Capítulo 8

Avaliação de arquiteturas

Com a alta demanda do mercado tecnológico, a avaliação da arquitetura é essencial para conseguir atender ao tempo do mercado. Por mais bem projetado e desenvolvido que um sistema seja, é possível que ocorram erros que não foram previstos durante o seu projeto ou mesmo em adição a novas funcionalidades (GONÇALVES, 2007; TAHCHIEV *et al.*, 2010).

Em geral, esse tipo de erro é difícil de ser identificado e, por consequência, de ser corrigido. São cenários que podemos identificar diariamente, conforme o sistema cresce em complexidade e em tamanho (GONÇALVES, 2007).

Sendo assim, criar rotinas de teste para o código e sua arquitetura é a melhor maneira de manter a qualidade. Além disso, é possível tentar capturar e isolar de maneira mais eficiente a origem de uma falha.

Esse é um processo de documentação, gestão e implantação de sistemas cada vez mais utilizado atualmente (GONÇALVES, 2007; TAHCHIEV *et al.*, 2010). Neste capítulo, vamos entender um pouco mais sobre essas rotinas e como podemos trabalhar com os testes a nosso favor.

1 Métodos de avaliação de arquiteturas

Existem diversas maneiras de garantir a qualidade do sistema por meio da avaliação com testes funcionais e técnicos. Dependendo da metodologia utilizada, pode-se aplicar vários níveis de testes, sendo que em alguns casos é possível, inclusive, automatizar os testes para entrarem no ciclo de integração e implantação contínua (GONÇALVES, 2007). São testes que podemos utilizar para garantir uma boa qualidade e avaliação de nossa arquitetura:

- **Teste de caixa branca:** conhecido também como teste lógico ou estrutural, tem como objetivo testar componentes internos de um módulo ou do sistema.
- **Teste de caixa preta:** verifica se um componente ou um sistema está correto avaliando o sistema de forma externa. Nesse caso, não verificamos o que o sistema está fazendo, apenas damos a entrada e comparamos com a saída.
- **Teste de regressão:** são testes realizados a cada nova versão de software e servem para verificar se outros pontos de falha no sistema persistem nas versões atuais.
- **Teste de unidade:** é orientado para verificar unidades menores de um software e, geralmente, aplicado a métodos de uma classe, laços de repetição, recursões, entre outros.
- **Teste de integração:** avalia se a comunicação entre módulos ou serviços distintos de um sistema funciona. Muitas vezes, os módulos funcionam bem separadamente, mas, quando conectados

para trabalharem em conjunto, problemas de compatibilidade entre as interfaces acabam surgindo.

- **Teste de carga:** avalia a capacidade do software sem que ele apresente erros.
- **Teste de usabilidade ou funcional:** avalia o comportamento de um sistema de acordo com as expectativas dos usuários.
- **Teste de estresse:** mede o comportamento do sistema quando se ultrapassa o seu limite, de modo a verificar como o software pode falhar nessas condições.

São vários os tipos de testes existentes, mas como decidir quais são os necessários? Será que preciso de todos eles em um projeto? Qual o custo, financeiro e em termos de tempo, para implementar todos? Como sempre, isso irá depender do tipo de software que você está projetando. Aplicações mais críticas, com um alto volume de informação e acesso, em geral, possuem rotinas de teste de estresse e/ou de carga. Já as aplicações mais simples podem se limitar a testes unitários e funcionais (TAHCHIEV *et al.*, 2010).

O tempo gasto para a construção dos testes, em geral, depende do grau de maturidade da equipe de desenvolvimento quanto à cultura de testes. Os testes serão manuais ou automatizados? Quais ferramentas serão utilizadas no processo de construção dos testes? Qual a afinidade da equipe com a ferramenta? São diversos os fatores que devemos considerar (GONÇALVES, 2007; TAHCHIEV *et al.*, 2010).

É importante lembrar que existem metodologias de desenvolvimento de software que tornam obrigatória a elaboração de testes logo no início da construção do sistema. A principal metodologia que temos no mercado é o TDD (test-driven development). Nessa metodologia, em primeiro lugar cria-se o teste para somente depois se desenvolver o código propriamente dito (TAHCHIEV *et al.*, 2010).

1.1 Construindo um teste unitário

Dentre os testes que avaliam a qualidade de um sistema, sem dúvidas o essencial, por se aplicar a qualquer cenário, é o teste unitário. Sendo assim, aqui utilizaremos um cenário de locação de veículo para ilustrar a implementação.

Para iniciar a construção, serão criadas as classes de modelos *Cliente*, *Carro* e *Locacao*. A primeira é a classe *Cliente*.

```
public class Cliente {

    private String nome;
    private String email;

    public Cliente() {}

    public Cliente(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

É uma classe simples, apenas com os atributos necessários e os métodos *getters* e *setters* para acesso aos atributos privados. Na sequência, a classe *Carro* seguirá o mesmo princípio.

```
public class Carro {

    private String modelo;
    private Integer disponiveis;
    private Double valor;

    public Carro() {}

    public Carro(String modelo, Integer disponiveis,
Double valor) {
        this.modelo = modelo;
        this.disponiveis = disponiveis;
        this.valor = valor;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public Integer getDisponiveis() {
        return disponiveis;
    }

    public void setDisponiveis(Integer disponiveis) {
        this.disponiveis = disponiveis;
    }

    public Double getValor() {
        return valor;
    }

    public void setValor(Double valor) {
        this.valor = valor;
    }
}
```

Para finalizar os modelos, temos a classe *Locacao*.

```
public class Locacao {
    private Cliente cliente;
    private Carro carro;
    private LocalDate dataRetirada;
    private LocalDate dataRetorno;
    private Double valor;

    public Cliente getCliente() {
        return cliente;
    }

    public void setCliente(Cliente cliente) {
        this.cliente = cliente;
    }

    public Carro getCarro() {
        return carro;
    }

    public void setCarro(Carro carro) {
        this.carro = carro;
    }

    public LocalDate getDataRetirada() {
        return dataRetirada;
    }
    public void setDataRetirada(LocalDate dataRetirada) {
        this.dataRetirada = dataRetirada;
    }

    public LocalDate getDataRetorno() {
        return dataRetorno;
    }

    public void setDataRetorno(LocalDate dataRetorno) {
        this.dataRetorno = dataRetorno;
    }
    public Double getValor() {
        return valor;
    }

    public void setValor(Double valor) {
        this.valor = valor;
    }
}
```

A próxima classe que é necessário criar é a classe que faz uso dos modelos e os integra com as regras de negócio. Para tornar o exemplo de fácil compreensão, utilizaremos uma classe chamada de *LocacaoService*, que indica o cenário para alugar um veículo.

```
public class LocacaoService {

    public Locacao alugaCarro(Cliente cliente, Carro carro)
throws LocacaoException, CarroNaoDisponivelException {
        if(cliente == null) throw new
LocacaoException("Cliente não informado");

        if(carro == null) throw new LocacaoException("Carro
não informado");

        if(carro.getDisponiveis() == 0) throw new
CarroNaoDisponivelException();

        Locacao locacao = new Locacao();
        locacao.setCliente(cliente);
        locacao.setCarro(carro);
        locacao.setDataRetirada(LocalDate.now());
        locacao.setDataRetorno(adicionaDias(LocalDate.now(),
3));
        locacao.setValor(carro.getValor());

         //Salvar a locacao, no banco de dados
         return locacao;
    }

}
```

Com o método *alugaCarro* definido, precisamos pensar em todos os cenários, as ações e as validações desejadas dentro desse método. Por exemplo, cada *if* que lança uma exceção deve ser testado como uma unidade de código – perceba, daí, a denominação "teste unitário". Vamos ver como fica o nosso arquivo de teste, lembrando que cada unidade possui um teste separado.

```
public class LocacaoTest {

    private LocacaoService service;

    @Rule
    public ErrorCollector erros = new ErrorCollector();

    @Rule
    public ExpectedException exception = ExpectedException.
none();

    @Before // Como se fosse o construtor do teste.
    public void inicializa() {
        System.out.println("Executa antes de realizar os
testes.");
        service = new LocacaoService();
    }

    @Test
    public void testeAlugaCarro() throws Exception {
        Cliente cliente = new Cliente("Gustavo");
        Carro carro = new Carro("Renault Clio 2012", 5, 10.0);

        Locacao locacao = service.alugaCarro(cliente, carro);

        erros.checkThat(locacao.getValor(),
is(equalTo(10.0)));
        erros.checkThat(ehMesmaData(LocalDate.now(), locacao.
getDataRetirada()), is(true));
        erros.checkThat(ehMesmaData(locacao.getDataRetorno(),
obterDataComDiferencaDias(3)), is(true));
   }

    @Test(expected = CarroNaoDisponivelException.class)
public void testeCarroNaoDisponivel() throws Exception{
        Cliente cliente = new Cliente("Martin");
        Carro carro = new Carro("Fiat Uno", 0, 10.0);

        service.alugaCarro(cliente, carro);
    }

    @Test
    public void testeVeiculoVazio() {
        Cliente cliente = new Cliente("Andrey");
```

```
        try {
            service.alugaCarro(cliente, null);
            fail(); // Da classe Assert
        } catch (CarroNaoDisponivelException |
LocacaoException e) {
            assertEquals("Carro não informado",
e.getMessage());
        }
    }

    @Test
    public void testeClienteVazio() throws
CarroNaoDisponivelException, LocacaoException {
        Carro carro = new Carro("Renault Clio 2012", 5, 10.0);

        exception.expect(LocacaoException.class);
        exception.expectMessage("Cliente não informado");

        service.alugaCarro(null, carro);
    }

    @After
    public void finaliza() {
        System.out.println("Executado após o teste.");
    }
}
```

São várias as anotações que temos dentro do arquivo Test. A primeira coisa é a criação de um atributo *LocacaoService*, que será compartilhado entre todos os testes que serão executados. Temos duas Rules, cada uma para coletar informações do resultado do teste – uma para coletar os erros ocorridos no projeto e depois exibi-los de uma única vez, e outra para capturar exceções lançadas. As anotações *Before* e *After* são executadas sempre, antes e depois de cada teste, respectivamente.

Para cada unidade, é necessário criar um método *void* com a anotação *Test*. Eles podem ser executados individualmente ou em conjunto. Nesse caso, é apresentado um teste para o aluguel do veículo, outro para veículo indisponível, outro para objeto veículo não criado e um último

para cliente não identificado. Perceba que são testes de sucesso e falha entre as unidades testadas. Isso é importante para garantir a qualidade da construção do sistema ao longo de todo o desenvolvimento.

Considerações finais

Neste capítulo, apresentamos como criar um teste unitário e como identificar os testes necessários para construir o sistema e manter sua qualidade e avaliação, além de sua arquitetura.

Referências

GONÇALVES, Edson. **Desenvolvendo aplicações web com JSP, Servlets, JavaServer Faces, Hibernate, EJB 3 Persistence e Ajax**. Rio de Janeiro: Ciência Moderna, 2007.

TAHCHIEV, Petar *et al.* **JUnit in action**. Nova York: Manning, 2010.

Sobre o autor

Andrey Araujo Masiero é professor e desenvolvedor de software com interesse em arte, tecnologia e cultura. Doutor e mestre em engenharia elétrica na área de inteligência artificial pelo Centro Universitário FEI e bacharel em ciência da computação pela mesma instituição, foi premiado com destaque acadêmico pelo projeto RoboFEI @Home, em 2015, e com o primeiro lugar na Exposição dos Trabalhos Acadêmicos da Ciência da Computação (EXPOCOM), tendo apresentado o trabalho de Sistema de Gerenciamento de Patterns, Anti-patterns e Personas (SIGEPAPP). Profissional com 16 anos de experiência no mercado, participou de projetos no Governo do Estado de São Paulo e projetos na iniciativa privada, como a integração entre os bancos Santander e Real.