Clean Code

Chapter 8: Boundaries

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

فصل هشتم: مرز ها

این فصل در رابطه با نحوه استفاده از کتابخانه ها و پکیج های خارجی در برنامه مون صحبت میکنه. منظور از مرز ها هم، مرز برنامه ما و پکیج خارجی که داریم استفاده میکنیم هستش. میگه این مرز باید تمیز باشه و کنترلش دست خودت باشه.

استفاده از کد های دیگران

همیشه یه تضادی بین نویسندگان پکیج ها و استفاده کنندگان آن پکیج وجود داره.

کسانی که پکیج ها رو مینویسن، تمایل دارن که اون پکیج قابل استفاده در انواع برنامه های مختلف باشه و تا جای ممکن محدودیت ها رو بر میدارن که کاربر بتونه در برنامه ازش استفاده کنه.

از طرفی کاربر ها دنبال پکیج هایی هستن که به طور خاص نیاز اون ها رو برآورده کنه

این تضاد باعث به وجود اومدن یه سری مشکل ها در مرز های برنامه ما میشن (همون طور که گفتیم، منظور از مرز ها، اون بخشی از برنامه ما میشه که میخواد ازپکیج های خارجی استفاده کنه)

برای مثال، یه نگاهی به Map جاوا میندازیم (java.util.Map):

این ها بخشی از امکاناتی هستن که Map به ما میده:

- clear() void - Map
- containsKey(Object key) boolean - Map
- containsValue(Object value) boolean - Map
- entrySet() Set - Map
- equals(Object o) boolean - Map
- get(Object key) Object - Map
- getClass() Class<? extends Object> - Object
- hashCode() int - Map
- isEmpty() boolean - Map
- keySet() Set - Map
- notify() void - Object
- notifyAll() void - Object
- put(Object key, Object value) Object - Map
- putAll(Map t) void - Map
- remove(Object key) Object - Map
- size() int - Map
- toString() String - Object
- values() Collection - Map
- wait() void - Object
- wait(long timeout) void - Object
- wait(long timeout, int nanos) void - Object

همونطور که معلومه، Map خیلی فیچر های زیادی به ما ارائه میده و خیلی انعطاف پذیره. اما فرض کنید ما میخوایم از Map استفاده کنیم، اما نمیخوایم کاربران بتونن از Map حذف کنن چیزی رو. اگر از خود Map استفاده کنیم، یک تابع clear هست که کاربر ها میتونن با استفاده ازش، هر چیزی رو که بخوان حذف کنن.

یا فرض کنید بخوایم فقط یک سری آبجکت های خاص بتونن در Map ذخیره بشن. خود Map چنین امکانی رو به ما نمیده.

فرض کنید لازم داریم که یک لیست از Sensor ها رو ذخیره کنیم در Map.

اگر بخوایم از خود Map استفاده کنیم، مجبوریم این کارو کنیم:

```
Map sensors = new HashMap();
```

و موقع خواندن، این طوری فراخوانی کنیم:

```
Sensor s = (Sensor)sensors.get(sensorId );
```

اما این اتفاق فقط یک بار نمیفته، بلکه بارها و بارها در برنامه مجبوریم برای خواندن، از این راه استفاده کنیم.

مشکلش این جاست که خواندن و cast کردن به شیء مد نظر به کاربر محول شده است

این کد کار میکنه، اما قطعا تمیز نیست! و خیلی خوانا نیست.

در صورت استفاده از Generic Type ها، اوضاع یه مقدار بهتر میشه:

```
Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId );
```

اما این پیاده سازی هم مشکلی که در ابتدا گفتیم، یعنی فراهم کردن امکاناتی که نمیخوایم باشه (مثلا امکان حذف) رو داره.

روش خیلی بهتر و تمیزتر برای استفاده از Map به این شکله:

```
public class Sensors {
  private Map sensors = new HashMap();

  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }

  //snip
}
```

با این کار، interface ای که مرز برنامه ما و پکیجی که استفاده میکنیم هست، مخفی میشه. (یعنی مثلا تابع get() که خود Map داره، از دید کاربر مخفی میشه و ما خودمون این مرز رو کنترل میکنیم)

الان این کلاسی که خودمون ساختیم، امکان بزرگتر شدن و افزودن فیچر های بیشتر رو داره. همچنین عملیات casting و استفاده از generic type ها درون خود این کلاس اتفاق میفته و از دید کاربر مخفی میمونه. ضمن این که این رابط، دقیقا پاسخگوی نیاز های برنامه ماست.

مهم: ما (نویسنده کتاب!) نمیگیم که هر جا که میخواید از Map استفاده کنید، حتما اون رو به این شکل encapsulate کنید. بلکه توصیه میکنیم که آبجکت هایی از Map رو در برنامه تون این ور اون ور پاس ندید! یعنی نیاید یه آبجکت از Map درست کنید و اون رو به تابع های مخالف بفرستید یا چنین آبجکتی رو return نکنید. اگر میخواید از خود Map استفاده کنید، سعی کنید درون یه کلاس یا چند تا کلاس نزدیک به هم استفاده کنید و فراتر نرید.

کاوش کردن و یادگیری مرز ها

استفاده کردن از کتابخانه ها و کدهای دیگران، ویژگی های جدیدی به کد ما اضافه میکند. اما وقتی میخوایم از یه کتابخانه یا کد دیگری استفاده کنیم، از کجا شروع میکنیم؟

نوشتن تست برای کتابخانه هایی که استفاده میکنیم وظیفه ما نیست، اما به نفعمونه که واسشون تست بنویسیم.

ما ممکنه یکی دو روز زمان صرف خوندن داکیومنت های کتابخانه ای که ازش استفاده میکنیم بشیم. بعد ازش استفاده میکنیم. اما وقتی کدمون به مشکل بخوره گاها سخت میشه فهمید مشکل از کد ما بوده یا از کتابخونه ای که داریم ازش استفاده میکنیم و ممکنه زمان زیادی رو واسه دیباگ کردنش صرف کنیم.

چرا از یه رویکرد دیگه استفاده نکنیم؟

به جای این که تو production code با کتابخونه خارجی آشنا بشیم، میتونیم یه سری تست بنویسیم که هم باعث آشنایی ما با کتابخونه میشه و هم اینکه تست نوشتیم!

Jim Newkirk به این روش میگه learning tests.

این تست ها باید جوری که ما میخوایم از اون کتابخونه استفاده کنیم نوشته بشن (فقط اون بخش هاییش که لازم داریم)

یک مثال از Learning tests

فرض کنید میخوایم نحوه استفاده از کتابخونه log4j رو یادبگیریم.

بعد از دانلود و نصب و کمی خوندن داکیومنت، میتونیم یه تست ساده بنویسیم:

```
@Test
public void testLogCreate() {
   Logger logger = Logger.getLogger("MyLogger");
   logger.info("hello");
}
```

بعد از ران کردن این تست، پیغام خطا داده میشه که به یه چیزی به اسم Appender احتیاج هست.

بعد از یه مقدار مطالعه مجدد داکیومنت، میفهمیم یه چیزی به اسم ConsoleAppender وجود داره. پس چنین چیزی رو میسازیم و دوباره امتحان میکنیم

به این شکل:

```
@Test
public void testLogAddAppender() {
   Logger logger = Logger.getLogger("MyLogger");
   ConsoleAppender appender = new ConsoleAppender();
   logger.addAppender(appender);
   logger.info("hello");
}
```

این سری متوجه میشیم که این Appender هیچ output stream ای نداره. بعد از یه کمی گشتن تو گوگل به چنین کدی میرسیم:

```
@Test
public void testLogAddAppender() {
   Logger logger = Logger.getLogger("MyLogger");
   logger.removeAllAppenders();
   logger.addAppender(new ConsoleAppender(
       new PatternLayout("%p %t %m%n"),
       ConsoleAppender.SYSTEM_OUT));
   logger.info("hello");
}
```

و میبینیم که کار کرد! و پیغام hello تو کنسول نمایش داده شد.

جالبه! حالا بخش ConsoleAppender.SystemOut رو حذف میکنیم و میبینیم که همچنان hello چاپ میشه تو خروجی. اما وقتی PatternLayout رو حذف میکنیم خطا میده که به output stream احتیاجه. و این رفتار خیلی عجیب غریبه!

یه کم که داکیومنت رو میخونیم، متوجه میشیم که constructor دیفالت ConsoleAppender کانفیگ نشده، که خیلی چیز خوبی نیست این کار و مثل یه باگ یا حداقل ناسازگاری تو log4j میمونه.

بعد از یه کم ور رفتن باهاش و گوگل کردن و اینا، در نهایت به چند تا تست زیر میرسیم. این خیلی خوبه! چون علاوه بر این که دانشمون تو اون زمینه بیشتر شده، اون دانش رو هم در قالب چند تا تست نوشتیم.

```
@Before
public void initialize() {
    logger = Logger.getLogger("logger");
    logger.removeAllAppenders();
    Logger.getRootLogger().removeAllAppenders();
}
```

```
@Test
public void basicLogger() {
    BasicConfigurator.configure();
    logger.info("basicLogger");
}
```

```
@Test
public void addAppenderWithStream() {
 logger.addAppender(new ConsoleAppender(
      new PatternLayout("%p %t %m%n"),
      ConsoleAppender.SYSTEM_OUT));
    logger.info("addAppenderWithStream");
}
```

```
@Test
public void addAppenderWithoutStream() {
 logger.addAppender(new ConsoleAppender(
      new PatternLayout("%p %t %m%n")));
    logger.info("addAppenderWithoutStream");
}
```

Learning Test ها بهتر از هیچی ان!

Learning test ها در نهایت هیچ هزینه ای برای ما ندارن. ما به هر حال باید api اون کتابخونه رو یاد بگیریم و نوشتن تست ها یه راه خیلی خوب واسه یاد گرفتنشونه.

نه تنها این تست ها هزینه نداره واسمون، بلکه از جهاتی خیلی هم مفیدن. مثلا اگه اون کتابخونه بعدا نسخه جدید ریلیز کنه، خیلی خیلی راحت میشه به نسخه جدید سوییچ کرد. چون که تست نوشتیم! بعد از ارتقا به نسخه جدید، یه بار تست ها رو ران میکنیم و اگه جاییش خطا داده، راحت میفهمیم نسخه جدید کجاها با نسخه قبلی فرق میکنه و میریم تو کدمون اصلاح میکنیم (البته طبق توضیحات قبلی اگه boundary نوشته باشیم واسه کتابخونه، فقط تو یک جا اصلاح میکنیم)

بنابراین نوشتن boundary test ها خیلی توسیه میشه (تست هایی که کتابخونه های خارجی رو تست میکنن)

در صورتی که این تست ها رو ننویسیم، با اومدن نسخه جدید اون کتابخونه ما تمایل داریم که به ورژن جدید مهاجرت نکنیم، چون نمیدونیم که کجاها ممکنه ناسازگاری پیش بیاره.

استفاده از کدی که هنوز وجود نداره

با استفاده از Adapter Pattern ! (خیلی پترن خوبیه!)

یه مرز دیگه وجود داره و اون مرز بین شناخته ها و بخش های شناخته نشده هستش.

خیلی وقت ها پیش میاد لازم میشه از یک سری API هایی در برنامتون استفاده کنید که هنوز نوشته نشدن (مثلا قراره توسط یه تیم دیگه یا یه میکروسرویس دیگه تامین بشه)

یه راهش اینه که صبر کنیم ببینیم کی آماده میشه و هر وقت آماده شد بریم و ازش استفاده کنیم تو کدمون

اما میشه از Adapter Pattern استفاده کنیم. به این صورت که قبلش هر جیزی که لازم داریم اون ای پی ای بهمون بده رو تو یه کلاس دیگه شبیه سازی میکنیم و هر وقت آماده شد، شبیه ساز رو حذف میکنیم و Adapter Pattern خودمون رو مینویسیم.

نویسنده میگه که سال ها قبل عضو تیمی بوده که داشتند رو یک محصول ارتباط رادیویی کار میکردند.

یک زیرسیستم بود به نام Transmitter که هنوز آماده نشده بود و قرار بود این تیم از اون استفاده کنه.
اطلاعات خیلی کمی هم از این زیرسیستم داشتن. از طرفی هم نمیخواستن منتظر بمونن تا این زیرسیستم آماده بشه و بعد ادامه بدن کارشون رو.

از طرفی یه چیز خیلی مبهمی از نحوه کاری که قرار بود این زیرسیستم انجام بده میدونستن، اما تقریبا میدونستن که قراره چه کاری رو انجام بده. میدونستن کاری که قراره انجام بده اینه: transmitter رو روی فرکانس داده شده تنظیم کنه و شکل آنالوگ اطلاعات داده شده رو ارسال کنه.

واسه این که کارشون معطل نشه، یه interface ساختن به اسم Transmitter و یه تابع واسش تعریف کردن به اسم transmit که دو تا پارامتر میگرفت: ۱)datastream ۲) فرکانس

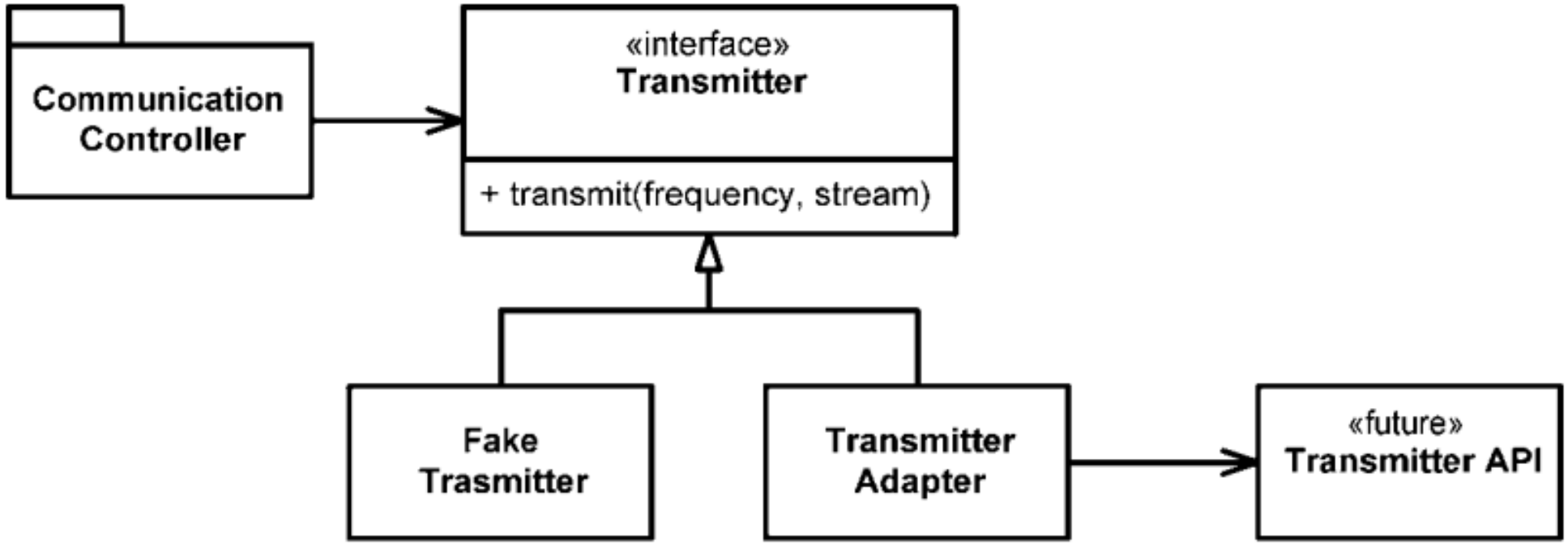

Figure 8-2
Predicting the transmitter

در شکل ۸-۲ میبینید که CommunicationsController رو با Transmitter (که تحت کنترلشون نبود) جدا کردند.

با استفاده از اینترفیسی که درست کرده بودند، CommunicationsController رو تمیز نگه داشتند.

زمانی که Transmitter API آماده شد، یک TransmitterAdapter نوشتند و ارتباط بین Transmitter API و Transmitter را تشکیل دادند.

استفاده از الگوی طراحی Adapter باعث شد ارتباط با API کپسوله شود (یعنی همه ی ارتباط با API در یک کلاس قرار گرفت) و برای تغییر ای پی ای تنها یک جا تغییر داده شود. همچنین باعث شد امکان نوشتن تست برای API خیلی راحت تر فراهم بشه.

جمع بندی (مرز های تمیز)

یه نرم افزار خوب، با کمترین هزینه و تلاش، با تغییرات تطبیق پیدا میکنه.

وقتی از کدی که در کنترل ما نیست استفاده میکنیم، باید خیلی مراقب باشیم و مطمئن باشیم که تغییرات اون کد، کمترین هزینه رو برای ما داشته باشن.

کد های خارجی، نیاز به مرز شفاف و تمیز و تست هایی برای بیان انتظارات خودمون از اون کد خارجی دارند.

ما نباید اجازه بدیم که بخش های زیادی از کد ما، راجع به کد خارجی بدونن (یعنی نبای کد ما به طور مسقیم و بدون مرز شفاف با کد خارجی، ارتباط برقرار کنن)

بهتره که به چیزی که خودمون تسلط داریم روش متکی باشیم تا چیزی که تسلط نداریم (کد های خارجی!).

مبادا اون ها ما رو کنترل کنن!

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