## ریپو گیت‌هاب

**Repository:** <https://github.com/elahekhodaverdi/SWT-Fall03>

**Commit Hash:**

## باگ‌های حل شده در کد‌‎‌

یک باگ در کد مربوط به maxSeatNumber در کد زیر وجود داشت:

| public int getMaxSeatsNumber() {  return tables.stream().map(Table::getSeatsNumber).max(Integer::compareTo).orElse(1); } |
| --- |

در قطعه کد بالا هنگامی که میزی وجود نداشته باشد مقدار 1 برمی‌گرداند که اشتباه است و باید 0 برگرداند.

با تست testGetMaxSeatsNumberWithNoTables این باگ پیدا شد.

## **سوال** اول

متدهای assume به طور کلی در مواقعی استفاده می‌شوند که می‌خواهیم مشخص کنیم تحت چه شرایطی تست ما معتبر است و باید اجرا شود. در صورتی که این شرایط برقرار نباشند، تست اجرا نمی‌شود. عدم برقراری شرط به معنای وجود ایراد در کد یا شکست تست نیست، بلکه به این معناست که در آن شرایط خاص، اجرای تست اطلاعات مفیدی ارائه نمی‌دهد.

زمانی که از متدهای assume استفاده می‌کنیم، اگر شرط برقرار نباشد، تست به‌طور خودکار fail نمی‌شود، بلکه skip می‌شود و به عنوان failed در نظر گرفته نمی‌شود.

| **description** | **method** |
| --- | --- |
| اگر عبارتی بگیرد که حاصل آن می‌شود false تست اجرا می‌شود در غیر اینصورت تست ignore می‌شود. | **assumeFalse** |
| اگر exception‌ای نباشد تست اجرا می‌شود و اگر باشد ignore/skip می‌شود. | **assumeNoException** |
| اگر با null یا یک آرایه‌ای از null ها فراخوانی شود تست ignore می‌شود. | **assumeNotNull** |
| دو آرگومان می‌گیرد؛ اولی مقداری ‌است که می‌خواهیم شرط را روی آن اعمال کنیم و بعدی یک matcher از Hamcrest که با کمک آن شرط را تعریف می‌کنیم | **assumeThat** |
| برعکس assumeFalse. اگر عبارتی بگیرد که حاصل آن می‌شود true تست اجرا می‌شود و در غیر اینصورت تست ignore می‌شود. | **assumeTrue** |

### assumeTrue:

نحوه کار کلی این متد در جدول توضیح داده شده است. ابتدا به دو فرم فراخوانی این تابع می‌پردازیم:

assumeTrue(boolean b)

assumeTrue(String message, boolean b)

در هر دو b یک عبارت است که در نهایت باید نوع آن boolean باشد.

کاربرد‌ها: از این متد همانند دیگر‌ متدها زمانی که می‌خواهیم تست تحت شرایط خاصی اجرا شود استفاده می‌کنیم مثال‌ها:

* در یک سیستم عامل خاص
* در دسترس بودن منابع مانند پایگاه داده
* تحت نسخه خاصی از جاوا و موارد مشابه
* یک مورد دیگر که در کلاس عنوان شد می‌توان گفت هنگام استفاده از مقادیر رندوم برای پارامترهای تست. اگر مقادیر تحت شرایطی صحیح باشند و همه مقادیر قابل قبول نباشند می‌توان نیز از این متد استفاده کرد.
* هر شرایط و محیط خاص دیگری که تحت آن تست باید اجرا شود و در غیر اینصورت باید Skip شود.

## سوال دوم

خیر. با استفاده از یک unit test نمی‌توانیم برای اطمینان از صحت عملکرد یک کد multi-thread اطمینان حاصل کنیم زیرا به دلیل قطعی نبودن و متغیر‌های بسیار هیچوقت نمی‌توانیم اطمینان حاصل کنیم که اگر برنامه برای n بار تست شده است و درست بوده برای بار بعدی نیز درست خواهد بود زیرا ممکن است race condition پیش بیاید (زمانی که به یک resources های مشترک دسترسی لازم است) یا حتی ممکن است با deadlock مواجه شویم. برای تست کردن چنین برنامه‌هایی می‌توانیم از mock کردن یا استفاده از concurrency utility‌های جاوا و یا boundary testing یا لاگ کردن(که نیازمند زمان و تلاش بیشتری برای رصد است) استفاده کنیم. از طرفی باید به این نکته توجه داشت که در این حالت می‌توان تست هایی را به صورت مکرر اجرا کرد و در صورتی که تعداد خوبی از این تست ها پاس شده باشند به میزان خوبی از عملکرد کد اطمینان پیدا کرد. اما باز هم بخاطر رندوم بودن راه مناسبی نخواهد بود.

## سوال سوم

از طرفی، با کمک کنسول شاید بتوانیم اطلاعات بیشتری در خروجی ببینیم، اما این می‌تواند گیج‌کننده باشد، مخصوصاً اگر با اروری مواجه شویم. یکی از اهداف استفاده از assert این است که به‌سادگی از موفقیت یا شکست تست مطلع شویم، بدون اینکه مجبور باشیم خطوط زیادی از خروجی کنسول را بخوانیم و چک کنیم. همچنین استفاده از کنسول یا نیازمند بررسی دقیق خروجی برای فهمیدن موفقیت یا شکست تست است، یا باید از ساختار استانداردی استفاده کنیم که در نهایت همان نتیجه‌ای را به دست می‌دهد که با استفاده از assert به آن می‌رسیم. اگر از روش اول (کنسول) استفاده کنیم، این کار بسیار وقت‌گیر و خسته‌کننده خواهد بود و با هدف اصلی ما از نوشتن تست‌ها در تضاد است، زیرا تست‌ها باید کار ما را آسان‌تر کنند. همچنین تمام تست‌هایی که به صورت مثال عنوان شده نوشته می‌شوند pass می‌شوند و fail نمی‌شوند، در نتیجه برای بررسی نتیجه تست‌ها هنگامی که از کنسول استفاده می‌کنیم، نیازمند نیروی انسانی خواهیم بود و این هزینه و زمان فرآیند تست را افزایش می‌دهد.

## سوال چهارم

### الف)

یک ایراد این تست نام‌گذاری اشتباه است و نامگذاری به این صورت است که نشان نمی‌دهد تست چه چیزی را تست می‌کند و انتظار چه نتیجه‌ای دارد.

اشکال دوم این است که expects یک keyword معتبر نیست و می‌توانستیم به جای آن از throws استفاده کنیم و یا حتی از Annotation‌ای مثل except استفاده کنیم. نکته دیگر این است که در صورت نوشتن throws نیز این تست‌ نیز fail می‌شود زیرا با نوشتن throw به این صورت اجبار کرده‌ایم که تست حتما throw کند و باید به جای این‌کار از assertion‌ها استفاده کنیم. Assertion ای که در این موقعیت استفاده می‌شود assertThrows است.

### ب)

مشکل این تست مستقل نبودن تست‌ها و وابستگی آن‌ها به هم است. در نتیجه نوشتن چنین تست‌هایی ترتیب در برنامه مهم است و تست ها به یکدیگر وابسته می‌شوند. برای جلوگیری از این مشکل باید به جای وابستگی تست ها به هم. از @BeforeEach استفاده کرده و حالت اولیه همه تست‌ها را یکسان بگیریم و هر آنچه هر تستی مازاد نیاز دارد را در همان تست هندل کنیم. برای مثال تست اصلاح شده به صورت زیر خواهد بود:

| public class TestCalculator {   Calculator fixture;   @BeforeEach  public void setUp() {  fixture = Calculator.getInstance();  fixture.setInitialValue(0);  }   @Test  public void testAccumulate() {  int result = fixture.accumulate(50);  Assertions.assertEquals(50, result);  }   @Test  public void testSubsequentAccumulate() {  fixture.accumulate(50);  int result = fixture.accumulate(100);  Assertions.assertEquals(150, result);  } } |
| --- |