند توصیفی بهتر از یه کامنت بلندیه که توصیف کنه کار تابع رو

هیچ اشکالی نداره اگه وقت زیادی واسه انتخاب اسم صرف کنید، حتی میتونی اسم های مختلفی رو انتخاب کنی

خیلی وقت ها سعی تو انتخاب اسم مناسب باعث میشه که بخشی از ساختار کد رو تغییر بدی

تو اسم های تابع یه کلاس، میتونی از لغت های شبیه هم استفاده کنی، طوری که انگار داره یه روایت گفته میشه. مثال:Listing3-7

آرگومان تابع

تعداد آرگومان ایده آل صفره، بعد یک بعد دو. از سه آرگومان باید تا حد ممکن اجتناب بشه، بیشتر از اون نباید استفاده بشه(مگر این که توجیه خیلی خوبی واسش باشه)

اگه تابع ها ورودی نداشته باشن خیلی راحت میتونی به صورت داستان کد رو بخونی

مهم* علت: آرگومان یه سطح انتزال پایین تر از تابعه. و مجبوری وارد جزئیات بشی.

مثال: Listing3-7 stringbuffer

مهم*: علت دیگه: تست کردن سخت میشه، باید همه ترکیبات ممکن واسه ورودی رو تست کنی.

آرگومان های out کار رو به مراتب سخت تر میکنه مثل .includeSetupPageInto(StringBuffer pageText)

آرگومان-فرم های رایج تابع تک ورودی (monadic)

- در دو حالت باید یه تابع تک ورودی باشه:
  - ۱- یه سوال راجع به ورودی پرسیده میشه(مثلا isExist)
  - ۲-یه عمل روی ورودی انجام میشه و تبدیل به یه چیز دیگه میشه و ریترن میشه
- نام گذاری تابع طوری باشه که تمایز این دو حالت مشخص بشه.
- حالت سوم کمتر مرسوم:ایونت، از تک ورودی استفاده میشه که State سیستم تغییر کنه.
- مثلا void passwordAttemptFailedNtimes(int attempts)

- ❊: StringBuffertransform(StringBuffer in) بهتر از void transform-(StringBuffer out)

آرگومان-فلگ

- استفاده از فلگ تو ورودی تابع کار خوبی نیست!
- چون باعث میشه تابع از "تک کاره" بودن خارج بشه
- بهتره دو تابع جدا بنویسیم، یکی برای true و یکی false
- در Listing3-7 بهتره به جای render(isSuite) این دو تابع رو بنویسیم: renderForSuite() و renderForSingleTest()

آرگومان- دو ورودی (Dyadic)

تابع با دو ورودی سخت تر قابل درکه نسبت به تک ورودی

مثلا writeField(name) و writeField(outputStream,name)

تابع دو ورودی بیشتر باعث باگ میشه، چو ممکنه ووردی ها اشتباه داده بشن

اما بعضی جاها لازمه مثلا new point(0,0)

Natural Ordering رو رعایت کنید

مثلا assertEqueals(expected,actual) جای ورودی هاش برعکسه

همیشه فکر کن ببین راه مناسبی هست که تبدیل کنی به تک آرگومانه. مثلا outputstream.writeField(name)

یا کلاس FieldWriter درست کنی که outputStream رو تو سازنده اش میگیره و... .

آرگومان- سه ورودی(Triads)

این حالت رو باید به شدت با دقت استفاده کنید و تنها در موارد لزوم

ترتیب ورودی ها هم مهمن و باید دقت شه

assertEquals(message, expected, actual) ترتیب خوبی نداره

assertEquals(1.0,amount, .001) خوبه(سومی مقدار خطاست)

آرگومان- شیء به عنوان ورودی

بعضی وقت ها که لازمه سه چهار تا ورودی بدیم، گاها میتونیم ورودی ها رو یه کلاس کنیم.

Circle makeCircle(double x, double y, double radius)

Circle makeCircle(Point center, double radius)

شاید تصور بشه این یه جور چیت کردنه، اما این طور نیست چون x و y بخشی از یه مفهوم هستن که بهتره یه ساختار مجزا واسش در نظر گرفته بشه با اسم مناسب

آرگومان- لیست ورودی ها

گاها تعداد متغیر ورودی هم متغیر است و معلوم نیست. مثل string.format.

بهتره این جور مواقع از لیست به عنوان ورودی استفاده بشه

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

تابع نباید اثر جانبی داشته باشه

تابع شما باید "یک کار" انجام بده و اثر جانبی هم نداشته باشه. گاهی اوقات این اثر جانبی پنهانه.

Listing 3-6

UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

مشکل: دو کار: چک پسورد و ایجاد سشن

کال کننده فقط میخواد پسورد چک کنه، نه این که سشن هم درست کنه

این باید دو تابع جدا باشه

اگه میخواد یه تابع باشه باید اسمش بشه checkPasswordAndInitializeSession

که قاعده "یک کار" رو نقض میکنه

آرگومان های خروجی (output arguments)

باید تابع رو جوری بنویسیم که double-take ایجاد نکنه! (نگاه دوباره به امضاء تابع)

مثلا تابع appendFooter(s) (s به فوتر یا فوتر به s)

```
public void appendFooter(StringBuffer report)
```

تابع رو جوری بنویسید که احتیاجی به چک کردن signture نداشته باشه (دیفالت پایتون!

اگر احتیاج به چک کردن signture داشته باشه یعنی double-take اتفاق افتاده.

بهتره این طوری باشه: report.appendFooter()

بنابراین از آرگومان های خروجی اجتناب کنید و از فرمت گفته شده استفده کنید.

Command-Query Seperation(CQS)

هر تابع باید یا یک کار انجام بده یا به یه سوال (کوئری) جواب بده، نه هر دو

اگر تابعی قراره یه مقدار برگردونه (کوئری) باید refentially transparent باشه و اثر جانبی نداشته باشه.

(یعنی بتونی تابع رو با مقدارش جایگزین کنی) مثلا اگه قراره ست کنه یه مقداری رو، نباید چیزی برگردونه.

این اشتباهه:

```
public boolean set(String attribute, String value);
```

چون موقع کال کردن این شکلی، معلوم نیست if داره چک میکنه "آیا قبلا ست شده" یا الان داره ست میکنه.

```
if (set("username", "unclebob"))...
```

میتونیم اسم تابع رو بزاریم setAndCheckIfExists اما بازم خیلی تو خوانایی if تاثیری نداره.

درستش:

```
if (attributeExists("username")) {
  setAttribute("username", "unclebob");
  ...
}
```

این بخش ها تو کتاب نیست:

این قاعده محدودیت هایی هم داره

مثلا تابع pop تو استک

یا تو برنامه های مالتی ترد:

این قواعد کلی هستن و نمیشه همه ی قواعد رعایت شن که با تجربه به دست میاد.

یه نکته دیگه جامانده از مثال قبل:
کلمه set هم فعله هم قید و confussion ایجاد میکنه.

```
private int x;
public int incrementAndReturnX() {
    lock x;   // by some mechanism
    x = x + 1;
    int x_copy = x;
    unlock x; // by some mechanism
    return x_copy;
}
```

```
private int x;
public int value() {
    return x;
}
void increment() {
    x = x + 1;
}
```

هنگام خطا Exception بدید و کد خطا رو برنگردونید.

عدم رعایت این نکته هم باعث شلوغ شدن کد و هم نقض قانون قبل میشه.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

راه صحیح؟ اول فکر، بعد صفحه بعد!

همیشه یادت باشه:

فرایند پردازش خطا با ساختار اصلی برنامه نباید قاطی شه.

تو مثال صفحه قبل قاطی شده بود، این جا کمتر

```
try {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
   logger.log(e.getMessage());
}
```

Extract Try/Catch

بلوک های try/catch فرایند اصلی برنامه رو با فرایند هندلینگ خطا ادغام میکنن.

روش صحیح:

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

هندلینگ خطا "یک کار" است

گفتیم هر تابع باید یک کار انجام بده

هندلینگ خطا هم یک کاره

بنابراین تابعی که هندلینگ خطا میکنه بهتر کار دیگه ای نکنه.

اگر کلمه try تو یه تابع بود، باید اولین کلمه تابع باشه و بعد از catch وfinally هیچ چیز دیگه ای نباشه.

در کتاب نیست: هندلینگ خطا باید تو لایه های بالاتر اتفاق بیفته.

وابستگی به Error Enum ها

استفاده از enum برای خطا، باعث میشه که در صورت تغییر، همه ی کلاس ها تغییر کنن و برنامه از اول کامپایل شه.

اما اگر از Exception ها استفاده شه، هر اکسپشن از کلاس قبلی مشتق میشه

و این باعث میشه Open-Close principle رعایت شه.

```
public enum Error {
   OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES,
   WAITING_FOR_EVENT;
}
```

Don’t Repeat Yourself

هیچ چیزی نباید تو برنامه تکرار بشه

در Listing3-1 یک چیزی ۴ بار تکرار شده بدون این که راحت بشه متوجه تکرار شدنش شد

اما در Listing3-7 تبدیل شده به یه تابع Include

Duplication ریشه بسیاری از مشکلات است

چگونه تابع های این شکلی بنویسیم؟

مثل نوشتن یه مقاله عمل کن

اول بنویس بدون توجه به موارد گفته شده

یعنی در ابتدا اگر تابع بزرگ شد، کد تکراری شد، اسم ها مناسب نبود مهم نیست.

اما بعد از اتمام آن بخش، حتما تابع را به تکه های مناسب تقسیم کن و اسم های خوب انتخاب کن و کد را ریفکتور کن

حرف پایانی

1. برنامه رو به چشم یه داستانی که باید تعریف شه ببینید، نه به چشم یه برنامه ای که باید بنویسید.
2. از امکانات زبانی که انتخاب کردید به بهترین نحو استفاده کنید که این داستان را به بهترین شکل ممکن بیان کنید
3. تابع ها افعال داستان شماست، فعل هایی که تو این داستان اتفاق میفته با تابع ها بیان میشن.