INSTITUTO DO EMPREGO
E FORMAÇÃO PROFISSIONAL

UPskill – Java

Linguagem de Programação Java – Projeto Orientado ao Objeto

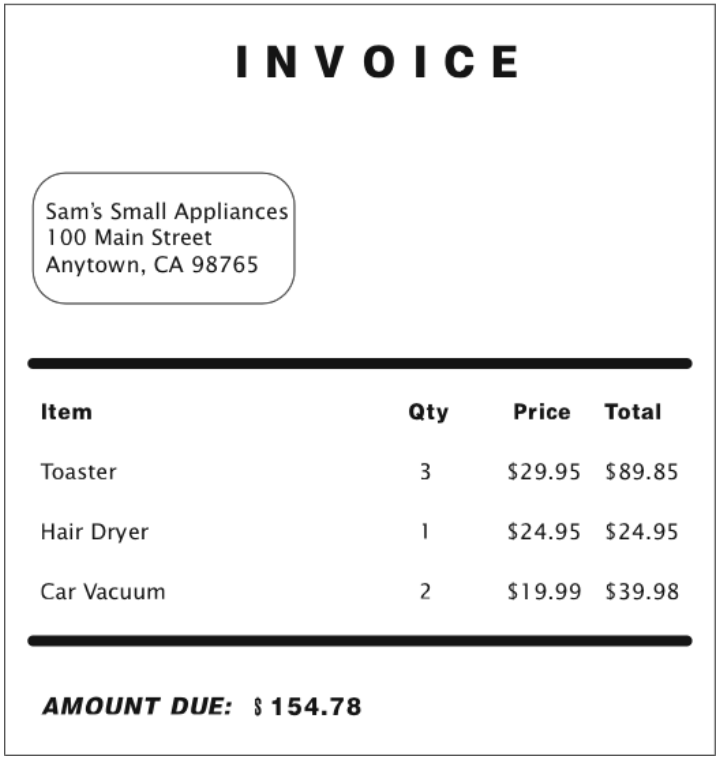
Adaptado de Donald W. Smith (TechNeTrain)

Conteúdos

- Classes e suas responsabilidades
- Relações entre classes:
  - Dependência
  - Associação
  - Agregação
  - Composição
  - Herança

Classes e suas Responsabilidades

- Para identificar novas classes, devemos analisar os substantivos existentes na descrição do problema
- Exemplo: Impressão de um recibo
  - Classes candidatas:
    - `Recibo`
    - `Item transacionado`
    - `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 escolhido para a 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)

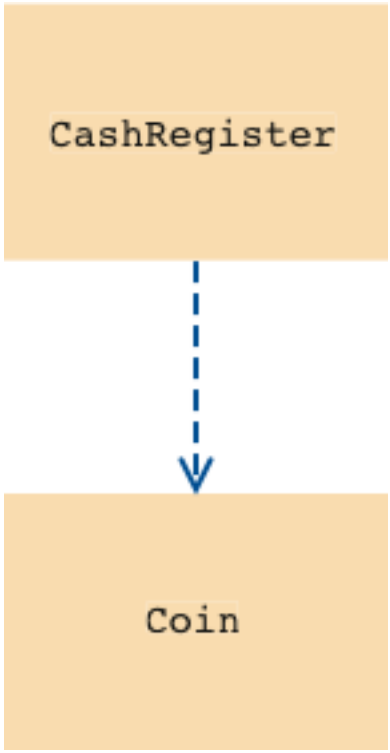
- Melhor alternativa: definir duas classes

```
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 utiliza 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

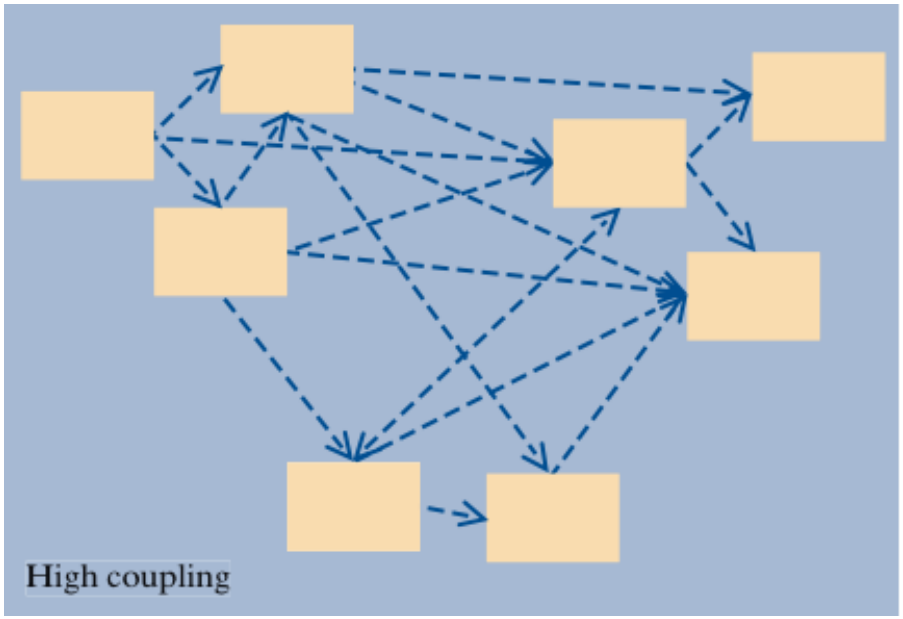
Relação de Dependência

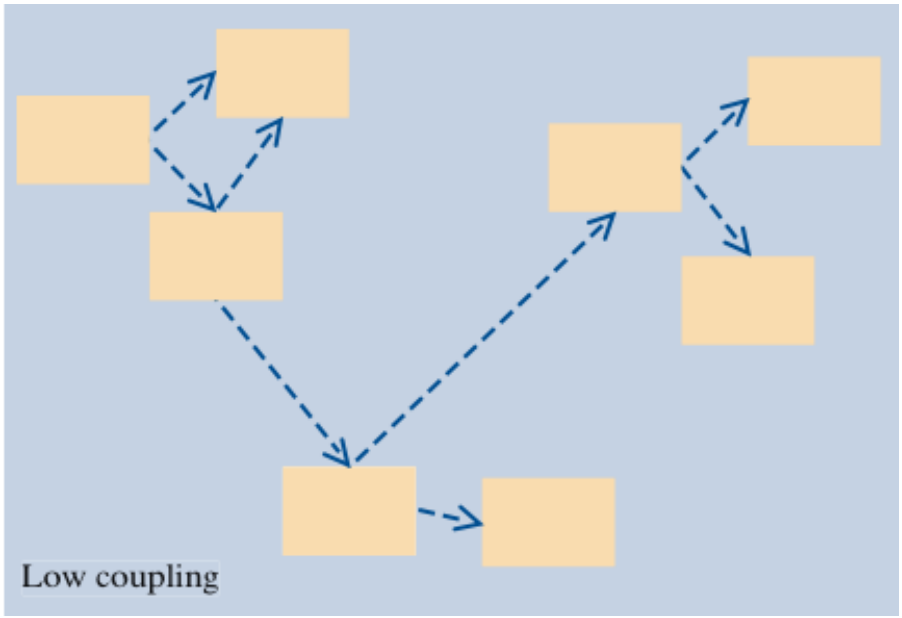


Acoplamento (*Coupling*) (1)

- 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)



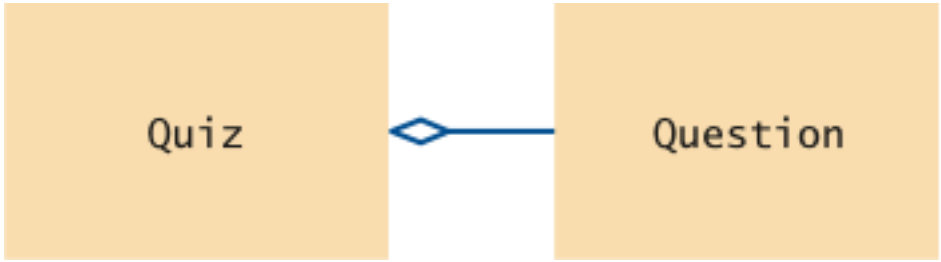


Associação

- Na programação orientada ao objeto, um objeto relaciona-se 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 array para os colecionar:

```
public class Quiz
{
   private 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 estas classes é de Agregação

```
public class Organization {
  private Person[] 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 *composição* 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

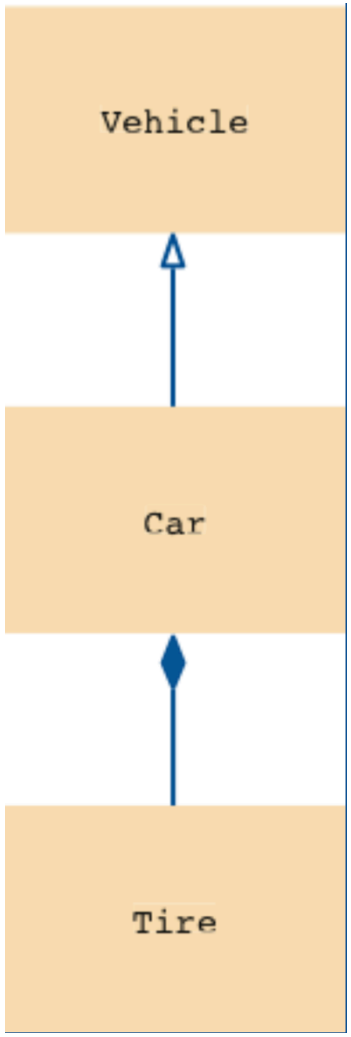
```
public class Car {
    // final will make sure engine is initialized
    private final Engine engine;

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

class Engine {
    private String type;
}
```

Herança (1)

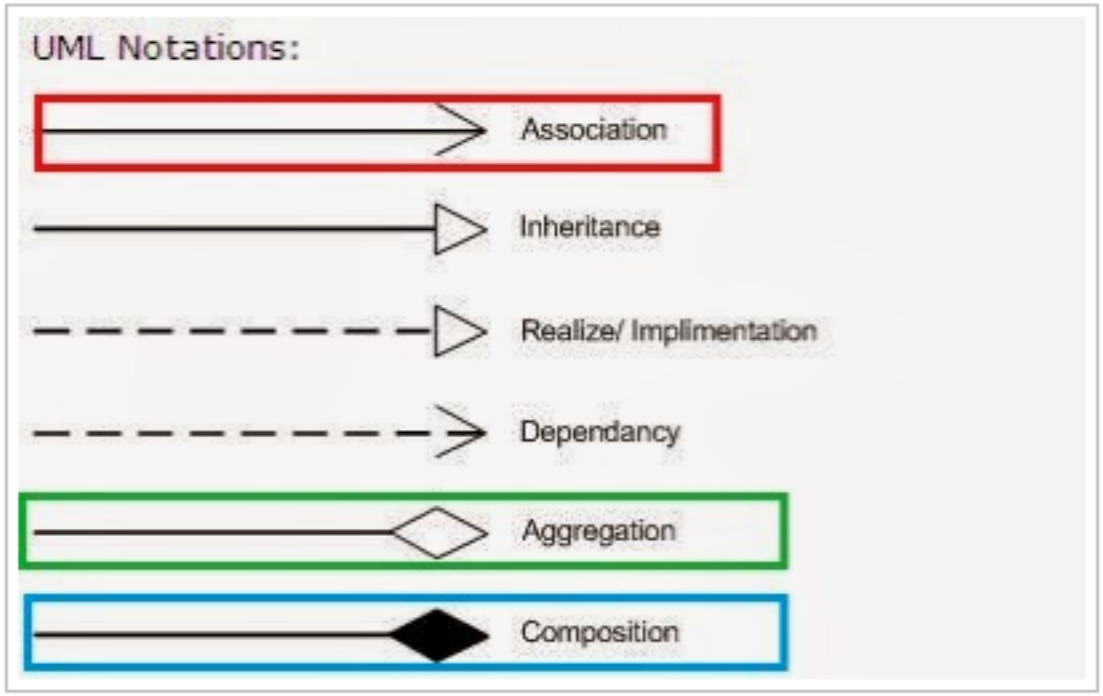
- **Herança** é uma relação entre uma classe mais genérica (**superclass**) e uma classe mais especializada (**subclass**)
  - Relação “is-a”
- Exemplo: qualquer carro *is-a* veículo; qualquer carro tem pneus
  - A classe `Car` é uma subclasse da classe `Vehicle`
  - A classe `Tire` é parte da classe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

Herança (2)

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



UML: Relações entre Objetos



Resumo

Identificação de 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 coesa 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 dependência entre classes (*coupling*) é uma boa prática
- Agregação: quando os objetos agregados podem existir sem serem parte de um objeto principal
- Composição: quando uma classe possui outra classe e a existência desta outra classe não faz sentido quando o seu "dono" é destruído

Resumo

Processo de desenvolvimento orientado ao objeto

- Iniciar o processo de desenvolvimento pela obtenção e documentação dos requisitos da aplicação
- Identificar classes e responsabilidades
- Usar diagramas UML para registar as relações entre classes
- 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