UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE - UFRN
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
TESTE DE SOFTWARE II

NEYLANE PEREIRA LOPES

GERAÇÃO AUTOMÁTICA DE TESTES

Profa.: Roberta Coelho

NATAL/RN
OUTUBRO/2024

SUMÁRIO

1 INTRODUÇÃO

Nesta atividade de testes automatizados, foi escolhido o projeto Java "Sistema Bancário", disponível no GitHub no repositório TesteSoftwareII-Sistema-Bancario. Este projeto foi desenvolvido especialmente para a disciplina de Teste de Software II e, no mesmo repositório, encontram-se os testes automatizados gerados para cada execução nas diferentes ferramentas.

Para a realização dos testes, foram selecionadas duas classes: `BankService`, que possui sete métodos, e `ApiBankCommunicationService`, também com sete métodos. O objetivo dos testes foi verificar a correta funcionalidade desses métodos e garantir que eles operassem conforme o esperado.

Um bug foi propositalmente injetado no método `realizeCredit` da classe `BankService`, onde a operação de crédito, que deveria realizar uma soma ao saldo da conta, foi substituída por uma subtração, resultando em um comportamento incorreto no contexto bancário. A imagem a seguir ilustra a alteração realizada:

```java
public double realizeCredit(int accountNumber, double value, boolean isTransfer) {
    if (value < 0) {
        return bankRepository.getAccountByAccountNumber(accountNumber).getBalance();
    }

    Account selectedAccount = bankRepository.getAccountByAccountNumber(accountNumber);
    selectedAccount.setBalance(selectedAccount.getBalance() - value);
    if (selectedAccount instanceof BonusAccount && !isTransfer) {
        int actualPunctuation = ((BonusAccount) selectedAccount).getPunctuation();
        ((BonusAccount) selectedAccount).setPunctuation(actualPunctuation + (int) (value / 100));
    }
    bankRepository.saveAccount(selectedAccount);
    return selectedAccount.getBalance();
}
```

2 COMO EXECUTAR NO TERMINAL LINUX

2.1. Randoop

Para a geração dos testes utilizando o Randoop, foi utilizada a versão 8 do Java e criado um projeto Maven. O processo foi repetido três vezes para cada classe. Abaixo estão os passos seguidos:

- **Compilar o projeto Maven:**

```
mvn clean package
```

- **Gerar testes com Randoop:**

```
java -Xmx3000m -classpath ./randoop-all-4.3.3.jar:./target/classes randoop.main.Main gentests --testclass=imd.ufrn.service.BankService --output-limit=100 --junit-output-dir=./src/test/java --junit-package-name=imd.ufrn
```

- **Executar os testes gerados:**

```
mvn test
```

2.2. EvoSuite

Para a geração dos testes com o EvoSuite, também foi utilizada a versão 8 do Java e um projeto Maven. O procedimento foi repetido três vezes para cada classe. Seguem os passos:

- **Compilar o projeto Maven:**

```
mvn clean package
```

- **Gerar testes com EvoSuite:**

```
java -jar ./evosuite-1.0.6.jar -class imd.ufrn.service.BankService -projectCP ./target/classes
```

- **Mover os testes gerados para o diretório de testes:**
  ```
  find ./evosuite-tests/imd/ufrn/service -type f -name "*.java" -exec cp {} ./src/test/java/imd/ufrn/ \;
  ```
- **Executar os testes gerados:**
  ```
  mvn test
  ```

2.3. ChatGPT

Para gerar os testes utilizando o ChatGPT, foi usada a versão GPT-4. O processo consistiu em criar um novo chat e solicitar os testes da seguinte forma: "Estou realizando uma atividade de testes de software, preciso que você gere testes JUnit para a seguinte classe Java:", fornecendo a classe desejada. Este procedimento foi repetido três vezes para cada classe. Seguem os passos para execução dos testes:

- **Compilar o projeto Maven:**
  ```
  mvn clean package
  ```
- **Executar os testes gerados:**
  ```
  mvn test
  ```

3 ANÁLISE DOS RESULTADOS

Para os testes gerados por cada ferramenta, foram analisados os seguintes pontos:

A. Quantas classes de testes foram geradas para cada classe em cada execução?

B. Quantos métodos de testes foram gerados em cada execução?

C. Quantas linhas de código em classes de testes foram gerados em cada execução?

D. O bug injetado pôde ser encontrado?

E. Qual característica dos testes gerados lhe chamou mais atenção?

3.1. Randoop

Foi gerada 1 classe de teste para a classe `BankService` em todas as 3 execuções. Em cada execução, foram gerados 80 métodos de teste. A classe de teste gerada continha 1.570 linhas de código em todas as execuções. Os testes gerados pelo Randoop apresentaram a mesma estrutura em todas as execuções, sugerindo um padrão de repetição, o que pode indicar pouca variação ou adaptação ao código analisado.

O bug injetado não pôde ser encontrado. Os testes tentaram realizar um crédito em uma conta usando o método `realizeCredit()`, mas, como o repositório `BankRepository` estava definido como `null`, isso resultou em uma `NullPointerException`. No entanto, o teste simplesmente capturou a exceção, validando que ela foi lançada, sem verificar se o comportamento correto foi alcançado. Como pode ser visualizado nas imagens abaixo:

```java
@Test
public void test31() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test31");
    imd.ufrn.repository.BankRepository bankRepository0 = null;
    imd.ufrn.service.BankService bankService1 = new imd.ufrn.service.BankService(bankRepository0);
    boolean boolean5 = bankService1.createAccount(0, 100, (double) 0);
    // The following exception was thrown during execution in test generation
    try {
        double double9 = bankService1.realizeCredit((int) (byte) 1, (double) 100.0f, true);
        org.junit.Assert.fail("Expected exception of type java.lang.NullPointerException; message: null");
    } catch (java.lang.NullPointerException e) {
        // Expected exception.
    }
    org.junit.Assert.assertTrue("'" + boolean5 + "' != '" + false + "'", boolean5 == false);
}
```

```
[INFO] Running imd.ufrn.RegressionTest
[INFO] Tests run: 80, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.173 s - in imd.ufrn.RegressionTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 80, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.799 s
[INFO] Finished at: 2024-10-11T15:08:14-03:00
[INFO] ------------------------------------------------------------------------
```

Para a classe `ApiBankCommunicationServiceTest`, também foi gerada 1 classe de teste em todas as execuções, com 49 métodos de teste e 943 linhas de código. As características dos testes gerados pode ser visualizadas a seguir:

```java
@Test
public void test54() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test54");
    imd.ufrn.repository.BankRepository bankRepository0 = null;
    imd.ufrn.service.BankService bankService1 = new imd.ufrn.service.BankService(bankRepository0);
    boolean boolean5 = bankService1.createAccount(0, 100, (double) 0);
    boolean boolean9 = bankService1.realizeTransfer((int) '#', 100, 0.0d);
    boolean boolean13 = bankService1.createAccount((int) (byte) -1, 100, (double) ' ');
    boolean boolean17 = bankService1.createAccount((int) (short) 10, (int) (short) 10, (double) '4');
    // The following exception was thrown during execution in test generation
    try {
        java.util.Optional<java.lang.Double> doubleOptional20 = bankService1.realizeDebit((int) '#', (double)
        org.junit.Assert.fail("Expected exception of type java.lang.NullPointerException; message: null");
    } catch (java.lang.NullPointerException e) {
        // Expected exception.
    }
    org.junit.Assert.assertTrue("'" + boolean5 + "' != '" + false + "'", boolean5 == false);
    org.junit.Assert.assertTrue("'" + boolean9 + "' != '" + false + "'", boolean9 == false);
    org.junit.Assert.assertTrue("'" + boolean13 + "' != '" + false + "'", boolean13 == false);
    org.junit.Assert.assertTrue("'" + boolean17 + "' != '" + false + "'", boolean17 == false);
}
```

Todos os testes foram executados com sucesso, e nenhum bug foi injetado nesta classe, como pode ser visualizado na figura abaixo.

```
[INFO] Results:
[INFO]
[INFO] Tests run: 49, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.861 s
[INFO] Finished at: 2024-10-12T14:41:08-03:00
[INFO] ------------------------------------------------------------------------
```

A característica dos testes gerados que mais chamou a atenção foi a alta quantidade de classes e métodos criados automaticamente, especialmente quando não se estabelece um limite claro. No entanto, a qualidade dos testes foi insatisfatória, pois o erro simples injetado, relacionado à operação de crédito, não foi identificado. Isso ocorreu devido à forma como os testes foram implementados, eles estavam configurados para capturar exceções, mas não realizavam uma verificação efetiva do comportamento correto, como comparar o saldo da conta antes e depois da operação de crédito, o que seria mais adequado para detectar o erro.

3.2. EvoSuite

Foi gerada 1 classe de teste para a classe `BankService` em todas as três execuções. Na primeira e terceira execuções, foram gerados 35 métodos de teste e na segunda foram gerados 33 métodos. Na primeira e terceira execuções, foram geradas 398 linhas de código, já na segunda foram 388 linhas.

O bug injetado não foi detectado, já que o EvoSuite considerou que a subtração no método de crédito fazia parte da regra de negócio e, portanto, não viu isso como um erro. A seguir um exemplo de teste gerado e o resultado da execução de todos os testes:

```java
@Test(timeout = 4000)
public void test21()  throws Throwable  {
    BankRepository bankRepository0 = new BankRepository();
    BankService bankService0 = new BankService(bankRepository0);
    boolean boolean0 = bankService0.createAccount(0, 3, 3);
    assertTrue(boolean0);

    double double0 = bankService0.realizeCredit(0, 4910.0, true);
    assertEquals((-4907.0), double0, 0.01);
}
```

```
[INFO] Results:
[INFO]
[INFO] Tests run: 35, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.340 s
[INFO] Finished at: 2024-10-12T17:07:27-03:00
[INFO] ------------------------------------------------------------------------
```

Foi gerada 1 classe de teste para a classe `ApiBankCommunicationServiceTest` em todas as 3 execuções. Onde foram gerados 7 métodos em cada classe para cada execução. Já a quantidade de linhas de código nas classes de teste gerada foram 52 em todas as execuções. Pode ser visualizado abaixo um exemplo de teste gerado:

```java
@Test(timeout = 4000)
public void test3()  throws Throwable  {
    ApiBankCommunicationService apiBankCommunicationService0 = new ApiBankCommunicationService();
    double double0 = apiBankCommunicationService0.credit(1769, (-991.324));
    assertEquals(0.0, double0, 0.01);
}
```

Nessa classe não foi injetado nenhum bug e todos os testes foram executados com sucesso, como pode ser visualizado na figura abaixo.

```
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.341 s
[INFO] Finished at: 2024-10-12T17:24:50-03:00
[INFO] ------------------------------------------------------------------------
```

A característica dos testes gerados que chamou mais a atenção foi em termos de qualidade, os testes gerados pelo EvoSuite foram mais elaborados em comparação ao Randoop. Mesmo assim, ele falhou em detectar o bug, pois não considerou as regras de negócio corretas do sistema bancário.

3.3. ChatGPT

Foi gerada 1 classe de testes para a classe BankService em todas as 3 execuções. Onde foram gerados 9 métodos na primeira execução, na segunda execuçao 10 e na terceira 8. Já a quantidade de linhas de código na única classe de teste gerada foram 152 na primeira execução, na segunda 174 e na terceira 119.

O bug injetado foi identificado com sucesso, pois entre os testes gerados para o método `realizeCredit()` havia um que realizava a operação de crédito, comparando o valor creditado com o saldo final. Dessa forma, o teste conseguiu detectar que o valor retornado não era o correto, permitindo a identificação do erro. A figura a seguir ilustra o teste que foi capaz de encontrar o bug, além do resultado da execução, que confirma a falha no comportamento esperado.

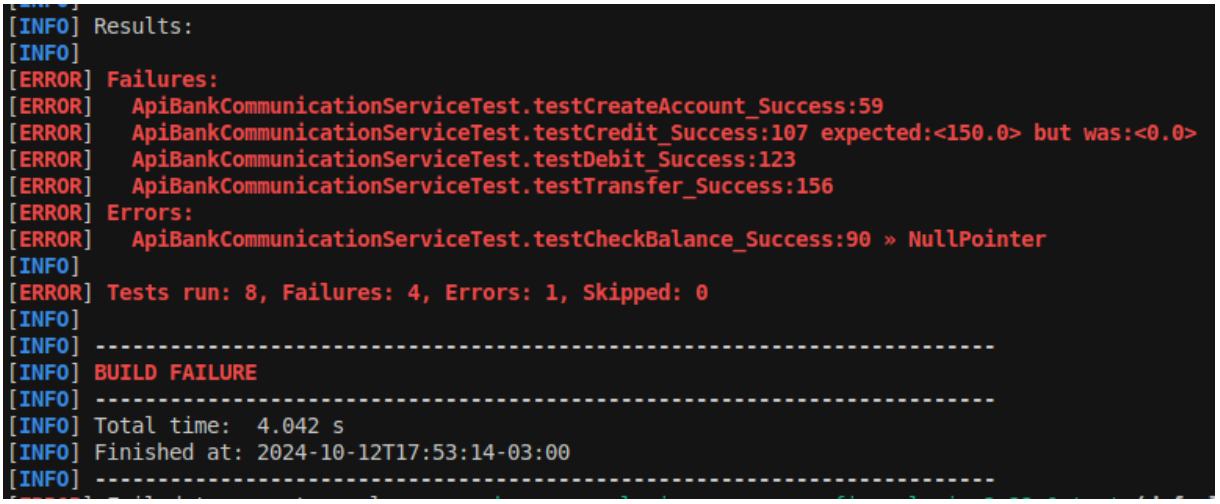
```java
// Teste para realizar crédito
@Test
public void testRealizeCredit_Success() {
    int accountNumber = 1234;
    double balance = 100.0;
    double creditValue = 50.0;

    Account account = new Account(accountNumber, balance);
    when(bankRepository.getAccountByAccountNumber(accountNumber)).thenReturn(account);

    double result = bankService.realizeCredit(accountNumber, creditValue, false);

    assertEquals(150.0, result, 0.01); // Verifique o novo saldo
    verify(bankRepository, times(1)).saveAccount(account); // Verifique se a conta foi salva
}
```

```
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   BankServiceTest.testRealizeCredit_Success:97 expected:<150.0> but was:<50.0>
[INFO]
[ERROR] Tests run: 9, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.530 s
[INFO] Finished at: 2024-10-11T16:14:22-03:00
[INFO] ------------------------------------------------------------------------
```

Foi gerada 1 classe de testes para a classe `ApiBankCommunicationServiceTest` em todas as 3 execuções, com 8 métodos gerados em cada uma delas. A quantidade de linhas de código para a classe de teste gerada foram 147 em todas as execuções.

Nesta classe, nenhum bug foi injetado. No entanto, para cenários mais complexos, os testes gerados apresentaram inconsistências. Alguns testes gerados falharam sem razão aparente, indicando que não faziam sentido no contexto da aplicação. No entanto, também foram gerados testes corretos, que executaram com sucesso. Esses comportamentos ambíguos podem ser visualizados nas figuras a seguir, destacando a falta de alinhamento dos testes com a lógica real da classe em cenários mais complicados.

```java
@Test
public void testDebit_Success() throws Exception {
    int accountNumber = 1;
    double debitValue = 50.0;
    String mockResponseJson = "50.0";

    // Simula a resposta da conexão
    when(connection.getResponseCode()).thenReturn(HttpURLConnection.HTTP_OK);
    InputStream inputStream = new ByteArrayInputStream(mockResponseJson.getBytes());
    when(connection.getInputStream()).thenReturn(inputStream);

    Optional<Double> result = apiBankCommunicationService.debit(accountNumber, debitValue);
    assertTrue(result.isPresent());
    assertEquals(50.0, result.get(), 0.01);
    verify(connection, times(wantedNumberOfInvocations:1)).getResponseCode();
}

@Test
public void testDebit_InsufficientFunds() throws Exception {
    int accountNumber = 1;
    double debitValue = 150.0;

    // Simula a resposta da conexão
    when(connection.getResponseCode()).thenReturn(HttpURLConnection.HTTP_OK);
    InputStream inputStream = new ByteArrayInputStream("null".getBytes());
    when(connection.getInputStream()).thenReturn(inputStream);

    Optional<Double> result = apiBankCommunicationService.debit(accountNumber, debitValue);
    assertFalse(result.isPresent());
    verify(connection, times(wantedNumberOfInvocations:1)).getResponseCode();
}
```



A característica mais impressionante dos testes gerados foi a capacidade de identificar o bug, mesmo sem informações explícitas sobre o contexto e as regras de negócio. Esse resultado mostrou que os testes conseguiram simular um cenário que reflete um ambiente real de um sistema bancário, realizando verificações adequadas. Isso demonstra uma abordagem inteligente, pois os testes conseguiram interpretar o funcionamento esperado do sistema, mesmo sem um detalhamento específico do comportamento de negócios, o que ressalta a adaptabilidade e a robustez dos testes gerados.

4 COMPARANDO TESTES DO CHATGPT COM RANDOOP E EVOSUITE

Ao comparar os testes gerados pelo ChatGPT com os do Randoop, ficou evidente que o ChatGPT produz testes com um nível maior de detalhamento em cada método. Enquanto o Randoop cria uma grande quantidade de testes, muitos deles não estão adequados ao contexto da aplicação, como a geração de exceções sem realmente testar a funcionalidade de forma significativa. Além disso, os testes gerados pelo Randoop são muito repetitivos, com métodos que capturam exceções sem validar corretamente os resultados esperados.

Por outro lado, o EvoSuite gerou testes mais semelhantes aos do ChatGPT em termos de qualidade. No entanto, o EvoSuite não conseguiu identificar um erro específico no código, pois interpretou a subtração no método de crédito como parte da regra de negócio correta, o que resultou em falhas na detecção do bug injetado.

Uma observação importante é que, em cenários onde a classe possui dependências externas que requerem o uso de mocks, o ChatGPT inicialmente teve dificuldade em identificar a forma correta de estruturar os testes. Nesses casos, o EvoSuite foi mais eficiente, conseguindo lidar melhor com as dependências injetadas e criando os mocks adequados para realizar os testes.

Tanto o EvoSuite quanto o ChatGPT apresentam pontos fortes, e não é possível afirmar de forma definitiva que um é superior ao outro em todos os aspectos. No entanto, o ChatGPT se destaca no cenário de testes pelo seu maior nível de detalhamento e compreensão do contexto do sistema, especialmente quando é fornecido um contexto adicional, como as regras de negócio. Mesmo sem essas regras, o ChatGPT conseguiu entender a dinâmica do sistema bancário, criando testes adequados e identificando erros que passaram despercebidos pelas outras ferramentas. Isso faz do ChatGPT uma ferramenta promissora para a geração automatizada de testes em cenários complexos.