

ریپو گیت هاب

Repository: <https://github.com/elahekhodaverdi/SWT-Fall103>

Commit Hash: ab6487b73fe3d6ec5733261801b893b980ceebcd

باگ های حل شده در کد

در قسمت reservationController وقتی getReservations را صدا می زنیم پارامتر تاریخ دلخواه است و می تواند نباشد. در کد داده شده در هنگام نبود تاریخ ما خطا برمی گردانیم. 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. به طور خاص، تمرکز بر روی این است که آیا مقادیر متغیرها، اشیاء، یا داده‌های ذخیره‌شده پس از اجرای عملیات به درستی 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 مراحل 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 اِدا 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 به درستی 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 اکشن‌های مدنظر ما اجرا شده یا نه و این کار

بدون بازرسی روی خروجی فانکشن به کمک این ابزار انجام پذیر است. با این ابزار می توان بررسی کرد که یک تابع مثلاً 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
- Dependency isolation
- Increased Test Coverage

انواع test-spies:

1. Spy: در این مدل بررسی می شود که آیا یک تابع فراخوانی شده و اگر بله چندین بار و با چه ورودی هایی
2. Stub: این مدل به ما کمک می کند تا برای هر تابع رفتار خاصی را تعریف کنیم بدین صورت که حین فراخوانی چه مقادیری را بازگرداند یا چه خطایی را بیندازد. به صورت کلی ما می توانیم با این روش سناریو های مختلف را بدون فراخوانی تابع شبیه سازی کنیم.
3. Mock: این مدل یک spy است که روی آن دسته ای از expectation ها ست شده است. در این مدل بنا بر این است که بررسی شود که آیا یک تابع با یک سری آرگومان مشخص فراخوانی می شود یا نه. از طرفی این مدل به نحوی کارساز است که مثلاً ما در رویه تست خود نیاز به فراخوانی تابعی برای خواندن دیتابیس داریم ولی چون این عمل زمان بر است از mock برای این منظور در تست ها بهره می بریم.

سوال سوم

(الف)

دو دلیل اصلی برای استفاده از Shared fixture به جای fresh fixture وجود دارد. دلیل اول سرعت است. گاهی می خواهیم فرایند تست سریع تر شود و استفاده از fresh fixture زمان زیادی برای راه اندازی fixture ها می گیرد، زیرا برای هر تست این فرایند دوباره انجام می شود. در این شرایط از Shared fixture استفاده می کنیم. هرچند پیچیدگی افزایش می یابد و تست ها به یکدیگر وابسته می شوند، که یافتن مشکل را دشوارتر می کند.

دلیل دوم این است که گاهی یک توالی طولانی از فعالیت ها 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 همان طور که به نظر می رسد برای تست مناسب نیست، زیرا عیب یابی آن نیز دشوار است. در این حالت، از یک shared fixture استفاده می کنیم و هر فعالیت را به عنوان یک تست جداگانه در نظر می گیریم که به ترتیب باید اجرا شوند. در نتیجه، یک زنجیره ای از تست ها خواهیم داشت.

دلایل دیگری نیز وجود دارد، مانند این که 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 به صورت 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 به صورت 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 به صورت جداگانه تعریف می‌شوند.

(ج)

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

- استفاده از ساختار داده‌های immutable

این داده‌ها 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 دیتابیس‌ها

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.