

# INF319 — Projeto e Implementação Orientados a Objetos

## Projeto Orientado a Objetos: Lista de Materiais

Luiz E. Busato

Instituto de Computação – UNICAMP  
buzato@ic.unicamp.br

Especialização em Engenharia de Software

# Roteiro

- 1 Especificação
- 2 Primeira Iteração
  - Análise
  - Projeto e Implementação
- 3 Segunda Iteração
  - Análise
  - Projeto e Implementação

# Lista de Materiais: Especificação

## Lista de Materiais

Uma lista de materiais é uma lista dos componentes necessários para a montagem de um item final a ser fabricado.

## Componentes

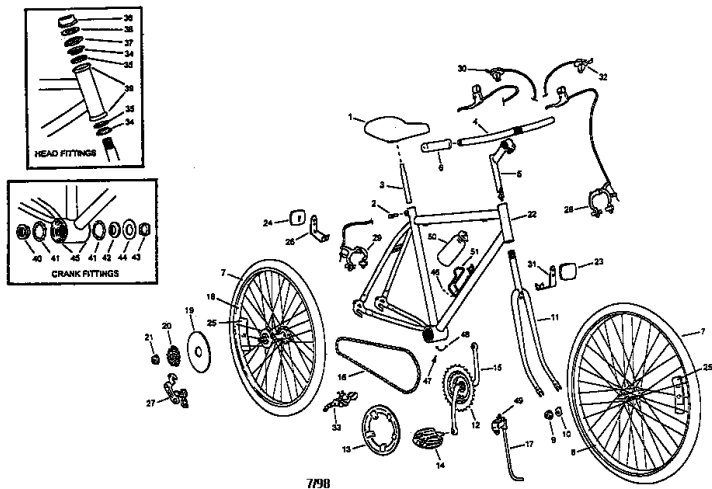
Todo componente é identificado unicamente e tem uma descrição e um custo. Um componente pode ser um componente atômico ou um componente composto, que possui a sua própria lista de materiais.

# Exemplo

## Bicicleta

Velocípede de duas rodas, de igual diâmetro, sendo a traseira acionada por um sistema de tração composto por pedais com múltiplas coroas e corrente que atua sobre múltiplos pinhões. Combinações diferentes de coroa e pinhão podem ser selecionadas utilizando um câmbio.

# Exemplo



# Desejo do Cliente/Caso de Uso

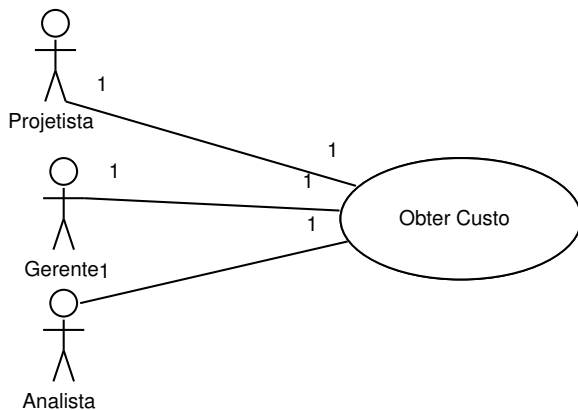
## Oportunidade

Uma indústria está analisando a possibilidade de contratar os seus serviços para elaborar um software para gerenciar completamente as listas de materiais de seus produtos, desde a sua elaboração, passando pela sua manutenção e pelo seu emprego nas linhas de montagem.

## Problema

Para avaliar a sua capacidade de levar o projeto adiante, a indústria contratou apenas o projeto de um caso de uso: **obter o custo total de uma lista de materiais**. Faça o projeto deste caso de uso, abstraindo outras componentes do sistema, e impressione o cliente.

# Caso de Uso



# Estórias

- Usuário deve ser capaz de calcular o custo total de uma lista de materiais.
- Usuário deve ser capaz de inserir um item na lista de materiais.
- Usuário deve ser capaz de remover um item da lista de materiais.

# Determinação das Classes

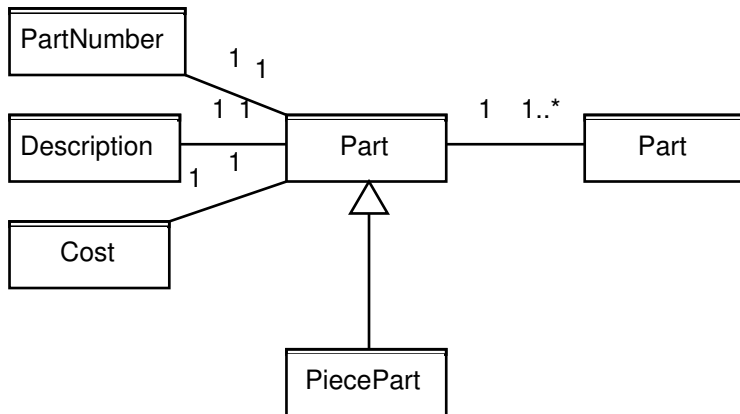
## Onde estão os objetos?

- Componente composto (Part)
- Componente atômico (PiecePart)
- Identificador único (PartNumber)
- Descrição (Description)
- Custo (Cost)

Vamos fazer o primeiro diagrama de classes sem nos preocuparmos com atributos e métodos, simplesmente associando os objetos identificados.

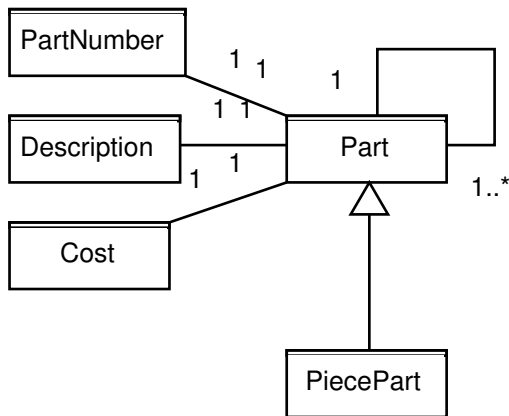
# Diagrama de Classes

Com o foco nas associações, ligamos todas as classes, mas há algo estranho...



# Diagrama de Classes, Corrigido

Não precisamos duplicar classes para representar auto-associações.



# Detalhamento dos Objetos

## Métodos e Atributos

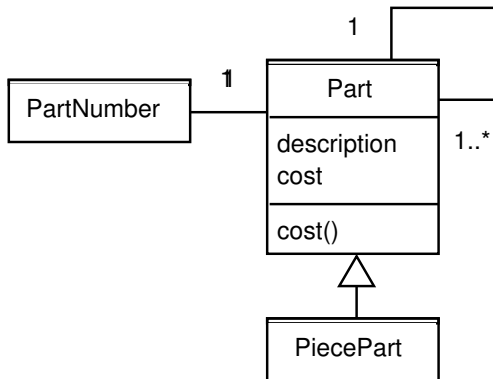
**Atributos:** descrição e custo.

**Métodos:** `cost()`, obtém o custo total deste componente, incluindo sub-componentes.

Há vários métodos de inspeção e ou modificação de atributos que não estão detalhados. Por exemplo, `GetDescription()` ou `SetDescription()`.

O identificador único permanece como associação. Isto não é estritamente necessário para atender o caso de uso, mas como não sabemos como este identificador é implementado, isolamos a nossa classe deste detalhe.

# Diagrama de Classes



# Implementação dos Atributos

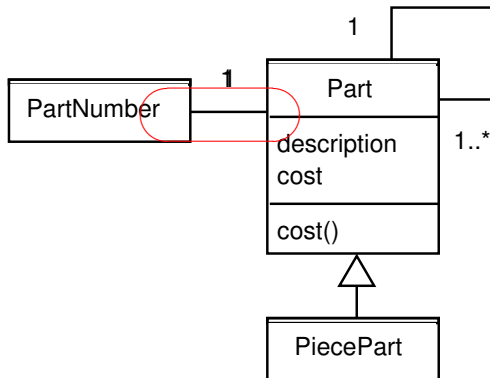
```
public class Part {  
    private String description;  
    private double cost;  
    private final PartNumber partNumber;  
}
```

## Observe

- Os atributos são **privados**.
- O identificador único é **constante**.

# Atributo ou Associação?

Do ponto de vista da modelagem, existe uma diferença significativa entre atributo e associação. Mas, não na implementação.



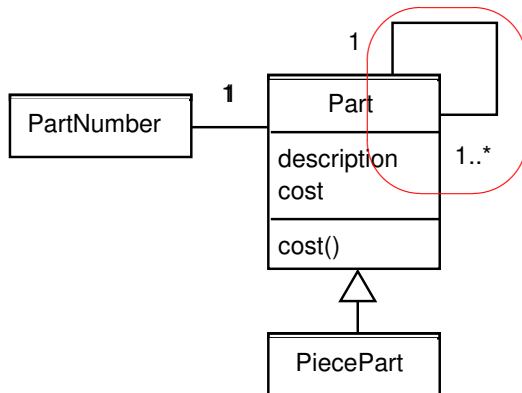
# Implementação

```
private final PartNumber partNumber;  
public Part(PartNumber thePartNumber,  
            String theDescription) {  
    description = theDescription;  
    partNumber = thePartNumber;  
    ...  
}
```

## Observe

- Associações são estabelecidas durante a construção do objeto.
- Apesar de modelados de forma diferente, não existe diferença na forma como a descrição e o identificador únicos são implementados.

# Cardinalidade



# Implementação

```
private Set parts;  
public Part(PartNumber thePartNumber,  
            String theDescription) {  
    ...  
    parts = new HashSet();  
}
```

## Observe

- Para implementar associações com cardinalidade  $1 - n$  ou  $n - n$  pode-se usar um objeto recipiente, como um conjunto em Java.

# Herança

```
public class PiecePart extends Part {  
    public PiecePart(PartNumber thePartNumber,  
                    String theDescription) {  
        super(thePartNumber, theDescription);  
    }  
}
```

## Observe

- A subclasse delega o trabalho de início a implementação da classe pai.
- Nesta modelagem, esta classe não implementa métodos. Isto é um indício de que algo está errado...

# Cálculo do Custo

```
public double cost() {  
    double totalCost = 0;  
    if (this instanceof PiecePart) {  
        totalCost = cost;  
    } else {  
        for (Iterator i = parts.iterator(); i.hasNext();) {  
            totalCost += ((Part) i.next()).cost();  
        }  
    }  
    return totalCost;  
}
```

## Observe

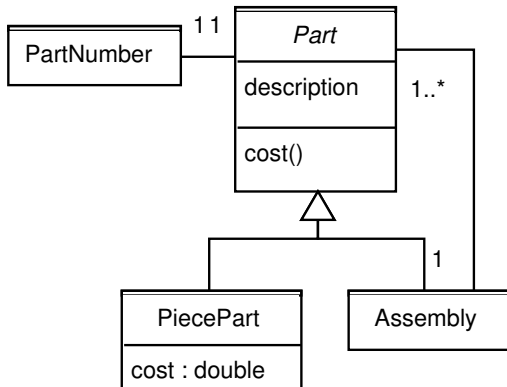
- O mesmo método funciona e integra as duas classes.
- O operador instanceof testa a classe de um objeto.

# Revisão da Análise

## Lições Aprendidas

- A experiência obtida na primeira iteração de análise, projeto e implementação indica que o projeto pode ser melhorado se a associação entre as partes for repensada:
  - A classe PiecePart não implementa métodos.
  - O mesmo método serve a duas classes distintas.
- A introdução de uma abstração específica para uma **montagem** (componente composto) resolve o problema.

# Novo Projeto



# Superclasse Abstrata

```
public abstract class Part {  
    private String description;  
    private final PartNumber partNumber;  
    public Part(PartNumber thePartNumber,  
                String theDescription) {  
        description = theDescription;  
        partNumber = thePartNumber;  
    }  
    public abstract double cost();  
}
```

## Observe

- A classe Part é **abstrata**, e possui um método abstrato `cost()` que deve ser implementado pelas subclasses concretas.

# Componente Atômico

```
public class PiecePart extends Part {  
    private double cost;  
    public PiecePart(PartNumber thePartNumber,  
                    String theDescription,  
                    double theCost) {  
        super(thePartNumber, theDescription);  
        cost = theCost;  
    }  
    public double cost() { ... }  
}
```

## Observe

- As associações e métodos relevantes a uma peça atômica foram movidos para a classe PiecePart.

# Componente Atômico

```
public class PiecePart extends Part {  
    ...  
    public double cost() {  
        return cost;  
    }  
}
```

## Observe

- A implementação de `cost()` de `PiecePart` usa o custo associado a uma parte atômica.

# Componente Composto

```
public class Assembly extends Part {  
    private Set parts;  
    public Assembly(PartNumber thePartNumber,  
                    String theDescription) {  
        super(thePartNumber, theDescription);  
        parts = new HashSet();  
    }  
    public double cost() { ... }  
}
```

## Observe

- As associações e métodos relevantes a composição foram movidas para a classe Assembly.

# Componente Composto

```
public class Assembly extends Part {  
    ...  
    public double cost() {  
        double totalCost = 0;  
        for (Iterator i = parts.iterator(); i.hasNext();) {  
            totalCost += ((Part) i.next()).cost();  
        }  
        return totalCost;  
    }  
}
```

## Observe

- A implementação de `cost()` de `PiecePart` agrega o custo de seus componentes, sem se importar com o seu tipo.
- Design Pattern: Composite (<http://www.vincehuston.org/dp/composite.html>)