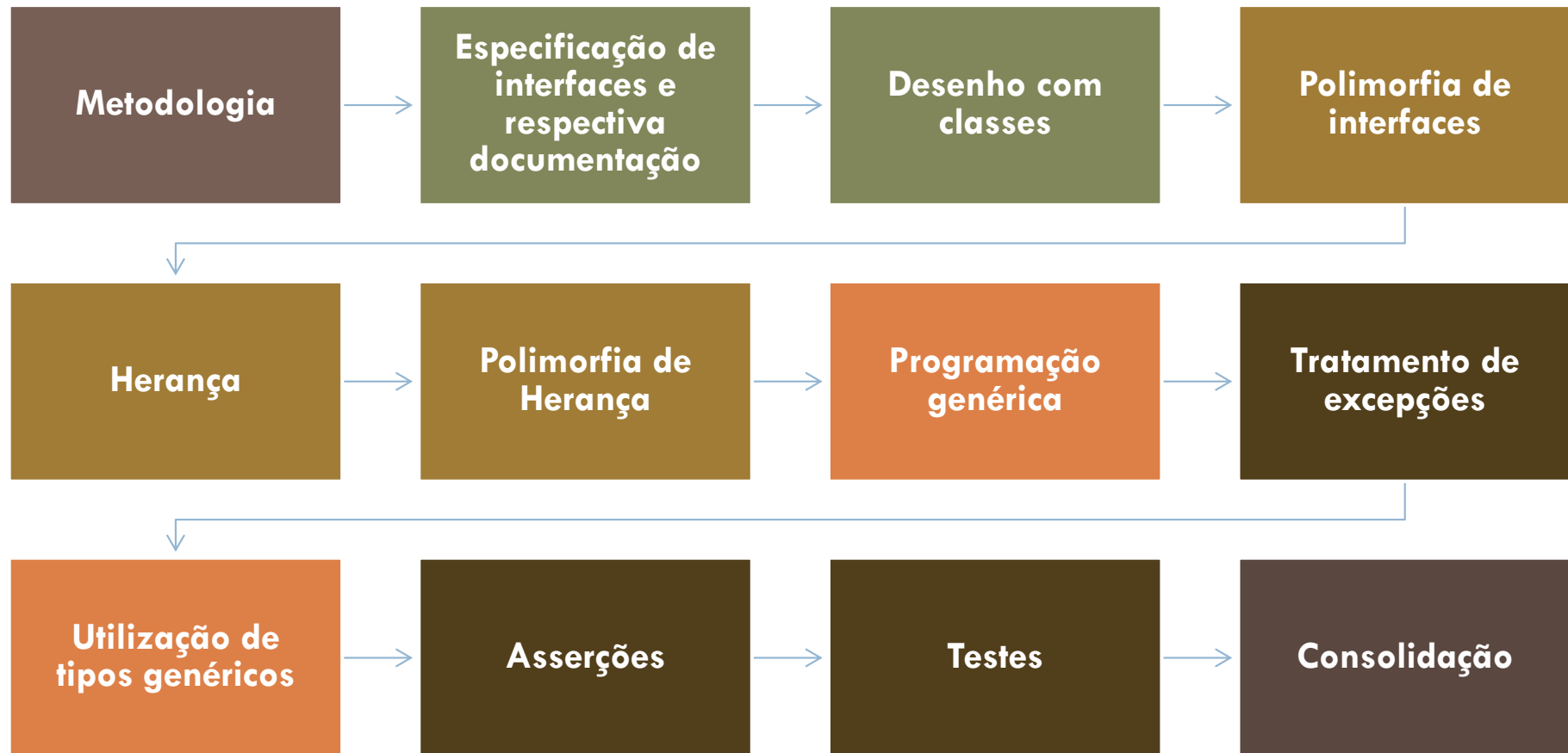


PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Testes

Programa

2



A aplicação PlayList

3

- Crie uma aplicação PlayList que permita gerir as suas músicas favoritas
 - Deve poder:
 - Adicionar músicas
 - Remover músicas
 - Classificar músicas
 - Pesquisar músicas
 - Listar músicas
 - Obter informações sobre as músicas



Já estamos a ver o filme...

4

- Uma classe com o interpretador de comandos
- Uma interface para a Playlist
- Uma classe que implementa a interface Playlist, tirando partido da Java Collections Framework
- Uma interface para representar o conceito de música
- Uma classe que implementa a interface que representa o conceito de música
- Experimenta-se cuidadosamente, a ver se está tudo ok...

5



TESTING

I FIND YOUR LACK OF TESTS DISTURBING.

9/9

0800 Antan started
1000 " stopped - antan ✓
1300 (032) MP - MC ~~1.982647000~~
2.130476415 (23) 4.615925059(-2)

(033) PRO 2 2.130476415
connect 2.130676415

Relays 6-2 in 033 failed special speed test
in relay " 10.000 test.

Relay
2145
Relay 3376

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545



Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.
1630 Antan started.
1700 closed down.

Bugs? Naaah...

7

THE DARK SIDE OF BUGS

PERCEIVED REALITY:

OH NO!
THIS LOOKS
LIKE A BUG



BUT I CAN'T RAISE IT
NOW, THEY'LL BLAME ME
FOR DELAYING THE RELEASE



AND THEY'LL SAY
I SHOULD HAVE
FOUND IT EARLIER



I'LL JUST PRETEND
I NEVER SAW IT



REALITY:

OK BUGS, THIS
TESTER IS
ABOUT TO SELL
HIS SOUL. LET'S
FINISH HIM OFF.



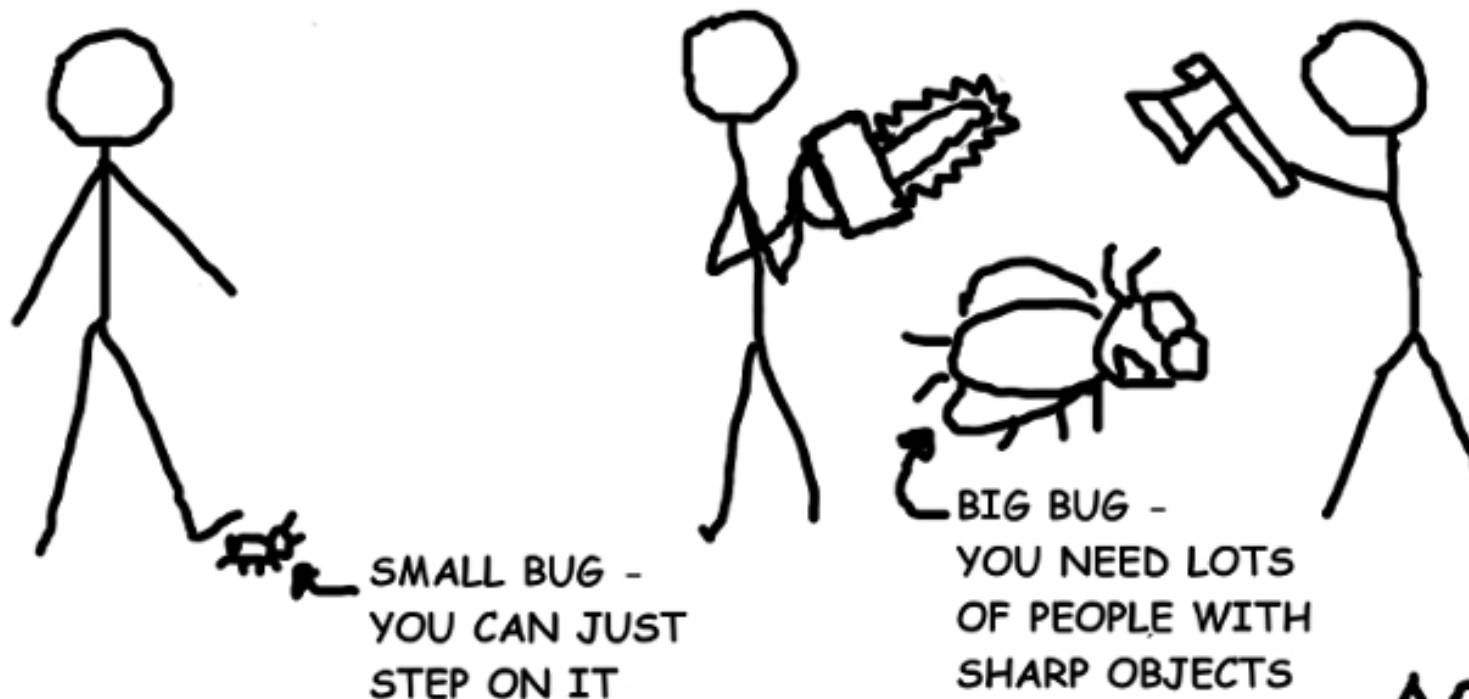
AG

Ok, ok, encontrei um *bug*. E agora?

8

WHY SHOULD WE "FIX" BUGS ASAP?

LIKE MANY LIVING CREATURES, BUGS GROW
IN SIZE THROUGHOUT THEIR LIFE. IT IS
DESIRABLE TO DISCOVER AND EXTERMINATE
BUGS SOON AFTER CONCEPTION.



SMALL BUG -
YOU CAN JUST
STEP ON IT

BIG BUG -
YOU NEED LOTS
OF PEOPLE WITH
SHARP OBJECTS

AG

Processo de desenvolvimento

9

Modelo em cascata

Análise

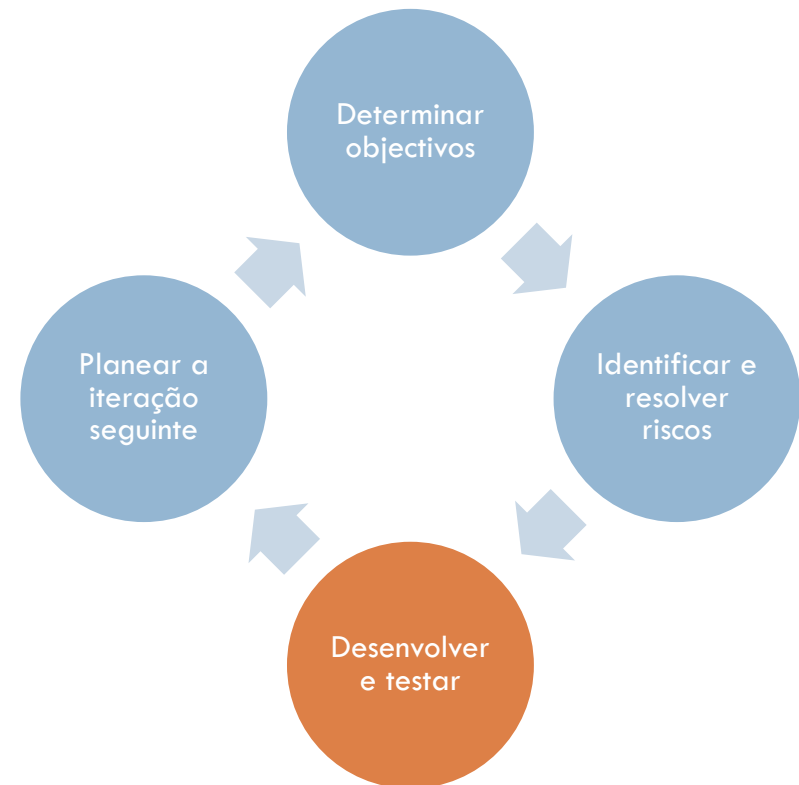
Desenho

Implementação

Verificação

Evolução

Modelo em espiral



Como garantir a qualidade do software

10

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

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

Várias técnicas para verificação da qualidade

11

- Revisão por pares da concepção (*design*), texto 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

12

- 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

13

- 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 exacto 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?

14

Comecemos por um exemplo...

Crie um caso de teste

15

- Criar um caso de teste é semelhante a criar uma nova classe
 - No Eclipse, faça:
File->New->JUnit Test Case
 - Escolha a versão mais recente do JUnit

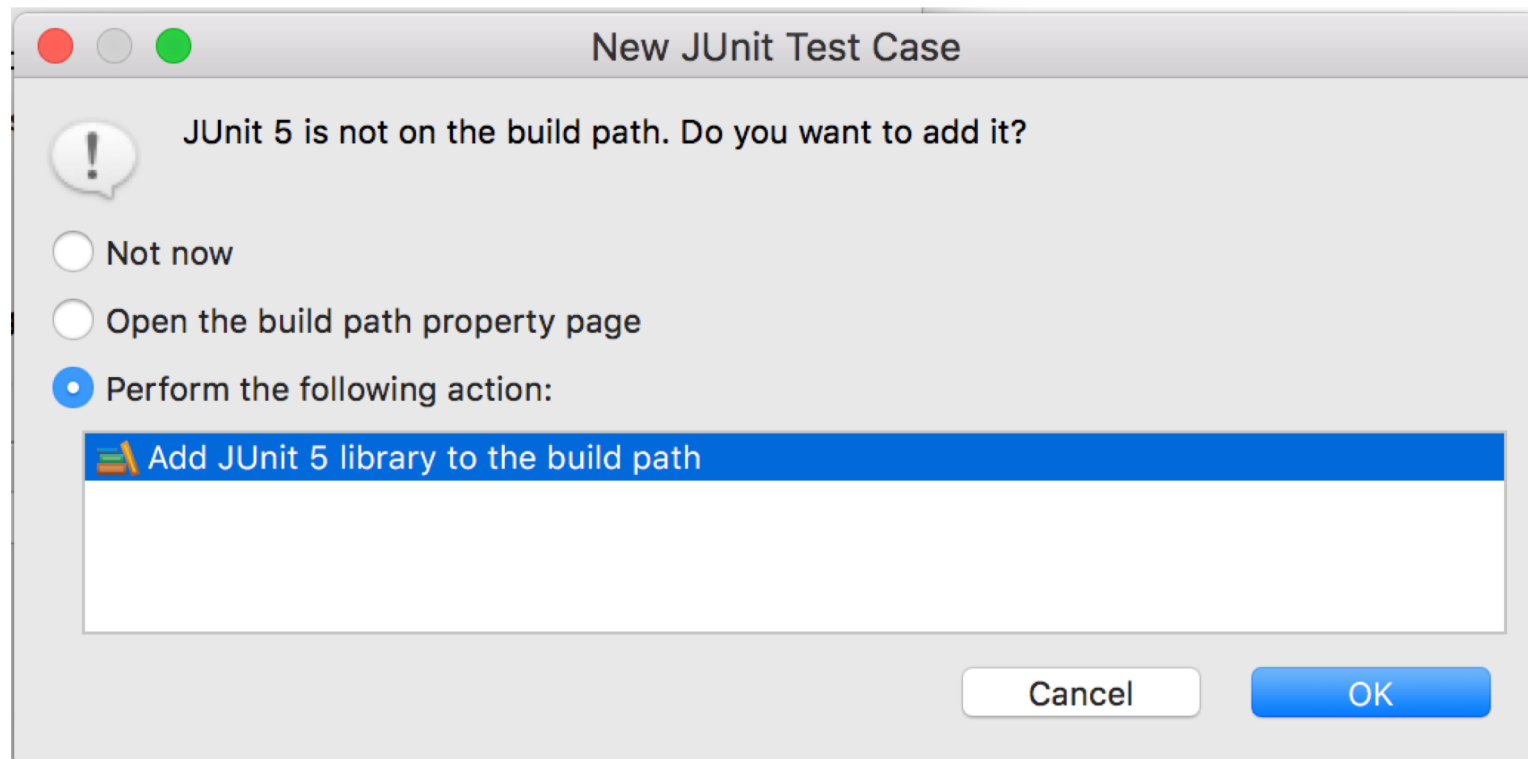
The screenshot shows the 'New JUnit Test Case' dialog box in the Eclipse IDE. The dialog is titled 'New JUnit Test Case' and contains the following fields and options:

- JUnit Test Case**: Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.
- Test Type**: Three radio buttons are present: 'New JUnit 3 test' (unselected), 'New JUnit 4 test' (unselected), and 'New JUnit Jupiter test' (selected).
- Source folder**: A text field containing '___JUnitT2/src' with a 'Browse...' button to its right.
- Package**: A text field containing 'tests' with a 'Browse...' button to its right.
- Name**: A text field containing 'ArrayTester'.
- Superclass**: A text field containing 'java.lang.Object' with a 'Browse...' button to its right.
- Which method stubs would you like to create?**: A group of checkboxes including 'setUpBeforeClass()', 'tearDownAfterClass()', 'setUp()', 'tearDown()', and 'constructor' (which is disabled).
- Do you want to add comments?**: A checkbox labeled 'Generate comments' with a link to 'Configure templates and default value here'.
- Class under test**: A text field containing 'tests.ArrayClassTester' with a 'Browse...' button to its right.
- Navigation**: At the bottom, there is a help icon (?), a '< Back' button, a 'Next >' button, a 'Cancel' button, and a 'Finish' button.

Adicione o JUnit ao seu ambiente

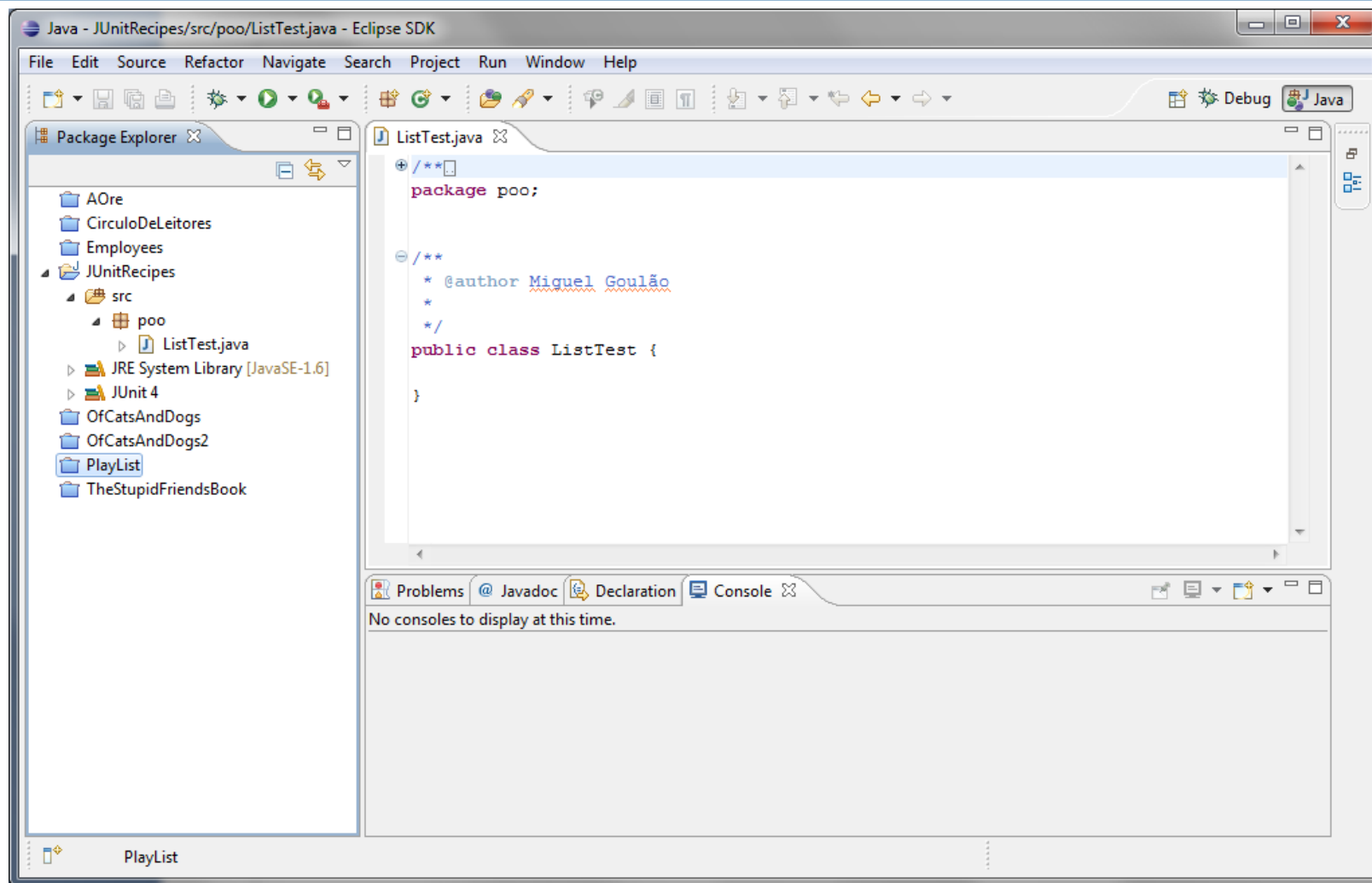
16

- Para poder usar o JUnit, terá de o incluir no caminho para a compilação do seu projecto



Vista do projecto

17



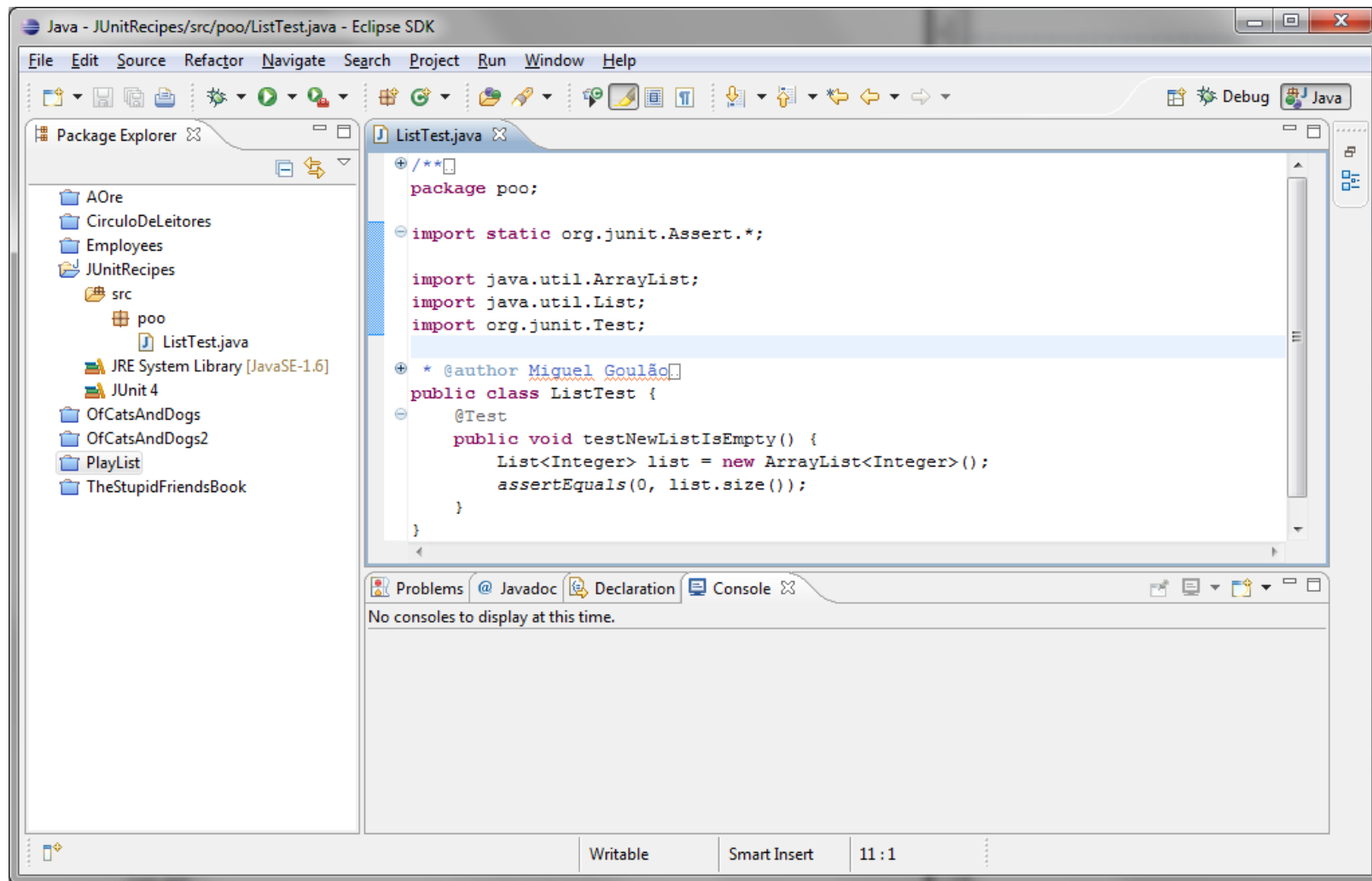
Vamos criar o nosso primeiro teste

18

- Cenário:
 - Criar uma lista nova
 - Testar se a lista, depois de criada, tem exactamente 0 elementos, ou seja, se está vazia
 - O teste é feito à custa de uma **asserção** do JUnit
- Nestes primeiros testes (e só nestes), vamos usar como cobaias classes e interfaces que sabemos funcionar bem, da Java Collections Framework
 - Na prática, é supérfluo testar bibliotecas estáveis

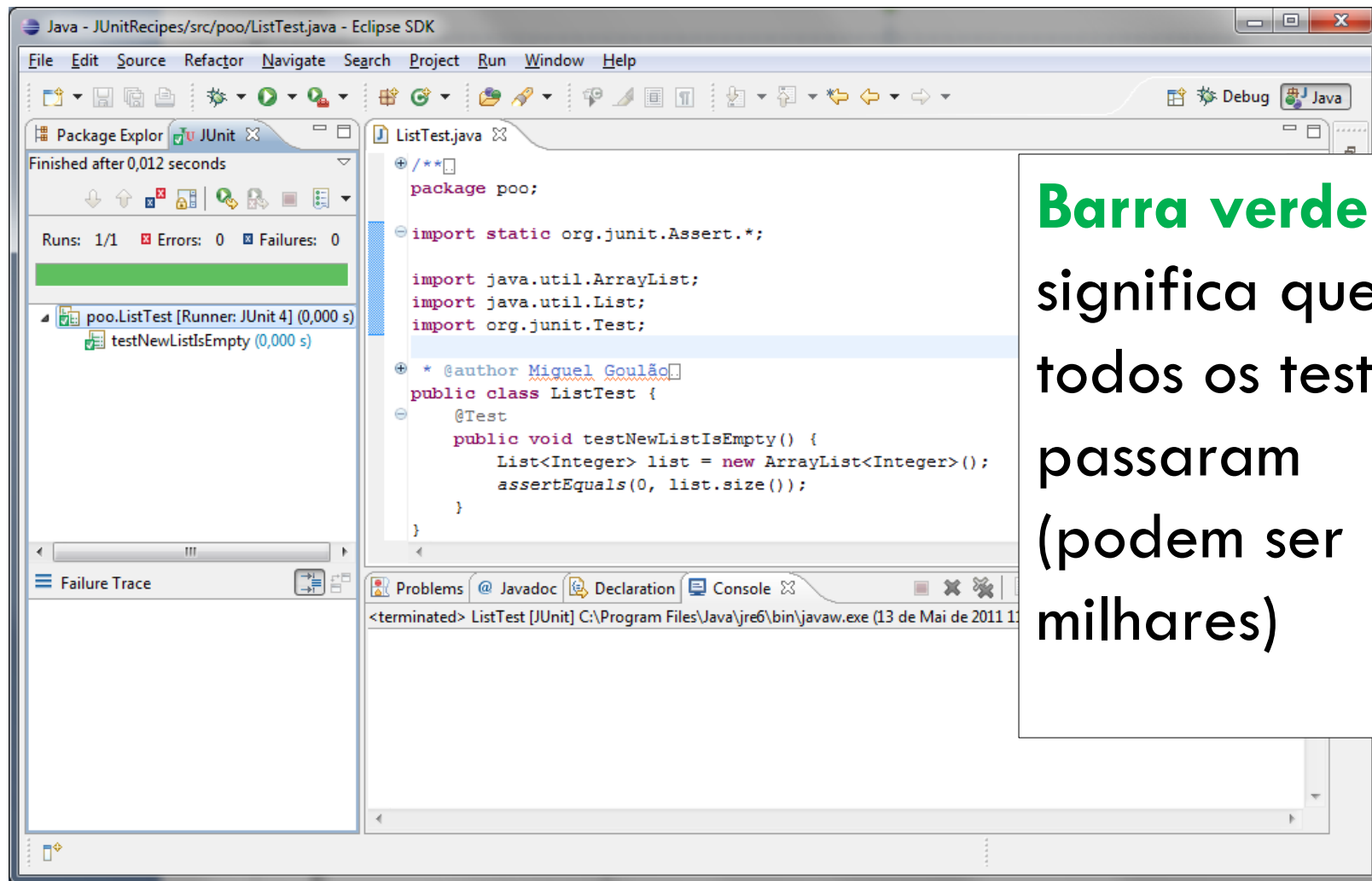
Quando criamos uma lista nova, ela está vazia

19



Corra o teste, para verificar que a lista que acabou de criar está mesmo vazia

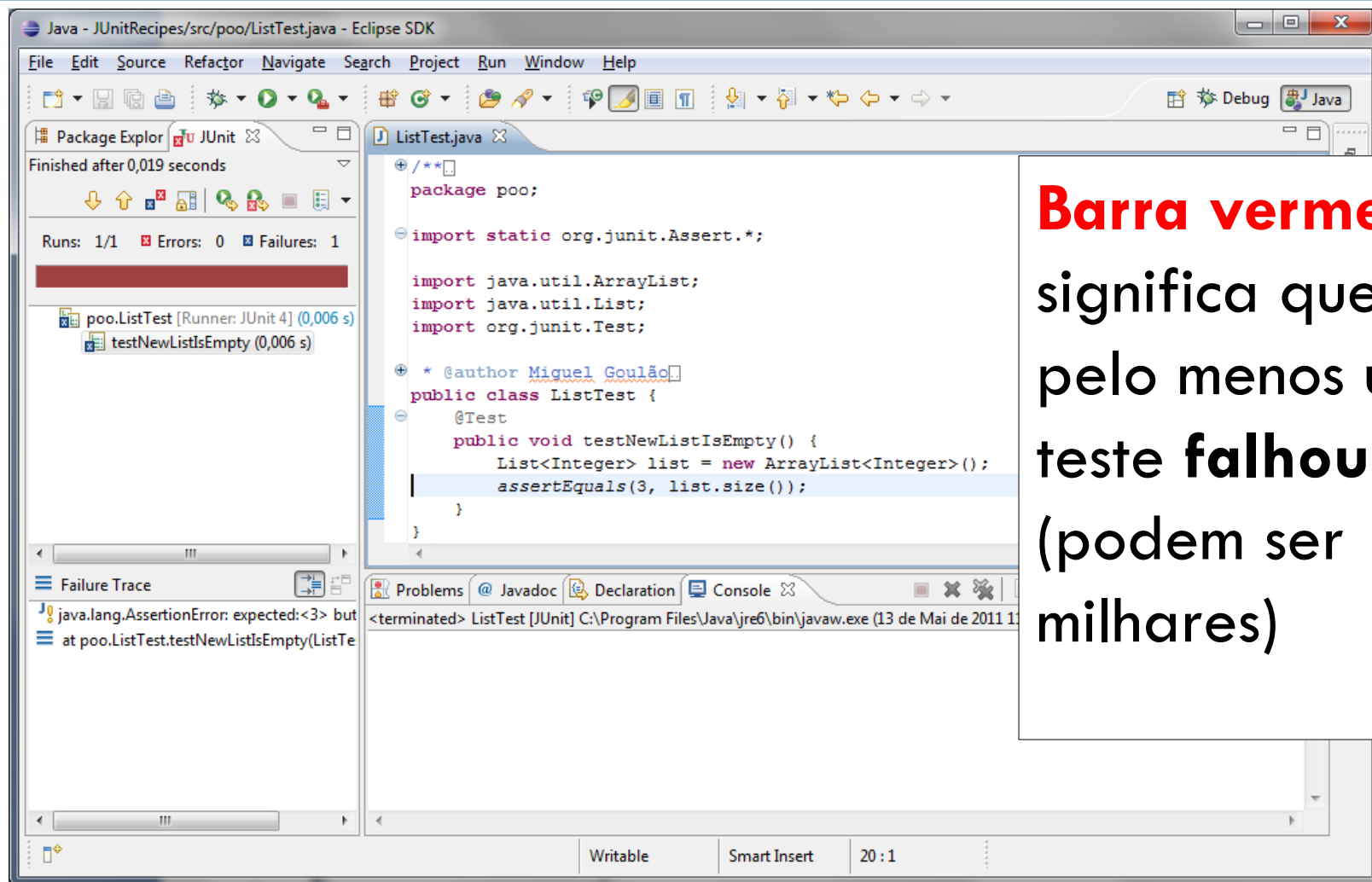
20



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

Experimente “estragar o teste”, admitindo que a nova lista teria 3 elementos

21



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

○ método `assertEquals()`

22

- ○ 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`)
 - Objectos (de quaisquer tipos)
- Se forem iguais, o teste passa
- Se forem diferentes, o teste não passa

Outro teste...

23

```
@Test
public void testListsContents() {
    List<String> list = new LinkedList<String>();
    List<String> other = new ArrayList<String>();
    list.add("Esta aula");
    list.add("está a ser");
    list.add("absolutamente fascinante!");

    for (String s: list)
        other.add(s);

    for (int i = 0; i < list.size(); i++)
        assertEquals(list.get(i), other.get(i));
}
```

Mesmo teste, 2ª versão, ainda mais curta

24

```
@Test
public void testListsContents() {
    List<String> list = Arrays.asList(new String[] {
        "Esta aula", "está a ser", "absolutamente fascinante!"});
    List<String> other = new ArrayList<String>();

    for (String s: list)
        other.add(s);

    assertEquals(list, other);
}
```

Construção do método de teste

25

- Criar o objecto 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 objectos a testar!

Quando especificar os testes?

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

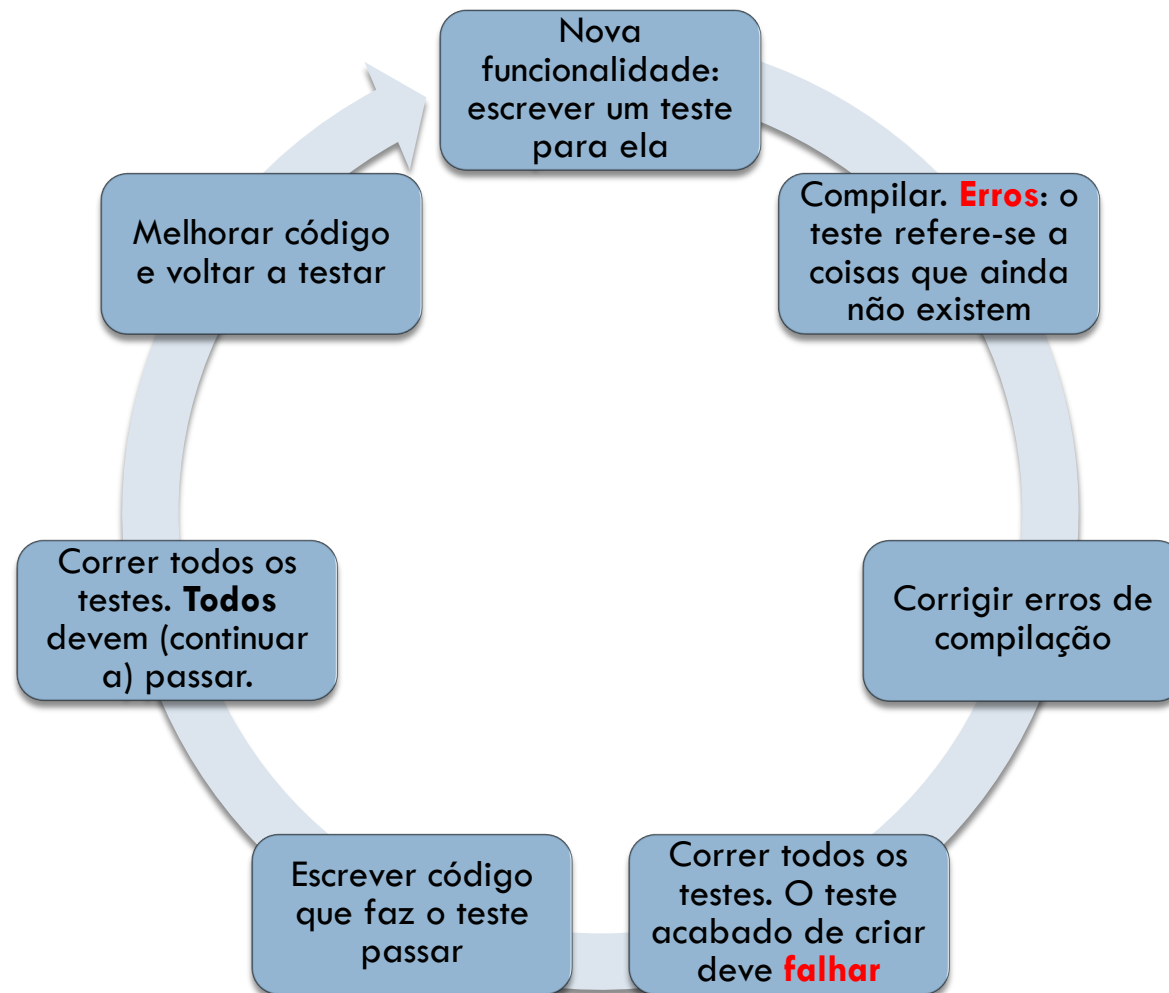
27

Desenvolvimento dirigido pelos testes



Desenvolvimento dirigido pelos testes

28



Desenvolvimento dirigido pelos testes

29

- 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

30

- 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 objecto 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

PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Test-Driven Development (exemplo)

A aplicação PlayList

32

- Crie uma aplicação PlayList que permita gerir as suas músicas favoritas
 - Deve poder:
 - Adicionar músicas
 - Remover músicas
 - Classificar músicas
 - Pesquisar músicas
 - Listar músicas
 - Obter informações sobre as músicas



Comecemos pela interface Music

33

```
public interface Music {  
    // Constantes  
    static final int MINIMUM_RATING = 1;  
    static final int MAXIMUM_RATING = 5;  
  
    // Metodos  
    String getName();  
    String getArtist();  
    int getDuration();  
    FileFormat getFormat();  
    int getRating();  
    Music convert(FileFormat fileFormat);  
}
```

Esta é a interface completa. Porém, é de notar que usando TDD as interfaces podem ser construídas por incrementos, tal como as classes.

Enumeração FileFormat

34

```
public enum FileFormat {  
    MP3, WAV, AIFF  
}
```

Implementação de MusicClass (esqueleto)

35

```
public class MusicClass implements Music {

    public MusicClass(String name) { }

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getArtist() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public int getDuration() {
        // TODO Auto-generated method stub
        return 0;
    }

    ...
}
```

Implementação de MusicClass (esqueleto)

36

```
...
@Override
public FileFormat getFormat() {
    // TODO Auto-generated method stub
    return null;
}
@Override
public int getRating() {
    // TODO Auto-generated method stub
    return 0;
}
@Override
public Music convert(FileFormat fileFormat) {
    // TODO Auto-generated method stub
    return null;
}
}
```

Normalmente...

37

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

Mas hoje as regras são outras

38

- 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**

Ooops, vários erros de compilação

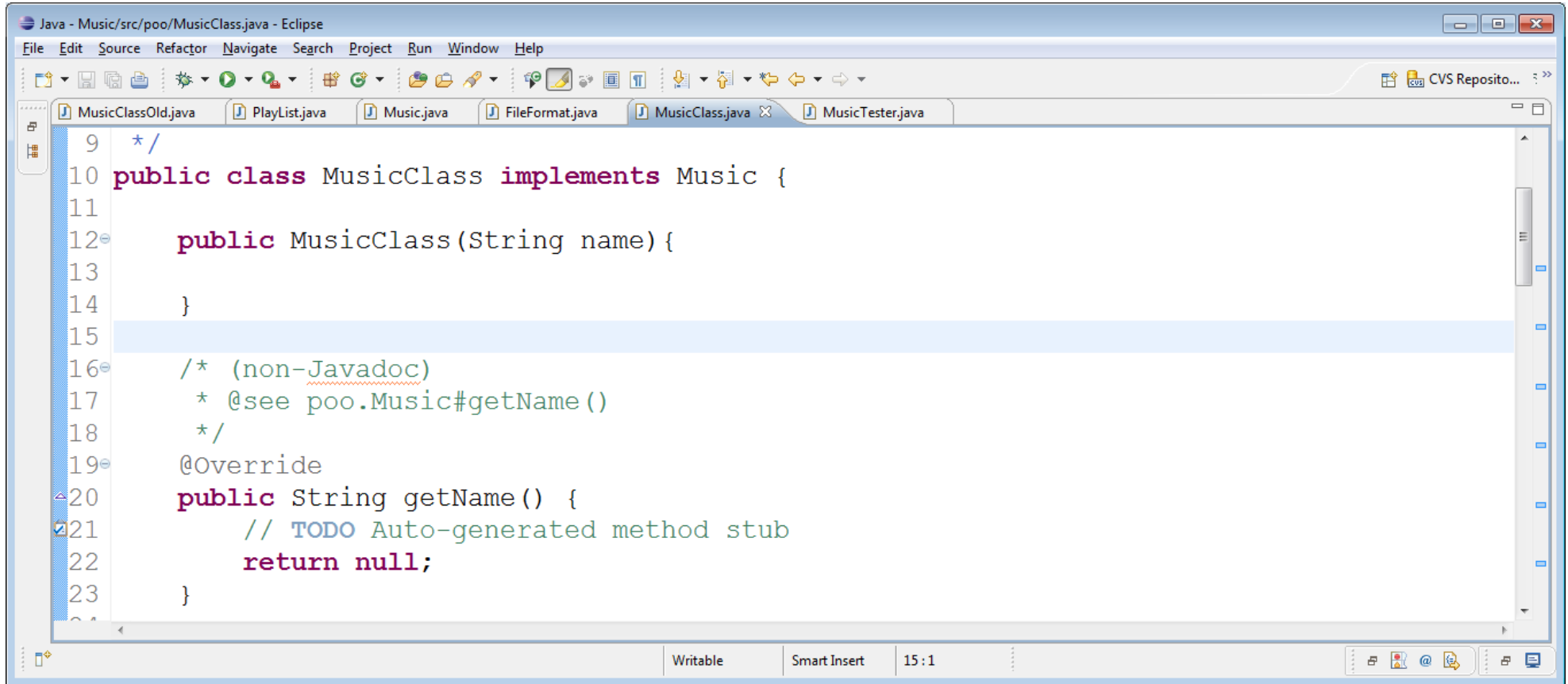
39

```
1 /**  
4 package poo;  
5  
6  
7 /**  
8  * @author Miguel Goulão  
9  *  
10 */  
11 public class MusicTester {  
12     @Test  
13     public void testSimpleConstructor() {  
14         assertEquals("Not on the test", new MusicClass("Not on the test").getName());  
15     }  
16 }
```

○ Perfeito, era exactamente isso que esperávamos!

Na classe MusicClass, falta o construtor

40

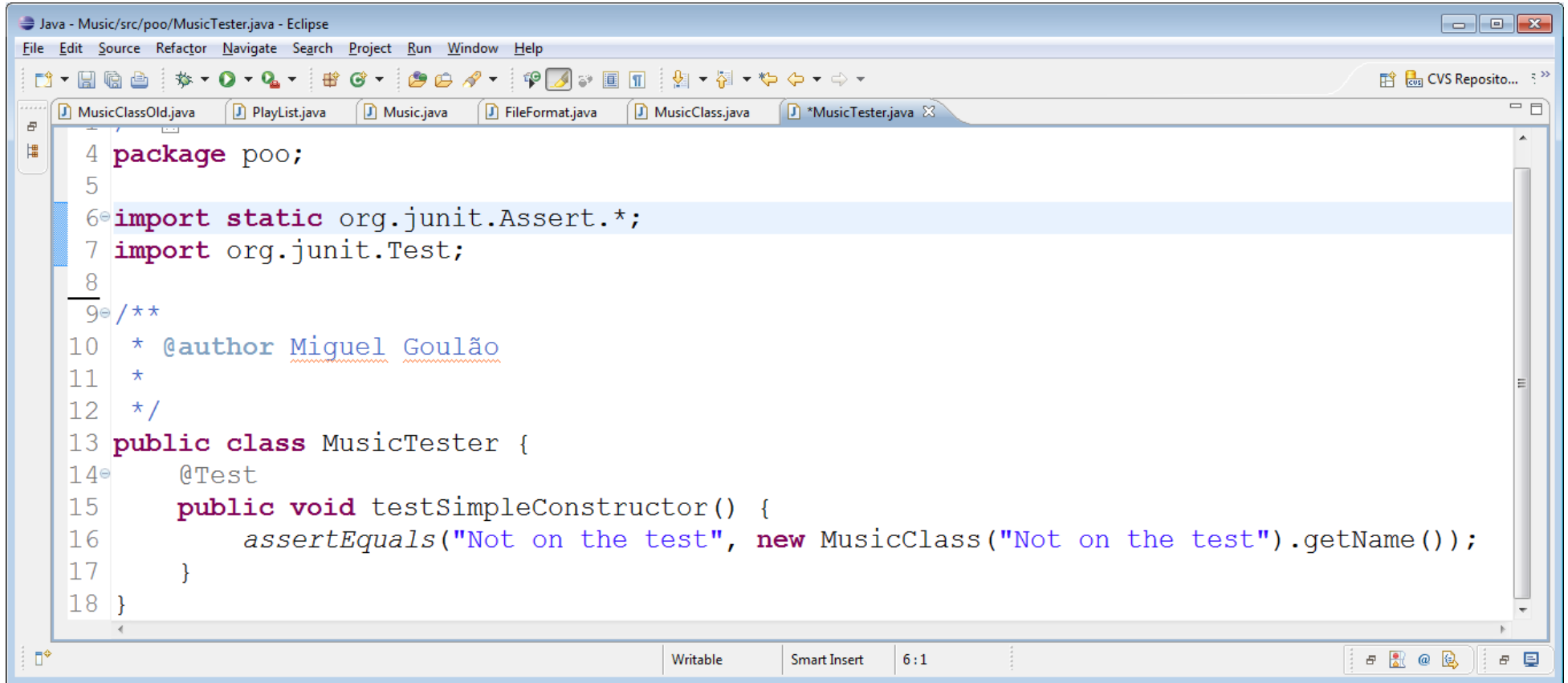


```
9  */
10 public class MusicClass implements Music {
11
12     public MusicClass(String name){
13
14     }
15
16     /* (non-Javadoc)
17      * @see poo.Music#getName()
18      */
19     @Override
20     public String getName() {
21         // TODO Auto-generated method stub
22         return null;
23     }
24 }
```

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

Na classe de teste, faltavam alguns imports

41

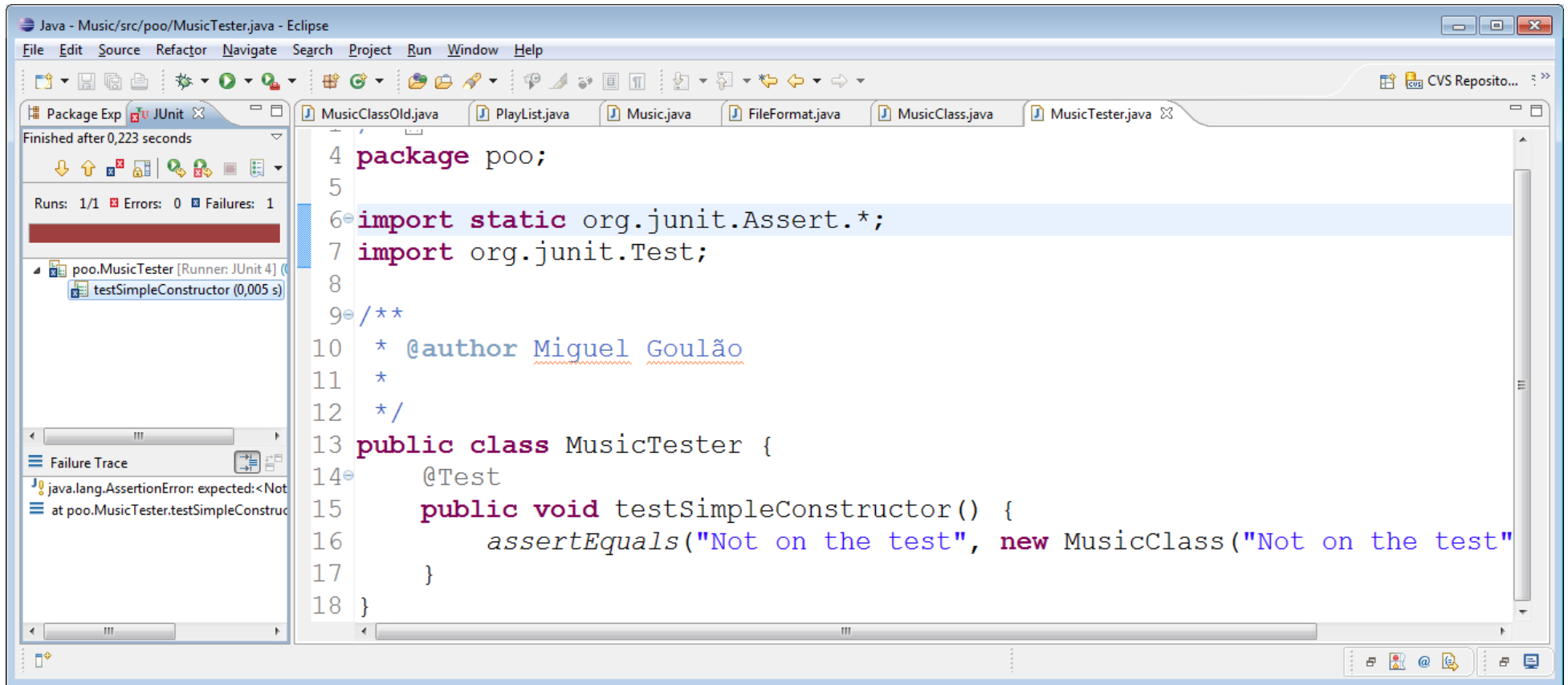


```
4 package poo;
5
6 import static org.junit.Assert.*;
7 import org.junit.Test;
8
9 /**
10  * @author Miguel Goulão
11  *
12  */
13 public class MusicTester {
14     @Test
15     public void testSimpleConstructor() {
16         assertEquals("Not on the test", new MusicClass("Not on the test").getName());
17     }
18 }
```

○ Sucesso! Toca a testar!

Afinal havia outra...

42



The screenshot shows the Eclipse IDE interface. The main editor displays the source code of `MusicTester.java` in the `poo` package. The code includes imports for `org.junit.Assert` and `org.junit.Test`, a Javadoc comment for the author Miguel Goulão, and a `@Test` annotated method `testSimpleConstructor()` that asserts the output of `new MusicClass("Not on the test")` is equal to "Not on the test".

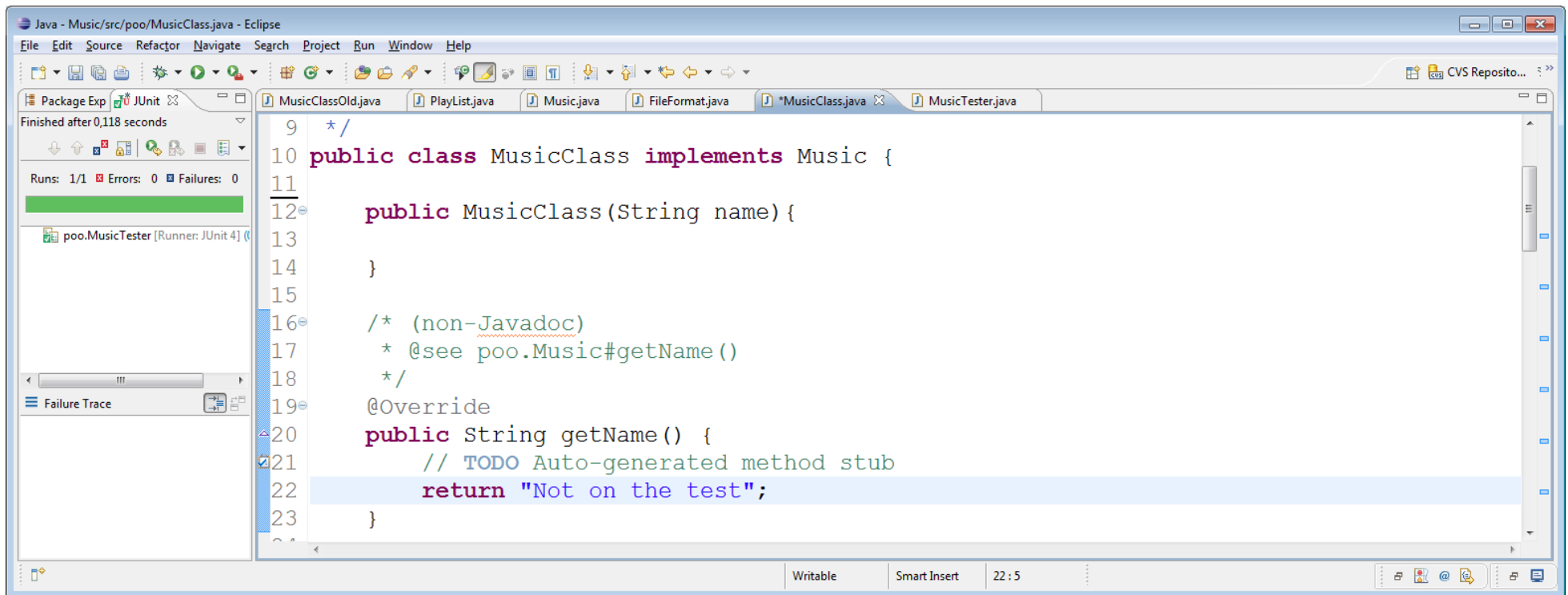
```
4 package poo;
5
6 import static org.junit.Assert.*;
7 import org.junit.Test;
8
9 /**
10  * @author Miguel Goulão
11  *
12  */
13 public class MusicTester {
14     @Test
15     public void testSimpleConstructor() {
16         assertEquals("Not on the test", new MusicClass("Not on the test")
17     }
18 }
```

On the left, the Package Explorer shows the project structure with `poo.MusicTester` selected. The JUnit view indicates that the test `testSimpleConstructor` failed after 0.005 seconds. The Failure Trace at the bottom shows the error: `java.lang.AssertionError: expected:<Not on the test> but was:<[null]>`.

- ... coisa a corrigir. Claro! O construtor ainda não faz nada!
- O selector também não...

Qual o “código mais simples que pode funcionar”?

43

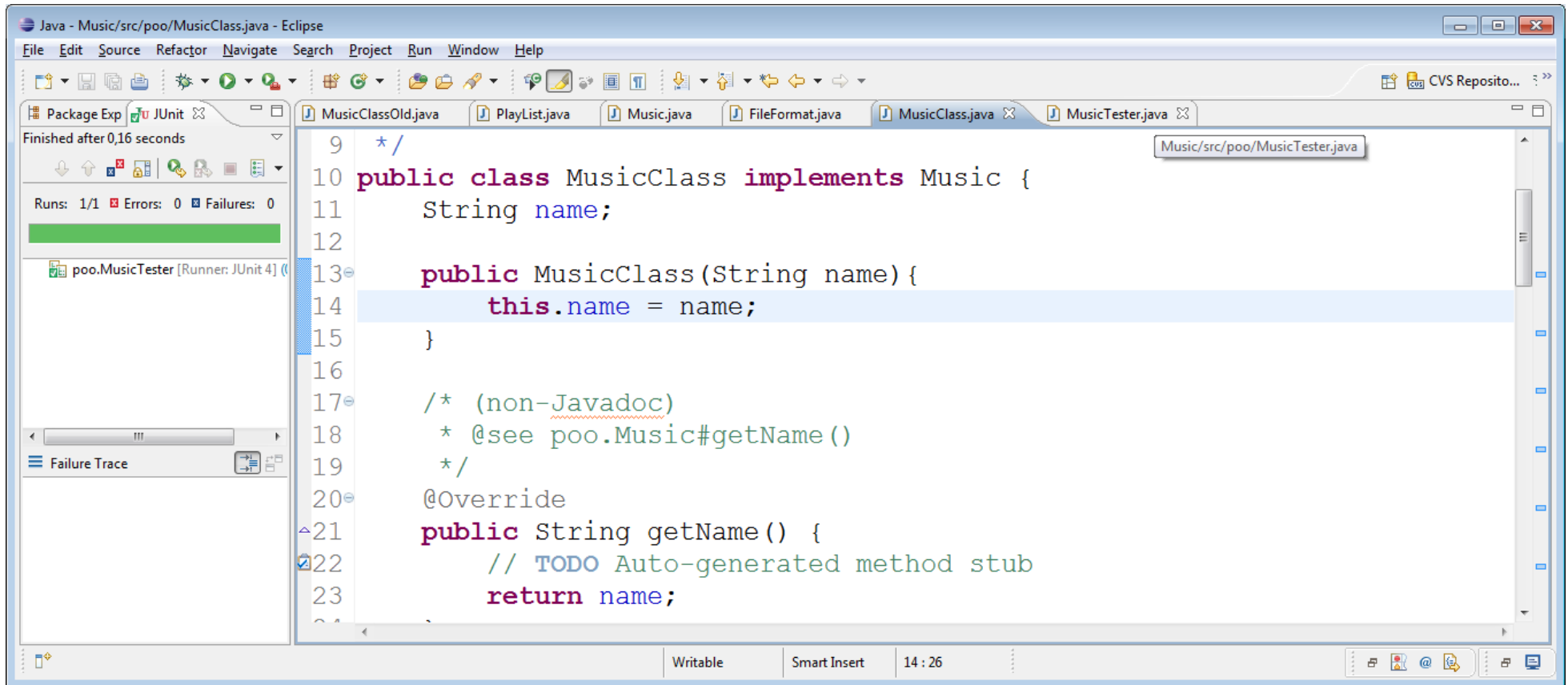


```
9  */
10 public class MusicClass implements Music {
11
12     public MusicClass(String name){
13
14     }
15
16     /* (non-Javadoc)
17      * @see poo.Music#getName()
18      */
19     @Override
20     public String getName() {
21         // TODO Auto-generated method stub
22         return "Not on the test";
23     }
24 }
```

- Parece uma “aldrabice”, mas o facto é que a funcionalidade mais geral ainda não está coberta por testes
- O facto é que este código passa o teste...

Corrigimos apenas o estritamente necessário

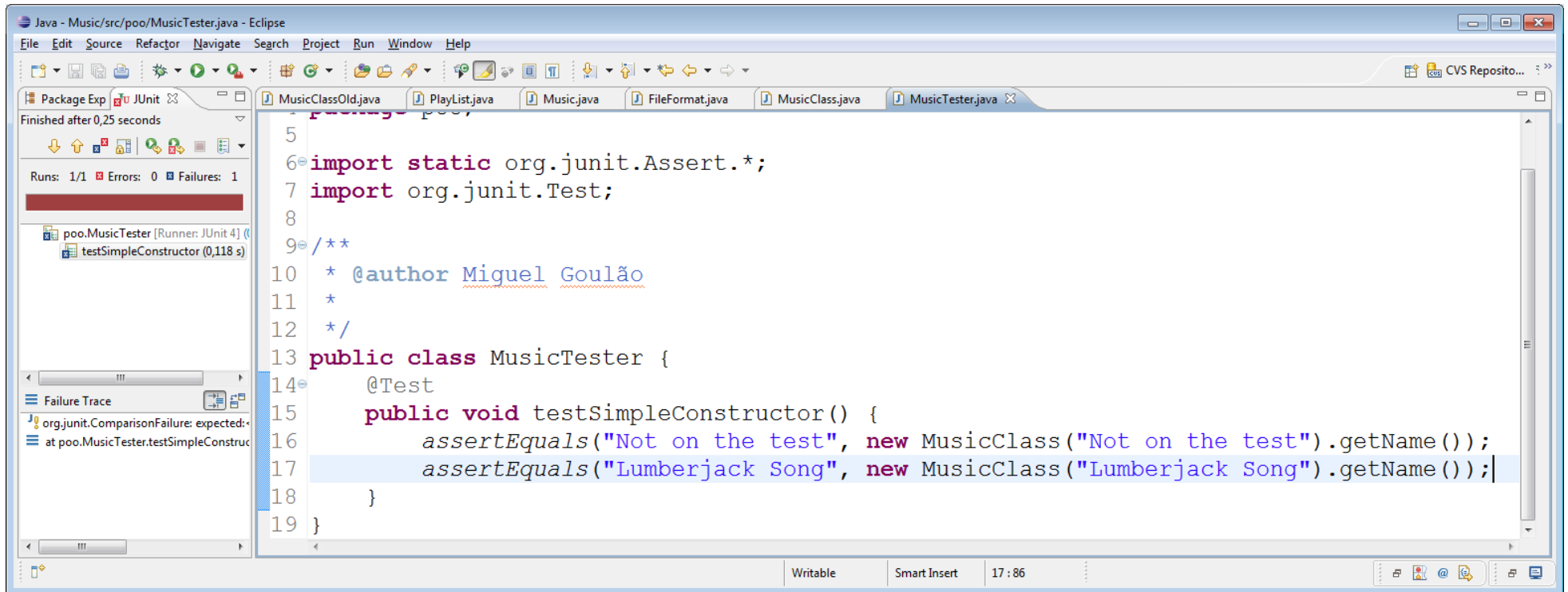
44



```
9  */
10 public class MusicClass implements Music {
11     String name;
12
13     public MusicClass(String name) {
14         this.name = name;
15     }
16
17     /* (non-Javadoc)
18      * @see poo.Music#getName()
19      */
20     @Override
21     public String getName() {
22         // TODO Auto-generated method stub
23         return name;
24     }
25 }
```

Transformemos o teste para ser mais geral

45



```
5
6 import static org.junit.Assert.*;
7 import org.junit.Test;
8
9 /**
10  * @author Miguel Goulão
11  *
12  */
13 public class MusicTester {
14     @Test
15     public void testSimpleConstructor() {
16         assertEquals("Not on the test", new MusicClass("Not on the test").getName());
17         assertEquals("Lumberjack Song", new MusicClass("Lumberjack Song").getName());
18     }
19 }
```

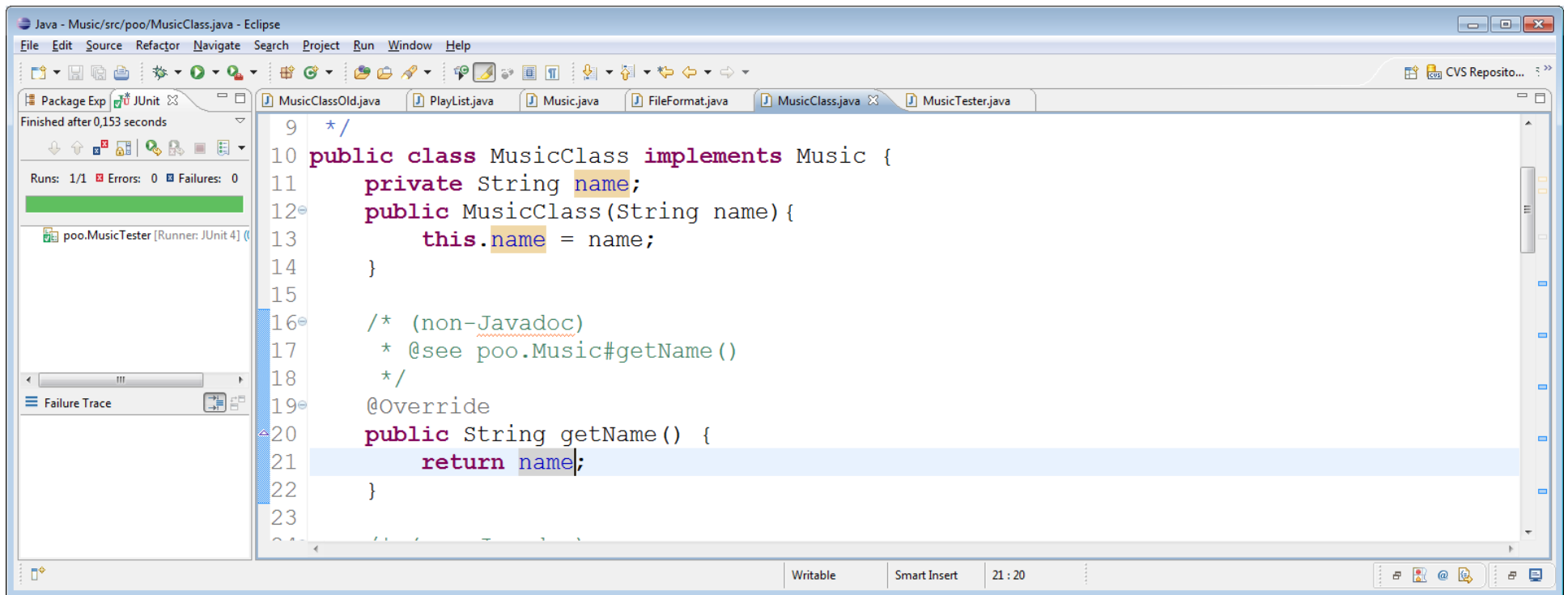
Failure Trace

- org.junit.ComparisonFailure: expected: ...
- at poo.MusicTester.testSimpleConstruc...

- Claro que o remendo de há pouco já não funciona
- Só podemos avançar para o próximo teste depois de correr com sucesso este

Vamos corrigir o código

46



```
9  */
10 public class MusicClass implements Music {
11     private String name;
12     public MusicClass(String name) {
13         this.name = name;
14     }
15
16     /* (non-Javadoc)
17      * @see poo.Music#getName()
18      */
19     @Override
20     public String getName() {
21         return name;
22     }
23 }
```

- Sucesso, podemos avançar para um construtor mais completo

Primeiro, construímos o teste

47

```
@Test
public void testFullConstructor() {
    Music m1 = new MusicClass("Ser Benfiquista", "Luis Picarra",
                               FileFormat.AIFF, 184, 2);
    assertEquals("Ser Benfiquista", m1.getName());
    assertEquals("Luis Picarra", m1.getArtist());
    assertEquals(184, m1.getDuration());
    assertEquals(FileFormat.AIFF, m1.getFormat());
    assertEquals(2, m1.getRating());
}
```

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

48

```
public class MusicClass implements Music {
    private String name;
    public MusicClass(String name){ this.name = name; }
    public MusicClass(String name, String artist, FileFormat format,
                       int duration, int rating) { }

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

    @Override
    public String getArtist() { return "Luis Picarra"; }

    @Override
    public int getDuration() { return 184; }

    @Override
    public FileFormat getFormat() { return FileFormat.AIFF; }

    @Override
    public int getRating() { return 2; }

    @Override
    public Music convert(FileFormat fileFormat) { return null; }
}
```

Temos de criar um teste em que o código não passe

49

```
@Test
public void testFullConstructor() {
    Music m1 = new MusicClass("Ser Benfiquista", "Luis Picarra",
                               FileFormat.AIFF, 184, 2);
    assertEquals("Ser Benfiquista", m1.getName());
    assertEquals("Luis Picarra", m1.getArtist());
    assertEquals(184, m1.getDuration());
    assertEquals(FileFormat.AIFF, m1.getFormat());
    assertEquals(2, m1.getRating());

    Music m2 = new MusicClass("Eu tenho dois amores", "Marco Paulo",
                               FileFormat.MP3, 194, 3);
    assertEquals("Eu tenho dois amores", m2.getName());
    assertEquals("Marco Paulo", m2.getArtist());
    assertEquals(194, m2.getDuration());
    assertEquals(FileFormat.MP3, m2.getFormat());
    assertEquals(3, m2.getRating());
}
```

Agora, alteramos o código, para passar no teste

50

```
public class MusicClass implements Music {
    private String name;
    private String artist;
    private FileFormat format;
    private int duration;
    private int rating;

    public MusicClass(String name){
        this.name = name;
    }

    public MusicClass(String name, String artist, FileFormat format,
                      int duration, int rating) {
        this.name = name;
        this.artist = artist;
        this.format = format;
        this.duration = duration;
        this.rating = rating; ;
    }
    @Override
    public String getName() {
        return name;
    }
}
```

Agora, alteramos o código, para passar no teste

51

```
@Override
public String getArtist() {
    return artist;
}

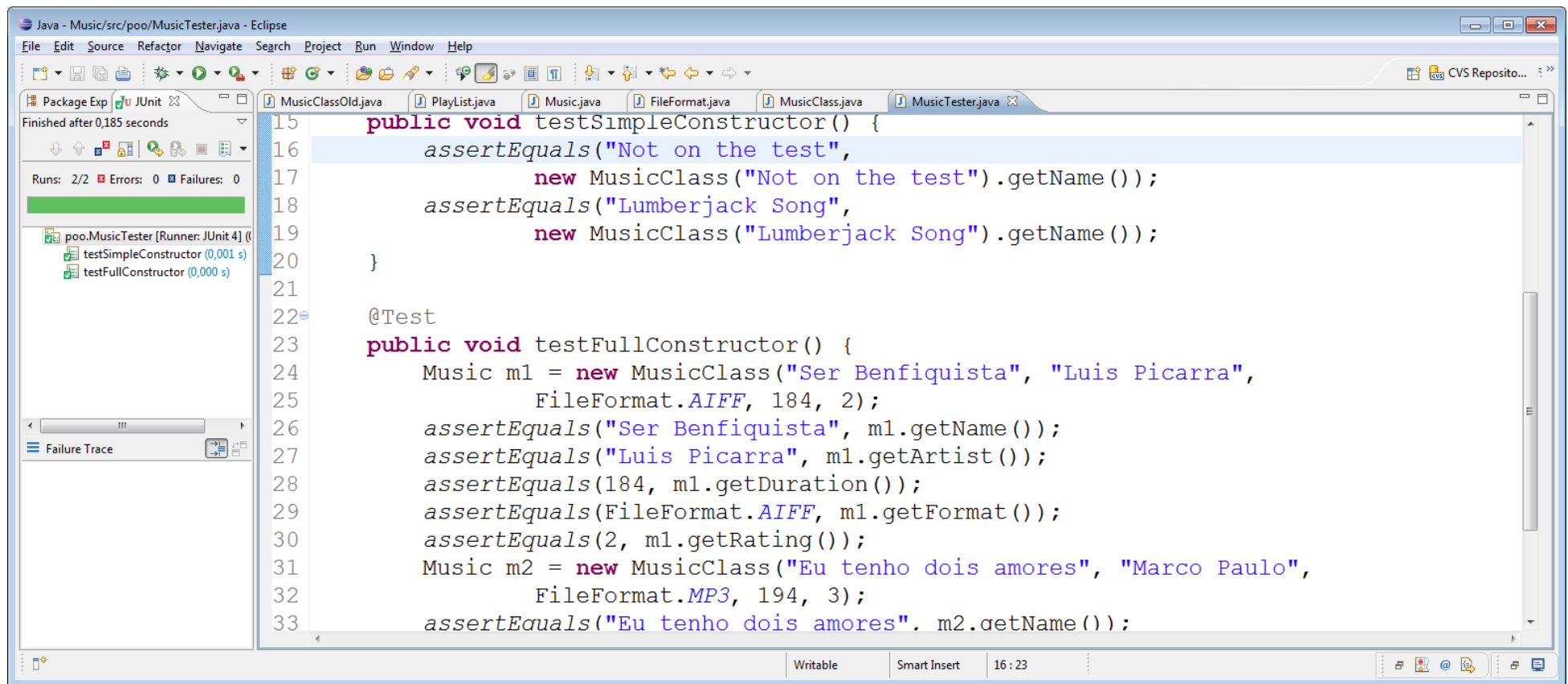
@Override
public int getDuration() {
    return duration;
}

@Override
public FileFormat getFormat() {
    return format;
}

@Override
public int getRating() {
    return rating;
}
...
}
```

Desta vez, ambos os testes passam

52



The screenshot shows the Eclipse IDE interface. The main editor displays the `MusicTester.java` file with the following code:

```
15 public void testSimpleConstructor() {
16     assertEquals("Not on the test",
17         new MusicClass("Not on the test").getName());
18     assertEquals("Lumberjack Song",
19         new MusicClass("Lumberjack Song").getName());
20 }
21
22 @Test
23 public void testFullConstructor() {
24     Music m1 = new MusicClass("Ser Benfiquista", "Luis Picarra",
25         FileFormat.AIFF, 184, 2);
26     assertEquals("Ser Benfiquista", m1.getName());
27     assertEquals("Luis Picarra", m1.getArtist());
28     assertEquals(184, m1.getDuration());
29     assertEquals(FileFormat.AIFF, m1.getFormat());
30     assertEquals(2, m1.getRating());
31     Music m2 = new MusicClass("Eu tenho dois amores", "Marco Paulo",
32         FileFormat.MP3, 194, 3);
33     assertEquals("Eu tenho dois amores", m2.getName());
```

On the left, the 'JUnit' tab shows the test results: 'Finished after 0,185 seconds', 'Runs: 2/2', 'Errors: 0', and 'Failures: 0'. Below this, a tree view shows the test classes: `poo.MusicTester [Runner: JUnit 4]`, `testSimpleConstructor (0,001 s)`, and `testFullConstructor (0,000 s)`. The 'Failure Trace' panel is empty.

○ Está tudo?

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

53

- 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

54

- Actividade 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 detectem esses *bugs* que podemos inadvertidamente introduzir durante as modificações

Algumas regras no desenvolvimento guiado por testes

55

- Só refabricamos código que passa em todos os testes existentes
- Só acrescentamos funcionalidades novas quando há um teste que o código actual 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 refabrição

56

```
public MusicClass(String name){  
    this.name = name;  
}
```

```
public MusicClass(String name, String artist, FileFormat format,  
                  int duration, int rating) {  
    this.name = name;  
    this.artist = artist;  
    this.format = format;  
    this.duration = duration;  
    this.rating = rating;  
}
```

```
public MusicClass(String name, String artist, FileFormat format,  
                  int duration, int rating) {  
    this(name);  
    this.artist = artist;  
    this.format = format;  
    this.duration = duration;  
    this.rating = rating;  
}
```

○ Código repetido, considerado um sinal de má programação

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

57

Testes para métodos equals ()

Propriedades do método `equals()`

58

- Reflexividade
 - Um objecto deve ser igual a si próprio
- Simetria
 - Se o objecto **x** é igual ao objecto **y**, então o objecto **y** é igual ao objecto **x**
- Transitividade
 - Se o objecto **x** é igual ao objecto **y** e o objecto **y** é igual ao objecto **z**, então o objecto **x** é igual ao objecto **z**
- Consistência
 - Para qualquer referência não nula dos objectos **x** e **y**, sucessivas invocações de **x.equals(y)** deverão retornar consistentemente **true**, ou consistentemente **false**, desde que não ocorram modificações em **x** ou **y**
- Não nulidade
 - Para qualquer referência não nula **x**, **x.equals(null)** retorna sempre **false**

Ingredientes para testar o equals()

59

- Um objecto de controlo, com o qual os restantes vão ser comparados
- Um objecto semelhante ao objecto de controlo, para o qual a operação equals() deve retornar true
- Um objecto diferente do objecto de controlo, para o qual a operação equals() deve retornar false
- Um quarto objecto que:
 - Se a classe a testar for final (ou seja, se for impossível criar sub-classes dessa classe), este valor deve ser null
 - Se a classe a testar não for final, este objecto deve ser parecido com o objecto de controlo, mas diferente, por ter propriedades adicionais

Reflexividade

60

```
@Test
public void testIdentity() {
    Music m1 = new MusicClass("Master of Puppets", "Metallica",
                               FileFormat.WAV, 452, 4);

    assertEquals(m1, m1);
}
```

- Um objecto tem de ser igual a si próprio
- Neste caso o teste passou, mesmo antes acrescentarmos o método **equals()** à classe **MusicClass**
 - É natural que passe o teste, quando se usa o **equals()** de **Object...**
 - ... e vai ter de continuar a passar depois de criarmos o método **equals()**

Temos de criar músicas para cada um dos testes?

61

- Sim, para ter o que testar
- Não temos é de o fazer repetidamente
 - Como evitar a repetição?
 - Factorizando o próprio código de testes, naturalmente!

Criação de músicas factorizada

62

```
public class MusicTester {  
    private Music sm1;  
    private Music sm2;  
    private Music m1;  
    private Music m2;  
    private Music m3;  
    private Music m4;
```

Começamos por declarar variáveis de instância para os nossos testes.

A anotação `@BeforeAll` indica que o método `setup()` deve ser **executado antes de cada um dos testes** da classe `MusicTester`.

```
@BeforeAll
```

```
public void setup() {  
    sm1 = new MusicClass("Not on the test");  
    sm2 = new MusicClass("Lumberjack Song");  
    m1 = new MusicClass("Ser Benfiquista", "Luis Picarra",  
                        FileFormat.AIFF, 184, 2);  
    m2 = new MusicClass("Eu tenho dois amores", "Marco Paulo",  
                        FileFormat.MP3, 194, 3);  
    m3 = new MusicClass("Master of Puppets", "Metallica",  
                        FileFormat.WAV, 452, 4);  
    m4 = new MusicClass("Master of Puppets", "Metallica",  
                        FileFormat.WAV, 452, 4);  
}
```

...

Criação de músicas factorizada

63

...

@Test

```
public void testSimpleConstructor() {  
    assertEquals("Not on the test", sm1.getName());  
    assertEquals("Lumberjack Song", sm2.getName());  
}
```

@Test

```
public void testFullConstructor() {  
    assertEquals("Ser Benfiquista", m1.getName());  
    assertEquals("Luis Picarra", m1.getArtist());  
    assertEquals(184, m1.getDuration());  
    assertEquals(FileFormat.AIFF, m1.getFormat());  
    assertEquals(2, m1.getRating());  
    assertEquals("Eu tenho dois amores", m2.getName());  
    assertEquals("Marco Paulo", m2.getArtist());  
    assertEquals(194, m2.getDuration());  
    assertEquals(FileFormat.MP3, m2.getFormat());  
    assertEquals(3, m2.getRating());  
}
```

...

Criação de músicas factorizada

64

```
@Test
public void testReflexive() {
    assertEquals(sm1, sm1);
    assertEquals(sm2, sm2);
    assertEquals(m1, m1);
    assertEquals(m2, m2);
    assertEquals(m3, m3);
}
```

- Depois desta refabricação da classe de testes, continuamos com a barra verde
 - Podemos, portanto, continuar a testar o `equals()`

Simetria

65

```
public void testSimetric() {  
    assertEquals(m3, m4);  
    assertEquals(m4, m3);  
}
```

- Este teste falha, com o `equals()` de `Object`
 - Temos, portanto, de voltar à classe `MusicClass` para corrigir o problema

Primeira tentativa

66

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    Music other = (Music) obj;  
    if (!artist.equals(other.getArtist()))  
        return false;  
    if (duration != other.getDuration())  
        return false;  
    if (format != other.getFormat())  
        return false;  
    if (!name.equals(other.getName()))  
        return false;  
    if (rating != other.getRating())  
        return false;  
    return true;  
}
```

- **other** pode ser **null**
- **other** pode não ser do tipo **Music**
- **name** e **artist**, de **this** ou de **other**, podem ser **null**
- Vamos testar um problema de cada vez, claro

Uma das músicas pode ser null

67

```
@Test
```

```
public void testNotNullMusic() {  
    assertFalse(sm1.equals(null));  
    assertFalse(sm2.equals(null));  
    assertFalse(m1.equals(null));  
    assertFalse(m2.equals(null));  
    assertFalse(m3.equals(null));  
    assertFalse(m4.equals(null));  
}
```

○ Estes testes falham todos!

○ Vários NullPointerExceptions

```
public boolean equals(Object obj) {  
    @Override  
    if (this == obj)  
        return true;  
    Music other = (Music) obj;  
    if (!artist.equals(other.getArtist()))  
        return false;  
    if (duration != other.getDuration())  
        return false;  
    if (format != other.getFormat())  
        return false;  
    if (!name.equals(other.getName()))  
        return false;  
    if (rating != other.getRating())  
        return false;  
    return true;  
}
```

Faltava um teste em equals()

68

```
@Test
public void testNotNullMusic() {
    assertFalse(sm1.equals(null));
    assertFalse(sm2.equals(null));
    assertFalse(m1.equals(null));
    assertFalse(m2.equals(null));
    assertFalse(m3.equals(null));
    assertFalse(m4.equals(null));
}
```

○ Estes testes já passam todos!

- Pelo menos, até alguém se lembrar de passar algo que não uma música como argumento para o equals()

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    Music other = (Music) obj;
    if (!artist.equals(other.getArtist()))
        return false;
    if (duration != other.getDuration())
        return false;
    if (format != other.getFormat())
        return false;
    if (!name.equals(other.getName()))
        return false;
    if (rating != other.getRating())
        return false;
    return true;
}
```

Tudo ok?

69

```
@Test
public void testNotMusic() {
    String s = "Music";
    assertFalse(sm1.equals(s));
    assertFalse(sm2.equals(s));
    assertFalse(m1.equals(s));
    assertFalse(m2.equals(s));
    assertFalse(m3.equals(s));
    assertFalse(m4.equals(s));
}
```

○ Estes testes falham todos!

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    Music other = (Music) obj;
    if (!artist.equals(other.getArtist()))
        return false;
    if (duration != other.getDuration())
        return false;
    if (format != other.getFormat())
        return false;
    if (!name.equals(other.getName()))
        return false;
    if (rating != other.getRating())
        return false;
    return true;
}
```

Faltava outro teste em equals()

70

```
@Test
public void testNotMusic() {
    String s = "Music";
    assertFalse(sm1.equals(s));
    assertFalse(sm2.equals(s));
    assertFalse(m1.equals(s));
    assertFalse(m2.equals(s));
    assertFalse(m3.equals(s));
    assertFalse(m4.equals(s));
}
```

- Estes testes passam todos!
 - Pelo menos até alguém se lembrar de criar uma música com o nome **null**
 - Ou um artista com o nome **null**

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (!(obj instanceof Music))
        return false;
    Music other = (Music) obj;
    if (!artist.equals(other.getArtist()))
        return false;
    if (duration != other.getDuration())
        return false;
    if (format != other.getFormat())
        return false;
    if (!name.equals(other.getName()))
        return false;
    if (rating != other.getRating())
        return false;
    return true;
}
```

Vamos ver se é desta

71

```
@Test
public void testNullArgs() {
    Music m = new MusicClass(null, null,
                             FileFormat.AIFF, 32, 2);
    Music n = new MusicClass(null, null,
                             FileFormat.AIFF, 32, 2);
    assertEquals(m,n);
}
```

○ Mais alguma coisa?

- Assim de repente, não!
- Mas sempre foi uma boa oportunidade de recordar o que devemos fazer para implementar um método equals correctamente

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (!(obj instanceof Music)) return false;
    Music other = (Music) obj;
    if (artist == null) {
        if (other.getArtist() != null)
            return false;
    } else if (!artist.equals(other.getArtist()))
        return false;
    if (duration != other.getDuration())
        return false;
    if (format != other.getFormat())
        return false;
    if (name == null) {
        if (other.getName() != null)
            return false;
    } else if (!name.equals(other.getName()))
        return false;
    if (rating != other.getRating())
        return false;
    return true;
}
```

Finalmente, temos a nossa classe `MusicClass` pronta

72

- Os construtores foram bem testados
- O `equals` está à prova de bala
- Para testar tudo isto, os gets tiveram de funcionar
- Está tudo testado?

73

Implementação e teste de PlayList

A interface PlayList

74

```
public interface PlayList {  
    String getName();  
    int getSize();  
    boolean contains(Music m);  
    void addMusic(Music m);  
    void deleteMusic(Music m);  
    int numMusics(String artist);  
    int numArtists();  
    int getDuration();  
    String mostProductiveArtist();  
    Iterator<Music> getIterator();  
}
```

Esqueleto de PlayListClass

75

```
public class PlayListClass implements PlayList {  
    public PlayListClass(String name) {}  
    @Override  
    public String getName() {return "";}  
    @Override  
    public int getSize() {return 0;}  
    @Override  
    public boolean contains(Music m) {return false;}  
    @Override  
    public void addMusic(Music m) {}  
    @Override  
    public void deleteMusic(Music m) {}  
    @Override  
    public int numMusics(String artist) {return 0;}  
    @Override  
    public int numArtists() {return 0;}  
    @Override  
    public int getDuration() {return 0;}  
    @Override  
    public String mostProductiveArtist() {return "";}  
    @Override  
    public Iterator<Music> getIterator() { return null; }  
}
```

- Esqueleto mínimo que satisfaz a interface
- Código compila, mas não faz nada interessante

PlayListClass, usando TDD

76

- Começamos por criar algum suporte aos testes, que vamos usar em todos os exemplos desta classe

```
public class PlayListTester {
    private Music sm1, sm2, m1, m2, m3, m4;
    private PlayList play, play2;

    @BeforeAll
    public void setUp() {
        sm1 = new MusicClass("Not on the test");
        sm2 = new MusicClass("Lumberjack Song");
        m1 = new MusicClass("Ser Benfiquista", "Luis Picarra", FileFormat.AIFF, 184, 2);
        m2 = new MusicClass("Eu tenho dois amores", "Marco Paulo", FileFormat.MP3,
                            194, 3);
        m3 = new MusicClass("Master of Puppets", "Metallica", FileFormat.WAV, 452, 4);
        m4 = new MusicClass("Master of Puppets", "Metallica", FileFormat.WAV, 452, 4);

        play = new PlayListClass("Cancoes da minha vida");
        play2 = new PlayListClass("Mais cancoes da minha vida");
    }
    ...
}
```

PlaylistClass, usando TDD

77

○ De seguida, testamos o construtor

```
@Test
public void testConstructor() {
    assertEquals(0, play.getSize());
    assertEquals(0, play.getDuration());
    assertEquals("Cancoes da minha vida", play.getName());
    assertEquals(0, play2.getSize());
    assertEquals(0, play2.getDuration());
    assertEquals("Mais cancoes da minha vida", play2.getName());
}
```

○ Claro que **falha**, temos de começar a preencher o esqueleto da classe PlaylistCalculator

Construímos o suficiente para passar no teste

78

```
public class PlayListClass implements PlayList {  
    private String name;  
  
    public PlayListClass(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public String getName() {  
        return this.name;  
    }  
    ...  
}
```

- Já passa no teste, podemos continuar, experimentando inserir e remover músicas

Testar métodos que não retornam nada

```
public interface PlayList {  
    String getName();  
    int getSize();  
    boolean contains(Music m);  
    void addMusic(Music m);  
    void deleteMusic(Music m);  
    int numMusics(String artist);  
    int numArtists();  
    int getDuration();  
    String mostProductiveArtist();  
    Iterator<Music> getIterator();  
}
```

○ que tem um método que não retorna nada de tão especial?

80

- Se um método não retorna nada, deve ter algum efeito secundário que seja indirectamente observável
 - Se não tiver nenhum efeito observável, será que o método é mesmo necessário?
 - Não altera o estado do objecto
 - Não permite consultar o estado do objecto
 - Talvez seja de considerar a hipótese de remover o método que aparentemente não faz nada da sua classe
 - Comece por apenas comentar o método, por via das dúvidas...
 - Se tiver um efeito observável, ainda que indirectamente, devemos construir o nosso teste de modo a observar esse efeito

Testar o método `addMusic()`

81

- Criar uma `PlayList` vazia
- A `PlayList` não contém a música, a princípio
- Acrescentamos a música
- Agora, a `PlayList` já tem a música
- Para ter confiança de que a lista funciona bem, o melhor é
 - Criar mais algumas músicas
 - Tentar acrescentar essas músicas à `PlayList`
 - Confirmar que as músicas acrescentadas passam, de facto, a pertencer à `PlayList`, enquanto que outras músicas se mantêm fora da `PlayList`

Acrescentamos um novo teste

82

```
@Test
public void testAddMusic() {
    assertFalse(play.contains(m1));
    assertFalse(play.contains(m2));
    assertFalse(play.contains(m3));
    play.addMusic(m1);
    assertTrue(play.contains(m1));
    assertFalse(play.contains(m2));
    play.addMusic(m2);
    assertTrue(play.contains(m1));
    assertTrue(play.contains(m2));
}
```

- O teste **falha**, como seria de esperar

Acrescentar músicas

83

```
public class PlayListClass implements PlayList {  
    private String name;  
    private SortedMap<String, Music> myList;  
  
    public PlayListClass(String name) {  
        this.name = name;  
        this.myList = new TreeMap<String, Music>();  
    }  
    private String key(Music m) {  
        return m.getName() + m.getArtist();  
    }  
    @Override  
    public boolean contains(Music m) {  
        return myList.containsKey(key(m));  
    }  
    @Override  
    public void addMusic(Music m) {  
        if ((m != null) && (!this.contains(m)))  
            myList.put(key(m), m);  
    }  
}
```

- Temos de criar uma variável de instância para guardar as músicas
- Actualizar o construtor
- Gerar chave apropriada
- Testar se música faz parte da colecção
- Acrescentar música à colecção, se não existir

Testar o método `deleteMusic()`

84

- Acrescentar músicas
- Verificar que as músicas inseridas ficaram na PlayList
- Remover algumas das músicas
- Verificar que as músicas removidas deixaram de estar na PlayList, enquanto que as não removidas se mantêm na PlayList

Testar o método `deleteMusic()`

85

```
@Test
public void testDeleteMusic() {
    testAddMusic();
    assertTrue(play.contains(m1));
    assertTrue(play.contains(m2));
    play.deleteMusic(m1);
    assertFalse(play.contains(m1));
    assertTrue(play.contains(m2));
}
```

- O teste **falha**, como seria de esperar
- O problema resolve-se especificando o corpo do método `deleteMusic`, na classe `Playlist`

```
public void deleteMusic(Music m) {
    myList.remove(key(m));
}
```

86

Testar selectores

Vale a pena testar um selector?

87

- Depende...

- Se o selector 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 indirectamente!

- Recordar testes ao construtor

- Se o selector faz algo mais complexo, vale a pena testar o selector

Testes a selectores

88

- Começar por inicializar alguns objectos
- Invocar os selectores a testar e verificar se os resultados são os esperados
- Já vimos alguns exemplos, em testes anteriores
- Vamos experimentar mais alguns, que envolvam cálculos sobre a colecção

Teste à operação `getSize()`

89

```
@Test
public void testSize() {
    assertEquals(0, play.getSize());
    play.addMusic(m1);
    assertEquals(1, play.getSize());
    play.addMusic(m1);
    assertEquals(1, play.getSize());
    play.addMusic(m2);
    assertEquals(2, play.getSize());
    play.addMusic(m3);
    assertEquals(3, play.getSize());
    play.addMusic(m4);
    assertEquals(3, play.getSize());
    play.deleteMusic(sm1);
    assertEquals(3, play.getSize());
    play.deleteMusic(m2);
    assertEquals(2, play.getSize());
    play.deleteMusic(m1);
    assertEquals(1, play.getSize());
    play.deleteMusic(m3);
    assertEquals(0, play.getSize());
}
```

- O teste **falha**, porque ainda temos a implementação por omissão de `getSize()`

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

- Vamos implementar correctamente e voltar a testar

```
public int getSize() {
    return myList.size();
}
```

90

Testar modificadores

Vale a pena testar um modificador?

91

- Depende...

- Se o modificador se limita a afectar 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 String setName() {
    this.name = name;
}
```

- Em todo o caso, pode testar estes métodos indirectamente, recorrendo aos selectores
- Se o modificador faz algo mais complexo, vale a pena testar o modificador
 - Rever exemplos de addMusic() e deleteMusic()