Relatório Trabalho Prático de Teste Baseado em Especificação

Método para adição de números inteiros muito, muito grandes

Breno Farias da Silva

Invalid Date

Sumário

Introdução

Este documento descreve o processo usado pelo autor do livro texto da disciplina, Mauricio Aniche, a ser usado o teste baseado em especificação para derivar os casos de teste para o método `adicaoNumerosGrandes()`.

Passos para execução do testes efetivo e sistemático

Passo 1 - Explorar o funcionamento do programa

Meu processo foi o seguinte:

- Vamos ver o programa trabalhando com a adição de dois números de um dígito. Vou passar as listas `[1]` e `[1]` como operandos;
  - <(`left`=[1], `right`=[1]), ([2])>
- Em seguida, vamos ver a adição de dois números de dois dígitos. Vou passar as listas `[1,5]` e `[1,0]` como operandos;
  - <(`left`=[1,5], `right`=[1,0]), ([2,5])>
- Por fim, vamos testar a adição de dois números de três dígitos. Vou passar as listas `[5,0,0]` e `[2,5,0]` como operandos;
  - <(`left`=[5,0,0], `right`=[2,5,0]), ([7,5,0])>

Passo 2 - Identificar as partições

Para cada entrada individualmente

Parâmetro `left`:

`left` é uma lista de inteiros representando os dígitos de um número. Como estamos lidando com listas, é importante considerar casos excepcionais como nulo e vazio, além de listas de diferentes comprimentos.

- `left` com valor `null`;
- `left` vazia;
- `left` com um único elemento;
- `left` com múltiplos elementos.

Parâmetro `right`:

`right` segue a mesma lógica de `left`, pois também é uma lista de dígitos.

- `right` com valor `null`;
- `right` vazia;
- `right` com um único elemento;
- `right` com múltiplos elementos.

Para combinações de entradas

As duas listas, `left` e `right`, possuem uma relação direta de dependência na operação de soma. Algumas combinações importantes de serem testadas são:

- Ambas `left` e `right` são nulas;
- Uma lista é nula e a outra não;
- Ambas as listas estão vazias;
- Uma lista está vazia e a outra contém elementos;
- Ambas as listas têm apenas um dígito;
- Uma lista tem mais dígitos que a outra;
- Ambas as listas têm múltiplos dígitos e o resultado gera “vai um” (carry);
- Soma sem necessidade de “vai um” (sem carry);
- Soma com múltiplos “vai um” encadeados.

Essas situações me parecem cobrir tanto casos normais quanto casos de excepcionais que podem impactar o comportamento da função.

Para saídas esperadas
Por fim, considerei os diferentes tipos de resultados possíveis. O método retorna uma lista de inteiros representando o número resultante da soma:

- Lista de inteiros
  - Lista com valor `null` (em caso de tratamento especial de erro, se houver);
  - Lista vazia (não esperado em somas válidas, mas importante considerar);
  - Lista com um único elemento (soma pequena);
  - Lista com múltiplos elementos (soma grande).
- Cada inteiro individualmente na lista de saída
  - Um dígito válido (0 a 9) para cada posição;
  - Ocasional aparecimento de um novo dígito à esquerda (por exemplo, 99 + 1 ⟶ 100, aumentando o tamanho da lista).

Passo 3 - Identificar os valores limite

No caso de `add()`, os limites mais relevantes aparecem quando ocorre uma transição no comportamento da soma, como por exemplo:

- Quando o resultado da adição passa de um único dígito para dois dígitos (isto é, quando há um "vai um" - carry);
- Quando uma das listas é maior que a outra e precisamos garantir que a função lida corretamente com tamanhos diferentes;
- Quando o resultado da soma gera um novo dígito mais à esquerda (por exemplo, somar 999 + 1 resultando em 1000).

Esses limites são importantes porque forçam o método a lidar com mudanças sutis no processamento da lista de dígitos.

Sempre que identificamos um limite, criamos dois testes: um para cada lado do limite (*in point* e *out point*). Aplicando isso, os limites e seus testes são:

- **Carry não ocorre / Carry ocorre**:
  - Soma de dois números sem gerar "vai um" (ex: 2 + 3 ⟶ 5);
  - Soma de dois números que gera "vai um" (ex: 5 + 7 ⟶ 12).
- **Tamanhos iguais / Tamanhos diferentes**:
  - `left` e `right` têm o mesmo número de dígitos;
  - `left` tem mais dígitos que `right` ou vice-versa.
- **Sem aumento de tamanho / Com aumento de tamanho**:
  - A soma não muda a quantidade de dígitos do número (ex: 25 + 13 ⟶ 38);
  - A soma aumenta a quantidade de dígitos do número (ex: 99 + 1 ⟶ 100).

Passo 4 - Derivar os casos de teste

Definir como diminuir o número de casos de teste

> ℹ Essa nota deve ser excluída do documento final, serve apenas para orientação do trabalho.
>
> Idealmente, combinaríamos todas as partições que criamos para cada uma das entradas: $4 \times 4 = 16$ testes.

Para evitar uma explosão de casos de teste, vou eliminar algumas combinações de partições que têm pouca probabilidade de revelar erros. A ideia é testar casos excepcionais apenas uma vez e não combiná-los com outras partições. Assim, listas `null` ou vazias serão testadas isoladamente e não combinadas com outros cenários. Além disso, situações específicas como listas de tamanhos diferentes e a ocorrência ou não de "carry" serão cobertas de forma direcionada, sem testar todas as combinações possíveis.

Portanto, seguindo essa ideia, na lista a seguir, marquei com um [x] as partições que serão testadas apenas uma vez:

- `left`:
  - `null` [x]
  - lista vazia [x]
  - lista com um elemento
  - lista com múltiplos elementos
- `right`:
  - `null` [x]
  - lista vazia [x]
  - lista com um elemento
  - lista com múltiplos elementos
- Comportamentos adicionais:
  - Soma sem "carry"
  - Soma com "carry"
  - Tamanhos iguais
  - Tamanhos diferentes
  - Soma que aumenta o número de dígitos do resultado

Derivar os casos de teste

Podemos derivar os casos de teste combinando adequadamente as partições.

- Casos de teste de exceção:
  - T01: `left == null`
  - T02: `right == null`
- Casos de teste de listas vazias:
  - T03: `left` é vazia
  - T04: `right` é vazia

- Casos de teste com listas de tamanho 1:
  - T05: Somar [2] e [3] (sem carry)
  - T06: Somar [5] e [7] (com carry)
- Casos de teste com listas de múltiplos elementos e tamanhos iguais:
  - T07: Somar [1, 2, 3] e [4, 5, 6] (sem carry)
  - T08: Somar [9, 9, 9] e [1, 0, 0] (com carry e aumento de dígito)
- Casos de teste com listas de múltiplos elementos e tamanhos diferentes:
  - T09: Somar [1, 2] e [3, 4, 5] (right maior que left)
  - T10: Somar [9, 9, 9] e [1] (left maior que right, com carry)
- Casos de teste de valor limite:
  - T11: Soma onde o carry acontece exatamente no último dígito (por exemplo, [9] + [1] ⟶ [0,1])

Esses testes cobrem tanto casos normais quanto exceções e valores limites, garantindo uma boa cobertura com um número reduzido de testes.

Passo 5 - Automatizar os casos de teste usando JUnit

Durante a implementação dos casos de teste utilizando o JUnit, a principal dificuldade encontrada foi garantir a correta comparação entre os resultados esperados e os obtidos, especialmente na manipulação de listas de inteiros representando números invertidos (pouco usual no dia a dia). Também foi necessário corrigir pequenos erros de uso do `assertEquals`, respeitando a ordem correta dos parâmetros (primeiro o valor esperado, depois o valor real). Além disso, foi importante padronizar todos os testes para o uso do JUnit 5, garantindo a utilização correta das anotações e métodos de asserção (`Assertions.assertEquals`, `Assertions.assertThrows`).

Outro ponto relevante foi pensar cuidadosamente nos casos de teste envolvendo **tamanhos diferentes de listas** e **propagação do carry** ao longo da adição, o que exigiu atenção especial para cobrir todas as possibilidades de comportamento do método `add`.

Passo 6 - Aumentar a suíte de testes por meio de experiência e criatividade

Ao revisar os testes desenvolvidos inicialmente, percebi que havia oportunidades de expandir a suíte de testes para cobrir cenários adicionais. Com base nisso, projetei novos testes para garantir uma cobertura ainda mais ampla:

- Testes para somar números de tamanhos diferentes onde o carry influencia no aumento do número de dígitos;
- Testes específicos para listas vazias (`left` ou `right` sendo vazios);
- Testes específicos para entradas `null`, garantindo que o método `add` lança exceções apropriadas (`IllegalArgumentException`).

Esses testes adicionais foram importantes para garantir que o método `add` fosse robusto frente a diferentes entradas e situações.

Execução dos casos de teste

A execução de todos os testes automatizados foi realizada utilizando o JUnit 5. Todos os testes passaram com sucesso, sem apresentar defeitos ou falhas.

Não foram encontrados defeitos durante a execução dos testes. Dessa forma, não houve necessidade de realizar testes de regressão, pois o comportamento da função `add` esteve consistente em todas as situações avaliadas.

Passo 5 - Automatizar os casos de teste usando JUnit

Durante a implementação dos casos de teste utilizando o JUnit, a principal dificuldade encontrada foi garantir a correta comparação entre os resultados esperados e os obtidos, especialmente na manipulação de listas de inteiros representando números invertidos (pouco usual no dia a dia). Também foi necessário corrigir pequenos erros de uso do `assertEquals`, respeitando a ordem correta dos parâmetros (primeiro o valor esperado, depois o valor real). Além disso, foi importante padronizar todos os testes para o uso do JUnit 5, garantindo a utilização correta das anotações e métodos de asserção (`Assertions.assertEquals`, `Assertions.assertThrows`).

Outro ponto relevante foi pensar cuidadosamente nos casos de teste envolvendo **tamanhos diferentes de listas** e **propagação do carry** ao longo da adição, o que exigiu atenção especial para cobrir todas as possibilidades de comportamento do método `add`.

Passo 6 - Aumentar a suíte de testes por meio de experiência e criatividade

Ao revisar os testes desenvolvidos inicialmente, percebi que havia oportunidades de expandir a suíte de testes para cobrir cenários adicionais. Com base nisso, projetei novos testes para garantir uma cobertura ainda mais ampla:

- Testes para somar números de tamanhos diferentes onde o carry influencia no aumento do número de dígitos;
- Testes específicos para listas vazias (`left` ou `right` sendo vazios);
- Testes específicos para entradas `null`, garantindo que o método `add` lança exceções apropriadas (`IllegalArgumentException`).

Esses testes adicionais foram importantes para garantir que o método `add` fosse robusto frente a diferentes entradas e situações.

Instruções para executar os testes

Para executar os testes do projeto Java, utilize os seguintes comandos Maven:

- `mvn install`: Instala todos os pacotes necessários para rodar o projeto.
- `mvn test`: Executa os casos de teste em JUnit.
- `mvn jacoco:report`: Gera o relatório Maven com o resultado dos testes.
- `mvn surefire-report:report`: Gera um relatório mais completo sobre a execução dos testes.
- `mvn test-compile org.pitest:pitest-maven:mutationCoverage`: Gera dados de teste de mutação, indicando quanto do código foi testado.

Esses comandos são úteis para garantir que o projeto esteja corretamente configurado e que todos os testes sejam executados e avaliados de maneira eficiente.