سوال اول: تست کردن متد های پرایوت

کنت بک در لینک ارائه شده و گفته است نه! پاسخ ما هم به این سوال منفی میباشد. اما چرا؟
در ابتدا توجه شود دلیلی داشته این متدها پرایوت انتخاب شده اند. تست کردن متدهای پرایوت فلسفه پرایوت بودن آنها را زیر سوال میبرد. این متدها وظیفه abstraction رو بر عهده دارند که با تست کردن آنها این ویژگی از آنها سلب میشود. به نوعی اینگونه از متدها، جزئیات پیاده سازی ای هستند که نباید به بیرون داده شوند. برخی مخالف هستند و اعتقاد دارند اینگونه سریع تر میتوان باگها را کشف کرد. هرچند پاسخ مناسب تری در این لینک داده شده است. واقعیت این است که عملکرد این متدها نیز برای کشف باگ و افزایش کیفیت کد لازم است چک شود اما نه به طور مستقیم. متدهای پرایوت موجود، در متدهای پابلیک صدا شده خواهند شد و با سناریو ی درست، ما میتوانیم نتیجه فراخوانی آنها را مشاهده و صحت عملکرد آنها را بررسی کنیم. با این کار، ما به رفتار و عملکرد کلاس به جای جزئیات داخلی پیاده سازی توجه خواهیم کرد که خود روش مناسبی برای تست میباشد.
برخی نیز ریفکتور منطق متدهای پرایوت به چندین متد پابلیک و تست آنها را مطرح میکنند. در برخی مواقع ناگزیر به تست متدهای پرایوت هستیم و بررسی آنها به وسیله متدهای پابلیک دشوار خواهد بود. در این لینک به این مورد اشاره شده و ما روش هایی همچون تغییر visibility، دسترسی dynamic و reflection را بدین منظور میتوانیم استفاده کنیم. در کل اگر طراحی کد مناسب باشد، انتظار میرود این مشکل رخ ندهد و وجود آن نشانه ای از نیاز به تغییر و ریفکتور کد میباشد.

سوال دوم: یونیت تیست و کد مولتی ترد

Unit test معمولاً برای تست واحدهای کوچک کد (معمولاً توابع و methodها) استفاده می‌شود تا از عملکرد صحیح آن واحدها اطمینان حاصل کنیم. یونیت تست‌ها معمولا به صورت مستقل از دیگر اجزای برنامه اجرا می‌شوند و باید تا حد ممکن از وابستگی به اجزای دیگر کد کاسته شوند. پس در نتیجه تست کردن کدی که multi-threaded است را به راحتی نمیتوان از unit test استفاده کرد. چون اجرا شدن کد به صورت multi-threaded به عوامل مختلفی بستگی دارد و عملا رفتار non-deterministic دارد.
در واقع به چندین علت unit test برای این کد ها مناسب نیست چون همانطور که گفته شد کد های multi-threaded میتوانند رفتار non-deterministic داشته باشند که عملا باعث میشود رفتارهای مختلف به ازای چیزهای یکسان مشاهده کنیم. همچنین ممکن است شامل race condition باشند یعنی یک سری رفتار های سیستم به علت بروز یک توالی یا گذر زمان به وجود بیاید که این امر در multi-threading ممکن است

و همچنین هنگام استفاده از آن و تست کردنش علاوه بر کد ما سیستم عامل و سخت افزار ما خاص منظوره تر تحت تاثیر قرار می گیرند که باعث میشود نتوانیم از unit test استفاده کنیم همچنین خود thread ها هم باهم interaction خواهند داشت که باز کار را پیچیده تر میکند. اما برای برخی قسمت های آن که ارتباطی به اجرا شدن به صورت multi-threaded را ندارند میتوان از آن استفاده کرد و از درست بودن آن قسمت اطمینان حاصل کرد. مثلا اگه کدی یکسان بین چندین thread اجرا میشود میتوان آن تکه یا صدا زدن method را از unit test بهره برد.

سوال سوم: بررسی نمونه تست های ارائه شده

testA:

مشکل این تست این است که همیشه pass میشود چون به جای assert کردن فقط نتیجه را print میکند.

به جای پرینت کردن باید کد زیر را نوشته شود تا اصلاح شود:

```
assertEqual(10, result);
```

testC:

در junit برای تست کردن throw کردن چیزی به نام except که در signature تابع نوشته شده است وجود ندارد بلکه باید از assertThrows استفاده کرد.

به صورت زیر:

```
assertThrows(Exception.class, () -> {
        New AnotherClass().process(badInput);
});
```

تست 3:

مشکل این تست این است که معلوم نیست تست testResourceAvailbility بلافاصله بعد از تست testInitialization اجرا شود و برای initialize کردن قبل از هر test از annotation به نام BeforeEach استفاده کرد که به صورت زیر میشود:

```
@BeforeEach
Public void initialization() {
        Configuration.initialize();
```

```
        ResourceManager.initializ();
}
```

لینک صفحه گیتهاب پروژه: https://github.com/Ipouyall/SoftwareTestingCourse