Encapsulamento

Prof. Hugo de Paula

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Departamento de Ciência da Computação

Sumário

Encapsulamento: ocultando informações

- Objetiva separar aspectos visíveis de um objeto ou classe de seus detalhes de implementação
- Interface:
  - tudo aquilo que o usuário do objeto vê/acessa.

Encapsulamento: ocultando informações

- Permite alterar a implementação de um objeto sem impactos em outros módulos do sistema.
- Permite que seus dados sejam protegidos de acesso ilegal.
- Em geral, desejamos ocultar determinados dados e/ou métodos do cliente/usuário da aplicação.

Ocultando informações

Exemplo:

- Acessar o campo quant e definir um estoque negativo pode invalidar o Produto.

Solução:

- Encapsulamento e interface pequena.

Modificadores de acesso

- Modificadores de acesso controlam a visibilidade dos componentes na aplicação.
- Ao nível da classe: **public** ou *package-private* (sem modificador explícito).
  - Classe declarada como **public** é visível a todas as classes do programa.
  - Classe sem modificador de acesso é visível apenas em seu pacote.
- Ao nível dos membros (atributos e métodos): **public**, **private**, **protected**, ou *package-private* (sem modificador explícito).

Modificadores de acesso

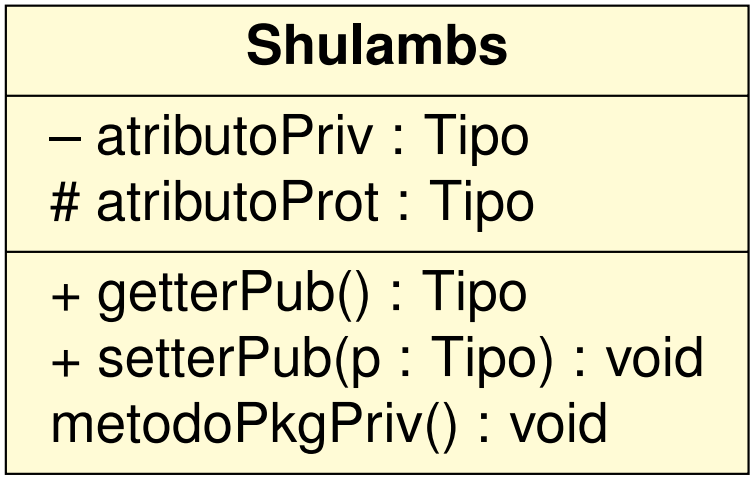
O Java possui 4 modificadores de acesso ao nível dos membros:

- **private**: membros declarados com acesso privado são acessíveis apenas na própria classe.
- *package-private*: membros declarados sem modificador de acesso são acessíveis apenas às classes dentro do mesmo pacote.
- **protected**: membros declarados com acesso protegido são acessíveis às classes do pacote e adicionalmente por suas subclasses.
- **public**: membros declarados com acesso público são acessíveis de qualquer lugar do programa.

Princípios da ocultação de informação

- Use o nível de acesso mais restrito e que faça sentido para um membro particular.
- Use **private** a menos que haja uma boa razão para não fazê-lo.
- Evite campos **public** exceto para constantes. Campos públicos aumentam o acoplamento em relação a uma implementação específica e reduz a flexibilidade do sistema a mudanças.

Encapsulamento na UML




```
class Shulambs {
    private Tipo atributoPriv;
    protected Tipo atributoProt;

    public Tipo getterPub() {
      ...
    }
    public void setterPub(Tipo p) {
      ...
    }

    void metodoPkgPriv() {
      ...
    }
}
```

Classe Produto: encapsulamento

```
public class Produto {
    private String descricao;
    private float preco;
    private int quant;

    public bool emEstoque() {
        return (quant > 0);
    }

    public Produto(String d, float p, int q)  {
        ...
    }

    public Produto() {
        ...
    }
}
```

Métodos de acesso (*getters* e *setters*)

- Métodos *get*: acessam o valor de um atributo privado.
  - Valores podem ser tratados antes de serem exibidos.
  - Ex: atributo booleano sendo exibido como V ou F atributo numérico e seu correspondente string.

  Métodos *set*: atribuem um valor a um atributo privado.
  - Valores devem ser validados/tratados antes de serem atribuídos.
  - Ex: número do `dia` numa classe Data depende do atributo `mes`.

Classe Produto: métodos de acesso (*getters* e *setters*)

```
public String getDescricao() { return descricao; }
public float getPreco() { return preco; }
public int getQuant() { return quant; }

public void setDescricao(String d) {
    if (d.length() >= 3)    descricao = d;
}
public void setPreco(float p) {
    if (preco > 0)   preco = p;
}
public void setQuant(int q) {
    if (quant >= 0)   quant = q;
}
public Produto(String d, float p, int q)
{
    setDescricao(d);
    setPreco(p);
    setQuant(q);
}
```

Classe Produto: acessando membros encapsulados

```
class Aplicacao {
    public static void main(String args[])
    {
        Produto p1 = new Produto();
        Produto p2 = new Produto("Shulambs,1.99F,200);

        p1.setDescricao("Cool Shulambs");
        p1.setPreco(2.49F);
        p1.setQuant(10);

        System.out.println("Produto: " + p1.getDescricao());
        System.out.println("Preço: " + p1.getPreco());
        System.out.println(" Estoque: " + p1.getQuant());

        System.out.println("Produto: " + p2.getDescricao());
        System.out.println("Preço: " + p2.getPreco());
        System.out.println(" Estoque: " + p2.getQuant());

    }
}
```

Quando não utilizar métodos de acesso

```
class Conta {
    private double limite;
    private double saldo;

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }
}
```

```
class Conta {
    private double saldo;
    private double limite;

    public Conta(double limite) {
        this.limite = limite;
    }

    public void depositar(double x) {
        this.saldo += x;
    }

    public void sacar(double x) {
        if(this.saldo + this.limite >= x) {
        this.saldo -= x;
        }
        else throw
          new Exception("Fundos insuficientes.");
        }

    public double getSaldo() {
        return this.saldo;
    }
}
```

Encapsulamento e o princípio da caixa preta

Envolve os princípios:

- Abstração
- Encapsulamento
- Interface
- Independência funcional
- Independência modular

Independência funcional e coesão

- Módulo: “grupo de comandos com uma função bem definida e o mais independente possível em relação ao resto do algoritmo”.
- Cada módulo deve cuidar de uma função específica, servindo a um propósito específico.
- É necessária coerência e unidade conceitual.

Independência funcional e coesão

- **Coesão**: Qualidade de uma coisa em que todas as partes estão ligadas umas às outras. Em software, todas as partes estão coerentemente relacionadas.
- Objetivo de um módulo em programação modular: alta coesão interna.
  - Facilita a manutenção.
  - Reduz efeitos colaterais e propagação de erros.
  - Dependência deve ser intra-modular: uso de estruturas internas ao módulo.

.

Independência modular e acoplamento

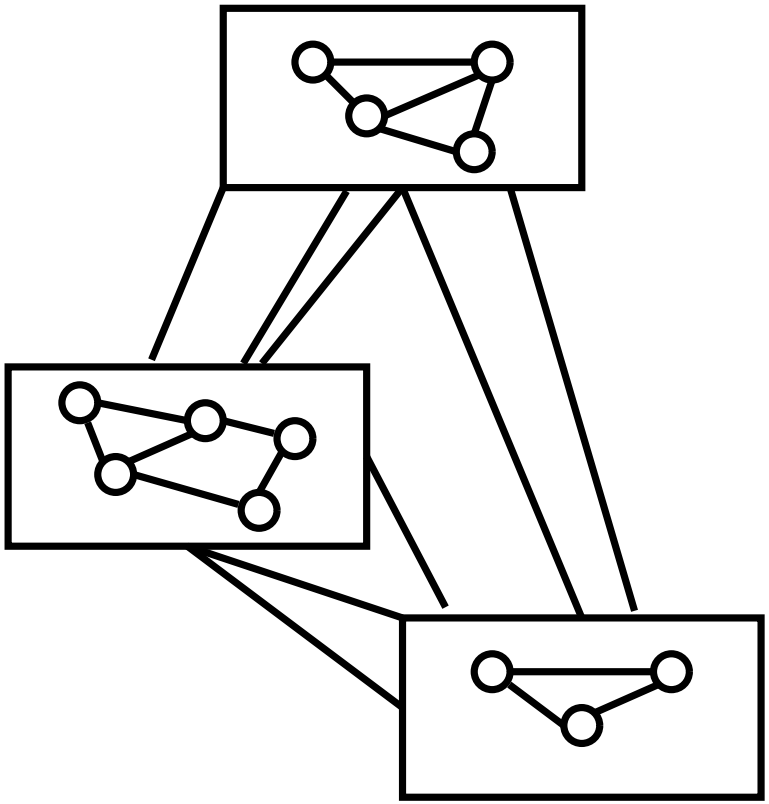
- Módulo: “grupo de comandos com uma função bem definida e o mais independente possível em relação ao resto do algoritmo”.
- A dependência pode ser medida pela quantidade de conexões entre os elementos de software.
- **Acoplamento**:
  - Medida da interconexão entre os elementos de software.
  - Situação ideal: baixo acoplamento.

Indicadores de baixo acoplamento

- **Tamanho**: quantidade de parâmetros e métodos públicos.
- **Visibilidade**: uso de parâmetros x uso de variáveis globais.
- **Flexibilidade**: facilidade na chamada (abordaremos no futuro).

Coesão e acoplamento

Alto acoplamento



Baixo acoplamento