به نام او

شهنام فیضیان 810100197
عرفان دارایی 810100139

آزمون نرم‌افزار
گزارش بخش تئوری پروژه 2

آدرس مخزن https://github.com/ShahnamFeyzian/Software-Testing-Course

شناسه آخرین کامیت 13d697ea71fbc1eefaef5a5c279cbb4c469fe04d

سوال اول

تفاوت State Verification و Behavior Verification:

State Verification یعنی اینکه بعد از اجرای تست، وضعیت نهایی سیستم رو بررسی کنیم؛ به این صورت که بعد از صدا زدن یک متد از سیستم تحت تست (SUT)، با استفاده از Assertion ها چک می‌کنیم که آیا وضعیت نهایی سیستم یا هر شی‌ای که برمی‌گردونه همون چیزیه که انتظار داشتیم یا نه. اینجا فقط خروجی‌های مستقیم سیستم رو نگاه می‌کنیم و کاری به اینکه سیستم چجوری با بقیه قسمت‌ها تعامل داره نداریم.

ولی Behavior Verification یه جورایی پویاتر و دقیق‌تره و به رفتار سیستم موقع اجرای تست دقت می‌کنه. اینجا داریم می‌بینیم که SUT موقع اجرا چه تعاملاتی با بقیه اجزا داره و به اصطلاح خروجی‌های غیرمستقیم (یعنی اون تماس‌های غیرمستقیم یا همون تعاملات با بقیه) رو چک می‌کنیم. برای این کار از چیزایی مثل Test Doubles استفاده می‌کنیم تا ببینیم که سیستم چه رفتارهایی از خودش نشون می‌ده.

پس فرق اصلی اینه که تو State Verification داریم وضعیت نهایی رو بعد از اعمال تغییرات بررسی می‌کنیم، اما تو Behavior Verification تمرکز روی تعاملات سیستم و خروجی‌های رفتاری غیر مستقیم شه.

استفاده از mock یا stub:

Behavior Verification:

برای این که بفهمیم آیا SUT (سیستم تحت آزمایش) درست با وابستگی‌هاش تعامل کرده یا نه، از Mocks استفاده می‌کنیم. توی Mocks می‌تونیم انتظارات خاصی رو برای فراخوانی متدها مشخص کنیم و بعد از اجرای تست ببینیم که آیا این انتظارات برآورده شدن یا نه. مثلاً اگر SUT باید یه متد خاص رو با پارامترهای مشخصی صدا بزنه، می‌تونیم توی Mock مشخص کنیم که این متد باید فقط یک بار با این پارامترها فراخوانی بشه.

State Verification:

اینجا به Stubs نیاز داریم. Stubs برای شبیه‌سازی وابستگی‌ها استفاده می‌شن و پاسخ‌های از پیش تعیین‌شده‌ای به SUT می‌دن. معمولاً از Stubs استفاده می‌کنیم تا وضعیت نهایی SUT رو بعد از اجرای متد هاش بررسی کنیم. مثلاً اگر SUT باید یه متد خاص رو فراخوانی کنه که یه وضعیت خاص رو برمی‌گردونه، می‌تونیم از Stub استفاده کنیم تا پاسخی که به SUT داده می‌شه، دقیقاً همون چیزی باشه که نیاز داریم.

Mocks بیشتر برای Behavior Verification خوبن چون بیشتر رو تعاملات و رفتار تمرکز دارن.

Stubs بیشتر برای State Verification کاربرد دارد چون کمک می‌کنن وضعیت نهایی SUT رو کنترل کنیم.

سوال دوم

Test spy:

Test Spyها ابزارهایی هستن که برای behavior verification یک سیستم تحت آزمایش (SUT) استفاده می‌شن. وقتی که می‌خوایم خروجی‌های غیرمستقیم SUT رو بررسی کنیم و نمی‌تونیم پیش‌بینی کنیم که این خروجی‌ها چه‌طور قراره باشن، Test Spyها می‌تونن خیلی

مفید باشن. به طور کلی، Test Spyها به ما این امکان رو می‌دن که اطلاعات غیرمستقیم رو ثبت و بررسی کنیم و با استفاده از اون‌ها می‌تونیم رفتار SUT رو بهتر درک کنیم.

علت های استفاده:

تأیید خروجی‌های غیرمستقیم: وقتی SUT به روش‌های دیگه‌ای تماس می‌گیره و نمی‌تونیم اون‌ها رو مستقیم ببینیم، Test Spy می‌تونه به ما کمک کنه تا این خروجی‌ها رو ضبط و تأیید کنیم.

دسترسی به خروجی‌ها قبل از تأیید: این ابزار به ما این امکان رو می‌ده که همه‌ی تماس‌های خروجی SUT رو قبل از انجام تأییدات روی آن‌ها بررسی کنیم.

شفافیت در Assertions: وقتی می‌خوایم Assertions واضحی داشته باشیم و فکر می‌کنیم که روش‌های معمول کافی نیستن، Test Spy می‌تونه به ما کمک کنه تا تأییدات رو به‌صورت شفاف‌تری انجام بدیم.

تسهیل تشخیص خطا: اگر تست ما به خاطر عدم تطابق با رفتار مورد انتظار متوقف نشه، Test Spy می‌تونه اطلاعات بیشتری در مورد خطاهای تست ارائه بده و به ما کمک کنه تا مشکل رو بهتر شناسایی کنیم.

تأمین نیازهای خاص تست: اگر تست ما نیاز به برابری خاصی داشته باشه که با تعریف معمولی برابری هماهنگ نباشه، Test Spy می‌تونه مفید باشه.

انواع Test Spyها:

1. Retrieval Interface: یک کلاس جداگانه تعریف می‌کنیم که اطلاعات ضبط شده رو ارائه می‌ده و تست از طریق این رابط به داده‌ها دسترسی پیدا می‌کنه.
2. Self Shunt: Test Spy و کلاس تست رو به یک شیء واحد ترکیب می‌کنیم. SUT به این شیء تماس می‌گیره و اطلاعات رو ذخیره می‌کنه.
3. Inner Test Double: Test Spy رو به‌صورت یک کلاس داخلی ناشناخته در متد تست پیاده‌سازی می‌کنیم و اطلاعات رو در متغیرهای محلی ذخیره می‌کنیم.
4. Indirect Output Registry: Test Spy می‌تونه پارامترها رو در یک مکان مشخص ذخیره کنه، مثل فایل یا شیء Registry، تا متد تست بتونه بهشون دسترسی پیدا کنه.

سوال سوم

الف)

استفاده از Fixture Shared نسبت به Fixture Fresh زمانی مناسب‌تره که:

1. صرفه‌جویی در زمان اجرا: اگر تست های ما نیاز به راه‌اندازی و تخریب مکرر یک fixture با هزینه بالا (مثل ارتباط با پایگاه داده) دارن ، استفاده از یک fixture مشترک می‌تونه زمان اجرای تست‌ها رو به طور قابل توجهی کاهش بده.
2. وجود وابستگی‌های پیچیده: اگر تست های ما وابستگی‌های پیچیده‌ای به یک fixture دارن و این وابستگی‌ها به طور مکرر در تست‌های مختلف استفاده میشن ، استفاده از fixture مشترک می‌تونه به کاهش تکرار و آسون تر شدن مدیریت کمک کنه.
3. استفاده از داده‌های ثابت: اگر داده‌های مورد استفاده در تست ها به ندرت تغییر می‌کنن و نیاز به بازسازی مکرر ندارن، می‌تونیم از fixture مشترک برای جلوگیری از بار اضافی ناشی از ایجاد مجدد اون استفاده کنیم.
4. تست‌های تکراری: اگر تست های ما نیاز به مقایسه نتایج یا رفتارهای مشابه دارن و از یک پایه مشترک استفاده می‌کنن، fixture مشترک می‌تونه به یکپارچگی نتایج کمک کنه.

ولی باید در نظر داشت که استفاده از fixture مشترک ممکنه به بروز مشکلاتی مثل تست‌های ناپایدار و غیرقابل تکرار منجر بشه، پس باید از راهکارهای مناسب برای مدیریت این مشکلات استفاده کرد.

ب)

مزایا:

1. اجرای مستقل تست‌ها:

Lazy Setup به هر تست اجازه می‌دهد که به‌صورت مستقل اجرا شود، حتی اگر نیاز به فیکسچر مشترکی داشته باشند. این به کاهش مشکلات مربوط به تست‌های یتیم کمک می‌کند. در مقابل، Suite Fixture Setup نیاز دارد که همه تست‌ها در یک کلاس خاص قرار بگیرد و نمی‌توانند به‌طور مستقل اجرا بشوند.

2. کاهش زمان ساخت:

Lazy Setup فقط فیکسچر رو در زمان نیاز (زمانی که اولین تست به آن نیاز دارد) ایجاد می‌کند، که ممکن است باعث کاهش زمان کلی تست‌ها شود. در مقابل، Suite Fixture Setup فیکسچر رو یک بار در ابتدای مجموعه تست‌ها می‌سازد که ممکن است در بعضی موارد نیاز به زمان بیشتری برای ساخت داشته باشد.

3. استفاده از Garbage Collection:

Lazy Setup می‌تواند به‌راحتی از Garbage Collection برای از بین بردن فیکسچرهای غیرضروری استفاده کند، به‌خصوص زمانی که فیکسچر نیازی به پاک‌سازی دستی ندارد.

معایب:

1. مشکلات در تخریب فیکسچر:

در Lazy Setup، تشخیص زمان تخریب فیکسچر سخت است زیرا هیچ متد مشخصی برای این کار وجود ندارد. ممکن است فیکسچر زودتر از موعد پاک شود یا باقی بماند. در مقابل Suite Fixture Setup از متدهای مشخصی برای پاک‌سازی فیکسچر استفاده می‌کند که باعث اطمینان از پاک‌سازی صحیح می‌شود.

2. نیاز به مدیریت بیشتر:

Lazy Setup ممکن است به کد بیشتری برای مدیریت فیکسچر مشترک نیاز داشته باشد به‌خصوص اگر بخواهیم آن را بین چندین کلاس تست به اشتراک بگذاریم. در

مقابل Suite Fixture Setup معمولاً نیازی به این نوع مدیریت ندارد، چون به‌طور خودکار فیکسچر رو در هر بار اجرای مجموعه تست می‌سازد و پاک می‌کند.

3. ممکن است منجر به تست‌های متقابل شود:

استفاده از Lazy Setup ممکن است در بعضی موارد منجر به تست‌های متقابل (Interacting Tests) شود چون فیکسچر بین تست‌ها به اشتراک گذاشته می‌شود. در مقابل Suite Fixture Setup به دلیل ایجاد یک فیکسچر مشترک برای کل کلاس، این مشکل رو تا حدی کاهش می‌دهد.

ج)

استفاده از Fresh Fixture:

با استفاده از Fresh Fixture، هر آزمون می‌تواند یک Fixture مستقل و تازه ایجاد کرده که هیچ‌گونه وابستگی به وضعیت آزمون‌های دیگر نداشته باشد. این رویکرد باعث می‌شود که هر آزمون به طور کامل از تغییرات در Fixtureهای دیگه ایمن باشد، اما هزینه‌های زمانی و منابع بیشتری را به دنبال دارد.

Lazy Setup با Garbage-Collected Teardown:

اگر از Lazy Setup استفاده کنیم، می‌توانیم از Garbage-Collected Teardown هم کمک بگیریم تا مطمئن شویم که هر بار که Fixture ایجاد می‌شود، تنها در آزمون‌های لازم استفاده بشود و بعد از اتمام آن‌ها به طور خودکار پاک‌سازی بشود. این کار از ایجاد مشکلات ناشی از تغییرات ناخواسته در Fixture جلوگیری می‌کند.

استفاده از Prebuilt Fixture:

اگر بخواهیم از یک Fixture مشترک استفاده کنیم اما از به هم ریختن آن توسط آزمون‌های دیگر جلوگیری کنیم، می‌توانیم یک Prebuilt Fixture ایجاد کنیم که تنها در ابتدای اجرا به‌کار گرفته میشود و سپس در هر آزمون از آن به‌صورت مستقل استفاده کنیم. با این کار، وضعیت Fixture از تغییرات ناخواسته محافظت می‌شود.

Isolation: ایجاد شرایط ایزوله برای هر آزمون به‌گونه‌ای که داده‌ها و وضعیت Fixture در هر آزمون به‌صورت مجزا باشد. این شامل استفاده از روش‌هایی مثل Mocking یا Stubbing برای جداسازی وابستگی‌ها می‌شود.