

# Programação Orientada por Objetos

## Testes

Prof. Cédric Grueau

Prof. José Sena Pereira

Departamento de Sistemas e Informática  
Escola Superior de Tecnologia de Setúbal  
Instituto Politécnico de Setúbal

2022/2023



# Sumário

- ▶ Testes
- ▶ Desenvolvimento dirigido pelos testes



## A aplicação CityAssets

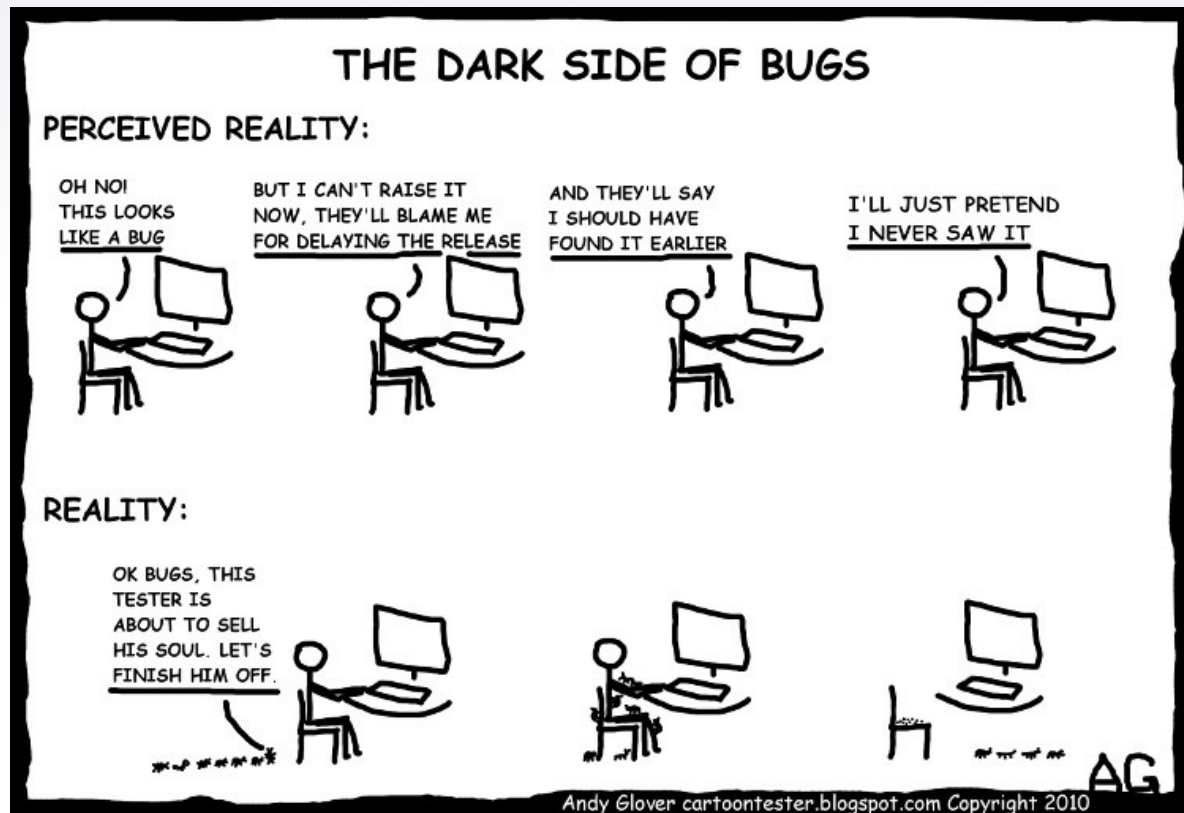
- ▶ Uma aplicação CityAssets que permita gerir os bens de uma câmara Municipal
  - ▶ Deve poder:
    - ▶ Mostrar o número de bens
    - ▶ Calcular o custo mensal de manutenção dos bens
    - ▶ Apresentar a informação de um dado bem



## Ideias para a solução...

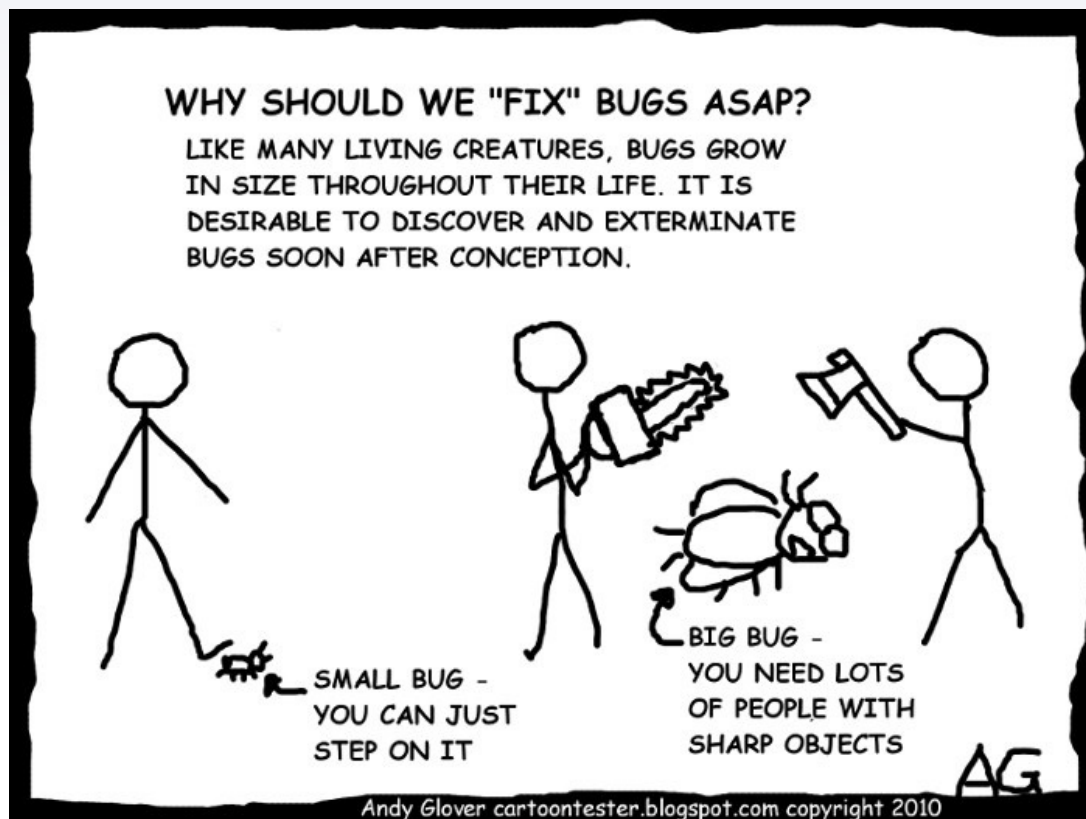
- ▶ Uma **classe** para um **bem**
- ▶ Uma **classe** que guarda um conjunto de bens, tirando partido da Java Collections Framework
- ▶ Uma **classe** que representa a câmara municipal
- ▶ Implementa-se esta solução e verifica-se se está tudo ok...

## E se houver erros?



Andy Glover [cartoontester.blogspot.com](http://cartoontester.blogspot.com) Copyright 2010

## Como lidar com erros?

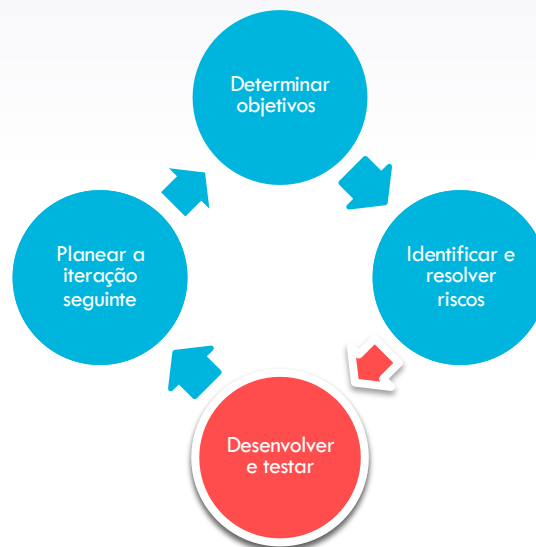


# Processo de desenvolvimento

## Modelo em cascata




## Modelo em espiral iterativo



# Como garantir a qualidade do software

- ▶ Durante TODO o processo
  - ▶ É fundamental
    - ▶ Identificar corretamente os requisitos
    - ▶ Desenhar uma boa solução
    - ▶ Implementar cuidadosamente a solução
    - ▶ Verificar a solução
    - ▶ Fazer evoluir corretamente a solução



Hoje, estamos sobretudo preocupados com esta parte, mas temos estado atentos a todas as restantes desde o início do ano!



# Várias técnicas para verificação da qualidade

- ▶ Revisão por pares da conceção (design), fonte (código) e outros entregáveis
- ▶ Verificação de qualidade com o auxílio de ferramentas
  - ▶ Frequentemente integradas no IDE
- ▶ Realização de testes sobre o código desenvolvido
- ▶ ...

# Testes unitários

- ▶ Focam-se numa parte específica da funcionalidade a que neste contexto vamos chamar unidades
  - ▶ Métodos
  - ▶ Classes
- ▶ Cada teste unitário corresponde a um **cenário de teste**
  - ▶ Tipicamente é implementado num **método de teste**
- ▶ Múltiplos cenários correspondem a múltiplos testes
  - ▶ Estes testes podem ser agrupados em **classes de teste**

# Testes unitários

- ▶ Porque gostamos de os fazer?
  - ▶ Automáticos – podem ser chamados com simples clicks
  - ▶ Escaláveis – tanto dão para programas pequenos como para sistemas com milhares de classes (ou mais)
  - ▶ Precisos – permitem identificar o ponto exato do programa que falhou
  - ▶ Programam-se como o resto do nosso Código
- ▶ E, no entanto...
  - ▶ Apenas testam unidades
    - ▶ E a integração e colaboração entre essas unidades, não se testa?

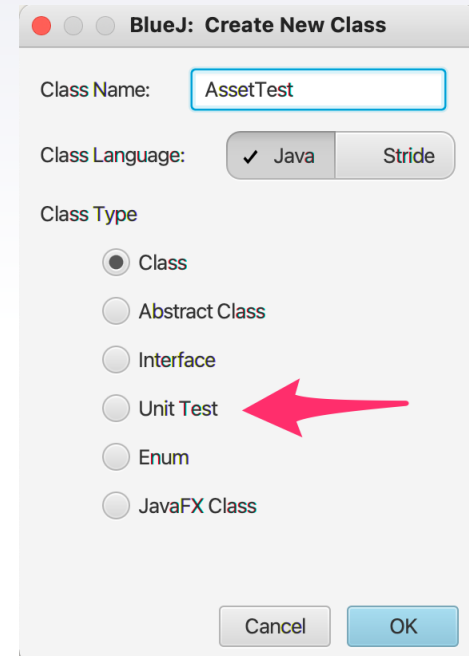
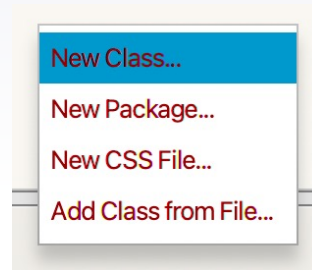


Exemplo



# Crie um caso de teste

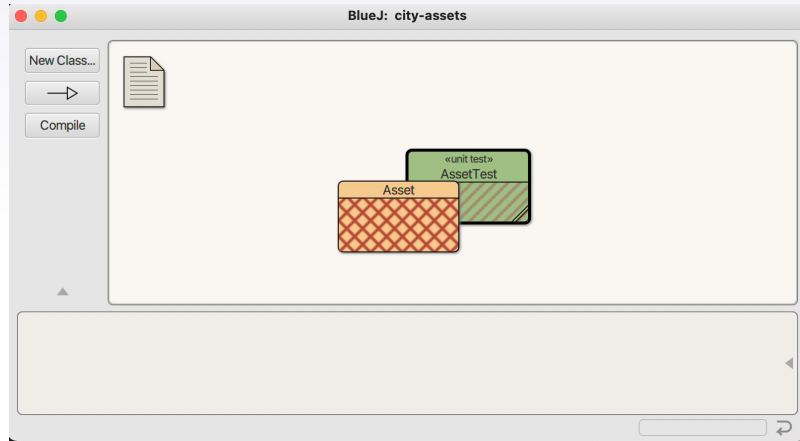
- ▶ Criar um caso de teste é semelhante a criar uma nova classe
  - ▶ No BlueJ, clique no botão da direita em cima do diagrama de classes e faça:
  - ▶ New-> Classe
  - ▶ Define o nome da classe com o sufixo Test
  - ▶ Escolhe o tipo de classe Unit Test



O Junit já está incluído por defeito no BlueJ.

Não é o caso do Apache Netbeans.

# Vista do projeto



- O BlueJ preenche a classe de teste criada com a importação das bibliotecas necessárias e uma estrutura com alguns métodos



## Vamos criar o nosso primeiro teste

- ▶ Cenário:
  - ▶ Criar um bem com um valor, um custo de manutenção e um número de 3 instâncias
  - ▶ Testar se o número de instâncias do bem, depois de criado, é exatamente 3
    - ▶ O teste é feito à custa de uma **asserção** do JUnit
- ▶ Nestes primeiros testes (e só nestes), vamos usar como cobaias construtores e seletores.
  - ▶ Na prática, como vamos ver mais a frente, nem sempre é necessário testar seletores

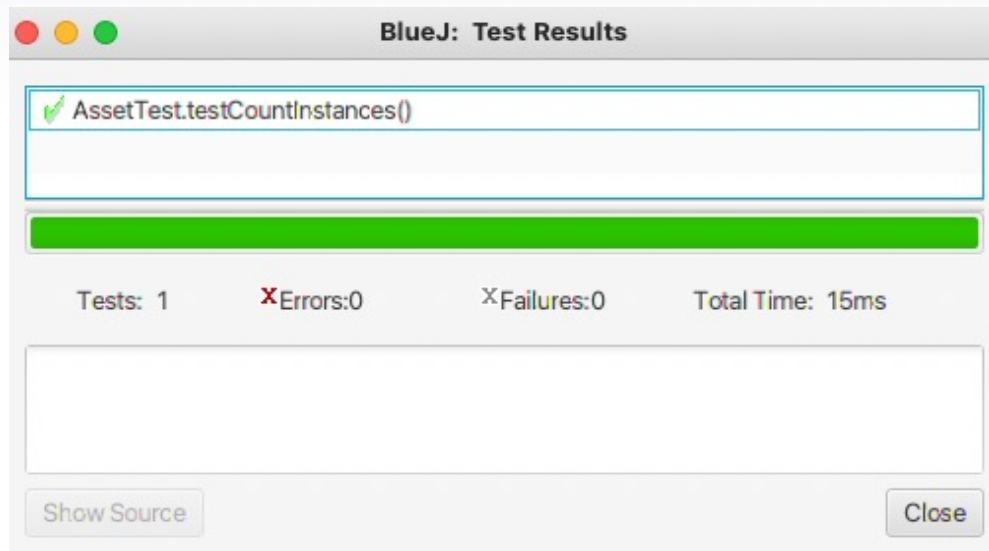
Quando criamos um novo bem

```
@Test  
public void testCountInstances(){  
    asset = new Asset(6700.0,29.0,3);  
    assertEquals(3,asset.getNumberOfInstances());  
}
```



# Correr o teste

- ▶ Para verificar que ao número de bens criados é mesmo 1

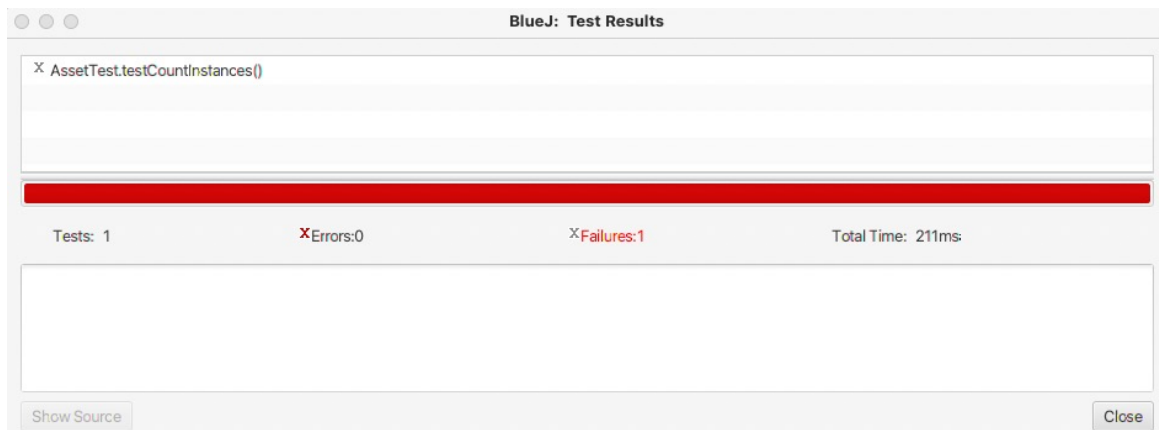


**Barra verde**  
significa que  
todos os testes  
passaram  
(podem ser  
milhares)

Experimente “estragar o teste”, admitindo que o número de testes deveria ser 2

```
@Test
public void testCountInstances(){
    asset = new Asset(6700.0,29.0,3);
    assertEquals(6,asset.getNumberOfInstances());
}
```

**Barra vermelha**  
significa que pelo menos um teste falhou (podem ser milhares)



## ○ método **assertEquals()**

- ▶ O método estático `org.junit.Assert.assertEquals` compara os seus dois argumentos, que podem ser:
  - ▶ Valores de tipos primitivos(`int`, `float`, `boolean`, `double`, `long`, `short`, `char`, `byte`)
  - ▶ Objetos (de quaisquer tipos)
- ▶ Se forem iguais, o teste passa
- ▶ Se forem diferentes, o teste não passa

## Outro teste...

```
@Test
public void testToString() {
    Asset asset = new Asset(15000.00,155.55);
    assertEquals("Capital 15000.00 € - custo mensal 155.55 €",asset.toString());
}
```

## Construção do método de teste

- ▶ Criar o objeto e colocá-lo num estado conhecido
- ▶ Invocar um método que retorna o resultado “real”
- ▶ Construir o resultado esperado
- ▶ Invocar:

**`assertEquals(valorEsperado, valorReal)`**

- ▶ Se tudo estiver bem, o teste passa

Nota: para que isto funcione bem, temos de implementar o método **`equals()`** na classe dos objetos a testar!



## Quando especificar os testes?

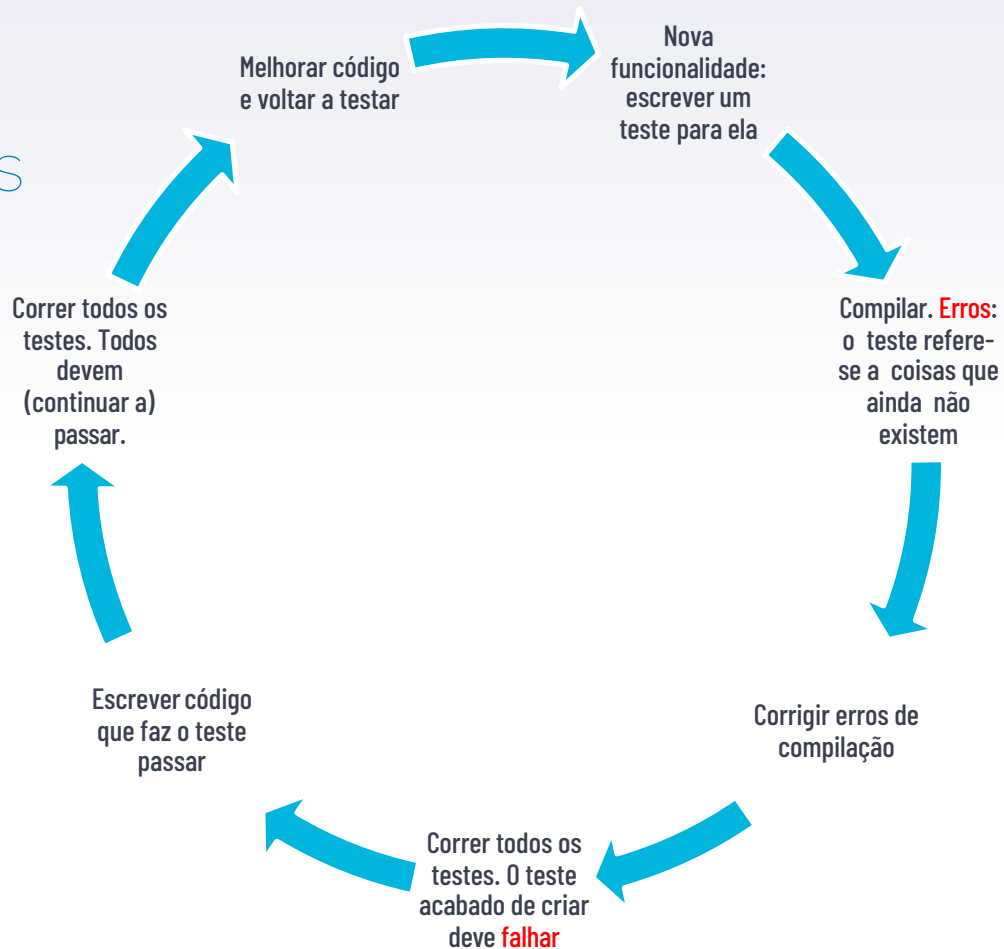
- ▶ Antes de desenvolver o código a testar?
- ▶ Depois de desenvolver o código a testar?

Test-Driven Development - TDD

# Desenvolvimento dirigido pelos testes



# Desenvolvimento dirigido pelos testes





# Desenvolvimento dirigido pelos testes

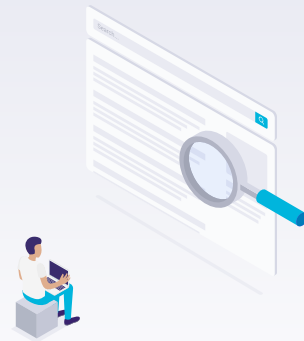
- ▶ Os testes **dirigem** o processo
  - ▶ **Antes** de acrescentar funcionalidade, escrever um teste
  - ▶ Depois de falhar o teste, acrescentamos a funcionalidade
- ▶ O que ganhamos com este processo?
  - ▶ **Controlo** sobre o processo de desenvolvimento
- ▶ **Princípios fundamentais:**
  - ▶ Nunca estar a mais de um teste de distância da **barra verde**
  - ▶ **Nunca** escrever novo teste quando se está com a **barra vermelha**
  - ▶ Correr **todos** os testes **frequentemente** (pelo menos uma vez por manhã ou tarde, de preferência várias)

# Funcionamento dos testes de unidade

- ▶ Independência dos testes
  - ▶ Ordem de execução dos testes deve ser sempre **irrelevante**
- ▶ Testar a interface, não a implementação
  - ▶ Os testes devem ser criados a pensar na interface
  - ▶ Nunca expor estado interno dum objeto para efeitos de teste
    - ▶ Se necessário, enriqueça a interface – mesmo que isso implique criar operações públicas só para facilitar os testes
- ▶ Não testar a plataforma
  - ▶ A API do Java está bem testada, não deve ter de a testar

Desenvolvimento dirigido pelos testes

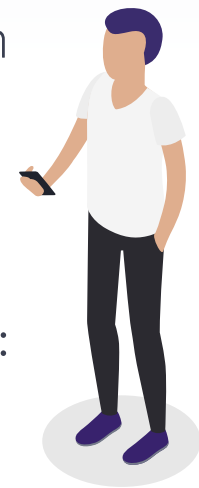
Test-Driven Development - TDD



# Exemplo

## A aplicação CityAssets

- ▶ Por uma questão de transparência para com os seus cidadãos, uma cidade decide conceber um sistema informático que lhe permita representar todos os bens (edifício, veículo,...) que possui.
- ▶ Um bem é definido pelo seu valor e custo mensal de manutenção. Também queremos saber o número de instâncias de cada bem.
- ▶ Para o mini sistema de informação que estamos a desenvolver, uma cidade pode ser considerada como uma classe de objetos que referencia todos os bens que possui. Esta classe deve oferecer os seguintes serviços:
  - Consultar a informação de um determinado imóvel,
  - Calcular o custo mensal total de manutenção dos bens,



## Comecemos pela classe **Asset** (Esqueleto)

- ▶ É de notar que as classes são construídas por incrementos.

```
public class Asset
{
    private double value;
    private double manutentionCost;
    private int numberOfInstances;

    public Asset(double value, double cost, int number) {}

    public double getValue(){}

    public double getManutentionCost(){}

    public void setManutentionCost(double cost){}

    public static int getNumberOfInstances(){}

    public String toString(){}
}
```



Normalmente... Se seguir o que foi feito em IPOO

1. Começaria por identificar variáveis e constantes a usar
2. Especificaria o construtor, seguido dos restantes métodos
3. Testaria no fim, ou à medida que os métodos fossem ficando prontos

## Regras específicas do TDD

- ▶ No desenvolvimento guiado pelos testes, tem de começar por especificar testes **antes** de escrever o código
- ▶ Começamos pelo construtor, dado que sem ele não vamos conseguir fazer mais nada
- ▶ Acrescente um teste simples ao construtor
  - ▶ Sim, esse mesmo, que **ainda não desenvolveu**

## Vários erros de compilação

- ▶ Resultado expectável nesta fase

```
@Test
public void testSimpleConstructor(){
    asset = new Asset(15000.0, 155.55, 1);
    assertEquals(15000.0, asset.getValue());
}
```



## Na classe `Asset`, falta o construtor

- ▶ Lembre-se, para começar, só queremos compilar...

```
public Asset(double value, double cost, int number)
{
}
```

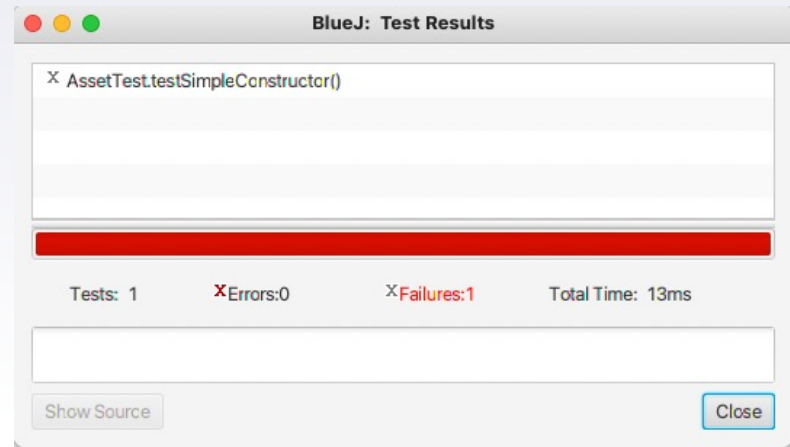
```
public double getValue()
{
    return 0.0;
}
```

```
public double getManutentionCost()
{
    return 0.0;
}
```

Para que o programa possa compilar, garantimos que os métodos retornam um valor *neutro* compatível com o tipo de retorno.

# Vamos testar...

- ▶ Naturalmente o teste dá erro uma vez que o construtor ainda não faz nada!
- ▶ Os seletores também não...



```
public Asset(double value, double cost, int number)
{
}

public double getValue()
{
    return 0.0;
}

public double getManutentionCost()
{
    return 0.0;
}
```


## Transformemos o teste para ser mais geral

- ▶ Só podemos avançar para o próximo teste depois de correr com sucesso este


```
@Test  
public void testSimpleConstructor(){  
    asset = new Asset(15000.0, 155.55, 1);  
    assertEquals(15000.0, asset.getValue());  
}
```

## Vamos corrigir o código da classe `Asset`

- Sucesso, podemos avançar para um seletor do atributo `manutentionCost`

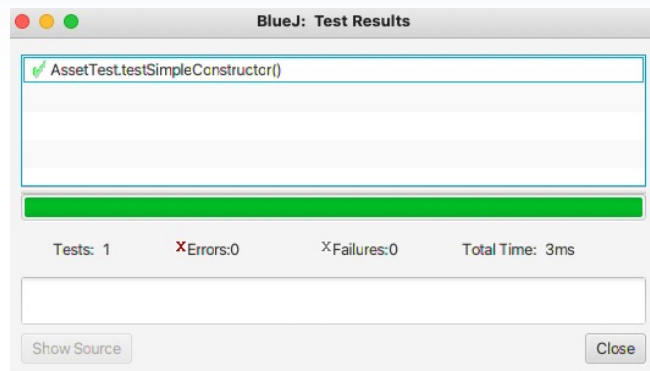


```
public Asset(double value, double cost, int number)
{
    this.value = value;
    this.manutentionCost = cost;
    this.numberOfInstances = number;
}
```



```
public double getValue()
{
    return value;
}
```

```
public double getManutentionCost()
{
    return 0.0;
}
```

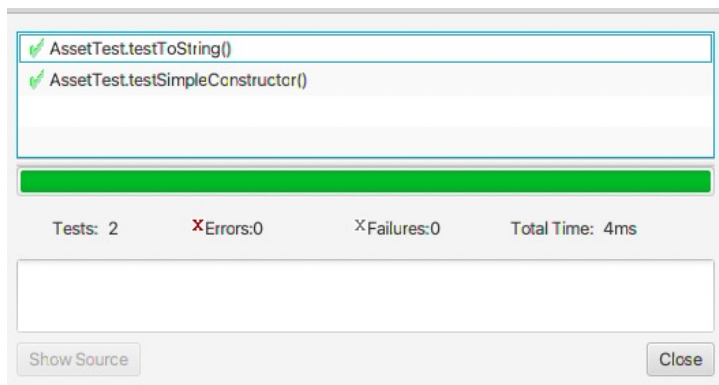


Primeiro, construímos o novo teste

```
@Test
public void testToString(){
    Asset asset = new Asset(15000.0,155.55,1);
    assertEquals("Capital 15000.0 € - custo mensal 155.55 € - 1 Instância(s).",asset.toString());
}
```

Este código é suficiente para passar o teste (e errado, claro)

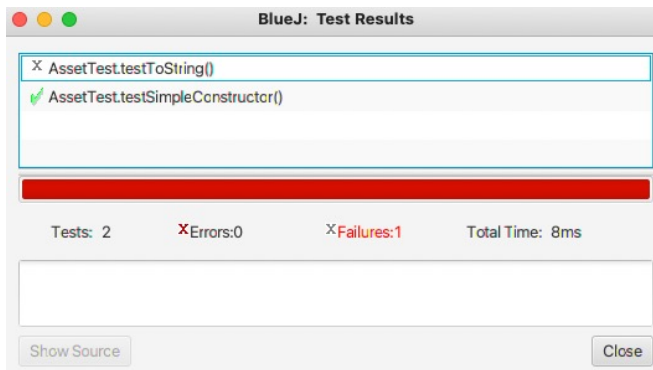
```
@Override  
public String toString(){  
    return "Capital 15000.0 € - custo mensal 155.55 € - 1 Instância(s).";  
}
```



# Temos de criar um teste em que o código não passe

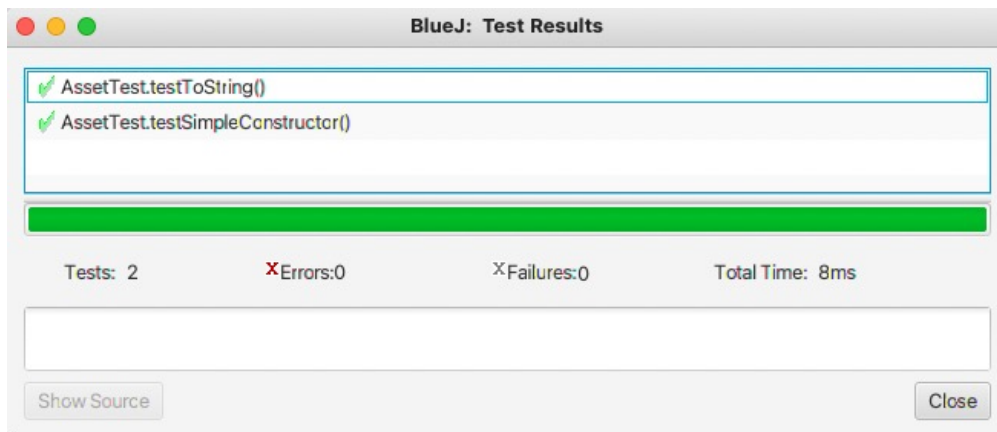
@Test

```
public void testToString(){  
    Asset asset = new Asset(15000.0,155.55,1);  
    Asset asset2 = new Asset(6700.0,29.0,3);  
    assertEquals("Capital 15000.0 € - custo mensal 155.55 € - 1 Instância(s).",asset.toString());  
    assertEquals("Capital 6700.0 € - custo mensal 29.0 € - 3 Instância(s).",asset2.toString());  
}
```



Agora, alteramos o código, para passar no teste

```
@Override
public String toString() {
    return "" + "Capital " + value +
        " € - custo mensal " + manutentionCost + " € - " +
        numberOfInstances + " Instância(s).";
}
```

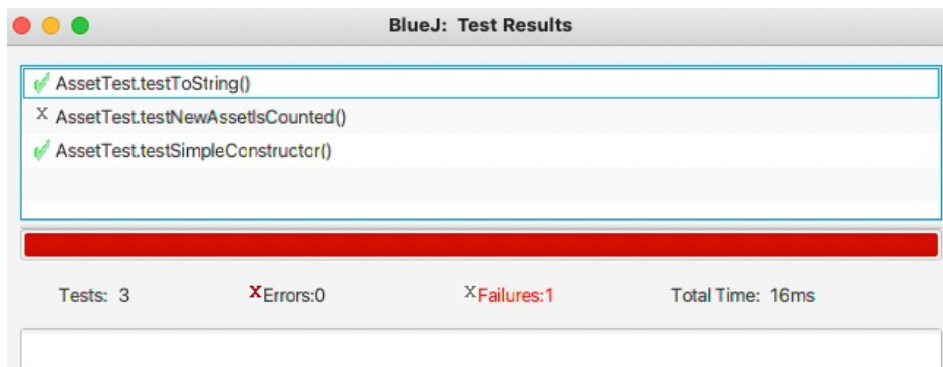




# Testar o seletor do número de bens

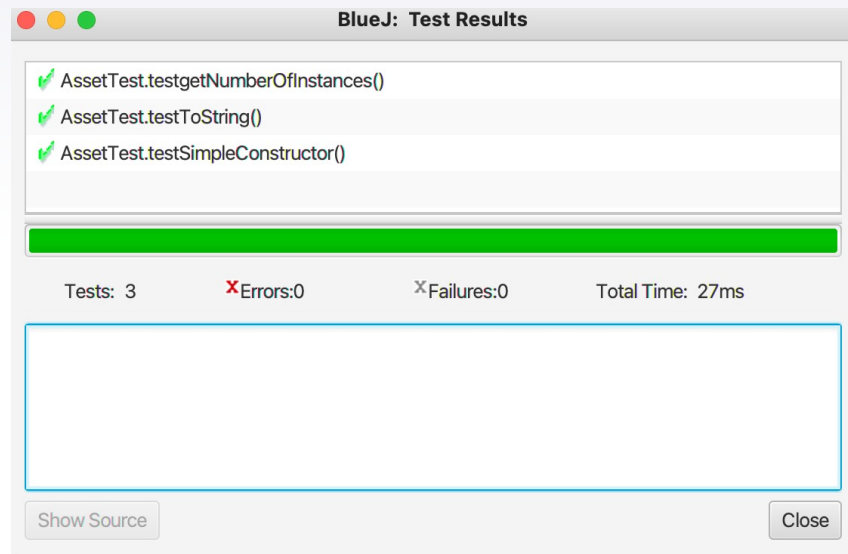
**@Test**

```
public void testgetNumberOfInstances() {  
    asset = new Asset(6700.0, 29.0, 2);  
    assertEquals(2, asset.getNumberOfInstances());  
}
```



► Agora, alteramos o código, para passar no teste

```
public int getNumberOfInstances()  
{  
    return numberOfInstances;  
}
```



Agora, passa o teste

# Definir uma fixture

```
public class AssetTest
{
    Asset asset, asset2;
    public AssetTest()
    {
        asset = new Asset(15000.0,155.55);
        asset2 = new Asset(6700.0,29.0);
    }

    public int getNumberOfInstances()
    {
        return numberOfInstances;
    }
}
```

```
@Test
    public void testToString(){
        assertEquals("Capital 15000.0 € - custo mensal 155.55 € - 1
Instância(s).",asset.toString());

        assertEquals("Capital 6700.0 € - custo mensal 29.0 € - 3
Instância(s).",asset2.toString());
    }

@Test
    public void testSimpleConstructor(){
        assertEquals(15000.0, asset.getValue());
        assertEquals(155.55, asset.getManutentionCost());
    }
}
```





Para quê fazer código incompleto até ter testes que o detectem?

- ▶ Queremos ter uma bateria de testes que cubra quer os casos particulares, quer o caso geral
- ▶ É arriscado deixar funcionalidades por testar
  - ▶ Queremos criar testes para **todas** as funcionalidades exigidas
  - ▶ Devemos tentar tornar impossível que uma implementação fictícia possa passar nos vários cenários de teste
- ▶ Portanto, só devemos acrescentar novo Código **depois** de ter um teste que falhe na sua ausência

# Podemos agora melhorar o código

- ▶ Atividade conhecida como **refabricação** (refactoring)
  - ▶ Devemos aperfeiçoar código que esteja a funcionar
    - ▶ Melhorando a sua estrutura interna
    - ▶ **Sempre** sem alterar o seu comportamento externo
- ▶ Para quê?
  - ▶ Tornar o código fonte tão fácil de compreender quanto possível
  - ▶ Preparar código para facilitar adição de novas funcionalidades
- ▶ Mas isto não tem riscos?
  - ▶ Pode haver **regressões**, i.e., podemos inadvertidamente introduzir bugs em código que estava a funcionar bem
    - ▶ Por isso, é fundamental ter testes que detetem esses bugs que podemos inadvertidamente introduzir durante as modificações

## Algumas regras no desenvolvimento guiado por testes

- ▶  Só *refabricamos* código que passa em todos os testes existentes
- ▶  Só acrescentamos funcionalidades novas quando há um teste que o código atual não passa
  - ▶ Se queremos acrescentar uma nova funcionalidade, acrescentamos um novo teste que “especifica” essa funcionalidade
  - ▶ Assim, garantimos ter pelo menos um teste para cada nova funcionalidade acrescentada

## Exemplo de refabricação

- Consideramos que a aplicação necessita de um segundo construtor que inicializa por defeito o número de bens a um. Uma solução para este segundo construtor poderia ser:

```
public Asset(double value, double cost)
{
    this.value = value;
    this.manutentionCost = cost;
    this.numberOfInstances = 1;
}
```

# Exemplo de refabricação

```
public Asset(double value, double cost, int number)
{
    this.value = value;
    this.manutentionCost = cost;
    this.numberOfInstances = number;
}

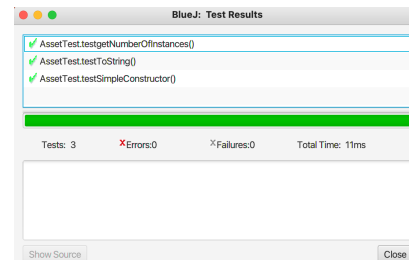
public Asset(double value, double cost)
{
    this.value = value;
    this.manutentionCost = cost;
    this.numberOfInstances = 1;
}

public Asset(double value, double cost) {
    this(value, cost, 1);
}
```

- Código repetido, considerado um sinal de má programação

- Versão refabricada, com o construtor simples a invocar o construtor mais complexo

O teste da versão refabricada é executado e confirma  
A correção do código refabricado.





Testar seletores



## Vale a pena testar um seletor?

- ▶ Depende...
  - ▶ Se o seletor se limita a retornar o valor de um campo, o teste é uma perda de tempo
    - ▶ Estes métodos podem ser simples demais para poder falhar por si só

```
@Override  
public String getName() {  
    return this.name;  
}
```

Pode testar estes métodos indiretamente!

- Recordar testes ao construtor
- Se o seletor faz algo mais complexo, vale a pena testar o seletor

# Testes a seletores

- ▶ Começar por inicializar alguns objetos
- ▶ Invocar os seletores a testar e verificar se os resultados são os esperados
- ▶ Já vimos alguns exemplos, em testes anteriores

# Testar modificadores



## Vale a pena testar um modificador?

- ▶ Depende...
  - ▶ Se o modificador se limita a afetar o valor de um campo, por cópia de um argumento, ou algo de simplicidade equivalente, o teste pode ser uma perda de tempo
    - ▶ Estes métodos podem ser simples demais para poder falhar por si só

```
@Override  
public void setName(String name)  
{  
    this.name = name;  
}
```

- Em todo o caso, pode testar estes métodos indiretamente, recorrendo aos seletores
- Se o modificador faz algo mais complexo, vale a pena testar o modificador

# Testar operações sobre coleções

- ▶ Operações que modificam a coleção
- ▶ Operações que visitam a coleção para recolher (acumular, filtrar, ...) informação
- ▶ Operações que comparam coleções

## Operações que modificam a coleção

- ▶ Inserções (rever **addAsset()**)
- ▶ Remoções (rever **deleteAsset()**)
- Reordenações

Operações que visitam a coleção para recolher informação





# Esqueleto da classe **Assets**

```
/**
 * Um conjunto de bens
 *
 * @author Cédric Grueau
 * @version marco 2022
 */

import java.util.ArrayList;

public class Assets
{
    ArrayList<Asset> assets;

    /**
     * Constructor for objects of class Assets
     *
     * @param assets um array de bens
     */
    public Assets(Asset[] assets)
    {

    }
}
```

```
public String getInfo(int indice)
{

}

public ArrayList<Asset> getAssets() {
}

public void removeAssets(Asset[] assets) {
}

public int size() {
}
}
```

# Esqueleto da classe **Assets**, versão “compilável”

```
/**
 * Um conjunto de bens
 *
 * @author Cédric Grueau
 * @version marco 2022
 */

import java.util.ArrayList;

public class Assets
{
    ArrayList<Asset> assets;

    /**
     * Constructor for objects of class Assets
     *
     * @param assets um array de bens
     */
    public Assets(Asset[] assets)
    {

    }
}
```

```
public String getInfo(int indice)
{
    return null;
}

public ArrayList<Asset> getAssets() {
    return null;
}

public void removeAssets(Asset[] assets) {
}

public int size() {
    return 0;
}
}
```

## Teste de **Assets (Asset[] assets)**

```
public class AssetsTest
{
    private Asset[] someAssets =
        {new Asset(130.0,0.0,10),new Asset(3500.0,250.0,5),
        new Asset(39000.0,1000.0,1)};
    private Assets assets;

    public AssetsTest() {}

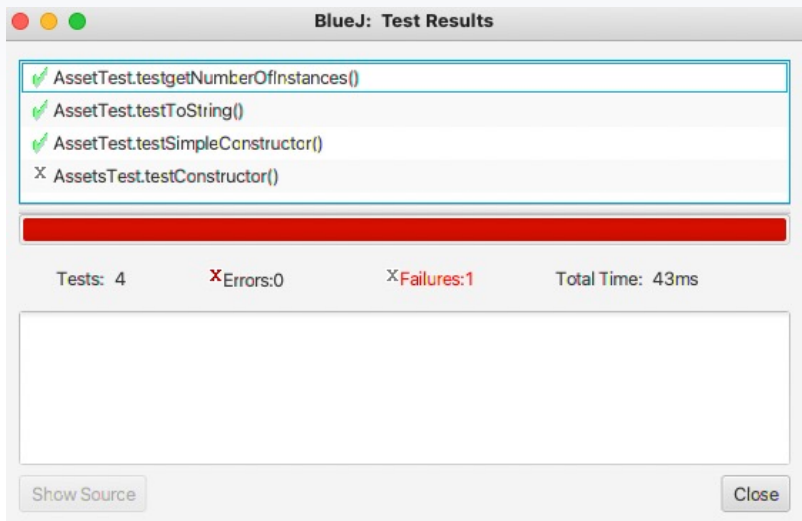
    @BeforeEach
    public void setUp(){ assets = new Assets(someAssets);}

    @Test
    public void testConstructor() {
        System.out.println("Teste do construtor");
        assertEquals(3, assets.size());
    }
}
```

Começamos por declarar variáveis de instância para os nossos testes.

A anotação `@Before` indica que o método `setUp()` deve ser executado antes de cada um dos testes da classe `AssetsTest`.

## Teste à operação `Assets (Asset[] assets)`



- ▶ O teste falha, porque o construtor e o método **size** ainda tem a implementação por omissão

```
public Assets (Asset[] assets) {}  
  
public int size() {  
    return 0;  
}
```

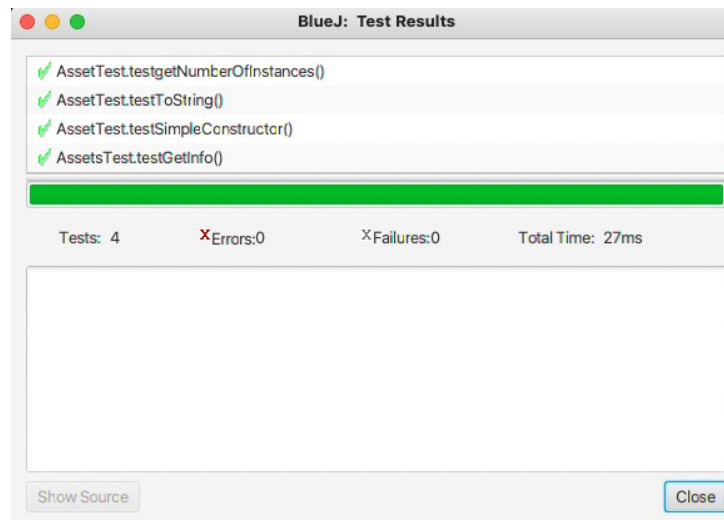
## Implementação de **Assets (Asset[] assets)**

```
public Assets (Asset[] assets)
{
    this.assets = new ArrayList<Asset>();
    for (Asset a : assets) {
        this.assets.add(a);
    }
}
```

## Implementação de **size()**

```
public int size() {  
    return assets.size();  
}
```

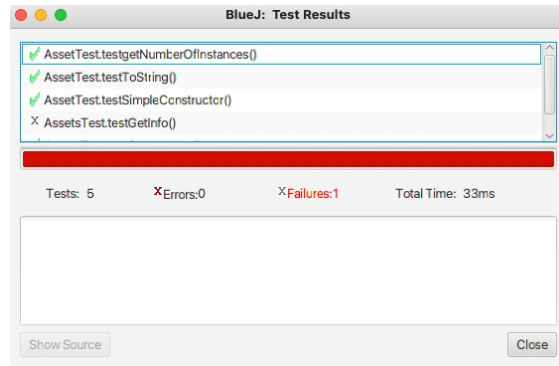
- ▶ O teste passa, após a implementação do construtor e do método **size**



# Teste de `getInfo(int indice)`

`@Test`

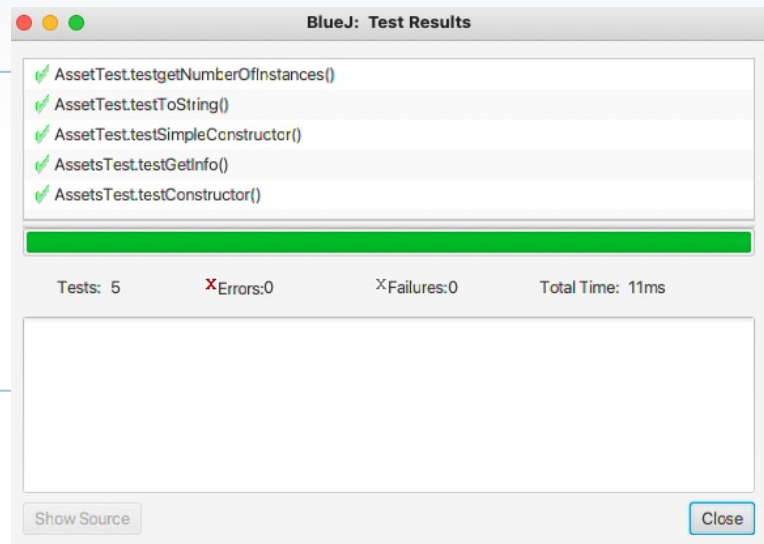
```
public void testGetInfo() {  
    assertEquals("Capital 3500.0 € - custo mensal 250.0 € - 5 Instância(s).", assets.getInfo(1));  
}
```



► O teste falha, porque o construtor e o método `getInfo()` ainda tem a implementação por omissão

## Implementação de `getInfo (int indice)`

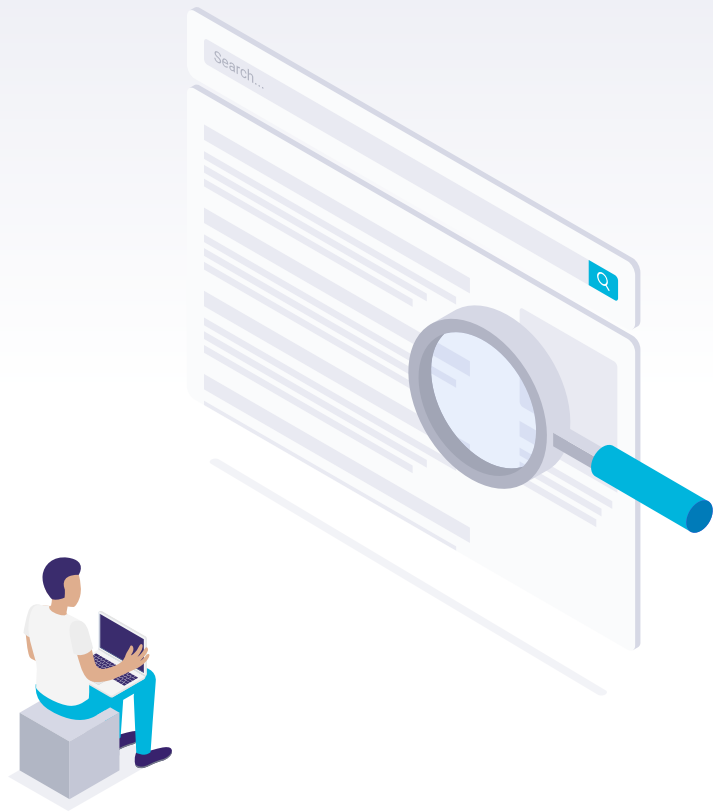
```
public String getInfo (int indice)
{
    return assets.get (indice) .toString();
}
```





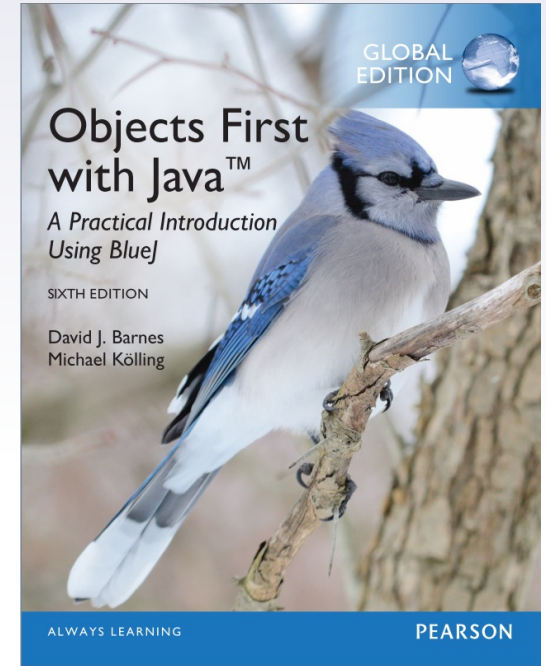
# Resumo

- ❑ Definição
- ❑ Processo de teste



# Bibliografia

- ▶ Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
  - ▶ Capítulo 9



# Instruções de afirmação do JUnit

- ▶ Boolean
  - ▶ `assertTrue(condition)`
  - ▶ `assertFalse(condition)`
- ▶ Null object
  - ▶ `assertNull(object)`
  - ▶ `assertNotNull(object)`
- ▶ Identical
  - ▶ `assertSame(expected, actual)`
  - ▶ `assertNotSame(expected, actual)`
- ▶ Assert Equals
  - ▶ `assertEquals(expected, actual)`
- ▶ Assert Array Equals
  - ▶ `assertArrayEquals(expected, actual)`
  - ▶ `assertEquals(expected[i], actual[i])`
  - ▶ `assertArrayEquals(expected[i], actual[i])`

# Exemplos de utilização de instruções de afirmações

```
@Test
public void testAssert(){

    //Declaração de variáveis
    String string1="JUnit";

    String string2="JUnit";
    String string3="test";
    String string4="test";
    String string5=null;
    int variable1=1;
    int variable2=2;
    int[] airethematicArray1 = { 1, 2, 3 };
    int[] airethematicArray2 = { 1, 2, 3 };

    // ...
```

```
//instruções de
assertEquals(string1,string2);

assertSame(string3, string4);

assertNotSame(string1, string3);

assertNotNull(string1);
assertNull(string5);
assertTrue(variable1<variable2);

assertArrayEquals(airethematicArray1, airethematicArray2);

}
```

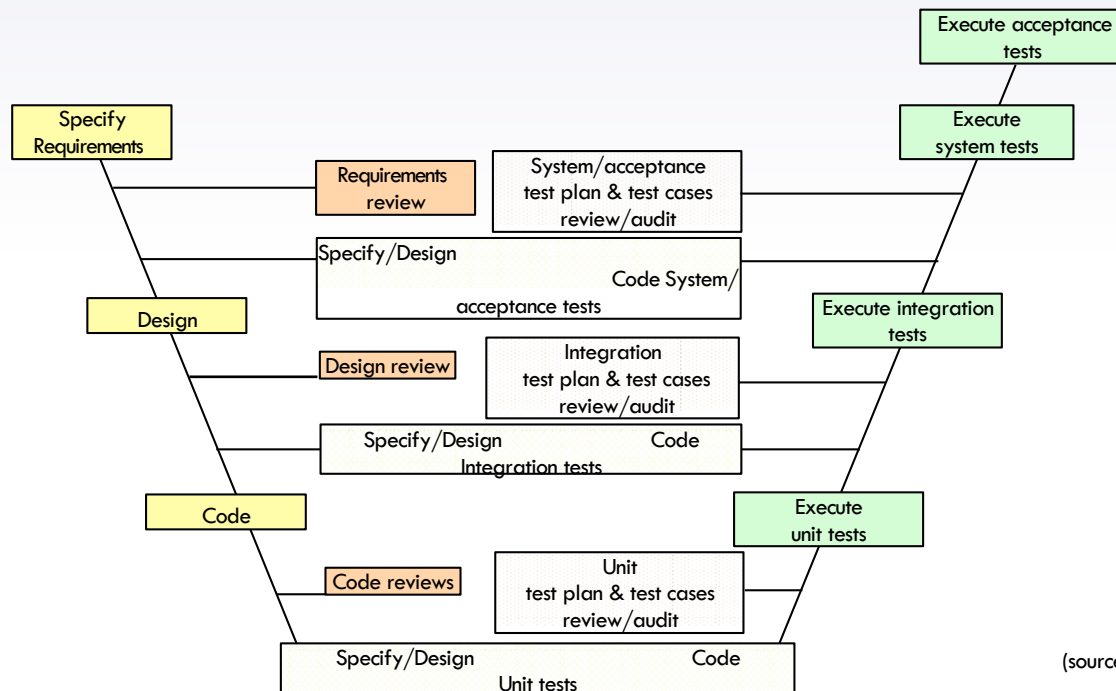
## O que é o teste de software?

- ▶ O teste de software consiste na verificação **dinâmica** de que o comportamento de um programa, de acordo com um conjunto **finito** de **casos de teste**, escolhidos de modo apropriado de entre um conjunto normalmente infinito de testes possíveis, cumpre o comportamento **esperado**

## Implicações da definição de teste

- ▶ O teste implica executar o programa com determinados inputs
- ▶ Normalmente, é impossível testar todas as situações possíveis, por serem infinitas
- ▶ Podemos e devemos usar estratégias de desenho de testes para criar bons casos de teste
- ▶ Dado um teste, temos de ser capazes de decidir se um resultado é ou não aceitável
  - ▶ Esta decisão é conhecida como o problema do oráculo

# Processo de teste (Modelo em V estendido)



(source: I. Burnstein, pg.15)

## Fases de teste – Teste Unitário

- ▶ Teste de unidades individuais, ou de grupos de unidades relacionadas
- ▶ Tipicamente, um tipo de teste sobre a API
- ▶ Tipicamente, da responsabilidade do programador
- ▶ Testes baseados em experiência, especificações e no Código
  - ▶ Mas o programador pode seguir estratégias com provas dadas!
- ▶ O principal objetivo é detetar defeitos funcionais e estruturais na unidade a testar



## Processo de testes – Testes de integração

- ▶ Teste em que os componentes são combinados e testados para avaliar a interação entre eles
- ▶ Normalmente, são da responsabilidade de uma equipa de testes independente de quem programou os componentes
- ▶ Os testes são baseados numa especificação do Sistema
  - ▶ Especificações técnicas, desenho do sistema
- ▶ O principal objetivo é detetar defeitos que ocorram devido a interações entre unidades diferentes, e que sejam visíveis ao nível da interface

## Fases de teste – Testes de sistema

- ▶ Testes realizados sobre um sistema completo, de modo a avaliar se o sistema está de acordo com os requisitos para ele especificados
  - ▶ Requisitos funcionais e não-funcionais (ex. Eficiência)
- ▶ Testes funcionais de caixa-negra, através da interface com o utilizador, normalmente baseados em informação recolhida no documento de requisitos
  - ▶ Estes testes ignoram completamente os detalhes de implementação do sistema
- ▶ Normalmente, são testes realizados por uma equipa independente

## Fases de teste – Testes de Aceitação

- ▶ Testes formais destinados a determinar se um sistema satisfaz, ou não, um determinado conjunto de critérios de aceitação, de modo a que o cliente possa decidir se deve ou não aceitar o sistema
- ▶ Testes formais realizados para permitir que o utilizador, cliente, ou outra entidade autorizada aceite um sistema, ou um componente
  - ▶ Tipicamente realizados pelo cliente
  - ▶ Frequentemente os testes são baseados num documento de requisitos, ou no manual do utilizador
  - ▶ O principal motivo é verificar se os requisitos e expectativas são atingidos

## Fases do teste – Testes de Regressão

- ▶ Repetição seletiva de testes de um sistema, ou de um componente, para verificar que:
  - ▶ As modificações feitas desde o último teste não provocaram efeitos indesejados
  - ▶ O sistema, ou componente, continuam a satisfazer os requisitos especificados

# ▶ Testes

- ▶ O mecanismo de testes pode demonstrar a existência de defeitos, mas nunca a sua ausência
- ▶ No contexto de POO, ficamos à um nível superficial da prática efetiva de testes de software