# Testes automatizados
# Engenharia de software I

Uma maneira para conseguir testar o sistema todo de maneira constante e contínua a um preço justo é automatizando os testes. Ou seja, escrevendo um programa que testa o seu programa. Esse programa invocaria os comportamentos do seu sistema e garantiria que a saída é sempre a esperada.

Um ponto que é sempre levantando em qualquer discussão sobre testes manuais versus testes automatizados é produtividade. O argumento mais comum é o de que agora a equipe de desenvolvimento gastará tempo escrevendo código de teste; antes o tempo era gasto apenas escrevendo código de produção. Portanto, essa equipe será menos produtiva.

A resposta para essa pergunta é: o que produtividade? Se produtividade for medida através do número de linhas de código de produção escritos por dia, talvez o desenvolvedor seja sim menos produtivo. Agora, se produtividade for a quantidade de linhas de código de produção sem defeitos escritos por dia, provavelmente o desenvolvedor será mais produtivo ao usar testes automatizados.

Além disso, se analisarmos o dia a dia de um desenvolvedor que faz testes manuais, podemos perceber a quantidade de tempo que ele gasta com teste. Geralmente o desenvolvedor executa testes enquanto desenvolve o algoritmo completo. Ele escreve um pouco, roda o programa, e o programa falha. Nesse momento, o desenvolve dor entende o problema, corrige-o, e em seguida executa novamente o mesmo teste.

Quantas vezes por dia o desenvolvedor executa o mesmo teste manual? O desenvolvedor que automatiza seus testes perde tempo apenas 1 vez com ele; nas próximas, ele simplesmente aperta um botão e vê a máquina executando o teste pra ele, de forma correta e rápida.

# Testes de Unidade

Imagine-se passeando em uma loja virtual qualquer na web. Ao selecionar um produto, o sistema coloca-o no seu carrinho de compras. Ao finalizar a compra, o sistema fala com a operadora de cartão de crédito, retira o produto do estoque, dispara um evento para que a equipe de logística separe os produtos comprados e te envia um e-mail confirmado a compra.

Desenvolvedores, quando pensam em teste de sofware, geralmente imaginam um teste que cobre o sistema como um todo. Um teste de unidade não se preocupa com todo o sistema; ele está interessado apenas em saber se uma pequena parte do sistema funciona.

Um teste de unidade testa uma única unidade do nosso sistema. Geralmente, em sistemas orientados a objetos, essa unidade é a classe. Em nosso sistema de exemplo, muito provavelmente existem classes como “CarrinhoDeCompras”, “Pedido”, e assim por diante. A ideia é termos baterias de testes de unidade separadas para cada uma dessas classes; cada bateria preocupada apenas com a sua classe.

Na prática, equipes acabam por executar poucos testes ralos, que garantem apenas o cenário feliz e mais comum.

# Escrevendo testes

Usaremos o JUnit, o framework de testes de unidade mais popular do mundo Java , Para testarmos uma classe que soma 2 valores

```
@Test
public void testSoma() {
	Calculos c = new Calculos();
	int a = 3;
	int b = 5;
	int retornoEsperado = 8;
	int retorno = c.soma(a, b);
	assertEquals(retornoEsperado, retorno);
}
```

# Casos de teste

números romanos

Numerais romanos foram criados na Roma Antiga e eles foram utilizados em todo o seu império. Os números eram representados por sete diferentes símbolos, listados na tabela a seguir.

- I, unus, 1, (um)
- V, quinque, 5 (cinco)
- X, decem, 10 (dez)
- L, quinquaginta, 50 (cinquenta)
- C, centum, 100 (cem)
- D, quingenti, 500 (quinhentos)
- M, mille, 1.000 (mil)

# O primeiro teste

Conhecendo o problema dos numerais romanos, é possível levantar os diferentes cenários que precisam ser aceitos pelo algoritmo: um símbolo, dois símbolos iguais, três símbolos iguais, dois símbolos diferentes do maior para o menor, quatro símbolos dois a dois, e assim por diante. Dado todos estes cenários, uns mais simples que os outros, começaremos pelo mais simples: um único símbolo.

Começando pelo teste “deve entender o símbolo I”. A classe responsável pela conversão pode ser chamada, por exemplo, de ConversorDeNumeroRomano, e o método converte(), recebendo uma String com o numeral romano e devolvendo o valor inteiro representado por aquele número:

```java
@Test
public void deveEntenderOSimboloI() {
	ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
	int numero = romano.converte("I");
	assertEquals(1, numero);
}
```

# Exercicio

Suponha agora que o projeto atual seja uma loja virtual. Essa loja virtual possui um carrinho de compras, que guarda uma lista de itens comprados. Um item possui a descrição de um produto, a quantidade, o valor unitário e o valor total desse item. Veja o código que representa esse carrinho:

```
public class CarrinhoDeCompras {
	private List<Item> itens;
	public CarrinhoDeCompras()
	{this.itens = new ArrayList<Item>();}
	public void adiciona(Item item)
	{this.itens.add(item);}
	public List<Item> getItens()
	{return Collections.unmodifiableList(itens);}
}
```

```java
public class Item {
	private String descricao;
	private int quantidade;
	private double valorUnitario;
	public Item(String descricao, int quantidade, double valorUnitario) {
		this.descricao = descricao;
		this.quantidade = quantidade;
		this.valorUnitario = valorUnitario;
	}
	public double getValorTotal()
	{	return this.valorUnitario * this.quantidade;	}
	// getters para os atributos
}
```

Agora imagine que o programador deva implementar uma funcionalidade que devolva o valor do item de maior valor dentro desse carrinho de compras. Pensando já nos testes, temos os seguintes cenários:

- Se o carrinho só tiver um item, ele mesmo será o item de maior valor.
- Se o carrinho tiver muitos itens, o item de maior valor é o que deve ser retornado.
- Um carrinho sem nenhum item deve retornar zero.