3

PROJETO ORIENTADO AO OBJETO

Paradigmas de Programação

LEI - ISEP

Objetivos

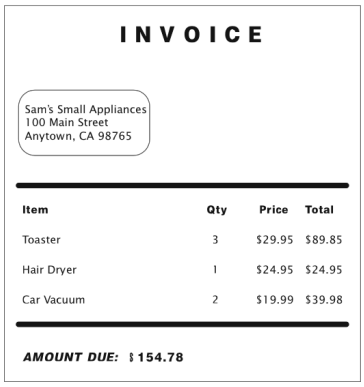
- Aprender a identificar novas classes e métodos
- Compreender os conceitos de coesão e de acoplamento (*coupling*)
- Identificar relações de herança, composição e dependência entre classes
- Utilizar *packages* para organizar aplicações

Conteúdos

- Classes e suas responsabilidades
- Relações entre classes (herança, composição e dependência)
- Packages

Classes e suas Responsabilidades

- Para identificar novas classes, analisar os substantivos existentes na descrição do problema
- Exemplo: Impressão de um recibo
  - Classes candidatas:
    - `Recibo`
    - `Item transaccionado`
    - `Cliente`



INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	$29.95	$89.85
Hair Dryer	1	$24.95	$24.95
Car Vacuum	2	$19.99	$39.98

AMOUNT DUE: $154.78

Classes e suas Responsabilidades (2)

- Conceitos presentes no domínio do problema são bons candidatos para classes
  - Exemplos:
    - Física: `Projétil`
    - Negócios: `CaixaRegistadora`
    - Jogo: `Personagem`
- O nome de uma classe deve descrever a classe

Coesão (1)

- Uma classe deve representar um único conceito
- A interface pública de uma classe é **coesa** se todas as suas características estão relacionadas com o conceito que a classe representa

Coesão (2)

- Esta classe não é coesa:

```
public class CashRegister
{
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   …
   public void enterPayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   . . .
}
```

- Envolve dois conceitos: *caixa registadora* e *moeda*

Coesão (3)

- Melhor alternativa: Definir duas classes:

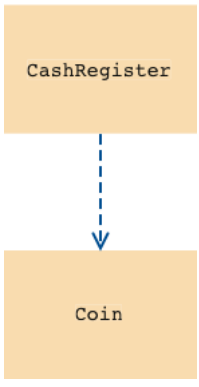
```
public class Coin
{
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}

public class CashRegister
{
   public void enterPayment(int coinCount, Coin coinType)
   { . . . }
   . . .
}
```

Relações entre Classes

- Uma classe **depende** de outra se usa objetos dessa classe
  - relação “conhece”
- `CashRegister` depende de `Coin` para determinar o valor do pagamento
- Visualização de relações: diagramas de classe
- **UML**: Unified Modeling Language
  - Notação para análise e projeto orientado ao objeto

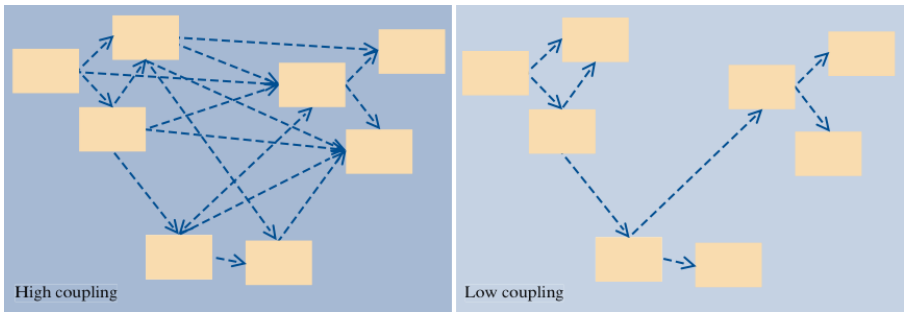
Relação de Dependência



Acoplamento (*Coupling*) (1)

- Se o nível de dependência entre classes é grande, o **acoplamento** entre classes é elevado
- Boa prática: minimizar o acoplamento entre classes
  - Alteração numa classe pode requerer a atualização de todas as classes acopladas
  - Utilizar uma classe numa outra aplicação requer utilizar todas as classes das quais ela depende

Acoplamento (*Coupling*) (2)

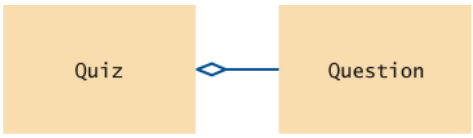


Associação

- Na programação orientada ao objeto, um objeto está relacionado com outro para usar funcionalidades e serviços fornecidos por esse objeto
- Esta relação entre dois objetos é conhecida como associação
- Composição e Agregação são formas de associação entre dois objetos

Agregação (1)

- Uma classe **agrega** outra se os seus objetos contêm objetos de outra classe
  - relação "has-a"
  - Exemplo: um teste (*quiz*) é formado por questões
  - A classe `Quiz` agrega a classe `Question`



Agregação (2)

- A identificação de relações de agregação ajuda na implementação de classes
- Exemplo: uma vez que um teste pode ter qualquer número de questões, usar um vetor para os colecionar:

```
public class Quiz
{
   private ArrayList<Question> questions;
   . . .
}
```

Agregação (3)

- Os objetos agregados podem existir sem serem parte de um objeto principal
- Exemplos:
  - Uma questão pode existir sem pertencer a um teste
  - Aluno numa Escola, quando a Escola encerra, o Aluno continua a existir e pode ingressar noutra Escola

Agregação (4)

- Uma vez que *Organization* tem *Person* como *employees*, a relação entre eles é a Agregação

```
public class Organization {
    private List employees;
}

public class Person {
    private String name;
}
```

Composição (1)

- Referimo-nos à associação entre dois objetos como composição, quando uma classe possui outra classe e a existência desta outra classe não faz sentido, quando o seu "dono" é destruído
- Relação "part-of"
- Exemplo: a classe Humano é uma *composition* de várias partes do corpo incluindo mão, braço e coração
  - Quando o objeto Humano morre, a existência de todas as partes do corpo deixam de ter utilidade

Composição (2)

- Outro exemplo de Composição é Car e as suas partes (motor, rodas, ...) – as partes individuais de um carro não funcionam isoladamente quando o carro é destruído

```
public class Car {
    //final will make sure engine is initialized
    private final Engine engine;

    public Car(){
       engine  = new Engine();
    }
}

class Engine {
    private String type;
}
```

Agregação e Composição

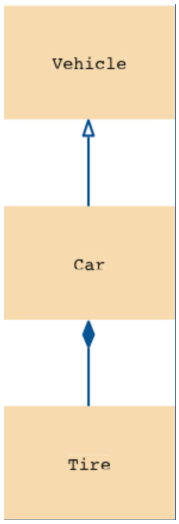
- São formas de associação

Herança (1)

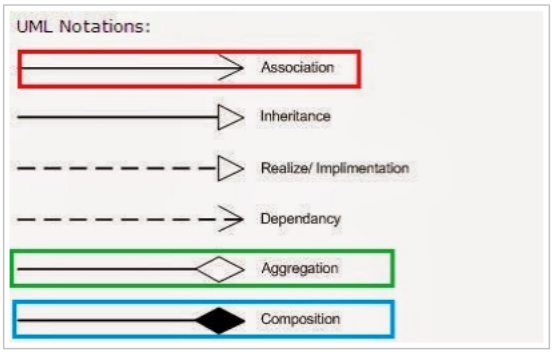
- **Herança** é uma relação entre uma classe mais genérica (**superclass**) e uma classe mais especializada (**subclass**)
  - Relação "is-a"
- Exemplo: qualquer carro *is a* veículo; qualquer carro tem pneus
  - A classe `Car` é uma subclasse da classe `Vehicle`; a classe `Car` agrega a classe `Tire`

Herança (2)

```
public class Car extends Vehicle
{
   private Tire[] tires;
   . . .
}
```



UML: Relações entre Objetos



Nested Classes

- Uma *nested class* é uma classe definida no interior de outra classe (*outer class*)
- Utilidade:
  - Permite agrupar classes que são usadas num único lugar – são utilizadas apenas por uma classe
  - Aumenta o encapsulamento – consideremos duas classes independentes A e B, em que B tem que aceder aos membros de A que normalmente seriam privados; escondendo B dentro de A, os membros de A podem ser declarados privados e B pode aceder-lhes
  - Conduz a código mais legível e de mais fácil manutenção – as classes interiores ficam junto ao local onde são usadas

Nested Classes

- Dividem-se em duas categorias:
  - Estáticas (declaradas com *static*, denominadas *static nested classes*)
  - Não estáticas (são chamadas *inner classes*)
- Uma *nested class* é um membro da classe que a contém
- *Non-static nested* classes (*inner classes*) têm acesso a outros membros da classe exterior, mesmo que declarados como privados
- *Static nested classes* não têm acesso a membros da classe exterior
- Como membro de uma classe exterior, uma *nested class* pode ser declarada `private`, `public`, `protected` ou *package private* (Uma classe exterior apenas pode ser declarada como `public` ou *package private*)

Static Nested Classes

- Tal como os métodos e variáveis de classe, uma *static nested class* está associada à sua *outer class*
- Tal como os métodos de classe estáticos, uma *static nested class* não tem acesso direto a variáveis e métodos de instância definidos na sua *outer class*: pode usá-los apenas através de uma referência de um objeto
- *Static nested classes* são acedidas usando o nome da *outer class*: `OuterClass.StaticNestedClass`
- Por exemplo, para criar um objeto da static nested class:

```
OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();
```

Inner Classes (1)

- Tal como os métodos e variáveis de instância, uma *inner class* está associada a uma instância da sua *outer class* e tem acesso direto aos métodos e atributos do objeto
- Pelo facto de uma *inner class* estar associada a uma instância, não pode definir membros estáticos
- Objetos que são instâncias de uma *inner class* existem no interior de uma instância da *outer class*

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Inner Classes (2)

- Uma instância de uma *Inner Class* apenas pode existir no interior de uma instância da *Outer Class* e tem acesso direto aos métodos e atributos da instância que a contém
- Para instanciar uma *Inner Class*, primeiro é necessário instanciar a *Outer Class*; Depois, criar o *inner object* dentro do *outer object*:

```
OuterClass.InnerClass innerObject = outerObject.new
InnerClass();
```

- Existem dois tipos adicionais de *Inner Classes*:
  - classes locais
  - classes anónimas

Classes Locais

- As Classes Locais são classes definidas num bloco, o qual é um grupo de zero ou mais instruções entre chavetas
  - Tipicamente encontram-se classes locais definidas no corpo de um método, num ciclo *for* ou numa instrução *if*
- Uma classe local tem acesso aos membros da sua *Outer Class*
- Uma classe local tem ainda acesso às variáveis locais (declaradas no mesmo bloco)
  - No entanto, uma classe local apenas pode aceder a variáveis locais se estas forem declaradas como *final*

Classes Anónimas (1)

- As classes anónimas permitem tornar o código mais compacto; permitem declarar e instanciar uma classe ao mesmo tempo
- São como as classes locais exceptuando o facto de não terem nome
- Devemos usá-las quando precisamos de usar uma classe local uma única vez
- Uma classe anónima é uma expressão (a sua declaração termina com ;)

Classes Anónimas (2)

```
public class AnonymousClasses {
 interface HelloWorld {
  public void greet();
  public void greetSomeone(String someone);
 }

 public void sayHello() {
  class EnglishGreeting implements
            HelloWorld {
   String name = "Todo o mundo";

   public void greet() {
    greetSomeone("world");
   }
   public void greetSomeone(String someone) {
    name = someone;
    System.out.println("Hello " + name);
   }
  }
  HelloWorld englishGreeting =
            new EnglishGreeting();
```

Classe Local que implementa a interface HelloWorld

```
  HelloWorld frenchGreeting = new HelloWorld() {
   String name = "Todo o mundo";
   public void greet() {
    greetSomeone("tout le monde");
   }
   public void greetSomeone(String someone) {
    name = someone;
    System.out.println("Salut " + name);
   }
  };
```

Classe Anónima que implementa a interface HelloWorld

```
  englishGreeting.greet();
  englishGreeting.greetSomeone("John");
  frenchGreeting.greet();
  frenchGreeting.greetSomeone("Claude");
 }

 public static void main(String... args) {
  AnonymousClasses myApp =
            new AnonymousClasses();
  myApp.sayHello();
 }
}
```

Classes Anónimas (3)

- Sintaxe: a expressão que define uma classe anónima é constituída por:
  - Operador *new*
  - O nome de uma interface a implementar ou uma classe a estender
  - Parênteses contendo os argumentos para um construtor, tal como numa expressão para criação de uma instância de uma classe normal
    - Nota: Quando se implementa uma interface, como não existe construtor, usa-se um par de parênteses vazio, tal como no exemplo
  - Um corpo, contendo a declaração do corpo da classe: no corpo são permitidas declarações de métodos, mas instruções não são permitidas

Classes Anónimas (4)

- Uma classe anónima tem acesso aos membros da sua *Outer Class*
- Uma classe anónima não tem acesso às variáveis locais do bloco onde se situa, a não ser que estas sejam declaradas como `final`
- As classes anónimas têm as mesma restrições que as classe locais no que diz respeito aos seus membros:
  - Não é possível declarar inicializadores estáticos ou interfaces numa classe anónima
  - Uma classe anónima pode ter membros estáticos desde que sejam variáveis constantes
- É possível declarar os seguintes elementos:
  - Atributos, métodos, inicializadores de instância, classes locais
- Não é possível declarar construtores

Packages

- ***Package***: um conjunto de classes relacionadas
- Alguns *packages* da biblioteca Java:

Package	Função	Exemplo
`java.lang`	Suporte à linguagem Java	`Math`
`java.util`	Classes e interfaces utilitárias	`Random`
`java.io`	*Input* e *output*	`PrintStream`
`java.awt`	Abstract Windowing Toolkit (gráficos 2D e *user interfaces* simples	`Color`
`java.applet`	Applets	`Applet`
`java.net`	*Networking*	`Socket`
`java.sql`	Acesso a base de dados	`ResultSet`
`javax.swing`	Swing *user interface*	`JButton`
`omg.w3c.dom`	Document Object Model para documentos XML	`Document`

Organização em *Packages* de Classes Relacionadas (1)

- Para incluir uma classe num *package*, inserir

```
package packageName;
```

como a primeira instrução do seu código

- O nome do *package* consiste em um ou mais identificadores separados por pontos

Organização em *Packages* de Classes Relacionadas (2)

- Por exemplo, para inserir a classe `BankAccount` class num *package* com o nome `com.horstmann`, o ficheiro `BankAccount.java` deve começar da seguinte forma:

```
package com.horstmann;

public class BankAccount
{
   . . .
}
```

- O **package por defeito (default package)** não tem nome, não se usando a instrução `package`

Importação de Packages

- É possível usar uma classe sem importá-la:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- No entanto não é prático especificar o seu nome completo
- A instrução *import* permite usar o nome curto da classe:

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
```

- É possível importar todas as classes de um *package*:

```
import java.util.*;
```

- Não é necessário importar classes do *package* `java.lang`
- Não é necessário importar outras classes pertencentes ao mesmo *package*

Nomes dos *Packages*

- Utilização de *packages* para evitar conflitos de nomes

```
java.util.Timer
```

vs.

```
javax.swing.Timer
```

- Recomendação: começar com o nome do domínio por ordem invertida:

```
pt.isep
```

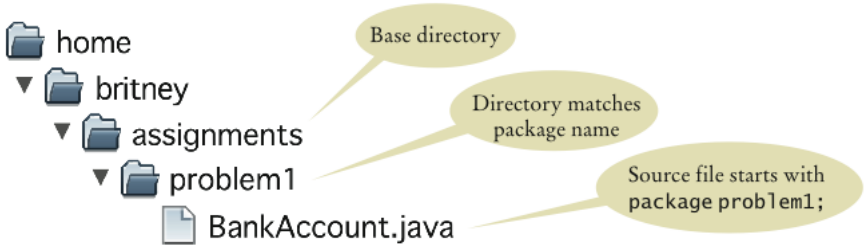
- `pt.ipp.isep.dei.esteves`: para as classes do Esteves (`esteves@dei.isep.ipp.pt`)

Estrutura de diretórios das Classes

- **Diretório base:** diretório que contém os ficheiros fonte dos nossos programas
- O caminho do ficheiro fonte de uma classe, relativo ao diretório base, deve corresponder ao nome do seu *package*
- Exemplo: se o diretório base é

  `/home/britney/assignments`

  colocar os ficheiros fonte das classe do *package* `problem1` no diretório:



Resumo

Identificar classes e suas responsabilidades

- Para identificar classes, procurar por substantivos na descrição do problema
- Conceitos do domínio do problema são bons candidatos de classes
- A interface pública de uma classe é coesiva se todas as suas características estão relacionadas com o conceito representado pela classe

Resumo

Relações entre classes e diagramas UML

- Uma classe depende de outra se utiliza objetos dessa classe
- A redução de dependências entre classes (*coupling*) é uma boa prática
- Uma classe agrega outra se os seus objetos contêm objetos de outra classe
- A relação de herança (relação *is-a*) é por vezes utilizada de forma inapropriada quando a relação *has-a* seria mais apropriada
- A relação de agregação (relação *has-a*) significa que objetos de uma classe contêm referências para objetos de outra classe

Resumo

Processo de desenvolvimento orientado ao objeto

- Iniciar o processo de desenvolvimento pela obtenção e documentação dos requisitos da aplicação
- Identificar classes, responsabilidades e colaboradores
- Usar diagramas UML para registar as relações entre classe
- Usar comentários `javadoc` (com o corpo dos métodos ainda vazios) para registar o comportamento das classes
- Após completar o projeto, passar à implementação das classes

Resumo

Nested Classes

- *Innerclasses* existem no seio de uma *Outerclass*
- Dois tipos de *nested classes:*
  - *static nested classes*
  - *innerclasses*
    - Estas compreendem ainda dois tipos adicionais de classes*:*
      - *local classes*
      - *anonymous classes*

Resumo

Packages

- Um *package* é um conjunto de classes relacionadas
- Usar packages para estruturar as classes na aplicação
- A diretiva `import` permite fazer referência a uma classe de um *package*, sem utilizar o prefixo do *package*