


Laboratório de Engenharia de Software

# INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho  
[ivan@inf.puc-rio.br](mailto:ivan@inf.puc-rio.br)





Laboratório de Engenharia de Software

## Programa – Capítulo 7

- Herança vs. Composição
- Interface
- Aplicação de Interface
- Exercício – Fila com uso de composição
- Exercício – Fila com uso de interface

© LES/PUC-Rio

Laboratório de Engenharia de Software	<b>Programa – Capítulo 7</b>	
	<ul style="list-style-type: none"><li>• <b>Herança vs. Composição</b></li><li>• Interface</li><li>• Aplicação de Interface</li><li>• Exercício – Fila com uso de composição</li><li>• Exercício – Fila com uso de interface</li></ul>	
© LES/PUC-Rio		

Laboratório de Engenharia de Software	<b>Herança vs. Composição (1)</b>	
	<ul style="list-style-type: none"><li>• A herança é um mecanismo de reutilização caixa branca, pois, frequentemente, expõe a estrutura das classes ancestrais;</li><li>• A herança é um mecanismo estático, não permitindo, assim, a reconfiguração dinâmica de um sistema;</li><li>• Ela aumenta o acoplamento entre uma classe e suas classes ancestrais;</li><li>• Toda uma hierarquia de classes tem de ser incluída no caso de reutilização.</li></ul>	
© LES/PUC-Rio		

## Herança vs. Composição (2)



Laboratório de Engenharia de Software

- A composição permite obter funcionalidades complexas por meio da colaboração de vários objetos que oferecem funções mais simples;
- É um mecanismo de reutilização caixa preta, pois os aspectos internos dos objetos não precisam ser expostos;
- Permite a reconfiguração dinâmica de um sistema.

© LES/PUC-Rio

## Herança vs. Composição (3)



Laboratório de Engenharia de Software

- A composição aumenta as chances da reutilização de classes;
- Ela favorece a criação de classes menores e mais coesas;
- Seu poder aumenta quando usada em conjunto com o polimorfismo;
- Para tal, prefira herança de interface em vez de herança de implementação.

© LES/PUC-Rio

## Exemplo



Seja a classe abaixo:

```
public class Lista {

    public boolean vazio();
    public boolean insIni(Object x); // insere x no inicio da lista
    public boolean insFin(Object x); // insere x no final da lista
    public Object retIni(); // remove o primeiro elemento da lista
    public Object retFin(); // remove o último elemento da lista
    public void posIni(); // posiciona o cursor no inicio da lista
    public Object prox(); // retorna o próximo elemento da lista

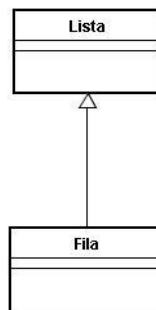
}
```

Construa uma classe **Fila** reutilizando a classe acima.

## Solução – Generalização



- Uma análise superficial do problema poderia levar à conclusão de que a generalização e a herança de classe seriam uma boa solução, como é mostrado na figura a seguir:



## Generalização – Problemas



Laboratório de Engenharia de Software

- Uma generalização, entretanto, pressupõe a existência de uma relação “é-um”;
- Entretanto, uma **fila** não é uma **lista**, embora algumas das propriedades de **listas** se apliquem a **filas**;
- A operação **insIni**, por exemplo, é aplicável a **listas**, mas não a **filas**, que, por definição, só podem ter novos elementos inseridos ao final das mesmas.

© LES/PUC-Rio

## Solução – Composição



Laboratório de Engenharia de Software

- Criar uma composição entre uma **Fila** e uma **Lista**;
- Isto é, fazer com que uma **Fila** seja composta por uma **Lista**;
- O programador irá visualizar apenas os métodos públicos da classe **Fila**;
- A **Fila** não irá implementar as operações;
- Ela irá delegar a execução de tais operações à **Lista** que ela contém.

© LES/PUC-Rio

## Composição



```
package Fila;

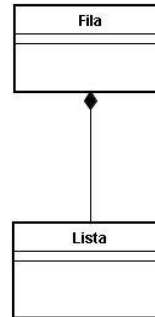
import Lista.*;

public class Fila {
    Lista ls=new Lista();

    public boolean enqueue(Object x) {
        return ls.insFin(x);
    }

    public Object dequeue() {
        return ls.retIni();
    }

    public boolean vazio() {
        return ls.vazio();
    }
}
```



© LES/PUC-Rio


## Delegação



- A **delegação** é uma maneira de tornar a composição um mecanismo de reutilização extremamente poderoso;
- Na **delegação**, um objeto recebe uma solicitação e delega a sua execução a um ou mais objetos;
- O objeto receptor atua, frequentemente, como coordenador da execução de uma solicitação;
- Por isso, muitas vezes é necessário que os objetos delegados consultem o estado do objeto receptor (**callback**);
- Para tal, o objeto receptor passa uma referência para si próprio (**this**) quando envia mensagens para os objetos delegados.

© LES/PUC-Rio

**Programa – Capítulo 7**




Laboratório de Engenharia de Software

- Herança vs. Composição
- **Interface**
- Aplicação de Interface
- Exercício – Fila com uso de composição
- Exercício – Fila com uso de interface

© LES/PUC-Rio

**Interface**



Laboratório de Engenharia de Software

- Uma interface é uma construção similar a uma classe abstrata que contém apenas métodos abstratos;
- Da mesma forma que uma classe abstrata, uma interface não pode ser instanciada;
- Seu objetivo é declarar algumas operações que serão implementados por uma ou mais classes;
- Diferentemente de uma classe abstrata, uma interface não possui implementação, apenas declarações de operações (cabeçalhos) e constantes.

© LES/PUC-Rio

14

## Exemplo (1)



- A classe `java.util.Arrays` possui um método, chamado `sort`, que ordena um array de objetos;
- Para usá-lo é preciso, entretanto, que a classe dos elementos do array implemente a interface `Comparable`:

```
public interface Comparable {
    int compareTo(Object o);
}
```

- Isto é, uma classe deve implementar o método `compareTo` para implementar `Comparable` e, por conseguinte, usar o método `sort`.

## Exemplo (2)



```
public class Empregado implements Comparable{
    private String nome;
    private double salario;

    public Empregado(String n,double s){
        nome=n;
        salario=s;
    }
    public int compareTo(Object o){
        Empregado e=(Empregado)o;
        if(this.salario>e.salario)
            return 1;
        else
            if(this.salario<e.salario)
                return -1;
            else
                return 0;
    }
    String getNome(){
        return nome;
    }
    public double getSalario(){
        return salario;
    }
}
```



## Exemplo (3)



```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Empregado[] lst={new Empregado("Joao",50.0),new Empregado("Ana",30.0),
            new Empregado("Paula",100.0),new Empregado("Carlos",10.0)};

        Arrays.sort(lst);

        for(Empregado e:lst)
            System.out.println(e.getNome()+" "+e.getSalario());
    }
}
```

## Problema do TAD Fila

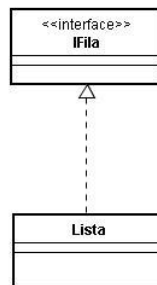


- Retornemos ao problema da criação de uma classe **Fila** a partir de uma classe **Lista** já existente;
- Já analisamos os efeitos decorrentes do uso de generalização na sua solução;
- Para contorná-los, criou-se uma composição entre a classe **Fila** e a classe **Lista** (uma **Fila** contém uma **Lista**);
- É, também, possível resolvê-lo usando-se uma interface.

## Solução – uso de uma interface (1)



- Definir uma interface chamada **IFila**;
- Fazer com que a classe **Lista** implemente a interface **IFila**;
- O uso da interface impede que o programador referencie as propriedades não aplicáveis a **filas**.



© LES/PUC-Rio

## Solução – uso de uma interface (2)



```

package Fila;

public interface IFila {
    public boolean vazio();
    public boolean insFin(Object x);
    public Object retIni();
}

-----

package Lista;
import Fila.*;

public class Lista implements IFila {
    public boolean vazio() { }
    public boolean insIni(Object x) { }
    public boolean insFin(Object x) { }
    public Object retIni() { }
    public Object retFin() { }
    public void posIni() { }
    public Object prox() { }
}
  
```

### Problema

Os nomes dos métodos da classe Lista têm de ser os mesmos das operações da interface IFila, embora os nomes **enqueue()** e **dequeue()** sejam os usualmente empregados.

Como resolver isso?

© LES/PUC-Rio

## Programa – Capítulo 7

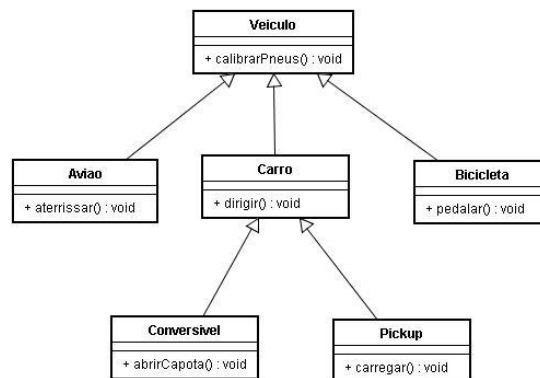


- Herança vs. Composição
- Interface
- **Aplicação de Interface**
- Exercício – Fila com uso de composição
- Exercício – Fila com uso de interface

## Problema (1)



- Seja a hierarquia de veículos mostrada a seguir:



## Problema (2)



- Todos os meios de transporte listados na hierarquia possuem pneus;
- Por isso, o método `calibrarPneus()` foi declarado no topo da hierarquia, sendo, dessa forma, herdado pelas demais subclasses;
- Apenas os aviões aterrissam. Logo, o método `aterrissar()` foi declarado na subclasse `Aviao`.

## Problema (3)



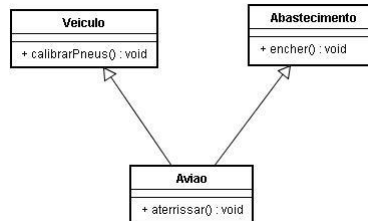
- Quase todos os veículos da hierarquia anterior podem ser abastecidos com combustível;
- Entretanto, se fosse definida uma implementação para tal na classe `Veiculo`, ela seria herdada por `Bicicleta`, que não pode ser abastecida;
- Por outro lado, definir métodos distintos para abastecimento nas classes `Aviao` e `Carro` introduziria uma redundância indesejável.

**Como resolver este problema?**

## Solução – Herança múltipla



- A linguagem C++ resolveria o problema com herança múltipla:



- A herança múltipla resolve alguns problemas, mas introduz outros. Por isso, Java oferece apenas herança simples.

## (Má) Solução - Um método para cada veículo



```

public class Posto {
    private double totGas=1000.00;
    private double totAlc=1000.00;
    private double totQrs=1000.00;

    public double reabastecer(Aviao a,TipoComb tipo,double qtd);
    public double reabastecer(Carro a,TipoComb tipo,double qtd);
}
  
```

**É indesejável que a classe Posto tenha de ser alterada sempre que um novo tipo de veículo que puder ser reabastecido seja inserido na hierarquia.**

## (Boa) Solução – A interface IUsaCombustivel



```
public interface IUsaCombustivel {  
    public double encher(TipoComb tipo, double qtd);  
}
```

## Um método para todos os veículos



```
public class Posto {  
    private double totGas=1000.00;  
    private double totAlc=1000.00;  
    private double totQrs=1000.00;  
  
    public double reabastecer(IUsaCombustivel v, TipoComb tipo,  
                             double qtd)  
    {  
    };  
}
```

## A classe Aviao implementa a interface



```
public class Aviao extends Veiculo implements IUsaCombustivel {
    private double capcTanque=5000.00;
    private double totComb=2000.00;

    public void aterrisar(){};
    public double encher(TipoComb tipo,double qtd) {
        if(tipo!=TipoComb.Querosene)
            return qtd;

        double falta=capcTanque-totComb;

        if(qtd>falta) {
            totComb=capcTanque;
            return qtd-falta;
        }
        else {
            totComb+=qtd;
            return 0.0;
        }
    }
}
```


## A classe Carro implementa a interface



```
public class Carro extends Veiculo implements IUsaCombustivel {
    private double capcTanque=50.00;
    private double totComb=20.00;

    public void dirigir(){};
    public double encher(TipoComb tipo,double qtd) {
        if(tipo!=TipoComb.Alcool &&
            tipo!=TipoComb.Gasolina)
            return qtd;
        double falta=capcTanque-totComb;
        if(qtd>falta) {
            totComb=capcTanque;
            return qtd-falta;
        }
        else {
            totComb+=qtd;
            return 0.0;
        }
    }
}
```

## Exemplo de reabastecimento




Laboratório de Engenharia de Software

```
Posto p=new Posto();  
Carro c=new Carro();  
  
p.reabastecer(c,TipoComb.Alcool,10.00);
```

© LES/PUC-Rio 31

## Considerações Finais (1)



Laboratório de Engenharia de Software


- Uma interface não pode ser instanciada, embora se possa declarar variáveis que se comportam como tal;
- Todas as operações definidas em um interface são públicas;
- Uma interface não possui variáveis de instância nem implementação de métodos (**exceto estáticos e default**);
- Todas as variáveis definidas em uma interface são tratadas como constantes estáticas (`public static final`);
- Uma classe pode implementar múltiplas interfaces.

© LES/PUC-Rio 32



Laboratório de Engenharia de Software

## Considerações Finais (2)




- Para que uma classe implemente uma interface deve-se
  - ✓ Declarar que a classe implementa (`implements`) a interface;
  - ✓ Fornecer uma implementação para cada operação declarada na interface.

© LES/PUC-Rio

33

Laboratório de Engenharia de Software

## Programa – Capítulo 7



- Herança vs. Composição
- Interface
- Aplicação de Interface
- **Exercício – Fila com uso de composição**
- **Exercício – Fila com uso de interface**

© LES/PUC-Rio