ریپو گیت‌هاب

Repository: https://github.com/elahekhodaverdi/SWT-Fall03
Commit Hash: 0b004e2e8981e2f20f498c640ce9db29a8aa8fd6

باگ‌های حل شده در کد

در قسمت reservationController وقتی getReservations را صدا می‌زنیم پارامتر تاریخ دلخواه است و می‌تواند نباشد. در کد داده شده در هنگام نبود تاریخ ما خطا برمی‌گردانیم. برای حل این مشکل کافی است که کد این متد را به نحو زیر تغییر دهیم و یک شرط اضافه کنیم.

```
if (date != null) {
   try {
       localDate = LocalDate.parse(date, DATE_FORMATTER);
   } catch (Exception ex) {
       throw new ResponseException(HttpStatus.BAD_REQUEST,
PARAMS_BAD_TYPE);
   }
}
```

و با این تست متوجه این باگ شدیم:

```
@Test
void testGetReservationsWhenDateIsMissing() throws UserNotManager,
TableNotFound, InvalidManagerRestaurant, RestaurantNotFound  {
   List<Reservation> reservations = new ArrayList<>();
   reservations.add(new Reservation(user, restaurant, table,
LocalDateTime.now().plusHours(-4)));
   reservations.add(new Reservation(user, restaurant, table,
LocalDateTime.now().plusHours(-6)));

   when(reserveService.getReservations(restaurant.getId(),
table.getTableNumber(), null))
           .thenReturn(reservations);

when(restaurantService.getRestaurant(restaurant.getId())).thenReturn(restaurant)
;

   Response response = reservationController.getReservations(restaurant.getId(),
table.getTableNumber(), null);

   assertNotNull(response);
   assertEquals(HttpStatus.OK, response.getStatus());
   assertTrue(response.isSuccess());
   assertEquals("restaurant table reservations", response.getMessage());
```

```
    assertEquals(reservations, response.getData());

    verify(restaurantService, times(1)).getRestaurant(restaurant.getId());
    verify(reserveService, times(1)).getReservations(restaurant.getId(),
table.getTableNumber(), null);
}
```

سوال اول

State Verification

در این رویکرد، هدف بررسی وضعیت نهایی سیستم پس از اجرای یک عملیات است. به‌طور خاص، تمرکز بر روی این است که آیا مقادیر متغیرها، اشیا، یا داده‌های ذخیره‌شده پس از اجرای عملیات به‌درستی به‌روزرسانی شده‌اند یا خیر. برای مثال این که آیا یک آبجکت به لیست اضافه شده است یا خیر یا آیا داده جدیدی به دیتابیس اضافه شده است یا خیر.
در این نوع اعتبارسنجی، فرآیند و مراحلی که برای رسیدن به این وضعیت طی می‌شود، اهمیت چندانی ندارد؛ بلکه نتیجه نهایی مهم است.
در اجرای State Verification می‌توانیم از Stub استفاده کنیم. Stubها آبجکت‌های ساده هستند که رفتارهای از پیش تعریف‌شده را تقلید می‌کنند تا به ما کمک کنند تمرکز خود را بر روی وضعیت نهایی سیستم معطوف کنیم.

Behavior Verification

در این رویکرد، هدف ما بررسی رفتار سیستم در حین اجرای عملیات است. این بدان معنی است که ما می‌خواهیم اطمینان حاصل کنیم که آیا سیستم کارهای مشخصی را در حین فرآیند انجام می‌دهد یا خیر. برای مثال آیا متدی با پارامترهای مشخصی فراخوانی می‌شود یا خیر یا اینکه مطمئن شویم متدی ابدا فراخوانی نمی‌شود و کارهایی از این قبیل. در این حالت، به جای بررسی نتیجه نهایی، تمرکز بر چگونگی اجرای عملیات است.
برای اجرای Behavior Verification می‌توانیم از Mock استفاده کنیم. Mockها علاوه بر تقلید رفتار آبجکت‌ها، می‌توانند فراخوانی متدها را track کنند و بررسی کنند که آیا متدهای مورد انتظار در جریان عملیات به‌درستی فراخوانی شده‌اند یا خیر.

سوال دوم

ابزار test-spy در unit-testing به منظور بررسی و مانیتور کردن رفتار توابع به کار گرفته می‌شوند. این ابزار به ما این امکان را می‌دهد تا بررسی کنیم که آیا یک سری اکشن های مدنظر ما اجرا شده یا نه و این کار

بدون بازرسی روی خروجی فانکشن به کمک این ابزار انجام پذیر است. با این ابزار می‌توان بررسی کرد که یک تابع مثلا چند بار اجرا شده و هر بار با چه آرگومان هایی اجرا شده است.
دلایل استفاده از این ابزار به شرح زیر است:

- Behavior Verification
- Indirect Outputs

انواع test-spies:

1. **Retrieval Interface** : در این مدل با تعریف یک retrieval interface روی test spy یک مدل داریم که اطلاعات را در اختیار قرار می‌دهد. پس از اینکه تست روی سیستم انجام شد از این رابط استفاده می‌کند تا خروجی‌های غیرمستقیم واقعی سیستم را از test spy دریافت کند. سپس این خروجی‌ها با بهره گیری از تابع assertion چک می‌شوند.

2. **Self Shunt** : در این الگو کلاس تست هم تست‌کننده و هم test spy را در سیستم تحت تست برعهده دارد و برای هر دو ایفای نقش می‌کند. در این روش به‌جای ایجاد یک test spy جداگانه کلاس تست خود را به عنوان جایگزین کلاس test spy به سیستم تحت تست معرفی می‌کند و از این طریق تعاملات و مقادیر ورودی به آن را ثبت می‌کند. به عنوان مثال اگر کلاس SUT وابسته به کلاس دیگری برای ثبت عملیات‌ها باشد کلاس تست می‌تواند با پیاده‌سازی رابط آن خود را به عنوان آن جایگزین کند. سپس با استفاده از instance variable ها در کلاس تست مقادیر واقعی پارامتر ها را ذخیره می‌کند. در انتها متد های تست با استفاده از assertion مقادیر را بررسی می‌کنند . این الگو به ساده‌سازی کد تست کمک می‌کند و برای تست‌های ساده و سریع بسیار مناسب است.
به صورت کلی می‌توان گفت که با ترکیب کلاس های test و test spy و ایجاد یک آبجکت واحد سیستم تحت تست با این کلاس واحد ارتباط گرفته تا مقادیر در این کلاس ذخیره و در نهایت بررسی شوند.

3. **Inner Test Double** : یک روش رایج برای پیاده‌سازی test spy به صورت یک test double که به صورت hard coded پیاده سازی شده است این است که آن را به‌صورت یک کلاس داخلی یا بلوکی در داخل متدی که وظیفه تست را بر عهده دارد بنویسیم و طوری تنظیم کنیم که مقادیر اصلی را در instance variable ها و یا local variable ها ذخیره کند که برای تابع تست قابل دسترسی باشند. این در واقع روش دیگری برای پیاده‌سازی **Self Shunt** محسوب می‌شود.

4. **Indirect Output Registry** : یک امکان دیگر برای پیاده‌سازی test spy این است که پارامترهای اصلی برنامه را در یک مکان مشخص و ذخیره کند تا توابع تست بتوانند به آنها دسترسی داشته باشند. برای مثال می‌توان این اطلاعات را در یک فایل، رجیستری، یا یک منبع ذخیره‌سازی دیگر نگه دارد. این روش به توابع تست اجازه می‌دهد که پس از اجرای تست های روی سیستم به اطلاعات مورد نیاز دسترسی پیدا کند و آن‌ها را برای تایید صحت خروجی‌ها بررسی کند.

سوال سوم

الف)

دو دلیل اصلی برای استفاده از Shared fixture به جای fresh fixture وجود دارد. دلیل اول سرعت است. گاهی می‌خواهیم فرایند تست سریع‌تر شود و استفاده از fresh fixture زمان زیادی برای راه‌اندازی fixtureها می‌گیرد، زیرا برای هر تست این فرایند دوباره انجام می‌شود. در این شرایط از Shared fixture استفاده می‌کنیم. هرچند پیچیدگی افزایش می‌یابد و تست‌ها به یکدیگر وابسته می‌شوند، که یافتن مشکل را دشوارتر می‌کند.

دلیل دوم این است که گاهی یک توالی طولانی از فعالیت‌ها داریم که هرکدام به قبلی وابسته هستند. اگر بخواهیم کل فرایند را تست کنیم، باید یک تست داشته باشیم که هر فعالیت را به ترتیب فراخوانی کند. هرچند، این روش همان‌طور که به نظر می‌رسد برای تست مناسب نیست، زیرا عیب‌یابی آن نیز دشوار است. در این حالت، از یک shared fixture استفاده می‌کنیم و هر فعالیت را به عنوان یک تست جداگانه در نظر می‌گیریم که به ترتیب باید اجرا شوند. در نتیجه، یک زنجیره‌ای از تست‌ها خواهیم داشت.

دلایل دیگری نیز وجود دارد، مانند این‌که ممکن است تست‌ها در محیطی انجام شوند که بین تست‌ها تغییری نمی‌کند یا تست‌ها تاثیری بر محیط نمی‌گذارند و به طور کلی تست‌ها تاثیری بر یکدیگر ندارند.

ب)

مزیت‌ها:

در Lazy Setup منابع فقط زمانی تعریف می‌شوند که به آن‌ها نیاز باشد. بنابراین، اگر یک تست به منابع خاصی نیازی نداشته باشد، آن‌ها تعریف نمی‌شوند. در مقابل، در Suite Fixture Setup ممکن است منابعی در ابتدا ایجاد شوند که در بسیاری از تست‌ها استفاده نمی‌شوند، اما هزینه آن‌ها پرداخت می‌شود. به همین دلیل Lazy Setup بهینه‌تر است و بازدهی بیشتری دارد.
همچنین، به دلیل این که منابع تنها در صورت نیاز تعریف می‌شوند، Lazy Setup از حافظه کمتری در مقایسه با Suite Fixture Setup استفاده می‌کند.
از طرفی، در Lazy Setup این امکان وجود دارد که برای هر تست، منابع متفاوتی به‌صورت اختصاصی تعریف شوند. در حالی که در Suite Fixture Setup تصور می‌شود که منابع ثابتی داریم که باید در تمام تست‌ها استفاده شوند، که این باعث ایجاد محدودیت‌هایی در تعریف منابع برای همه تست‌ها می‌شود.

معایب:

در Lazy Setup، چون منابع در هر تست جداگانه تعریف می‌شوند، بسیاری از تست‌ها ممکن است به منابع یکسانی نیاز داشته باشند و این منجر به تکرار کد خواهد شد. در مقابل، در Suite Fixture Setup منابع یک‌بار برای تمام تست‌ها تعریف می‌شوند و از تکرار کد جلوگیری می‌شود.

در Lazy Setup، باید در هر تست منابع مورد نیاز مشخص شوند که این کار باعث افزایش پیچیدگی می‌شود. در حالی که در Suite Fixture Setup، منابع یک‌بار و به‌صورت مرکزی تعریف می‌شوند و این کار مدیریت آن‌ها را ساده‌تر می‌کند.
اگر تست‌های زیادی نیاز به تعریف منابع یکسان داشته باشند، در مقایسه با Suite Fixture Setup زمان بیشتری نیاز است، زیرا هر بار منابع برای هر تست به‌صورت جداگانه تعریف می‌شوند.

ج)

به صورت کلی باید شرایطی را فراهم کرد که در صورت تغییر بتوان به استیت اولیه برگشت. می‌توان از روش های زیر کمک گرفت:

- **استفاده از ساختار داده های immutable**

این داده ها به ما کمک می‌کنند که حتی در صورت تغییر در آنها یک کپی از آنها تهیه شده و داده اصلی immune باقی بماند

- **استفاده از قابلیت transaction rollback در دیتابیس ها**

با استفاده از این قابلیت می‌توان تمامی تغییرات لازم در یک تست را در یک تراکنش قرار داده و آن را در انتها rollback کنیم.

- **استفاده از snapshot در دیتابیس ها**

با تهیه snapshot از دیتابیس در ابتدای تست و بازگردانی آن در انتها می‌توان از داده ها محافظت کرد.

توجه شود این ویژگی برای سایر متغیر ها نیز قابل اجراست.

- **استفاده از setup/teardown hooks**
- **تبدیل کردن داده ها به تایپ read-only**

در این مدل باید توجه شود که اجازه هرگونه تغییر در داده ها از تست سلب می‌شود.