PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 23-25

Testes

A aplicação PlayList

- 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...

- 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...

TESTING

I FIND YOUR LACK OF TESTS DISTURBING.

9/9

0800 Arctan started { 1.2700 9.037 847 025
1000 " stopped - arctan ✓ 9.037 846 995 correct
13" sec (032) MP - MC ~~1.9821447000~~ ~~2.13047641515 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 11.00 test.
Relays changed

Relay 3145
Relay 3370

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

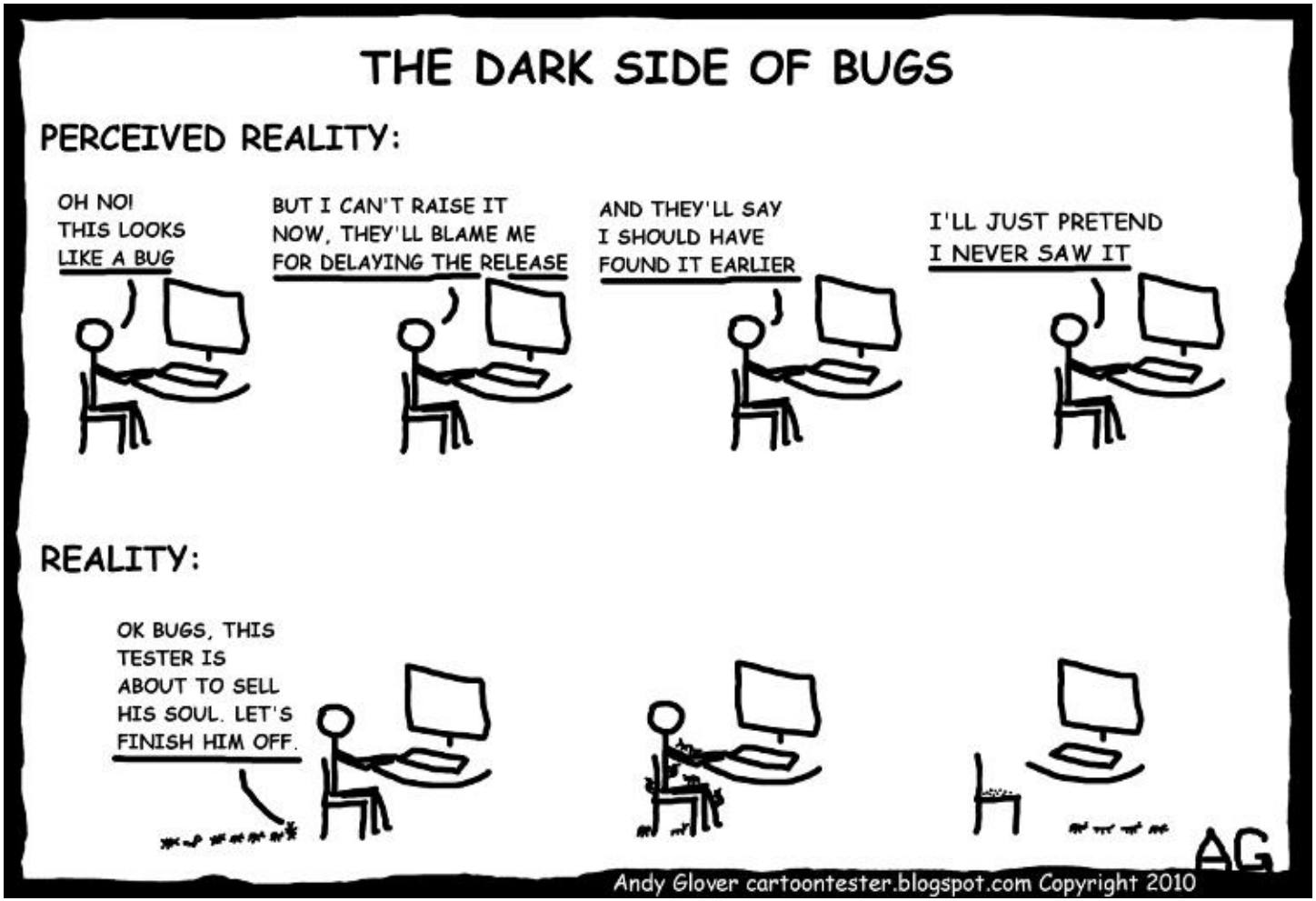
1545 

Relay #70 Panel F
(moth) in relay.

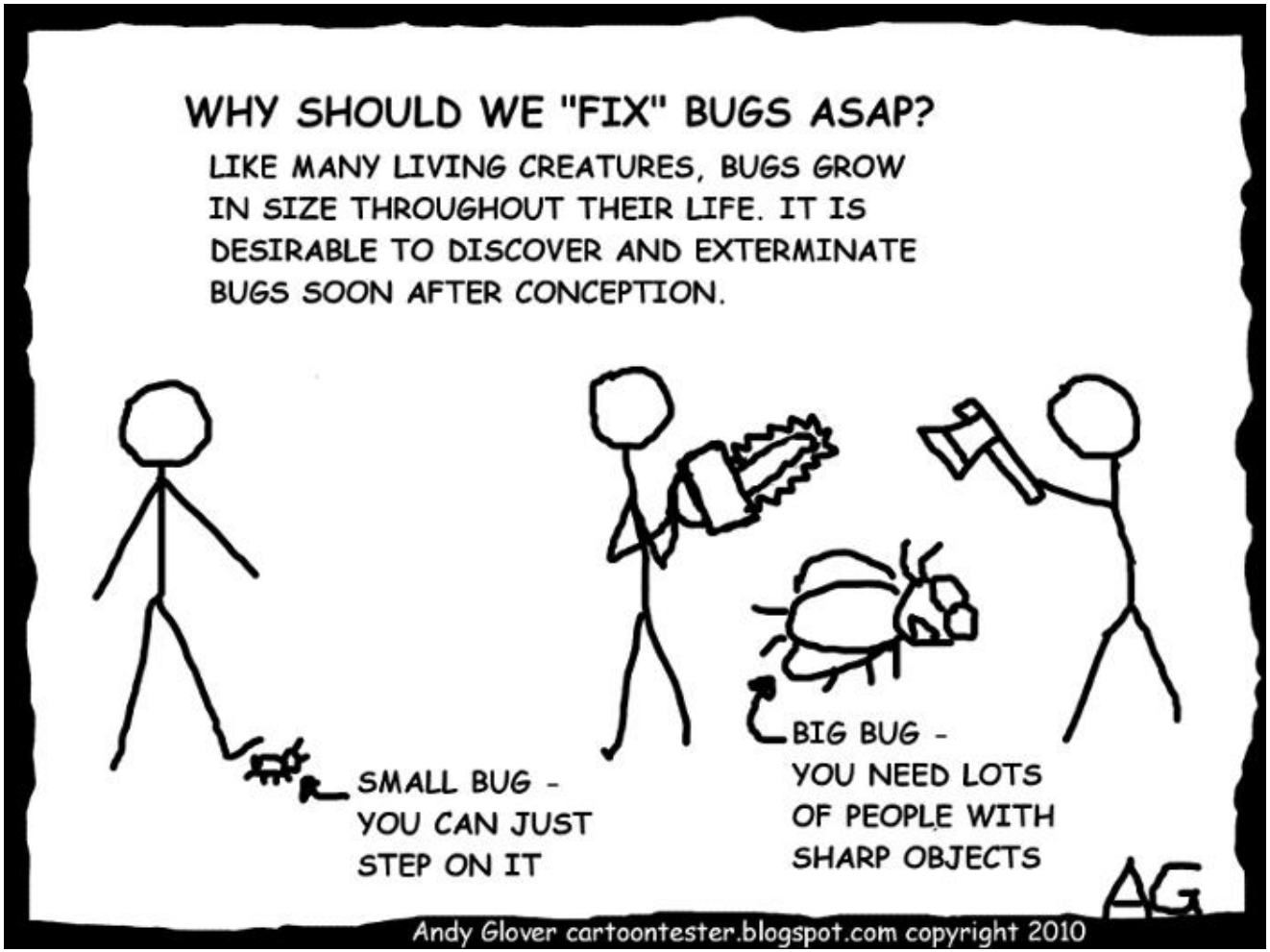
First actual case of bug being found.

~~1545~~ 1630 Arctangent started.
1700 closed down.

Bugs? Naaaah...

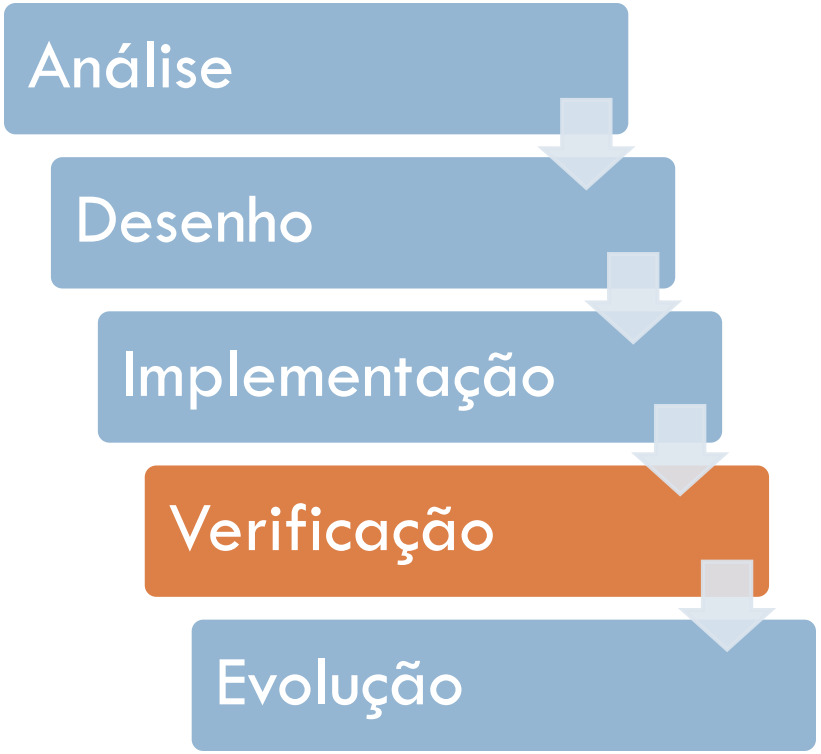

THE DARK SIDE OF BUGS
PERCEIVED REALITY:
OH NO! THIS LOOKS LIKE A BUG
BUT I CAN'T RAISE IT NOW, THEY'LL BLAME ME FOR DELAYING THE RELEASE
AND THEY'LL SAY I SHOULD HAVE FOUND IT EARLIER
I'LL JUST PRETEND I NEVER SAW IT
REALITY:
OK BUGS, THIS TESTER IS ABOUT TO SELL HIS SOUL. LET'S FINISH HIM OFF.
AG
Andy Glover cartoontester.blogspot.com Copyright 2010

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

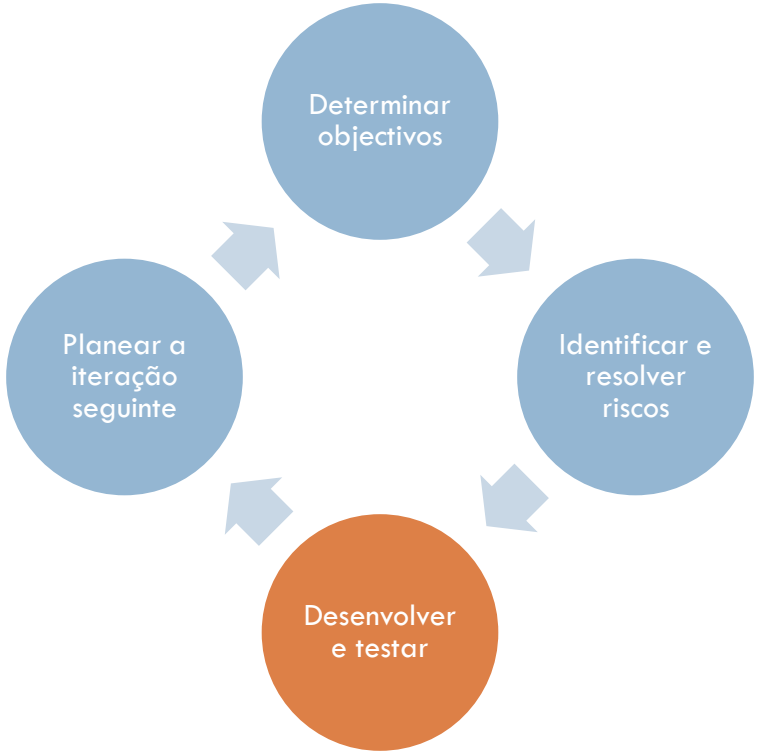


Processo de desenvolvimento

Modelo em cascata



Modelo em espiral



Como garantir a qualidade do software

- 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 fase, mas temos estado atentos a todas as restantes desde o início do curso!

Várias técnicas para verificação da qualidade

- Revisão do desenho, código e outros entregáveis por pares
- 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
  - 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

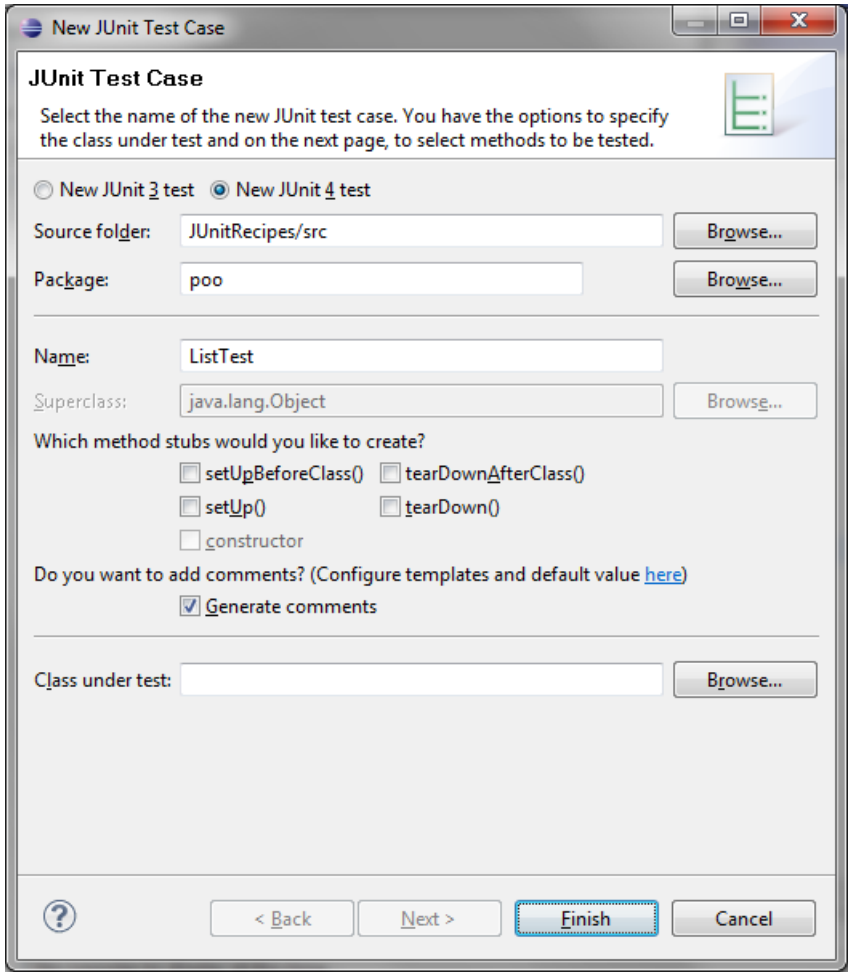
Testes unitários

- Porque gostamos de os fazer?
  - Automáticos
  - Escaláveis
  - Precisos
  - 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?

Comecemos por um exemplo...

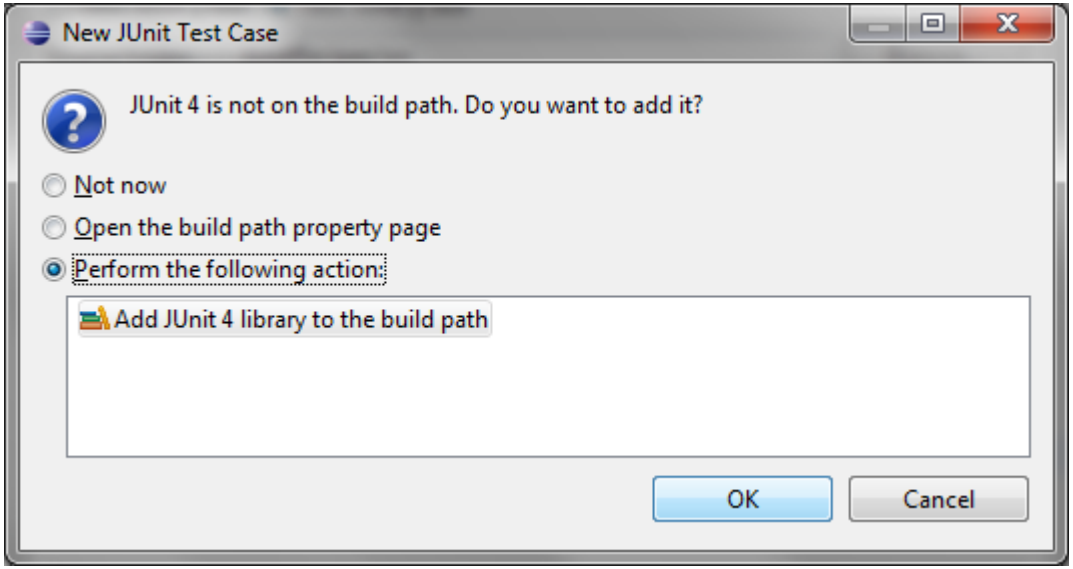
Crie um caso de teste

- 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

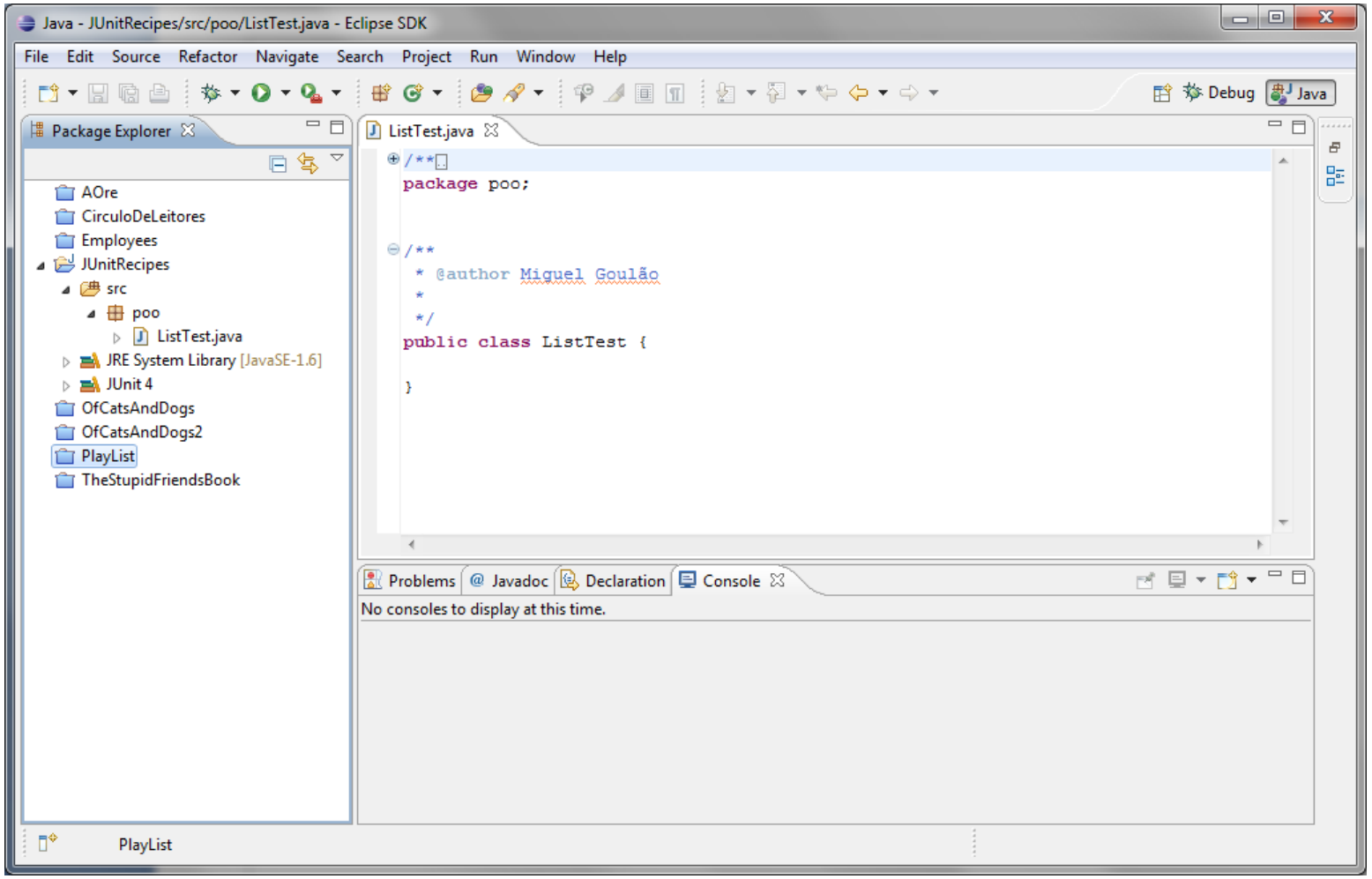


Adicione o JUnit ao seu ambiente

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



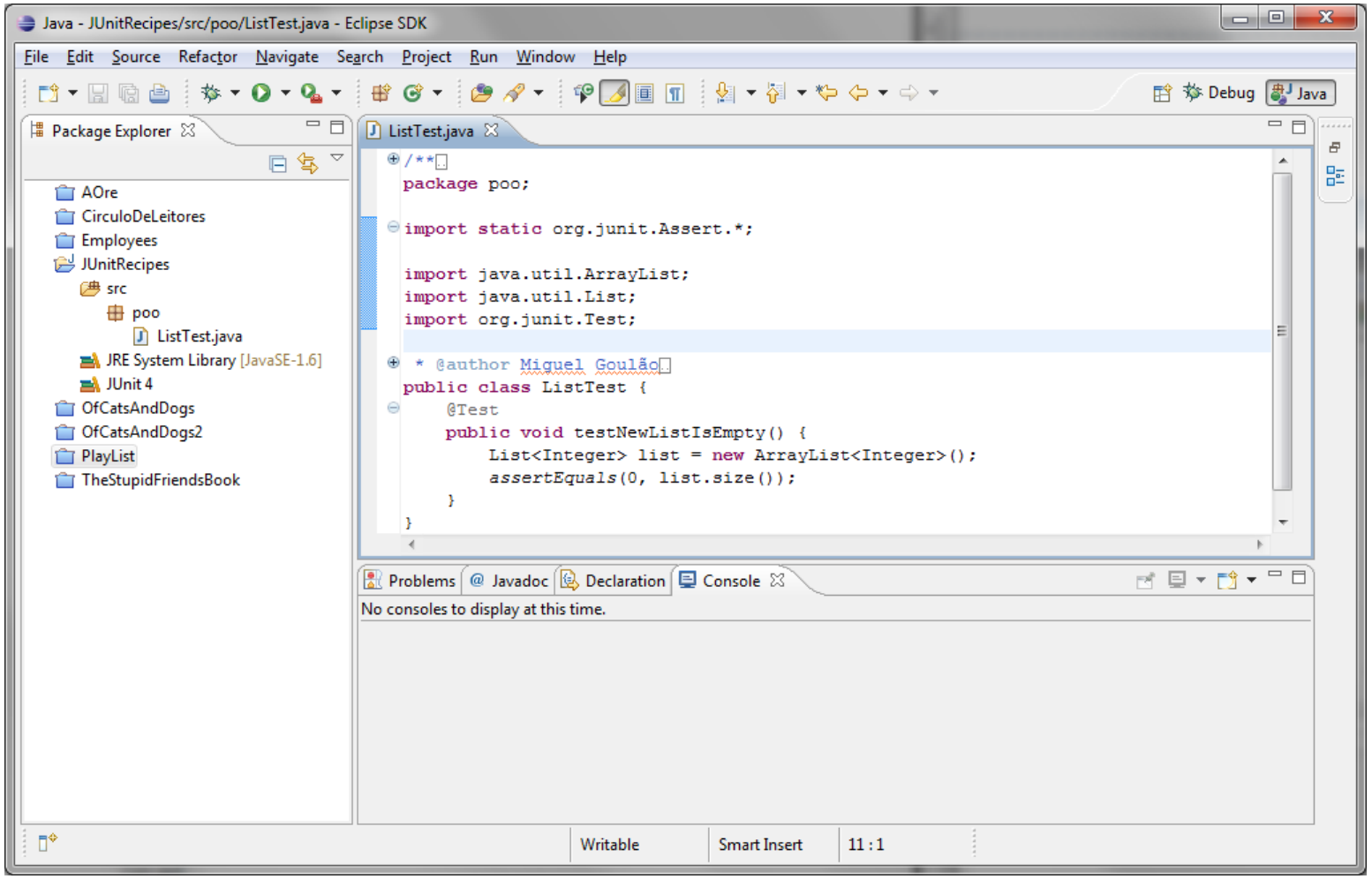
Vista do projecto



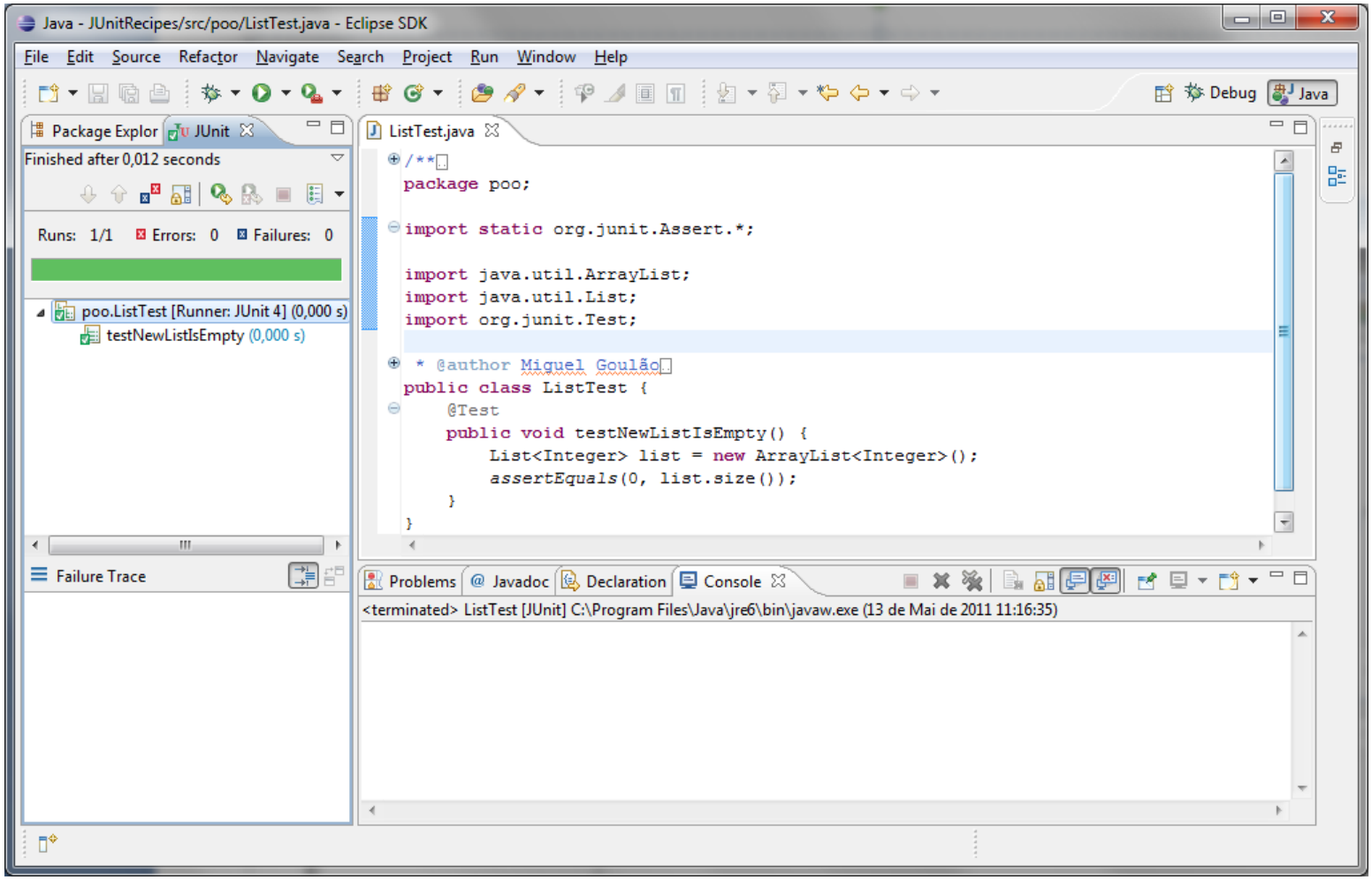
Vamos criar o nosso primeiro teste

- 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**
- Nestes primeiros testes, vamos usar como cobaias classes e interfaces que sabemos funcionar bem, da Java Collections Framework

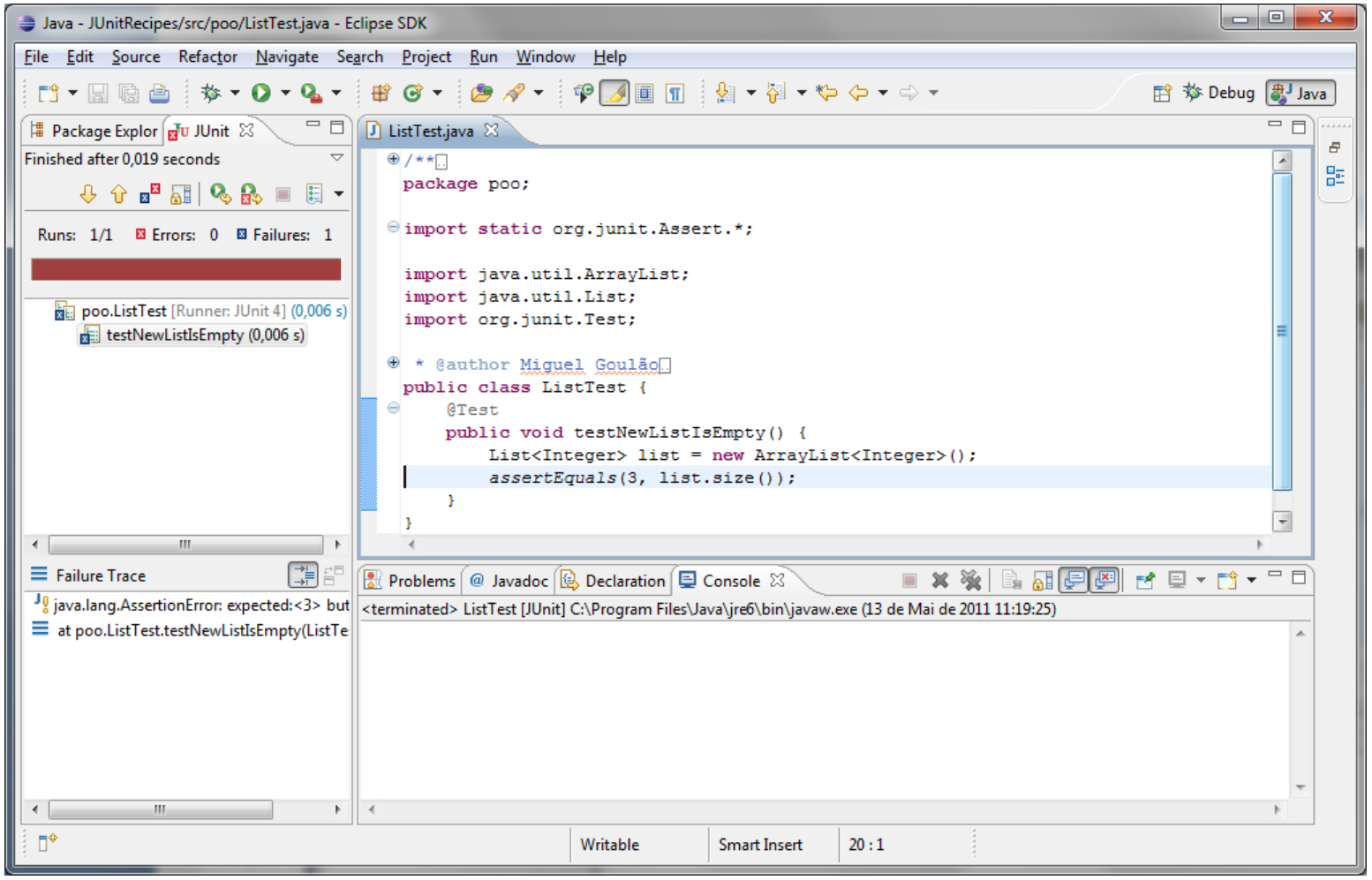
Quando criamos uma lista nova, ela está vazia



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



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



O método `assertEquals()`

- O método **`assertEquals`** compara os seus dois argumentos, que podem ser:
  - Valores de tipos primitivos (**`int`**, **`float`**, **`boolean`**, **`double`**, **`long`**, **`short`**, **`char`**, **`byte`**)
  - Objectos
- Se forem iguais, o teste passa
- Se forem diferentes, o teste não passa

Mesmo teste, 2ª versão, mais curta


```
@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, 3ª versão, ainda mais curta


```
@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

- 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

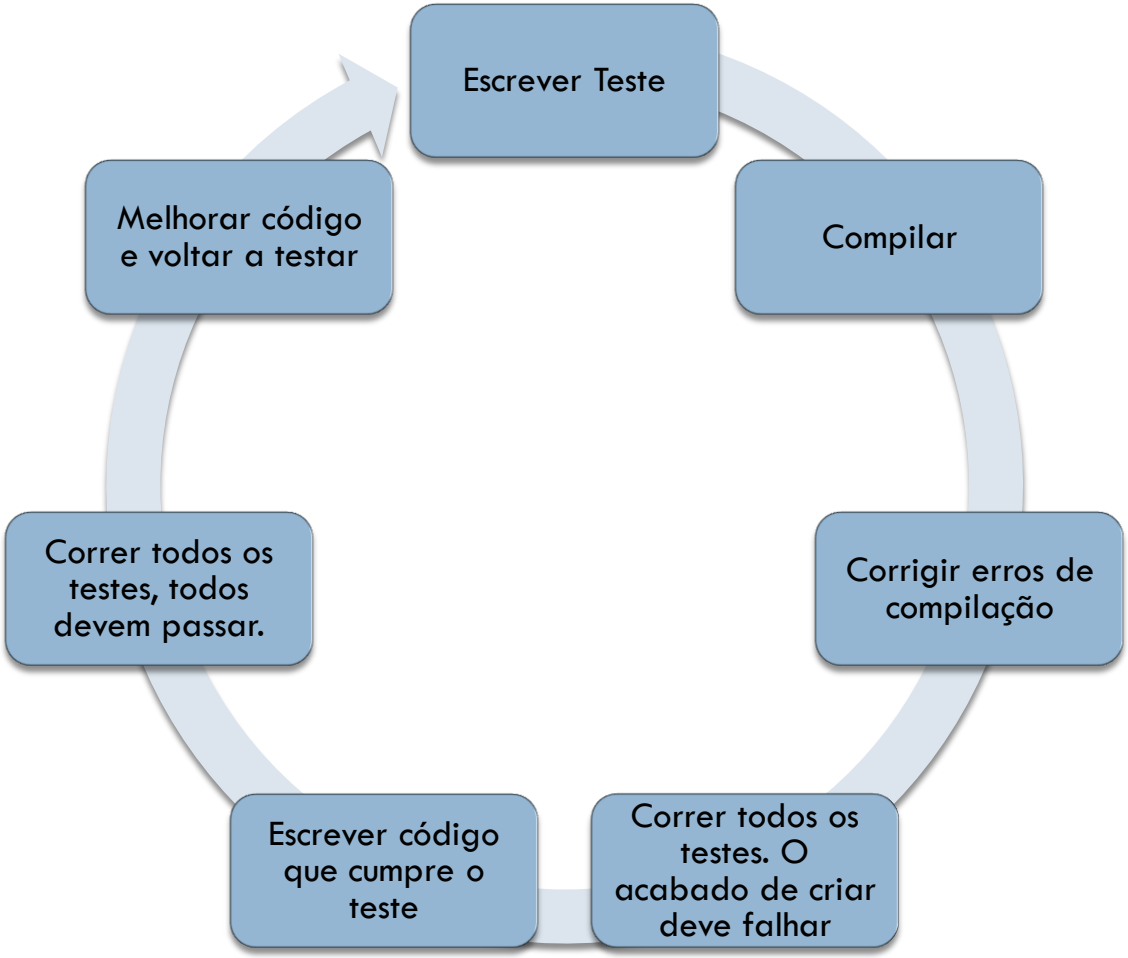
- **Nota:** para que isto funcione bem, temos de implementar o método `equals()` na classe dos objectos a testar!

Quando especificar os testes?

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

Desenvolvimento dirigido pelos testes

Desenvolvimento dirigido pelos testes



Desenvolvimento dirigido pelos testes

- Princípios fundamentais
  - Os testes dirigem o processo
    - Antes de acrescentar funcionalidade, escrever um teste
    - Depois de falhar o teste, acrescentamos a funcionalidade
  - Controlo sobre o processo
    - Nunca estar a mais de um teste da "barra verde"

Funcionamento dos testes de unidade

- Independência dos testes
- Testar a interface, não a implementação
  - Os testes devem ser criados a pensar na interface
  - Nunca deve expor o estado interno de um objecto para efeitos de teste
  - Se necessário, enriqueça a interface
- Não testar a plataforma
  - A API do Java está bem testada, em princípio não deve ter de a testar

PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Test-Driven Development (exemplo)

A aplicação PlayList

- 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


```
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);
}
```

Enumeração FileFormat


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

Implementação de MusicClass (esqueleto)


```
public class MusicClass implements Music {
  @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)


```
  ...
  @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;
  }
}
```

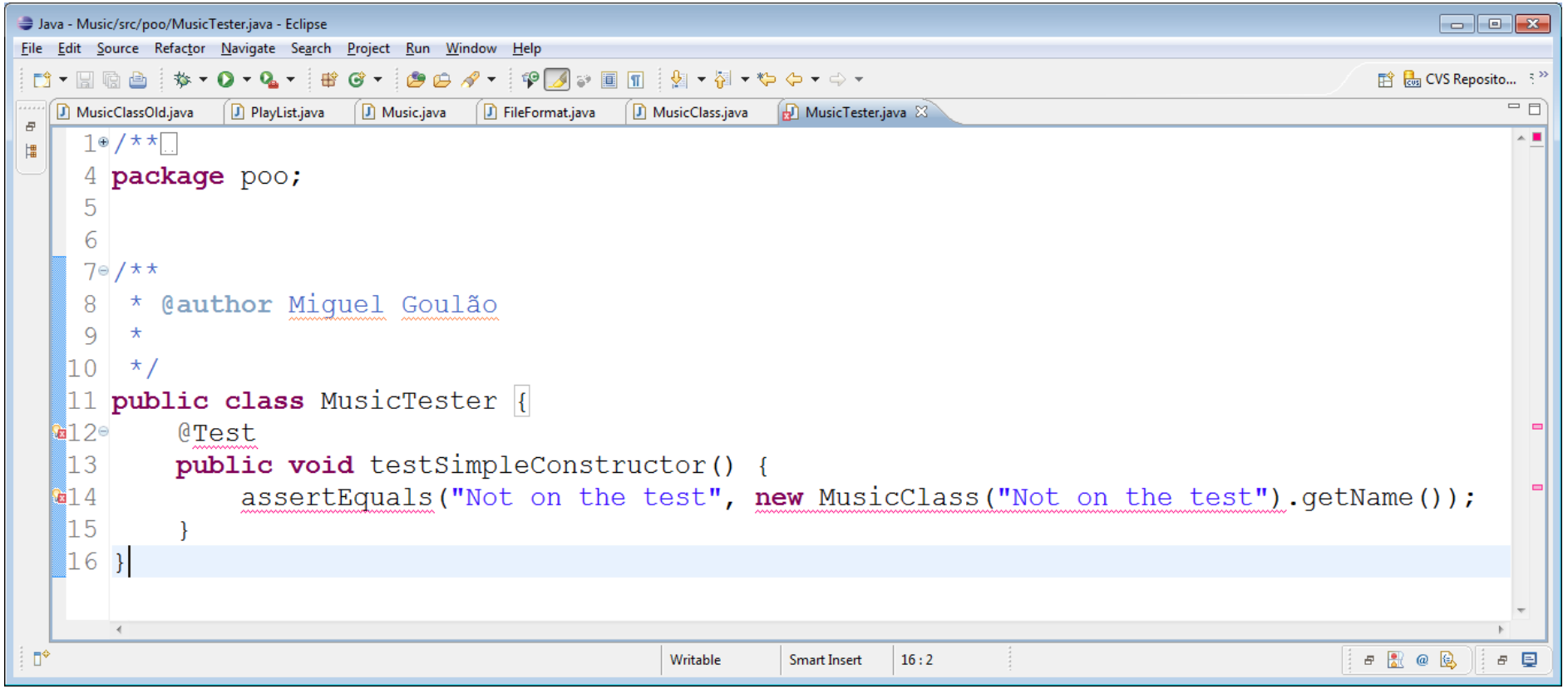
Normalmente...

- 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

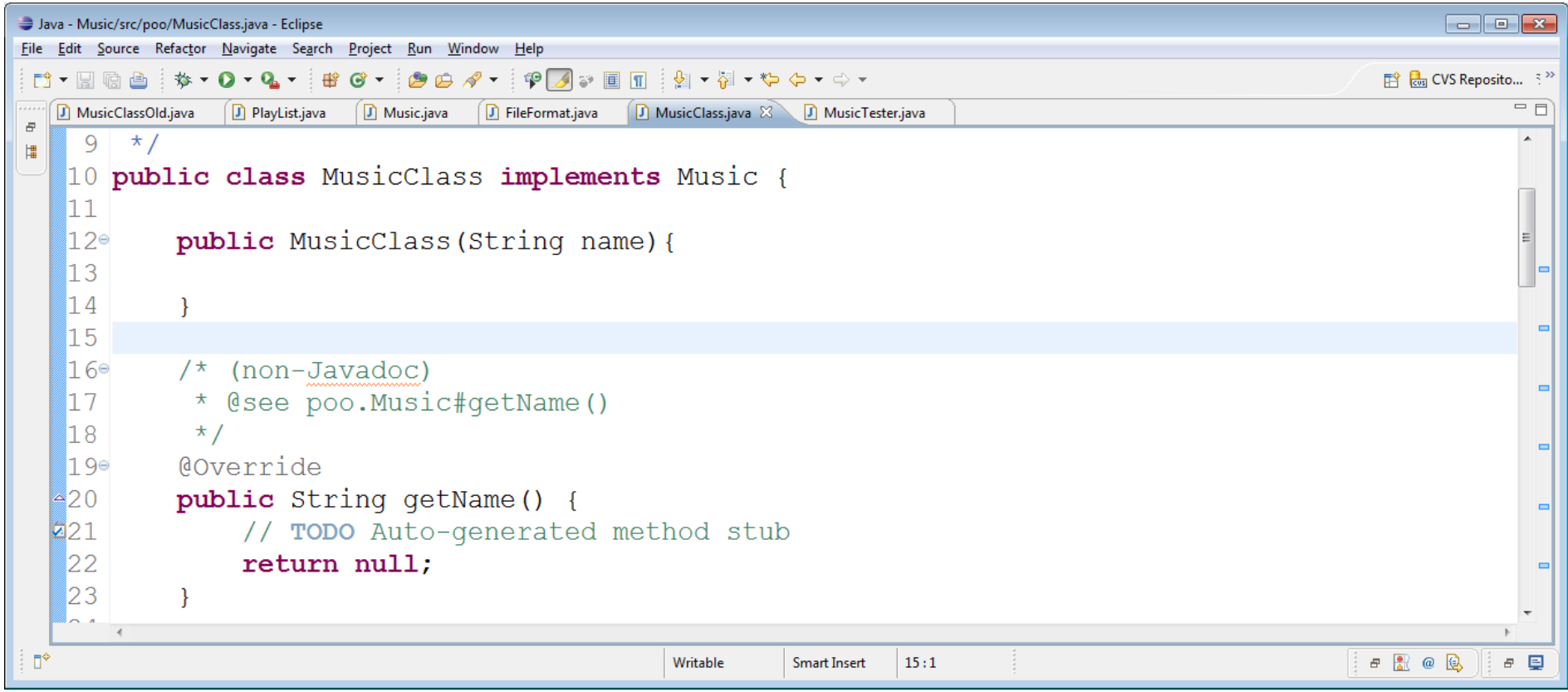
- No desenvolvimento guiado pelos testes, tem de começar por especificar testes antes de escrever o código
- Comecemos pelo construtor, dado que sem ele não vamos conseguir fazer mais nada
- Acrescente um teste simples ao construtor
  - Sim, esse mesmo, que ainda não desenvolveu

Ooops, vários erros de compilação



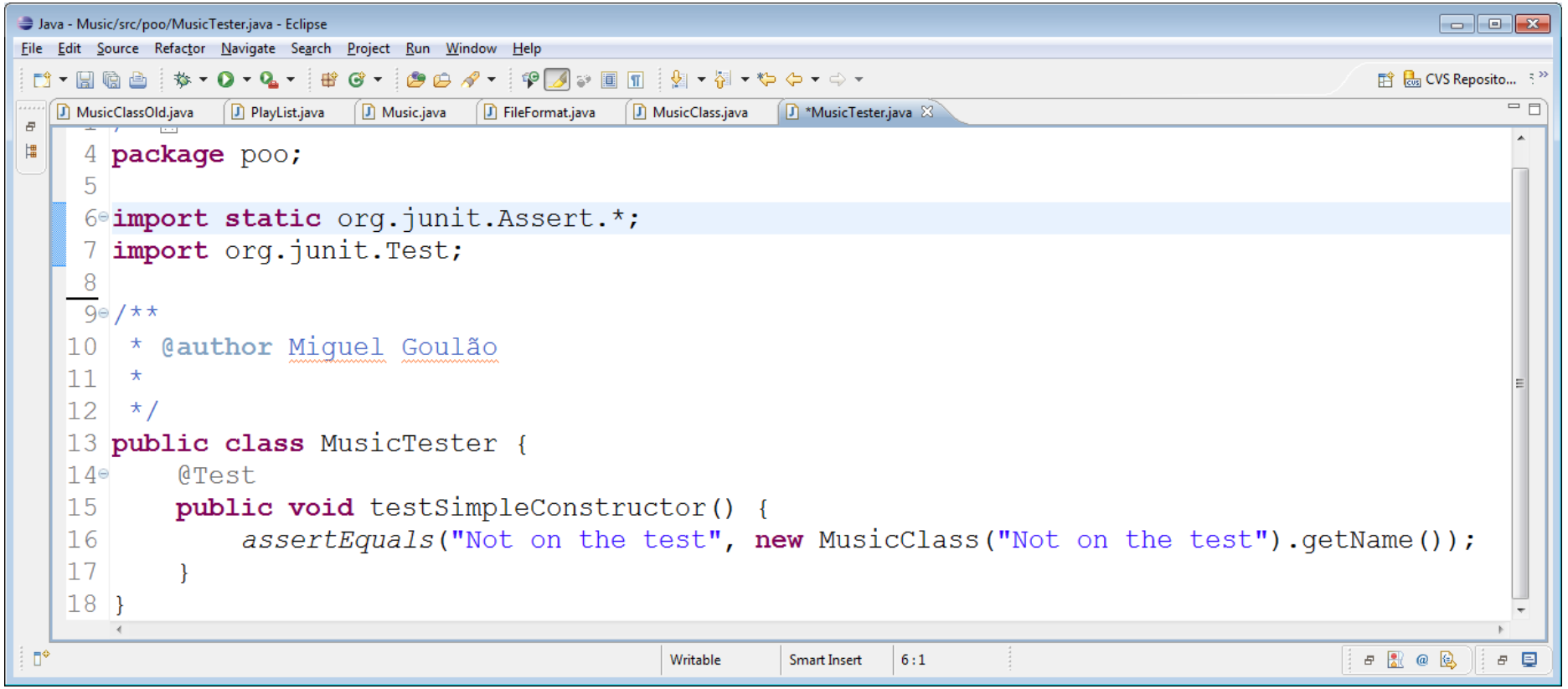
- Perfeito, era exactamente isso que esperávamos! ☺

Na classe MusicClass, falta o construtor



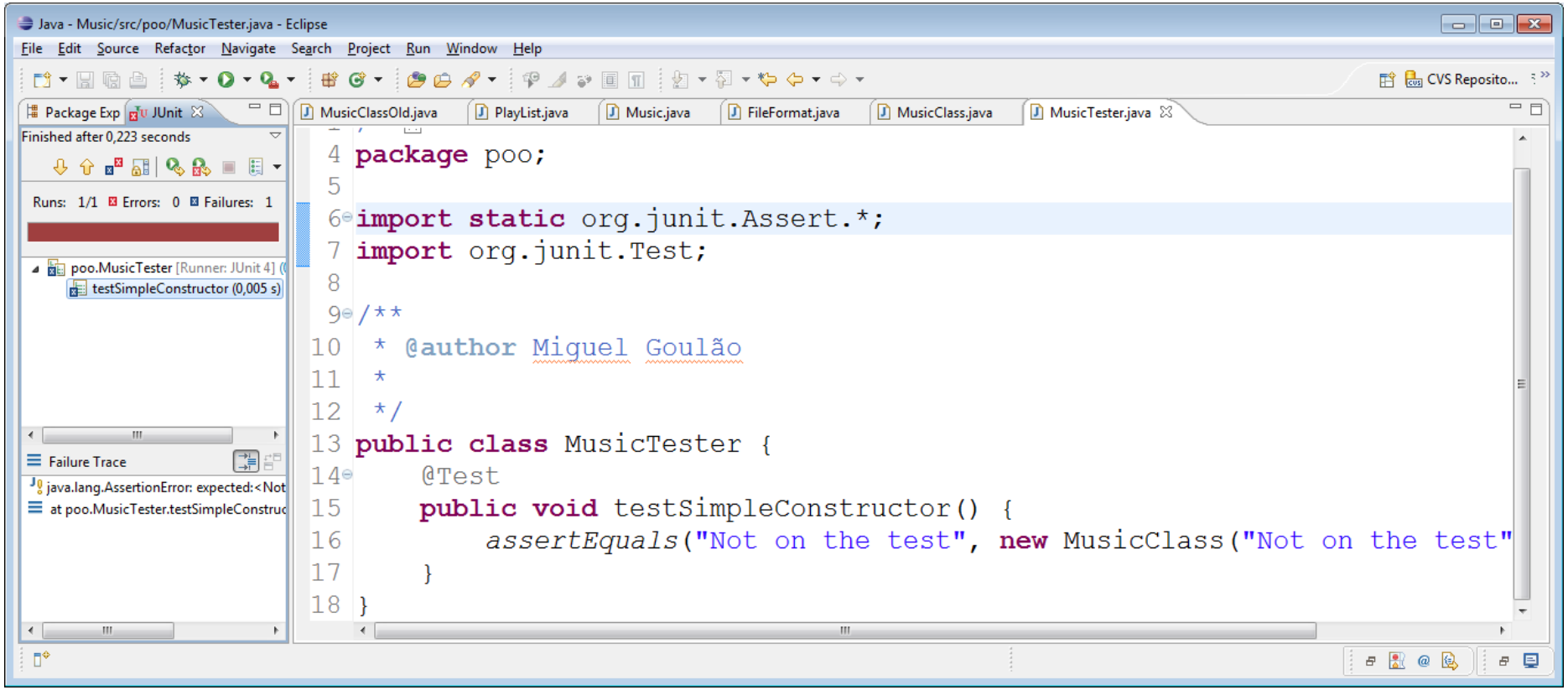
- Lembre-se, para começar, só queremos compilar...

Na classe de teste, faltavam alguns imports



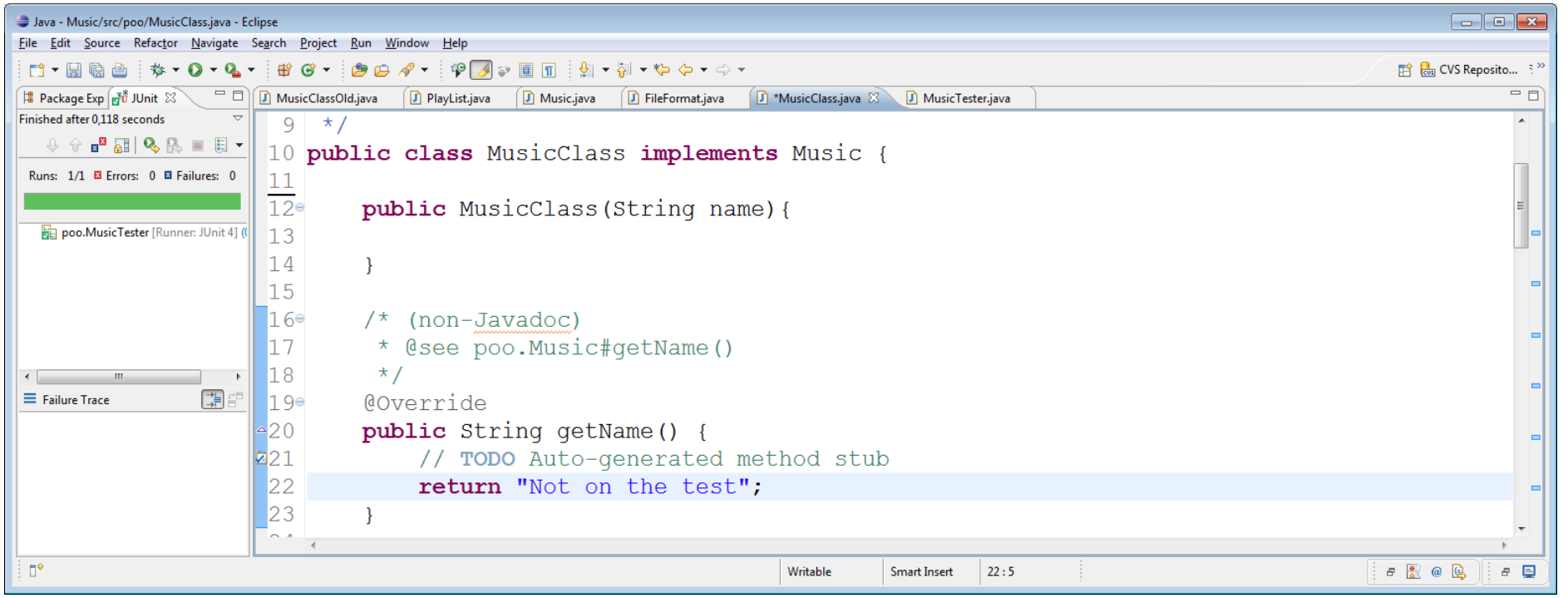
- Sucesso! ☺ Toca a testar!

Afinal havia outra...



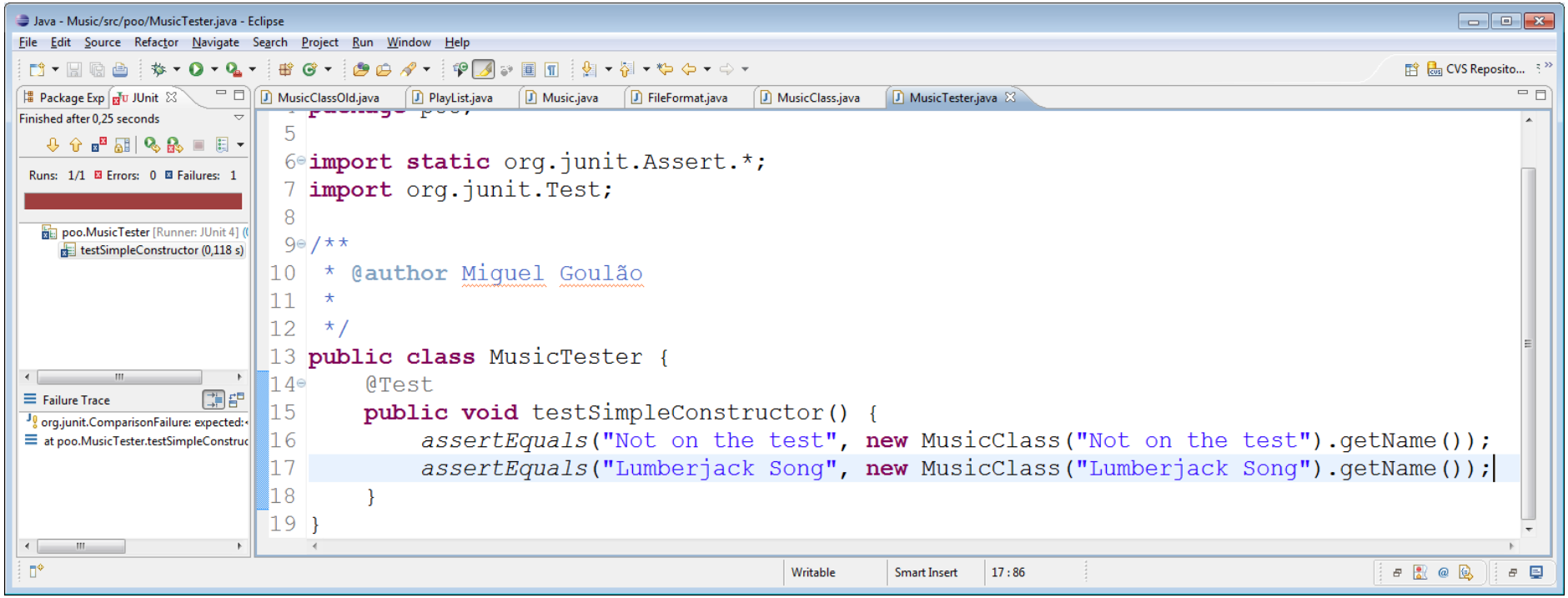
- ... coisa a corrigir. Claro! O construtor ainda não faz nada!
  - O selector também não...

Qual o código mais simples que pode funcionar?



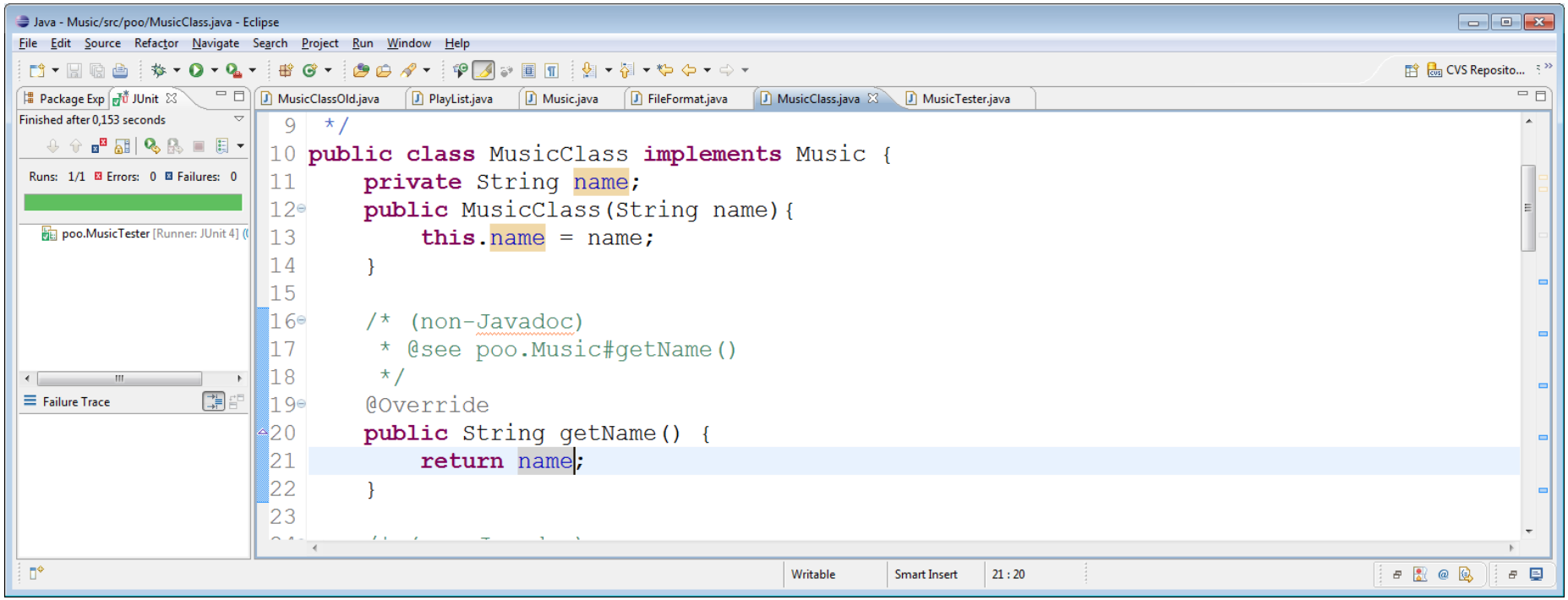
- Parece uma aldrabice, mas o facto é que a funcionalidade mais geral ainda não está coberta por testes
- O facto é que este código passa o teste

Transformemos o teste para ser mais geral



- 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



- Sucesso, podemos avançar para um construtor mais completo

Primeiro, construímos o teste


```
@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)


```
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


```
@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


```
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

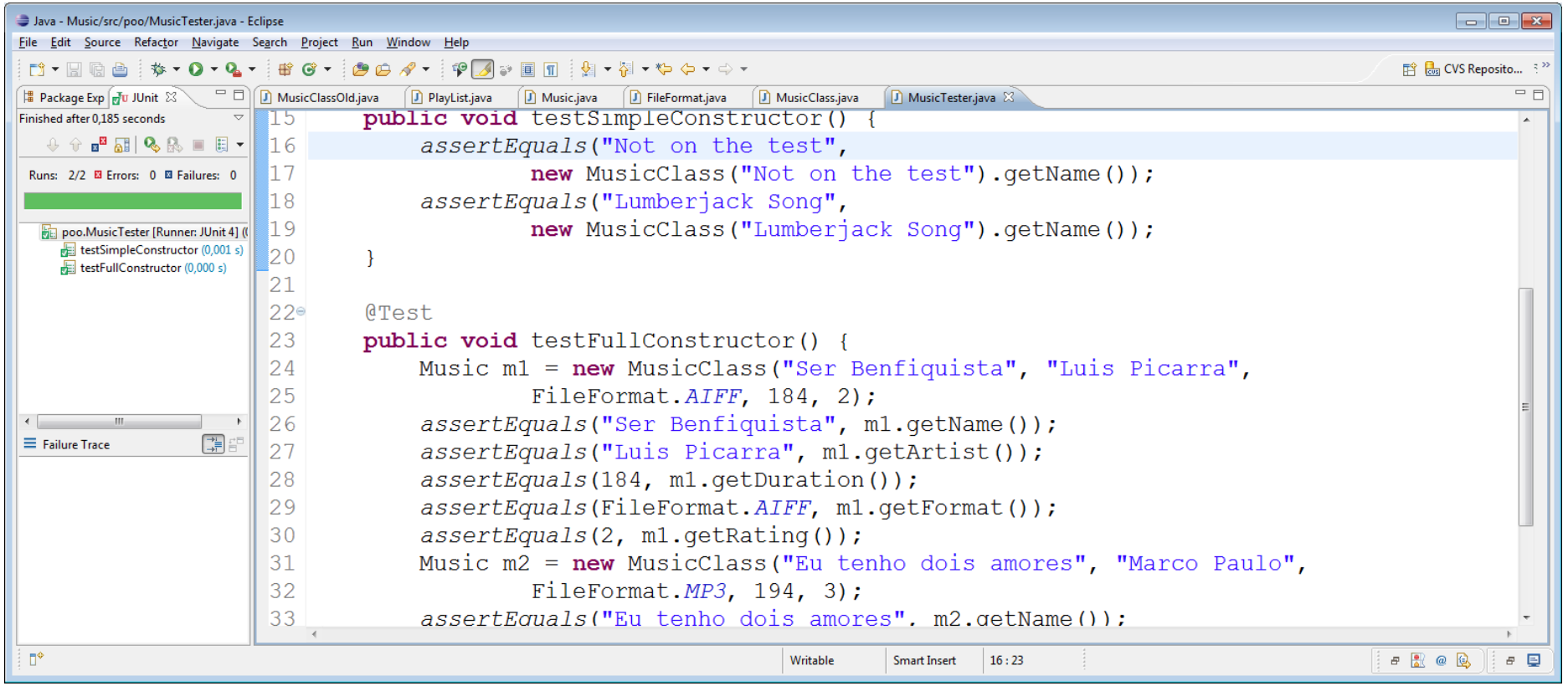

```
  @Override
  public String getArtist() {
    return artist;
  }

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

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

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

Desta vez, ambos os testes passam



- Está tudo?

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

- Esta operação é conhecida como **refabricação**
  - Devemos aperfeiçoar código que esteja a funcionar
    - Melhorando a sua estrutura interna
    - Não alterando o seu comportamento externo
  - Para quê?
    - Tornar o código fonte mais fácil de compreender
    - Preparar o código para facilitar a adição de novas funcionalidades
  - Qual é o risco?
    - 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

- Só refabricamos código que passa em todos os testes existentes
- Só acrescentamos funcionalidades novas quando há um teste que o código actual não passa
  - Assim, garantimos ter pelo menos um teste para cada nova funcionalidade acrescentada

Exemplo de refabricação


```
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;
}
```

- Código repetido, considerado um sinal de má programação

```
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;
}
```

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

Testes para métodos `equals()`

Propriedades do método `equals()`

- 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()

- 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


```
@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?

- Sim, para ter o que testar
- Não temos é de o fazer sempre
  - Como evitar a repetição?
    - Factorizando, como é evidente!

Criação de músicas factorizada


```
public class MusicTester {
   private Music sm1;
   private Music sm2;
   private Music m1;
   private Music m2;
   private Music m3;
   private Music m4;
   @Before
   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);
   }
   ...
```

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 `MusicTester`.

Criação de músicas factorizada


```
...
@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


```
@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


```
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

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



```
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;
}
```

- Falha na reflexividade, quando comparamos, por exemplo, **sm1** consigo próprio
  - Recorde que sm1 foi construído com o construtor mais simples, que não inicializava todas as variáveis de instância
    - Ou mudamos o equals()
    - Ou mudamos o construtor simples, para inicializar todas as variáveis de instância, nem que seja com valores por omissão

Vamos mudar o construtor simples


```
public MusicClass(String name){
  this.name = name;
  this.artist = "";
  this.format = FileFormat.MP3; //Este, ou outro qualquer
  this.duration = 0;
  this.rating = 0;
}
```

- Agora já passamos no teste
- Estamos satisfeitos?

Ainda há tanta coisa que pode correr mal...


```
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;
}
```

- **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


```
@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 NullPointerAssignments

```
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()`


```
@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?


```
@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()`


```
@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


```
@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

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

- Está tudo testado?

TESTING

I FIND YOUR LACK OF TESTS DISTURBING.

Implementação e teste de PlayList