Testes em Sistemas Legados

Escrevendo testes em sistemas legados

Conteúdo

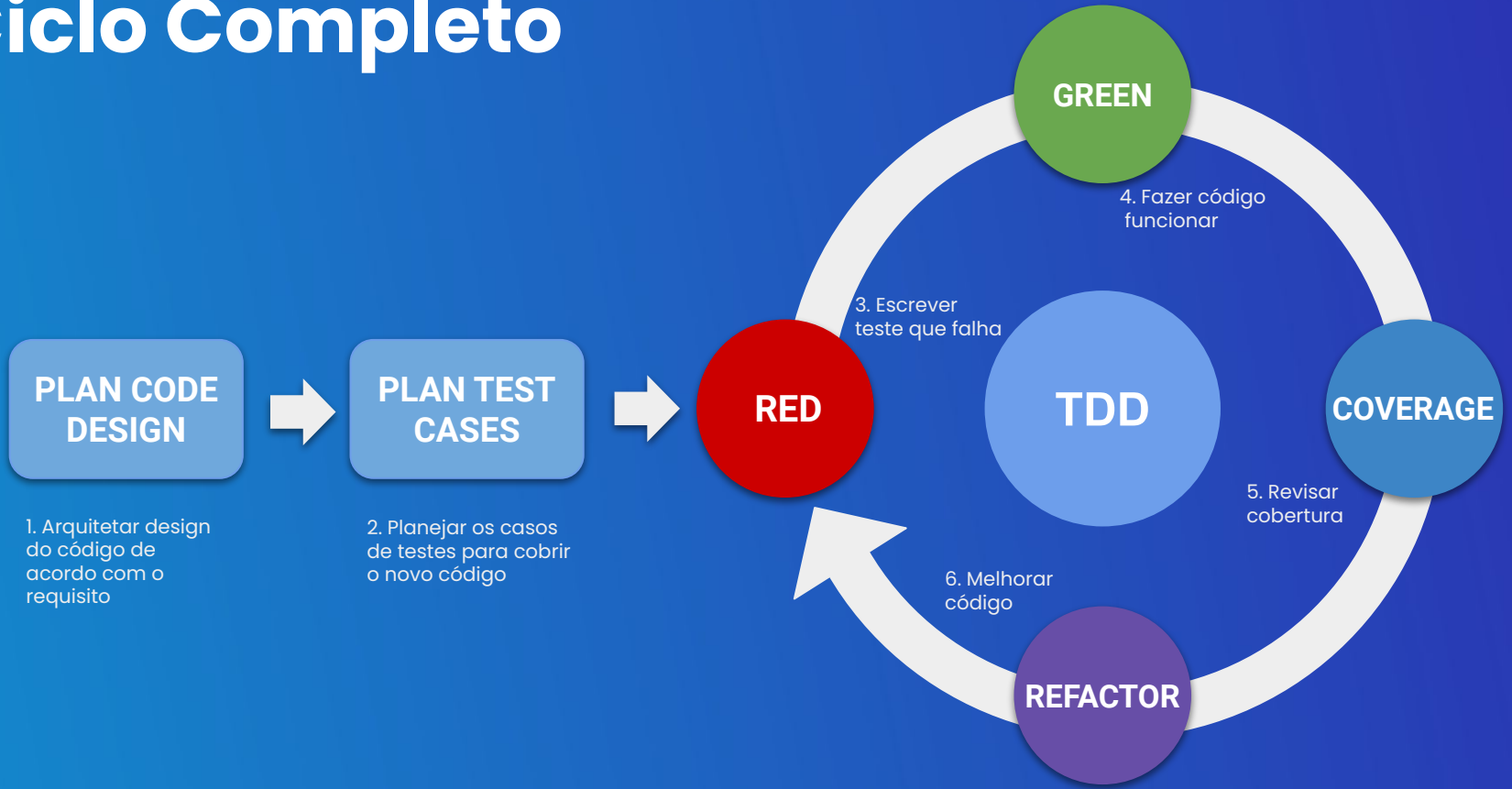
- Motivação
- Planejamento
- Boas Práticas
- Métodos Privados
- Cobertura
- Legado
- Exemplo
- Exercícios

"Os testes mostram a presença, não a ausência, de bugs."
-- Edsger Dijkstra

Motivação

- Testes nos ajudam a entregar um produto com menos falhas (maior qualidade)
- Testes nos ajudam a desenvolver um produto de forma mais rápida (reduz o ciclo: alterar -> iniciar aplicação -> testar)
- Testes nos dão segurança para melhorar a aplicação (refatorar o código, atualizar bibliotecas, adicionar novas funcionalidades)

Ciclo Completo



Boas Práticas

- **Cada método deve testar apenas um cenário**
- Crie métodos de teste pequenos
- Um teste não pode depender do resultado de outro teste
- O nome do método de teste deve expressar o que o teste deve fazer
- Nome do método deve começar com “deve...” ou “naoDeve”... Ex:
  - deveCalcularValorDescontoParaItensPedido
  - naoDevePermitirDescontoParaItensEmPromocao

Métodos Privados

- Testar métodos privados é um code smell (má prática). Devemos sempre testar os métodos privados de forma indireta, ou seja, através de métodos públicos.
- Foi realizado um estudo para tentar viabilizar os testes unitários no sistema legado sem testar diretamente métodos privados.

Métodos Privados

- Devido a complexidade atual do código legado, a conclusão foi que as melhorias na arquitetura iriam deixar o código legado ainda mais complexo ou exigiria uma grande refatoração.
- Neste caso, entre optar por aumentar a complexidade do código legado ou testar diretamente os métodos privados, achamos mais viável a última opção.

Métodos Privados

- Deve-se esclarecer que o teste de métodos privados só deve ser realizado quando não houver outras alternativas.
- Por exemplo, ao desenvolver um novo módulo, deve-se se preocupar com o design do código, para que seja testado apenas o métodos públicos e os privados sejam testados de forma indireta, como recomendam as boas práticas.

Métodos Privados

- Para testar métodos privados, pode-se utilizar uma ferramenta de mock como o PowerMock. Porém o PowerMock permite fazer muitas outras coisas. E queremos manter as coisas mais simples o possível.
- Por isso, utilizaremos um método utilitário para isto:
  - MethodTestUtils.invokePrivateMethod(objeto, “nomeMetodo”, params);

Legado

- Testar novas funcionalidades, isolando o código novo do código legado **(testar somente o código novo)**
- Não testar dependências (parâmetros do sistema, chamadas ao banco de dados ou chamadas a outros serviços)

Legado

```
public double calculaDesconto(Produto produto) {
    double desconto = 0;
    if (ConfigSistema.isUsaDescontoQuantidade()) {
        desconto += calculaDescontoPorQuantidade(produto);
    }
    if (ConfigSistema.isUsaDescontoEstoque() {
        desconto += calculaDescontoPorEstoque(produto);
    }
    ...
}
```


```
private double calculaDescontoPorQuantidade(Produto produto) {
  ...
}

private double calculaDescontoPorEstoque(Produto produto) {
  ...
}
```

Cobertura

- A cobertura de testes indica a porcentagem do código fonte que foi testada
- Deve-se cobrir o máximo possível do código. Mas não necessariamente um código com 100% de cobertura está 100% livre de erros.

Cobertura

- Lembre-se que os testes mostram a **presença** e não a **ausência** de erros no sistema.
- A qualidade dos cenários de testes é tão importante quanto a cobertura.
- Por isso a importância de pensar nos cenários de teste.
- De verificar se os cenários de teste contemplam todas as regras de negócio e todos os fluxos de exceção.

Cobertura

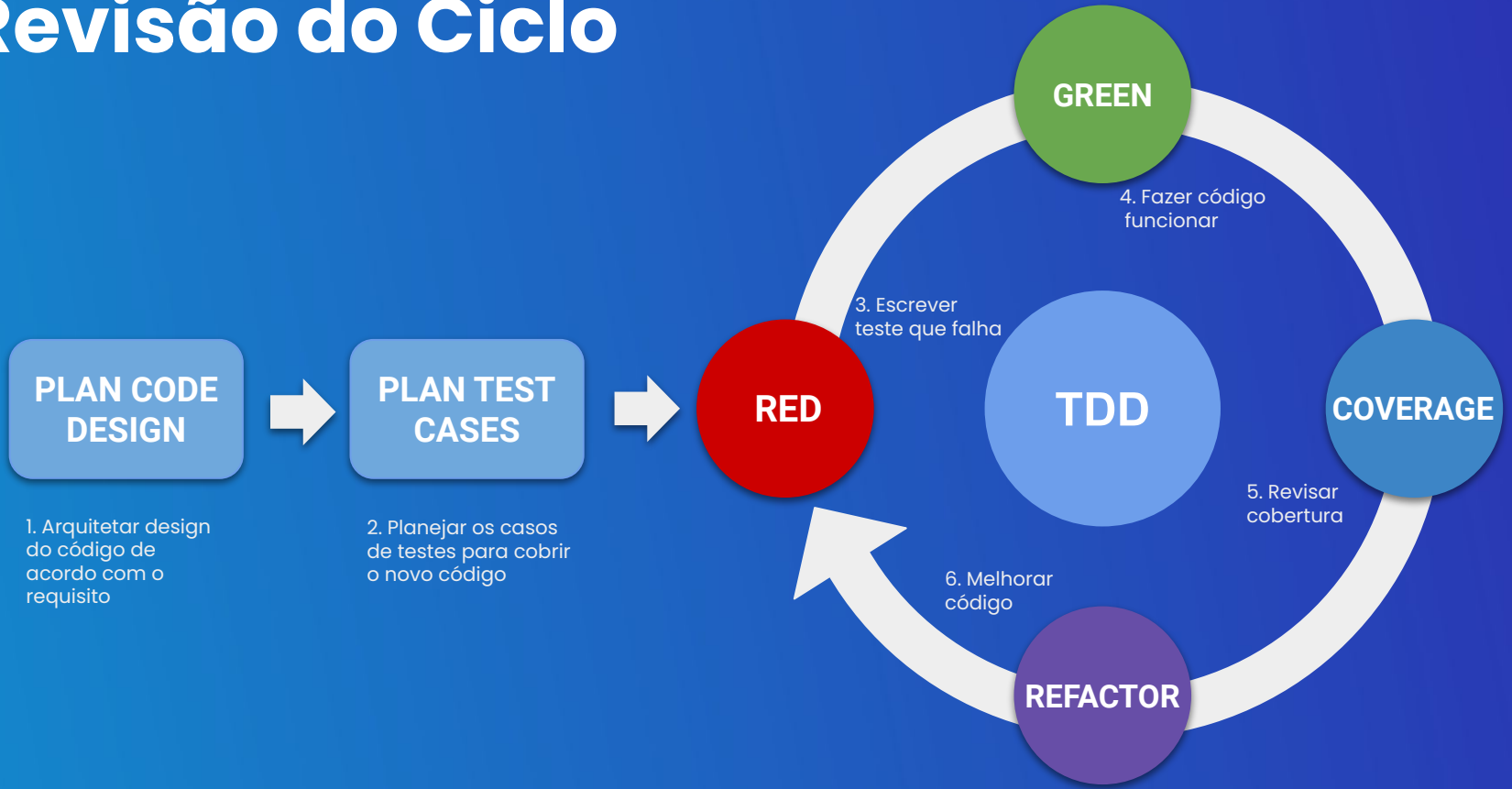
- Não é necessário escrever testes para todos os métodos de todas as classes do sistema
- A cobertura vai sendo aumentada a medida que os testes vão sendo realizados.
- A cobertura é realizada de forma indireta. Ou seja, ao testar um método público, todos métodos privados e protegidos que são chamados por ele, são testados indiretamente.

Exemplo

Incluir uma validação para kilometragem do veículo conforme as regras abaixo:

- Quilometragem inicial deve ser maior que zero
- Quilometragem final deve ser maior que zero
- Quilometragem final deve ser maior que a inicial

Revisão do Ciclo



Planejar Implementação

1) Fazer o planejamento da implementação:
    - Verificar onde será realizada a implementação
    - Verificar se será criado um novo módulo ou um novo componente
    - Evitar aumentar o código legado. Sempre crie novos métodos quando tiver que adicionar uma funcionalidade a algo já existente
    - Pensar na testabilidade da nova implementação

Planejar os testes

2) Planejar os casos de testes conforme as regras de negócio:

#	kmInicial	kmFinal	Pass/Fail	Resultado Esperado
1	0	1	FAIL	“Quilometragem inicial deve ser maior que 0”
2	1	0	FAIL	“Quilometragem final deve ser maior que 0”
3	-1	1	FAIL	“Quilometragem inicial deve ser maior que 0”
4	1	-1	FAIL	“Quilometragem final deve ser maior que 0”
5	2	1	FAIL	“Quilometragem final deve ser maior que a inicial”
6	2	2	FAIL	“Quilometragem final deve ser maior que a inicial”
7	1	2	PASS	Sucesso

Implementar os Testes

3) Implementar os testes inicialmente falhando:
    - Escrever um esboço do método a ser implementado apenas para compilar a classe de teste
    - Escrever os métodos de teste conforme os cenários planejados
    - Como a implementação ainda é só um esboço os testes irão falhar

Implementar a Funcionalidade

4) Implementar a funcionalidade e fazer o testes passar:
    - Realizar a implementação da funcionalidade propriamente dita
    - Rodar novamente os testes e ajustar o código até todos os testes estiverem passando

Verificar a Cobertura

5) Verificar a cobertura de código:

   - Rodar o plugin de cobertura de código e verificar se os cenários de testes cobrem o código da funcionalidade
   - Se tiver algum fluxo não coberto, escrever os testes para este fluxo

Melhorar o Código

6) Avaliar a necessidade de melhorias e limpeza do código:
   - Verificar se o código está limpo
   - Eliminar warnings e problemas exibidos pela IDE
   - Verificar se não existe problemas de performance
   - Verificar se não há violações de segurança
   - Verificar se a implementação pode ser encapsulada em um novo módulo ou um novo componente

Exercícios

Exercícios - Desconto por Estoque

Implementar no sistema uma regra para conceder um desconto adicional conforme o estoque do produto:

- 20% caso o produto tenha 1000 unidades ou mais em estoque
- 10% caso o produto tenha 500 unidades ou mais em estoque
- 5% caso o produto tenha 100 unidades ou mais em estoque
- 0% caso o produto tenha menos de 100 unidades em estoque

Exercícios – Testes a Serem Realizados

#	Estoque	Pass/Fail	Resultado Esperado
1	1001	PASS	Desconto de 20%
2	1000	PASS	Desconto de 20%
3	501	PASS	Desconto de 10%
4	500	PASS	Desconto de 10%
5	101	PASS	Desconto de 5%
6	100	PASS	Desconto de 5%
7	99	PASS	Nenhum desconto

Conclusão

- Evitar criar testes para o código legado devido a sua complexidade e suas dependências
- Testar apenas as novas implementações, que devem ser planejadas para a realização de testes
- Planejar os cenários de teste antes da implementação
- Cada cenário de teste deve ser implementado em um método de teste separado
- Objetivo geral é melhorar a qualidade do código legado de forma gradativa a partir das novas implementações

Referências