

Aula 04

Correcção

Abordagens Sistemáticas à Programação

Programação II, 2020-2021

v1.6, 01-03-2020

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

1 Tipos de Dados Abstratos

2 Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

1 Tipos de Dados Abstratos

2 Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve **o que** um tipo de dados representa sem ditar **como** o faz.
- Um TDA deve ter tudo o que precisamos de saber para **utilizar** o tipo de dados.
- Assim, o utilizador do tipo pode **abstrair-se** dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A **abstração** é a melhor forma de lidar com a complexidade.

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... . (*Definição mais completa adiante.*)

- Um TDA descreve o que um tipo de dados representa sem ditar como o faz.
- Um TDA deve ter tudo o que precisamos de saber para utilizar o tipo de dados.
- Assim, o utilizador do tipo pode abstrair-se dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A abstração é a melhor forma de lidar com a complexidade.

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve **o que** um tipo de dados representa sem ditar **como** o faz.
- Um TDA deve ter tudo o que precisamos de saber para **utilizar** o tipo de dados.
- Assim, o utilizador do tipo pode **abstrair-se** dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A **abstração** é a melhor forma de lidar com a complexidade.

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve **o que** um tipo de dados representa sem ditar **como** o faz.
- Um TDA deve ter tudo o que precisamos de saber para **utilizar** o tipo de dados.
- Assim, o utilizador do tipo pode **abstrair-se** dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A **abstração** é a melhor forma de lidar com a complexidade.

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve **o que** um tipo de dados representa sem ditar **como** o faz.
- Um TDA deve ter tudo o que precisamos de saber para **utilizar** o tipo de dados.
- Assim, o utilizador do tipo pode **abstrair-se** dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A **abstração** é a melhor forma de lidar com a complexidade.

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve **o que** um tipo de dados representa sem ditar **como** o faz.
- Um TDA deve ter tudo o que precisamos de saber para **utilizar** o tipo de dados.
- Assim, o utilizador do tipo pode **abstrair-se** dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A **abstração** é a melhor forma de lidar com a complexidade.

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve **o que** um tipo de dados representa sem ditar **como** o faz.
- Um TDA deve ter tudo o que precisamos de saber para **utilizar** o tipo de dados.
- Assim, o utilizador do tipo pode **abstrair-se** dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A **abstração** é a melhor forma de lidar com a complexidade.

Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

```
public class Fraction {  
    private int num;  
    private int den;  
    public Fraction add(Fraction f) { ... }  
    public Fraction multiply(Fraction f) { ... }  
}
```

- É um tipo para representar a posição de um pixel no ecrã.

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel move(int dx, int dy) { ... }  
    public double distanceTo(Pixel) { ... }  
}
```

- Ambos usam dois atributos inteiros para representar os dados.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

```
public class Fraction {  
    private int num;  
    private int den;  
    public Fraction add(Fraction f) { ... }  
    public Fraction multiply(Fraction f) { ... }  
}
```

- E um tipo para representar a posição de um pixel no ecrã.

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel move(int dx, int dy) { ... }  
    public double distanceTo(Pixel) { ... }  
}
```

- Ambos usam dois atributos inteiros para representar os dados.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

```
public class Fraction {  
    private int num;  
    private int den;  
    public Fraction add(Fraction f) { ... }  
    public Fraction multiply(Fraction f) { ... }  
}
```

- E um tipo para representar a posição de um pixel no ecrã.

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel move(int dx, int dy) { ... }  
    public double distanceTo(Pixel) { ... }  
}
```

- Ambos usam dois atributos inteiros para representar os dados.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

```
public class Fraction {  
    private int num;  
    private int den;  
    public Fraction add(Fraction f) { ... }  
    public Fraction multiply(Fraction f) { ... }  
}
```

- E um tipo para representar a posição de um pixel no ecrã.

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel move(int dx, int dy) { ... }  
    public double distanceTo(Pixel) { ... }  
}
```

- Ambos usam dois atributos inteiros para representar os dados.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

```
public class Fraction {  
    private int num;  
    private int den;  
    public Fraction add(Fraction f) { ... }  
    public Fraction multiply(Fraction f) { ... }  
}
```

- E um tipo para representar a posição de um pixel no ecrã.

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel move(int dx, int dy) { ... }  
    public double distanceTo(Pixel) { ... }  
}
```

- Ambos usam dois atributos inteiros para representar os dados.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

```
public class Fraction {  
    private int num;  
    private int den;  
    public Fraction add(Fraction f) { ... }  
    public Fraction multiply(Fraction f) { ... }  
}
```

- E um tipo para representar a posição de um pixel no ecrã.

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel move(int dx, int dy) { ... }  
    public double distanceTo(Pixel) { ... }  
}
```

- Ambos usam dois atributos inteiros para representar os dados.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma $\frac{N}{D}$, por exemplo $\frac{7}{3}$.

```
public class Fraction {  
    private int N; // Numerador  
    private int D; // Denominador  
    public Fraction add(Fraction f) { return }  
    public Fraction multiply(Fraction f) { return }  
}
```

- Mas também pode ser representada na forma mista $2\frac{1}{3}$.

```
public class Fraction {  
    private int I; // Parte inteira  
    private int N; // Numerador  
    private int D; // Denominador (maior que N)  
    public Fraction add(Fraction f) { return }  
    public Fraction multiply(Fraction f) { return }  
}
```

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma $\frac{N}{D}$, por exemplo $\frac{7}{3}$.

```
public class Fraction {
    private int N; // Numerador
    private int D; // Denominador
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- Mas também pode ser representada na forma mista $1\frac{N}{D}$, por exemplo $2\frac{1}{3}$.

```
public class Fraction {
    private int I; // Parte inteira
    private int N; // Numerador
    private int D; // Denominador (maior que N)
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma $\frac{N}{D}$, por exemplo $\frac{7}{3}$.

```
public class Fraction {
    private int N; // Numerador
    private int D; // Denominador
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- Mas também pode ser representada na forma mista $1\frac{N}{D}$, por exemplo $2\frac{1}{3}$.

```
public class Fraction {
    private int I; // Parte inteira
    private int N; // Numerador
    private int D; // Denominador (maior que N)
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma $\frac{N}{D}$, por exemplo $\frac{7}{3}$.

```
public class Fraction {
    private int N; // Numerador
    private int D; // Denominador
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- Mas também pode ser representada na forma mista $1\frac{N}{D}$, por exemplo $2\frac{1}{3}$.

```
public class Fraction {
    private int I; // Parte inteira
    private int N; // Numerador
    private int D; // Denominador (maior que N)
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma $\frac{N}{D}$, por exemplo $\frac{7}{3}$.

```
public class Fraction {
    private int N; // Numerador
    private int D; // Denominador
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- Mas também pode ser representada na forma mista $1\frac{N}{D}$, por exemplo $2\frac{1}{3}$.

```
public class Fraction {
    private int I; // Parte inteira
    private int N; // Numerador
    private int D; // Denominador (maior que N)
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma $\frac{N}{D}$, por exemplo $\frac{7}{3}$.

```
public class Fraction {
    private int N; // Numerador
    private int D; // Denominador
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- Mas também pode ser representada na forma mista $1\frac{N}{D}$, por exemplo $2\frac{1}{3}$.

```
public class Fraction {
    private int I; // Parte inteira
    private int N; // Numerador
    private int D; // Denominador (maior que N)
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

Tipos de Dados Abstratos

Abordagens Sistemáticas à Programação

Testando o programa por fora

Testando o programa por dentro

Associando um significado aos programas

Formalizando uma Especificação

Programação por Contrato

Programação por Contrato em Java

- As classes são uma forma de implementar tipos de dados abstratos.
- As operações do TDA são expostas pelas declarações de **membros públicos** de uma classe e constituem a sua **interface**.
- Os **membros privados** e os **corpos dos métodos** constituem a **implementação**.
- **Classe = Interface + Implementação;**

```
public class Data {  
    public Data() { }  
    public Data(int dia, int mes, int ano) { }  
    public int dia() { }  
    public int mes() { }  
    public int ano() { }  
    public boolean equals(Data outra) { }  
    private  
}
```

- As classes são uma forma de implementar tipos de dados abstratos.
- As operações do TDA são expostas pelas declarações de **membros públicos** de uma classe e constituem a sua **interface**.
- Os **membros privados** e os **corpos** dos métodos constituem a **implementação**.
- **Classe = Interface + Implementação**;

```
public class Data {  
    public Data() { ... }  
    public Data(int dia, int mes, int ano) { ... }  
    public int dia() { ... }  
    public int mes() { ... }  
    public int ano() { ... }  
    public boolean equals(Data outra) { ... }  
    private ...  
}
```


- As classes são uma forma de implementar tipos de dados abstratos.
- As operações do TDA são expostas pelas declarações de **membros públicos** de uma classe e constituem a sua **interface**.
- Os **membros privados** e os **corpos** dos métodos constituem a **implementação**.
- **Classe = Interface + Implementação**;

```
public class Data {  
    public Data() { ... }  
    public Data(int dia, int mes, int ano) { ... }  
    public int dia() { ... }  
    public int mes() { ... }  
    public int ano() { ... }  
    public boolean equals(Data outra) { ... }  
    private ...  
}
```

- As classes são uma forma de implementar tipos de dados abstratos.
- As operações do TDA são expostas pelas declarações de **membros públicos** de uma classe e constituem a sua **interface**.
- Os **membros privados** e os **corpos** dos métodos constituem a **implementação**.
- Classe = Interface + Implementação;

```
public class Data {  
    public Data() { ... }  
    public Data(int dia, int mes, int ano) { ... }  
    public int dia() { ... }  
    public int mes() { ... }  
    public int ano() { ... }  
    public boolean equals(Data outra) { ... }  
    private ...  
}
```

- As classes são uma forma de implementar tipos de dados abstratos.
- As operações do TDA são expostas pelas declarações de **membros públicos** de uma classe e constituem a sua **interface**.
- Os **membros privados** e os **corpos** dos métodos constituem a **implementação**.
- **Classe = Interface + Implementação**;

```
public class Data {  
    public Data() { ... }  
    public Data(int dia, int mes, int ano) { ... }  
    public int dia() { ... }  
    public int mes() { ... }  
    public int ano() { ... }  
    public boolean equals(Data outra) { ... }  
    private ...  
}
```

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Exercitar o programa para diferentes entradas;
 - Construção de programas orientada ao teste (TOD: Test Driven Development);
 - Analisar o código do programa de acordo com sua concepção;
- Testando o programa por **dentro**:
 - Corresponde à verificação interna para dentro do próprio programa?
 - Se sim, então o programa sob teste encontra-se a ser falha (e pode agir em conformidade);

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

- O factor de qualidade mais importante é a **correção**;
- Assim sendo, como verificar se um programa está **correcto**?
- Testando o programa por **fora**:
 - Executar o programa para diferentes casos;
 - Construção de programas orientada ao teste (*TDD: Test Driven Development*);
 - Árbitro **externo** ao programa decide da sua correcção.
- Testando o programa por **dentro**:
 - Será possível trazer o árbitro para dentro do próprio programa?
 - Se for, então o programa saberá quando está em falha (e pode agir em conformidade);

Abordagens Sistemáticas à Programação (2)

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perante o significado) do software.
 - Melhoramos a legibilidade.
 - Potencializamos a correção.

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perceber o significado) do software.
 - Melhoramos a legibilidade.
 - Potenciamos a correcção.

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perceber o significado) do software.
 - Melhoramos a legibilidade.
 - Potenciamos a correcção.

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perceber o significado) do software.
 - Melhoramos a legibilidade.
 - Potenciamos a correcção.

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perceber o significado) do software.
 - Melhoramos a legibilidade.
 - Potenciamos a correcção.

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perceber o significado) do software.
 - Melhoramos a legibilidade.
 - Potenciamos a correcção.

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma **semântica**).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.
- O **significado deve ficar explícito** no próprio código fonte. Desse modo:
 - Facilitamos a compreensão (perceber o significado) do software.
 - Melhoramos a legibilidade.
 - Potenciamos a correcção.

Abordagens Sistemáticas à Programação (3)

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Para procurar atingir esse objetivo devemos:

- Adotar normas sugestivas de design, métodos, notação
- Documentar adequadamente o código, bases de dados, etc.
- Dar um significado claro e inequívoco aos elementos do código
- Analisar a estrutura do código a nível de módulos. Esta é uma abordagem ainda mais profunda e crítica que as anteriores.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Para procurar atingir esse objetivo devemos:
 - Atribuir nomes sugestivos às classes, métodos, variáveis.
 - Documentar adequadamente o código. Bons comentários devem conter significados que não sejam evidentes no próprio código.
 - Anotar o elemento de software com **asserções**. Esta é uma abordagem ainda mais poderosa e eficaz que as anteriores.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Para procurar atingir esse objetivo devemos:
 - Atribuir nomes sugestivos às classes, métodos, variáveis.
 - Documentar adequadamente o código. Bons comentários devem conter significados que não sejam evidentes no próprio código.
 - Anotar o elemento de software com **asserções**. Esta é uma abordagem ainda mais poderosa e eficaz que as anteriores.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Para procurar atingir esse objetivo devemos:
 - Atribuir nomes sugestivos às classes, métodos, variáveis.
 - Documentar adequadamente o código. Bons comentários devem conter significados que não sejam evidentes no próprio código.
 - Anotar o elemento de software com **asserções**. Esta é uma abordagem ainda mais poderosa e eficaz que as anteriores.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Para procurar atingir esse objetivo devemos:
 - Atribuir nomes sugestivos às classes, métodos, variáveis.
 - Documentar adequadamente o código. Bons comentários devem conter significados que não sejam evidentes no próprio código.
 - Anotar o elemento de software com **asserções**. Esta é uma abordagem ainda mais poderosa e eficaz que as anteriores.

Abordagens Sistemáticas à Programação (4)

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma extensiva, que resume o nível de contrato de software código nesse ponto do programa.
 - Uma **documentação** da funcionalidade do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser executado automaticamente sempre que o programa é executado.

Abordagens Sistemáticas à Programação (4)

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

- As **asserções** são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma **pré-condição** desse elemento.
- Se suceder ao elemento, então diz-se que é uma **pós-condição**.
- Cada asserção incluída no código fonte pode ser vista como
 - Uma **especificação** expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
 - Uma **documentação** do funcionamento do código, sem o risco de ficar incoerente com o código.
 - Um **teste**, que pode ser verificado sistematicamente sempre que o programa é executado.

Exemplo

- Este programa está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Não sabemos! Depende do que é suposto ele fazer.
- Especificação:
 - Calcular o quociente q de r entre y como um número inteiro de x por y.

- Este programa está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Não sabemos! Depende do que é suposto ele fazer.
- Especificação:
 - Calcula o quociente q e o resto r como resultados da divisão inteira de x por y .

- Este programa está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Não sabemos! Depende do que é suposto ele fazer.
- Especificação:
 - Calcula o quociente q e o resto r como resultados da divisão inteira de x por y .

- Este programa está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Não sabemos! Depende do que é suposto ele fazer.
- Especificação:
 - Calcula o quociente q e o resto r como resultados da divisão inteira de x por y .

- Este programa está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Não sabemos! Depende do que é suposto ele fazer.
- Especificação:
 - Calcula o quociente q e o resto r como resultados da divisão inteira de x por y .

Exemplo

- Este programa calcula o quociente q e o resto r como resultados da divisão inteira de x por y . Está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- TALVEZ!** De acordo com a especificação podemos provar que no final:

$$x = y * q + r,$$

que é a *propriedade fundamental da divisão*.

- Este programa calcula o quociente q e o resto r como resultados da divisão inteira de x por y . Está correcto?

```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- TALVEZ!** De acordo com a especificação podemos provar que no final:

$$x = y * q + r,$$

que é a *propriedade fundamental da divisão*.

- Este programa calcula o quociente q e o resto r como resultados da divisão inteira de x por y . Está correcto?

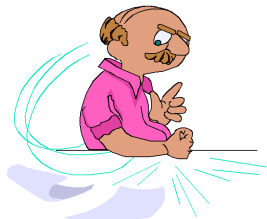
```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- TALVEZ!** De acordo com a especificação podemos provar que no final:

$$x = y * q + r,$$

que é a *propriedade fundamental da divisão*.

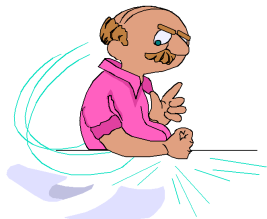
Algum tempo mais tarde



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

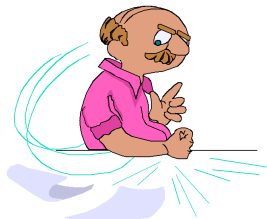
- Logo a especificação está incompleta.
- Devíamos ter "dito" que o programa só se aplica se $y > 0 \wedge x \geq 0$.

- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

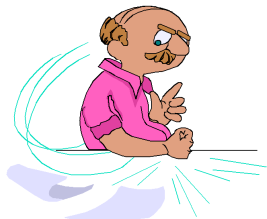
- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!
- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se $y > 0 \wedge x \geq 0$.



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!

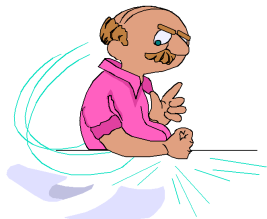
- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se $y > 0 \wedge x \geq 0$.



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!

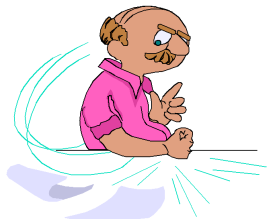
- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se $y > 0 \wedge x \geq 0$.



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!

- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se $y > 0 \wedge x \geq 0$.

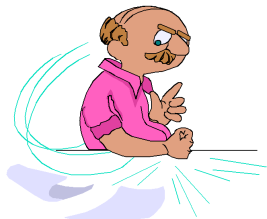


```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se $y > 0 \wedge x \geq 0$.

- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!

Algum tempo mais tarde

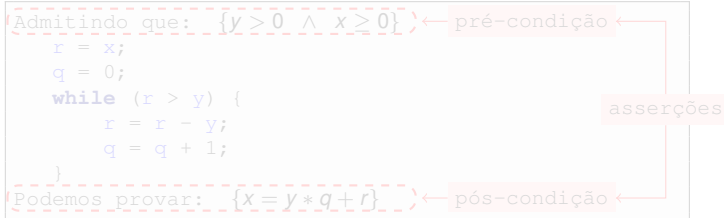


```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se $y > 0 \wedge x \geq 0$.

- O programa não está correcto!
- Não termina quando $y = 0$!
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de x ou y também são problemáticos!

Exemplo



Exemplo

```
Admitindo que:  $\{y > 0 \wedge x \geq 0\}$  ← pré-condição
  r = x;
  q = 0;
  while (r > y) {
    r = r - y;
    q = q + 1;
  }
Podemos provar:  $\{x = y * q + r\}$  ← pós-condição
```

asserções

Exemplo

```
Admitindo que:  $\{y > 0 \wedge x \geq 0\}$  ← pré-condição  
   $r = x;$   
   $q = 0;$   
  while ( $r > y$ ) {  
     $r = r - y;$   
     $q = q + 1;$   
  }  
Podemos provar:  $\{x = y * q + r\}$  ← pós-condição
```

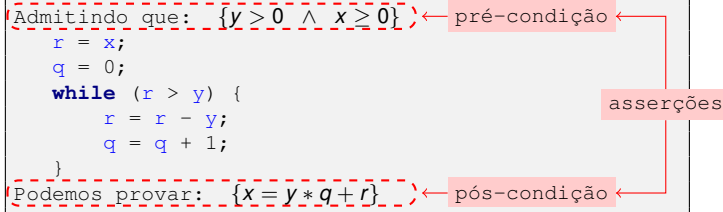
asserções

Exemplo

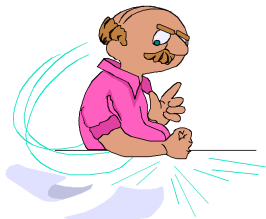
```
Admitindo que:  $\{y > 0 \wedge x \geq 0\}$  ← pré-condição  
   $r = x;$   
   $q = 0;$   
  while ( $r > y$ ) {  
     $r = r - y;$   
     $q = q + 1;$   
  }  
Podemos provar:  $\{x = y * q + r\}$  ← pós-condição
```

asserções

Exemplo



Algum tempo mais tarde



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

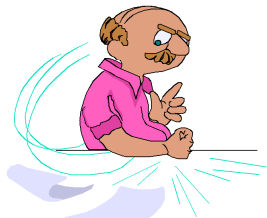
- Oops! É um erro ... vamos ver...

- O programa ainda não está correcto!
- Quando $x = 6$ e $y = 3$ o resultado é:

$$q = 1 \wedge r = 3$$

- em vez de:

$$q = 2 \wedge r = 0$$



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

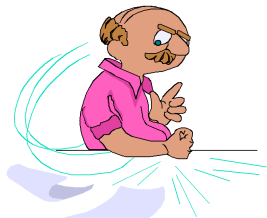
- O programa ainda não está correcto!
- Quando $x = 6$ e $y = 3$ o resultado é:

$$q = 1 \wedge r = 3$$

- em vez de:

$$q = 2 \wedge r = 0$$

- Oops! É um erro ... vamos ver...



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

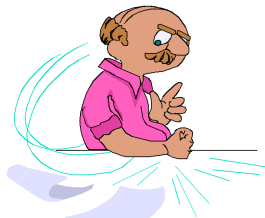
- O programa ainda não está correcto!
- Quando $x = 6$ e $y = 3$ o resultado é:

$$q = 1 \wedge r = 3$$

- em vez de:

$$q = 2 \wedge r = 0$$

- Oops! É um erro ... vamos ver...



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

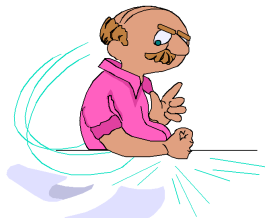
- O programa ainda não está correcto!
- Quando $x = 6$ e $y = 3$ o resultado é:

$$q = 1 \wedge r = 3$$

- em vez de:

$$q = 2 \wedge r = 0$$

- Oops! É um erro ... vamos ver...



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- Oops! É um erro ... vamos ver...

- O programa ainda não está correcto!
- Quando $x = 6$ e $y = 3$ o resultado é:

$$q = 1 \wedge r = 3$$

- em vez de:

$$q = 2 \wedge r = 0$$

Exemplo

Admitindo que: $\{y > 0 \wedge x \geq 0\}$

```
r = x;
```

```
q = 0;
```

```
while( (r >= y) ) {
```

```
    r = r - y;
```

```
    q = q + 1;
```

```
}
```

rectificação

nova pós-condição

Podemos provar: $\{x = y * q + r \wedge r < y\}$

Exemplo

Admitindo que: $\{y > 0 \wedge x \geq 0\}$

```
r = x;
```

```
q = 0;
```

```
while (r >= y) {
```

```
    r = r - y;
```

```
    q = q + 1;
```

```
}
```

rectificação

nova pós-condição

Podemos provar: $\{x = y * q + r \wedge r < y\}$

Exemplo

Admitindo que: $\{y > 0 \wedge x \geq 0\}$

```
r = x;
```

```
q = 0;
```

```
while( (r >= y) ) {
```

```
    r = r - y;
```

```
    q = q + 1;
```

```
}
```

rectificação

nova pós-condição

Podemos provar: $\{x = y * q + r \wedge r < y\}$

Exemplo

Admitindo que: $\{y > 0 \wedge x \geq 0\}$

```
r = x;
```

```
q = 0;
```

```
while( (r >= y) ) {
```

```
    r = r - y;
```

```
    q = q + 1;
```

```
}
```

rectificação

nova pós-condição

Podemos provar: $\{x = y * q + r \wedge r < y\}$

Exemplo

Admitindo que: $\{y > 0 \wedge x \geq 0\}$

```
r = x;
```

```
q = 0;
```

```
while( (r >= y) ) {
```

```
    r = r - y;
```

```
    q = q + 1;
```

```
}
```

rectificação

nova pós-condição

Podemos provar: $\{x = y * q + r \wedge r < y\}$

Formalizando uma Especificação

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:

- P é a **pré-condição** de A ;
- Q é a **pós-condição** de A .

- Significado:

- Qualquer execução de A , começando num estado que satisfaz a **asserção** P , deverá terminar num estado que satisfaz Q .

- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .
- Significado:
 - Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .
- Significado:
 - Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .
- Significado:
 - Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .

- Significado:

- Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .

- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .
- Significado:
 - Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .
- Significado:
 - Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

- Considere-se qualquer bloco de instruções A . A sua formulação em **lógica de Hoare** pode ser expressa como:

$$\{P\} A \{Q\}$$

- P e Q são **asserções**:
 - P é a **pré-condição** de A ;
 - Q é a **pós-condição** de A .
- Significado:
 - Qualquer execução de A , começando num estado que satisfaça P deverá terminar num estado que satisfaça Q .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- A **Programação por Contrato (PpC)** é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas **pré-condições e pós-condições**.
- Cada tipo de dados, deve especificar as condições **invariantes**.
- A essa especificação, quando feita por asserções, dá-se o nome de **contrato do módulo**.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

- A **Programação por Contrato** (PpC) é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas **pré-condições** e **pós-condições**.
- Cada tipo de dados, deve especificar as condições **invariantes**.
- A essa especificação, quando feita por asserções, dá-se o nome de **contrato do módulo**.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

- A **Programação por Contrato** (PpC) é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas **pré-condições** e **pós-condições**.
- Cada tipo de dados, deve especificar as condições **invariantes**.
- A essa especificação, quando feita por asserções, dá-se o nome de **contrato do módulo**.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

- A **Programação por Contrato** (PpC) é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas **pré-condições** e **pós-condições**.
- Cada tipo de dados, deve especificar as condições **invariantes**.
- A essa especificação, quando feita por asserções, dá-se o nome de **contrato do módulo**.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

- A **Programação por Contrato** (PpC) é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas **pré-condições** e **pós-condições**.
- Cada tipo de dados, deve especificar as condições **invariantes**.
- A essa especificação, quando feita por asserções, dá-se o nome de **contrato do módulo**.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

- A **Programação por Contrato** (PpC) é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas **pré-condições** e **pós-condições**.
- Cada tipo de dados, deve especificar as condições **invariantes**.
- A essa especificação, quando feita por asserções, dá-se o nome de **contrato do módulo**.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

Contratos de Funções

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    assert x >= 0; ← pré-condição

    double result;
    ...

    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
    ↑
    pós-condição
}
```

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    assert x >= 0; ← pré-condição

    double result;
    ...

    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
    ↑
    pós-condição
}
```

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    (assert x >= 0;) ← pré-condição

    double result;
    ...

    (assert Math.abs(result*result-x) <= NEAR_ZERO;)
    return result;
} ← pós-condição
```

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    (assert x >= 0;) ← pré-condição

    double result;
    ...

    (assert Math.abs(result*result-x) <= NEAR_ZERO;)
    return result;
} ← pós-condição
```

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    (assert x >= 0;) ← pré-condição

    double result;
    ...

    (assert Math.abs(result*result-x) <= NEAR_ZERO;)
    return result;
} ← pós-condição
```

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    assert x >= 0;
    double result;
    ...
    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
}
```

pré-condição

pós-condição

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    assert x >= 0; ← pré-condição

    double result;
    ...

    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
} ← pós-condição
```

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    assert x >= 0; ← pré-condição

    double result;
    ...

    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
    ↑
    pós-condição
}
```

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato associado à especificação de funções é definido pelas **pré-condições** e **pós-condições** da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
    assert x >= 0; ← pré-condição

    double result;
    ...

    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
}
```

↑ pós-condição

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos.
- Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe `Data` poderá ter o seguinte invariante:

```
valida(dia(),mes(),ano())
```

- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam sempre uma data válida.

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos.
- Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe `Data` poderá ter o seguinte invariante:

```
valida(dia(),mes(),ano())
```

- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam **sempre** uma data válida.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos.
- Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe `Data` poderá ter o seguinte invariante:

```
valida(dia(),mes(),ano())
```
- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam **sempre** uma data válida.

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos.
- Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe `Data` poderá ter o seguinte invariante:

```
valida(dia(),mes(),ano())
```
- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam **sempre** uma data válida.

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos.
- Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe Data poderá ter o seguinte invariante:

```
valida(dia(),mes(),ano())
```

- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam **sempre** uma data válida.

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos.
- Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe Data poderá ter o seguinte invariante:

```
valida(dia(),mes(),ano())
```

- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam **sempre** uma data válida.

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

Tipo de Dados Abstrato (TDA)

É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

Distribuição de Responsabilidades

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A PpC permite uma distribuição simples e clara de responsabilidades entre o módulo e os seus clientes:

	Obrigações	Benefícios
Cliente	Tem de garantir as pré-condições do módulo	Sabe que pós-condições e invariante são garantidos
Módulo	Tem de garantir o invariante e as pós-condições	Sabe que pré-condições são garantidas

Distribuição de Responsabilidades

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A PpC permite uma distribuição simples e clara de responsabilidades entre o módulo e os seus clientes:

	Obrigações	Benefícios
Cliente	Tem de garantir as pré-condições do módulo	Sabe que pós-condições e invariante são garantidos
Módulo	Tem de garantir o invariante e as pós-condições	Sabe que pré-condições são garantidas

A PpC permite uma distribuição simples e clara de responsabilidades entre o módulo e os seus clientes:

	Obrigações	Benefícios
Cliente	Tem de garantir as pré-condições do módulo	Sabe que pós-condições e invariante são garantidos
Módulo	Tem de garantir o invariante e as pós-condições	Sabe que pré-condições são garantidas

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Obviamente, a escolha dos contratos a associar a cada módulo (função, objecto) está nas mãos de quem o implementa.
- No entanto, como regra deve optar-se por contratos tão fortes quanto necessário para garantir implementações simples e para manter uma boa sensibilidade a falhas, mas sem restringir desnecessariamente o domínio de utilização nem complicar demasiado as condições.
- Por exemplo, no caso dos objectos do tipo `Data`, faz todo o sentido definir como invariante que as datas sejam válidas, já que torna bastante mais simples a compreensão e utilização destes objectos. Nunca será necessário lidar com datas absurdas como por exemplo 31 de Fevereiro de 2000.

- Obviamente, a **escolha dos contratos** a associar a cada módulo (função, objecto) está nas mãos de quem o implementa.
- No entanto, como regra deve optar-se por **contratos** tão **fortes** quanto necessário para garantir implementações simples e para manter uma boa sensibilidade a falhas, mas sem restringir desnecessariamente o domínio de utilização nem complicar demasiado as condições.
- Por exemplo, no caso dos objectos do tipo `Data`, faz todo o sentido definir como invariante que as datas sejam válidas, já que torna bastante mais simples a compreensão e utilização destes objectos. Nunca será necessário lidar com datas absurdas como por exemplo 31 de Fevereiro de 2000.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Obviamente, a **escolha dos contratos** a associar a cada módulo (função, objecto) está nas mãos de quem o implementa.
- No entanto, como regra deve optar-se por **contratos** tão **fortes** quanto necessário para garantir implementações simples e para manter uma boa sensibilidade a falhas, mas sem restringir desnecessariamente o domínio de utilização nem complicar demasiado as condições.
- Por exemplo, no caso dos objectos do tipo `Data`, faz todo o sentido definir como invariante que as datas sejam válidas, já que torna bastante mais simples a compreensão e utilização destes objectos. Nunca será necessário lidar com datas absurdas como por exemplo 31 de Fevereiro de 2000.

- Obviamente, a **escolha dos contratos** a associar a cada módulo (função, objecto) está nas mãos de quem o implementa.
- No entanto, como regra deve optar-se por **contratos** tão **fortes** quanto necessário para garantir implementações simples e para manter uma boa sensibilidade a falhas, mas sem restringir desnecessariamente o domínio de utilização nem complicar demasiado as condições.
- Por exemplo, no caso dos objectos do tipo `Data`, faz todo o sentido definir como invariante que as datas sejam válidas, já que torna bastante mais simples a compreensão e utilização destes objectos. Nunca será necessário lidar com datas absurdas como por exemplo
31 de Fevereiro de 2000.

- Sintaxe:

```
assert booleanExpression [: expression ];
```

- Semântica:

- * Se `booleanExpression` for `true`, a assertão passa.
- * Se for `false`, a assertão falha e é produzida uma mensagem de erro, normalmente precedida pelo nome da propriedade a verificar, e produz um `AssertionError` com o conteúdo que foi verificado e falhou.
- * `expression` é uma expressão opcional que produz uma `String`, que permite dar uma mensagem personalizada a falha.

- Sintaxe:

```
assert booleanExpression [: expression ] ;
```

- Semântica:

- Se `booleanExpression` for `true`, a asserção passa.
- Se for `false`, a asserção falha e é gerado um erro, que normalmente provoca a terminação do programa e produz um relatório com o contexto que antecedeu a falha.
- `expression` é uma expressão opcional, geralmente uma `String`, que permite dar informação adicional sobre a falha.

- Sintaxe:

```
assert booleanExpression [: expression ] ;
```

- Semântica:

- Se `booleanExpression` for `true`, a asserção passa.
- Se for `false`, a asserção falha e é gerado um **erro**, que normalmente provoca a terminação do programa e produz um relatório com o contexto que antecedeu a falha.
- `expression` é uma expressão opcional, geralmente uma `String`, que permite dar informação adicional sobre a falha.

- Sintaxe:

```
assert booleanExpression [: expression ] ;
```

- Semântica:

- Se `booleanExpression` for `true`, a asserção passa.
- Se for `false`, a asserção falha e é gerado um erro, que normalmente provoca a terminação do programa e produz um relatório com o contexto que antecedeu a falha.
- `expression` é uma expressão opcional, geralmente uma `String`, que permite dar informação adicional sobre a falha.

- Sintaxe:

```
assert booleanExpression [: expression ] ;
```

- Semântica:

- Se `booleanExpression` for `true`, a asserção passa.
- Se for `false`, a asserção falha e é gerado um **erro**, que normalmente provoca a terminação do programa e produz um relatório com o contexto que antecedeu a falha.
- `expression` é uma expressão opcional, geralmente uma `String`, que permite dar informação adicional sobre a falha.

- Sintaxe:

```
assert booleanExpression [: expression ] ;
```

- Semântica:

- Se `booleanExpression` for `true`, a asserção passa.
- Se for `false`, a asserção falha e é gerado um **erro**, que normalmente provoca a terminação do programa e produz um relatório com o contexto que antecedeu a falha.
- `expression` é uma expressão opcional, geralmente uma `String`, que permite dar informação adicional sobre a falha.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` OU `-ea`):
`java -ea Prog`
- Para desactivar (`-disableassertions` OU `-da`):
`java Prog` OU `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` ou `-ea`):
`java -ea Prog`
- Para desactivar (`-disableassertions` ou `-da`):
`java Prog` OU `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` ou `-ea`):
`java -ea Prog`
- Para desactivar (`-disableassertions` ou `-da`):
`java Prog` OU `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` ou `-ea`):
`java -ea Prog`
- Para desactivar (`-disableassertions` ou `-da`):
`java Prog` ou `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` ou `-ea`):
`java -ea Prog`
- Para desactivar (`-disableassertions` ou `-da`):
`java Prog` ou `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` ou `-ea`):
`java -ea Prog`
- Para desactivar (`-disableassertions` ou `-da`):
`java Prog` ou `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

Asserções em Java: Exemplos

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

```
1 public class Assert1 {  
2     public static void main(String[] args) {  
3         assert false;  
4     }  
5 }  
Exception in thread "main" java.lang.AssertionError  
at Assert1.main(Assert1.java:3)
```

```
1 public class Assert2 {  
2     public static void main(String[] args) {  
3         assert false: "disparate!";  
4     }  
5 }  
Exception in thread "main" java.lang.AssertionError: disparate!  
at Assert2.main(Assert2.java:3)
```

Asserções em Java: Exemplos

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

```
1 public class Assert1 {  
2     public static void main(String[] args) {  
3         assert false;  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError
at Assert1.main(Assert1.java:3)

```
1 public class Assert2 {  
2     public static void main(String[] args) {  
3         assert false: "disparate!";  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError: disparate!
at Assert2.main(Assert2.java:3)

Asserções em Java: Exemplos

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

```
1 public class Assert1 {  
2     public static void main(String[] args) {  
3         assert false;  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError
at Assert1.main(Assert1.java:3)

```
1 public class Assert2 {  
2     public static void main(String[] args) {  
3         assert false: "disparate!";  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError: disparate!
at Assert2.main(Assert2.java:3)

Asserções em Java: Exemplos

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

```
1 public class Assert1 {  
2     public static void main(String[] args) {  
3         assert false;  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError
at Assert1.main(Assert1.java:3)

```
1 public class Assert2 {  
2     public static void main(String[] args) {  
3         assert false: "disparate!";  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError: disparate!
at Assert2.main(Assert2.java:3)

Asserções em Java: Exemplos

Correcção

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

```
1 public class Assert1 {  
2     public static void main(String[] args) {  
3         assert false;  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError
at Assert1.main(Assert1.java:3)

```
1 public class Assert2 {  
2     public static void main(String[] args) {  
3         assert false: "disparate!";  
4     }  
5 }
```

Exception in thread "main" java.lang.AssertionError: disparate!
at Assert2.main(Assert2.java:3)

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (javadoc) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em `Java` nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

Tipos de Dados
Abstratos

Abordagens
Sistemáticas à
Programação

Testando o programa por
fora

Testando o programa por
dentro

Associando um significado
aos programas

Formalizando uma
Especificação

Programação por Contrato

Programação por Contrato
em Java