Universidade Estadual de Campinas

Instituto de Computação

Verificação, Validação e Teste de Software

Atividade 3 - Geração de Teste de Software

Sergio Sebastian Pezo Jimenez	298813
João Victor Costa Vaccari	298794
Gabriel Eduardo Mangione Mamonde	298792

Campinas

2025

Introdução

Nesse documento será possível encontrar a execução de algumas técnicas de testes funcionais como o **particionamento por classes de equivalência** (Parte 1) e a **análise de valor limite** (Parte 2), assim como a **geração de conjuntos de testes dos valores limites** (Parte 3) e a **criação dos mesmos utilizando JUnit** (Parte 4). Por motivos de estética e praticidade decidimos fornecer um documento externo para a Parte 3 que será enviado juntamente com esse. O código para os testes criados para a Parte 4 se encontram no link disponível na seção correspondente: um fork do repositório disponível em https://github.com/CaioRhoden/mc646-online-shop .

Parte 1: Particionamento por Classes de Equivalência

Nessa seção está presente o particionamento por classes de equivalência. Cada classe representa um grupo de valores que são tratados de forma semelhante pelo sistema, de modo que testar apenas um valor de cada classe já é suficiente para verificar o comportamento esperado. Assim, identificamos tanto as classes **válidas** (que representam entradas aceitáveis pelo sistema) quanto às classes **inválidas** (que representam entradas que devem ser rejeitadas ou tratadas como erro). Foi criada uma tabela com 11 linhas para cada um dos Requisitos Funcionais e as seguintes colunas: **Condições de entrada**, **Classes válidas**, **Classes inválidas**. Importante notar que em parênteses ao lado de cada classe corresponde um identificador, que será útil para a "cobertura" na Parte 3.

Condições de entrada	**Classes válidas**	**Classes inválidas**
Tamanho t do campo "Título do produto"	3 <= t <=100 (1)	t < 3 (2) , t >100 (3)
Tamanho k do campo "Palavra-chave"	0 <= k <= 200 (4)	k > 200 (5)
Tamanho d do campo "Descrição"	d == 0 (6), d >= 50 (7)	d < 50 (d==0 excluído) (8)
Quantidade A de avaliações	1 <= A <= 10 (9)	A < 1 (10) , A > 10 (11)
Valor p do preço do produto	1 <= p <= 9999 (12)	p < 1 (13) , p > 9999 (14)
Quantidade E de produtos no estoque	E >= 0 (15)	E < 0 (16)
Valor de STATUS	STATUS = IN_STOCK (17), STATUS = OUT_OF_STOCK (18), STATUS = PREORDER (19), STATUS = DISCONTINUED (20)	Qualquer conjunto de string que não seja: IN_STOCK, OUT_OF_STOCK, PREORDER, DISCONTINUED (21)
Valor "peso" relativo ao peso do produto	peso >= 0 (22)	peso < 0 (23)
Tamanho D das dimensões do produto	0 <= D <= 50 (24)	D < 0 (25), D > 50 (26)

Data de adição do produto	1758743676 < tsA <= ts atual (27) *(tsA = TimeStampAdição)	tsA > ts atual (28), tsA < 1758743676 (29), tsA = vazia (30)
Data de modificação	tsA <= tsM <= ts atual (31), tsM = vazia (32) *(tsM = TimeStampModificação)	tsM > ts atual (33), tsM < tsA (34)

Parte 2: Análise de Valor Limite

Depois de ter encontrado as classes de equivalência, passamos a análise de valor limite. Essa análise é uma técnica complementar que foca na testagem dos valores localizados nas fronteiras entre classes de equivalência. Análises como essas são necessárias porque a maioria dos problemas em códigos se encontram nessas fronteiras. Foram indicados **1** valor abaixo do limite, **1** dentro do limite e **1** fora do limite, para cada valor de fronteira. Separamos os valores em **Classes válidas** (dentro do limite) e **Classes inválidas** (fora do limite).

Condições de entrada	**Classes válidas**	**Classes inválidas**
Tamanho t do campo “Título do produto”	3, 4, 99, 100	2, 101
Tamanho k do campo “Palavra-chave”	0, 1, 199, 200	201
Tamanho d do campo “Descrição”	0, 49, 50	1, 51
Quantidade A de avaliações	1, 2, 9, 10	0, 11
Valor p do preço do produto	1, 2 , 9998, 9999	0, 10000
Quantidade E de produtos no estoque	0, 1	-1
Valor de STATUS	IN_STOCK, OUT_OF_STOCK, PREORDER, DISCONTINUED	Qualquer string fora das quatro apresentadas fora das classes válidas.
Valor “peso” relativo ao peso do produto	0, 1	-1
Tamanho D das dimensões do produto	0, 1, 49, 50	-1, 51
Data de adição do produto	1758743677, 1758743678,timestamp atual	1758743676,
Data de modificação	1758743677, 1758743678, timestamp atual	1758743676, timestamp atual + (86400s)

Parte 3: Geração de Conjunto de Testes

A tabela de casos de teste anexada (arquivo test_cases_coverage.html) foi gerada automaticamente utilizando a ferramenta Equivalence Partition Organizer. Após a geração inicial, valores de parametrização específicos foram estabelecidos para cada classe de equivalência e ponto de limite, garantindo testes precisos.

O conjunto de testes foi dividido e otimizado da seguinte forma:

- Casos Válidos (TC1 a TC4 - 4 testes): Estes casos de teste foram projetados para cobrir as classes de equivalência válidas e os valores de limite válidos para todos os campos.

- Casos Inválidos (TC5 a TC21 - 17 testes): Para os cenários inválidos, a estratégia foi a de "um caso de teste por falha". Isso significa que cada um dos 17 casos tem como objetivo testar uma única condição de falha ou classe de equivalência inválida.

O conjunto de casos de teste TC1 a TC21 garante uma cobertura completa das classes de equivalência e valores-limite definidos para os requisitos funcionais. Os casos TC1 a TC4 representam os cenários válidos, assegurando que o sistema aceite entradas corretas para título, palavra-chave, descrição, avaliações, preço, estoque, status, peso, dimensões e datas. Já os casos TC5 a TC21 exploram cenários inválidos, cobrindo erros como títulos muito curtos ou longos, palavras-chave com excesso de caracteres, descrições fora do intervalo permitido, avaliações fora da faixa de 1 a 10, preços inválidos, estoque negativo, status inexistente, peso negativo, dimensões fora do intervalo permitido, além de datas de adição e modificação incorretas.

Dessa forma, todos os requisitos funcionais são contemplados: o título deve ter entre 3 e 100 caracteres, a palavra-chave até 200 caracteres, a descrição vazia ou maior que 50, avaliações entre 1 e 10, preço entre 1 e 9999, estoque não negativo, status limitado aos quatro valores pré-definidos, peso não negativo, dimensões entre 0 e 50, data de adição válida e data de modificação coerente com a de adição e o tempo atual. Conclui-se, portanto, que a análise de cobertura evidencia que todas as classes foram testadas ao menos uma vez, e a análise de requisitos funcionais demonstra que tanto os cenários válidos quanto os inválidos foram contemplados, garantindo a adequação e completude do plano de testes.

Parte 4: Criação dos testes em JUnit

Agora é o momento para a implementação dos casos de teste.
O objetivo principal é identificar quais dos testes passaram corretamente sem problema, e para aqueles que não, tentar de oferecer recomendações para a melhora. Importante ressaltar que utilizamos a técnica conhecida como AAA, ou seja Arrange, Act e Assert, para fornecer mais robustez aos nossos testes.

Um exemplo de teste seria:

```java
@Test
public void testProductValidation_TC5_TitleTooShort() {
    // Arrange: Cria um produto que tem o titulo muito pequena (kewords < 3 chars)
    Product product = createProductSample(
        1L,
        "aa", // Invalid title (less than 3 chars)
        "keywords",
        null,
        5,
        10,
        "10x10x10",
        new BigDecimal("99.99"),
        ProductStatus.IN_STOCK,
        1.5,
        Instant.now(),
        Instant.now()
    );

    // Act: Validar o produto
    Set<ConstraintViolation<Product>> violations = validator.validate(product);

    // Assert: Se espera uma violação para o campo title
    assertEquals(1, violations.size(), "Should have one violation");
    ConstraintViolation<Product> violation = violations.iterator().next();
    assertEquals("title", violation.getPropertyPath().toString(), "Violation should be on 'title'");
    assertTrue(
        violation.getMessage().contains("must be between 3 and 100"),
        "Violation message should indicate size constraint"
    );
}
```

Os resultados que obtivemos foram os seguintes:

```
[INFO] Running myapp.service.ProductServiceTest
[INFO] Tests run: 21, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.591 s -- in myapp.service.ProductServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 21, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco:0.8.12:report (post-unit-test) @ sample-app ---
[INFO] Loading execution data file /home/sergio/Documents/unicamp/testing/atividade-3-testing/target/jacoco.exec
[INFO] Analyzed bundle 'Sample App' with 81 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  29.306 s
[INFO] Finished at: 2025-09-28T23:40:38-03:00
[INFO] ------------------------------------------------------------------------
```

Como podemos observar todos os testes passaram com sucesso. Todavia, para os testes que utilizam o Timestamp isso não significa muita coisa, enquanto não existe nenhuma verificação sobre as entradas, então eventualmente todas as entradas serão aceitas.

Conclui-se que os testes criados em JUnit conseguem avaliar com acurácia os valores limites das classes de equivalência identificadas. Isso, de forma alguma, pode ser entendido como um software sem erros, enquanto, nem sempre quando todos os testes sucedem se caracteriza uma ausência de problemas.

Também tem uma explicação no README.md do GitHub.

Link fork: https://github.com/thsergitox/atividade-3-testing