گزارش پروژه چهارم آزمون نرم افزار

محمد مهدی صمدی (۸۱۰۱۰۱465) - مجید صادقی نژاد (۸۱۰۱۰۱۴۵۹)

گزارش PIT test coverage

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	98%	219/223	80%	123/153	82%	123/150

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.example.demo.features.order.service	4	99%	128/129	76%	70/92	77%	70/91
com.example.demo.features.user.service	4	97%	91/94	87%	53/61	90%	53/59

دامنه‌ی سفارش و پرداخت

نتایج پوشش کد

Name	Line Coverage		Mutation Coverage		Test Strength	
LoggingConfirmationSender.java	100%	17/17	22%	2/9	22%	2/9
OrderCheckoutService.java	100%	15/15	89%	8/9	89%	8/9
OrderServices.java	100%	39/39	89%	17/19	89%	17/19
SandboxPaymentGateway.java	98%	57/58	78%	43/55	80%	43/54

در این تصویر تعداد mutate های survived و killed مشخص است.
مشاهده می‌شود که کلاس LoggingConfirmationSender اسکورهای پایینی گرفته. به این علت است که این کلاس صرفا برای لاگ کردن است و همیشه یک خروجی یکسان دارد. در نتیجه track کردن استیت آن دشوار است زیرا فقط و فقط پیام‌هایی لاگ می‌شوند. در واقع هدف این کلاس این است که در هر شرایطی void ریترن کند (مثلاً اگر ایمیل صحیح، غلط، اسپم یا هرچه بود) پس در تست‌ها هم صرفاً همین مورد چک شده و منطقی است که score کمی بگیرد. به همین دلیل ممکن است mutant هایی که تولید می‌شود رفتار کاملا مشابه نسخه اصلی داشته باشند.

تحلیل چند Survived Mutant

متد charge در SandboxPaymentGateway

- **بخش تغییر یافته:** خط ۵۷، شرط مربوط به سقف مبلغ مجاز: if (amount > MAX_ALLOWED_AMOUNT)
- **تغییر انجام شده:** عملگر > به >= تغییر داده شده.

- **چرا زنده ماند؟** تست charge_returnsFalse_whenAmountExceedsSandboxLimit از عدد 1000.01 استفاده می‌کند. تستی نداریم که دقیقاً عدد 1000.0 را چک کند. به همین دلیل، وقتی کد به «بزرگتر یا مساوی ۱۰۰۰» تغییر می‌کند، منطق برنامه برای عدد ۱۰۰۰ عوض می‌شود اما چون این مرز را تست نکرده‌ایم، متوجه خطا نشدیم.

۲. متد calculateRiskScore در SandboxPaymentGateway

- **بخش تغییر یافته:** خط ۸۹، اولین پله محاسبه ریسک بر اساس مبلغ: if (amount > 200)
- **تغییر انجام شده:** این شرط را برعکس شده یا مرز آن را به amount >= 200 تغییر داد.
- **چرا زنده ماند؟** در تست‌ها فقط مبالغ 50.0 (خیلی کمتر) و 900.0 (خیلی بیشتر) را چک کردیم. هیچ تستی وجود ندارد که عدد 200.0 یا 200.01 را بررسی کند تا ببیند آیا واقعاً ۳۰ امتیاز ریسک در آن نقطه اضافه می‌شود یا خیر.

۳. متد sendConfirmation در LoggingConfirmationSender

- **بخش تغییر یافته:** خط ۲۵، جایی که در صورت نال بودن کاربر هشدار ثبت می‌شود: log.warn("Skipped confirmation: user details missing for amount {}", amount);
- **تغییر انجام شده:** فراخوانی این متد (void method call) حذف شده (Removed call to log).
- **چرا زنده ماند؟** تست sendConfirmation_doesNothing_whenUserIsNull فقط از assertDoesNotThrow استفاده می‌کند. این دستور فقط بررسی می‌کند که کد خطا ندهد. از آنجایی که حذف شدن یک دستور "Log" باعث کرش کردن برنامه نمی‌شود، تست شما همچنان پاس می‌شود و متوجه نمی‌شود که عملیات اصلی (ثبت لاگ) دیگر انجام نمی‌شود.

۴. متد hasTooManyDecimalPlaces در SandboxPaymentGateway

- **بخش تغییر یافته:** خط ۱۳۸، بررسی تعداد ارقام اعشار: return value.scale() > 2;
- **تغییر انجام شده:** عملگر > به >= تغییر یافته یا مقدار ثابت 2 را با عدد دیگری جایگزین شده.
- **چرا زنده ماند؟** در تست charge_returnsFalse_whenAmountHasTooManyDecimalPlaces شما عدد 12.479 (سه رقم اعشار) را تست کردیم که بزرگتر از ۲ است. اگر کد به scale() >= 2 تغییر کند، عددی مثل 12.47 (دو رقم اعشار) که باید مجاز باشد، غیرمجاز تلقی می‌شود. چون تستی برای عدد دقیق با **دو رقم اعشار** نداریم که خروجی true را تایید کند، این تغییر منطق زنده می‌ماند.

دامنه‌ی برنامه ی غذایی و انبار

نتایج پوشش کد

کلاس	Line Coverage	Branch Coverage
MealPlanService	(50/51) 98%	(25/26) 96.2%

Branch Coverage	Line Coverage	کلاس
(8/8) 100%	(8/8) 100%	NutritionBasedCalorieEstimator
(4/4) 100%	(13/15) 87%	PantryItem
(8/10) 80%	(20/20) 100%	SimplePantryInventory

کلاس‌های NutritionBasedCalorieEstimator و SimplePantryInventory پوشش بسیار بالایی دارند چون منطق اصلی‌شان با چند سناریوی ساده قابل پوشش است و شاخه‌های کمی دارند. PantryItem پوشش پایین‌تری دارد (87%) که نشان می‌دهد که یا بعضی getter ها توسط تست‌ها اجرا نشده‌اند، یا برخی مسیرهای ساده (مثل مسیرهای برنگشتن و برگشتن در حالت‌های خاص) کمتر پوشش داده شده‌اند. MealPlanService هم پوشش خط خیلی خوبی دارد، چون تست ها مسیر های ورودی نامعتبر، عدم کفایت منابع، و سناریوی تولید پلن را پوشش داده‌اند. Branch Coverage نشان می‌دهد چند درصد از شاخه‌های شرطی حداقل یک بار توسط تست‌ها اجرا شده‌اند. بنابراین ممکن است Line Coverage بالا باشد ولی هنوز بعضی حالت‌های شرط‌ها اجرا نشده باشند. در این پروژه، MealPlanService فقط یک branch را کامل پوشش نداده (25/26) و SimplePantryInventory هم ۲ branch از ۱۰ branch را پوشش نداده (8/10) که معمولاً مربوط به حالت‌های کم‌تر رخ‌دادنی ورودی‌هاست.

نتایج Mutation Testing

Name	Line Coverage		Mutation Coverage		Test Strength	
MealPlanService.java	98%	50/51	84%	21/25	84%	21/25
NutritionBasedCalorieEstimator.java	100%	8/8	100%	10/10	100%	10/10
PantryItem.java	87%	13/15	70%	7/10	88%	7/8
SimplePantryInventory.java	100%	20/20	94%	15/16	94%	15/16

خلاصه نتایج PIT برای ۴ کلاس هدف:

Mutation Score	No Coverage	Survived	Killed	Generated	کلاس
(21/25) 84%	0	4	21	25	MealPlanService
(10/10) 100%	0	0	10	10	NutritionBasedCalorieEstimator
(7/10) 70%	2	1	7	10	PantryItem
(15/16) 94%	0	1	15	16	SimplePantryInventory

جمع کل:
61 جهش تولید شد، 53 تا کشته شد، 6 تا زنده ماند، و 2 جهش پوشش داده نشد. Mutation Coverage کلی 87% بود.

تحلیل چند Survived Mutant

- MealPlanService - متد generateWeeklyPlan
جهش: dailyCalorieTarget / mealsPerDay به ضرب (*) تبدیل شده .
چرا زنده مانده: در تست‌ها سناریوی تولید پلن با mealsPerDay=1 اجرا شده؛ در این حالت تقسیم و ضرب نتیجه یکسان می‌دهند، پس رفتار تغییری نمی‌کند و تست شکست نمی‌خورد.
یک تست با mealsPerDay != 1 (مثلا 3) و چند محصول با کالری‌های متفاوت نزدیک به هدف هر وعده، می‌تواند ترتیب انتخاب غذا را تغییر دهد و این mutant را بکشد.

- MealPlanService - متد scoreProducts
جهش: targetPerMeal - calories به targetPerMeal + calories تبدیل شده
چرا زنده مانده: تست فعلی بیشتر روی محدودیت تکرار هفتگی و قابل تکمیل بودن برنامه تمرکز دارد و به این حساس نیست که دقیقا نزدیک‌ترین غذا به هدف کالری انتخاب شود یا نه. اگر معیار امتیازدهی خراب شود ولی برنامه هنوز تکمیل شود، تست‌ها ممکن است متوجه نشوند.
اگر در نیازمندی پروژه نزدیک بودن به هدف کالری مهم است، نوشتن تستی که انتخاب را بر اساس نزدیکی کنترل کند منطقی است وگرنه شاید ارزش زیادی نداشته باشد.

- MealPlanService – متد scoreProducts
جهش: حذف scored.sort (لیست امتیازها مرتب نمی‌شود).
چرا زنده مانده: در تست، لیست محصولات به همان ترتیبی داده شده که بعد از مرتب‌سازی هم همان ترتیب می‌ماند (عملاً ورودی از قبل مرتب بوده). پس حذف sort اثری روی خروجی نگذاشته است.

- PantryItem - متد reserve
جهش: شرط servingsRequested <= 0 به یک مرز متفاوت تغییر کرده (changed conditional boundary).
چرا زنده مانده: حتی اگر شرط از <= 0 به < 0 تبدیل شود، برای ورودی 0 کد وارد بدنه می‌شود ولی کم کردن صفر از موجودی تغییری ایجاد نمی‌کند؛ بنابراین از دید تست‌ها رفتارش یکسان است و mutant عملاً معادل است.

- SimplePantryInventory - متد requestedServings
جهش: مرز شرط servings > 0 ? servings : 1 تغییر کرده (conditional boundary).
چرا زنده مانده: تست ها مقدار defaultServingSize را همیشه مثبت گذاشته‌اند (۲ و ۳). بنابراین حالتی مثل 0 یا مقدار غیرمعتبر اصلاً بررسی نشده و جهش کشف نشده است.

پاسخ به سوالات تئوری

سوال ۱

گاهی اگر پوشش (coverage) را فقط روی نسخه سالم اندازه بگیریم، درباره کیفیت تست‌ها برداشت غلط می‌کنیم چون وجود یک باگ واقعی می‌تواند مسیر اجرای برنامه را تغییر دهد (مثلاً زودتر crash کند، یا وارد شاخه دیگری شود).

مثال: فرض کنید در نسخه سالم، تست‌ها مسیر محاسبه کالری و مرتب‌سازی محصولات را کامل اجرا می‌کنند و Line/Branch Coverage بالا است. اما در نسخه buggy، به خاطر یک باگ (مثلاً تقسیم بر صفر یا یک شرط اشتباه) برنامه قبل از رسیدن به بخش مرتب‌سازی throw می‌کند و تست‌ها اصلاً به آن قسمت‌ها نمی‌رسند. اگر ما فقط به coverage نسخه clean نگاه کنیم، فکر می‌کنیم تست‌ها همه مسیرها را پوشش داده‌اند، در حالی که روی نسخه buggy عملاً بسیاری از کدها اجرا نشده‌اند و تست‌ها بیشتر به خاطر crash شدن متوقف می‌شوند.

این تفاوت می‌تواند نتیجه‌گیری درباره mutation testing را هم منحرف کند: ممکن است در نسخه سالم تعداد زیادی mutant کشته شود (به نظر تست‌ها قوی‌اند)، اما یک باگ واقعی که رفتار را زودتر قطع می‌کند باعث می‌شود همان تست‌ها در عمل به بخش‌های مهم نرسند. یعنی پوشش بالا روی نسخه سالم الزاما به معنی اثرگذاری بالا برای پیدا کردن خطای واقعی نیست.

سوال ۲

معیارهای statement/branch coverage فقط می‌گویند کد اجرا شد یا نه، اما الزاماً نمی‌گویند «خروجی آن درست بررسی شد یا نه. weak mutation هم معمولا روی رسیدن به حالت داخلی متفاوت تمرکز دارد، ولی strong mutation نیاز دارد که این تفاوت واقعا به یک خروجی یا رفتار قابل مشاهده تبدیل شود و تست آن را ببیند.

strong mutation معمولا وقتی خیلی قوی‌تر از statement/branch عمل می‌کند که:

- کد مسیرهای زیادی دارد ولی تست ها فقط اجرا می‌کنند و assertion دقیق ندارند (مثلاً فقط بدون exception اجرا شود).
- منطق محاسباتی یا شرطی حساس است (مثل امتیازدهی، مرتب‌سازی، انتخاب بهترین گزینه) و با یک تغییر کوچک ممکن است خروجی عوض شود، اما اگر تست خروجی را دقیق چک نکند coverage همچنان بالا می‌ماند.
- بخش‌هایی از کد اثر شان غیرمستقیم است (مثل مرتب‌سازی لیست یا محاسبه score) و اگر تست فقط برنامه تکمیل شد را چک کند، بسیاری از خطاها پنهان می‌مانند.

به همین دلیل تکیه صرف بر coverage می‌تواند حس کاذب ایجاد کند: ممکن است line/branch بالا باشد، ولی تست ضعیف باشد و خیلی از خطاهای واقعی و بسیاری از mutant ها بدون اینکه تست شکست بخورد از زیر دست تست ها رد شوند. Mutation testing دقیقاً این نقطه ضعف را بهتر نشان می‌دهد.