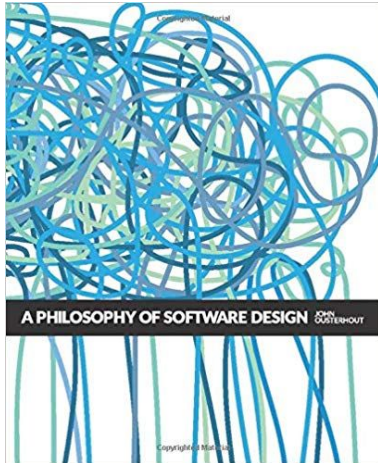


Propriedades de Projeto

Profa. Fabíola S. F. Pereira

fabiola.pereira@ufu.br

"O problema mais fundamental em Ciência da Computação é a tarefa de decomposição de problemas: como dividir um problema complexo em partes que possam ser resolvidas de forma independente" -- John Ousterhout



Definição de Projeto de Software

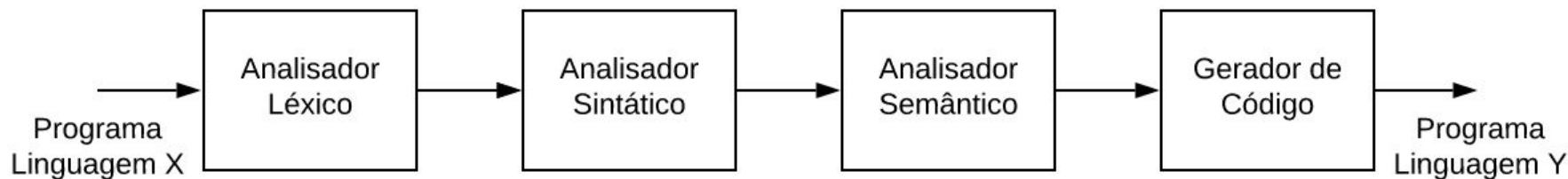
- Frase de Ousterhout é uma excelente definição
- Projeto:
 - Quebrar um "problema grande" em partes menores
 - Resolução (ou implementação) das partes menores resolvem (ou implementam) o "problema grande"

Projeto de Software

- Desenho ou proposta de solução

Projetar = Quebrar em partes menores

- Exemplo: compilador



Módulos

- Em Engenharia de Software:
 - Partes menores = módulos (pacotes, componentes, classes, etc)

Mais um conceito: Abstração

- Conceito que permite "usar" um módulo sem conhecer detalhes de sua implementação
- Exemplo: scanner (analisador léxico)
 - Implementar um scanner pode ser difícil
 - Mas usar é simples:

```
String token;
```

```
token = Scanner.next_token();
```

Na aula de hoje

- Propriedades de "bons projetos" de software
 - Integridade Conceitual
 - Ocultamento de Informação
 - Coesão
 - Acoplamento

Propriedades de Projeto

Integridade Conceitual

Integridade Conceitual

Primeiro Nome	Último nome	Idade
Maria	Silva	23
José	Fonseca	45 anos

O que incomoda na tabela?

Integridade Conceitual

- Funcionalidades de um sistema devem ser coerentes
- Sistema não pode ser um "amontoadado" de funcionalidades sem nenhuma coerência ou consistência

Exemplos

- Botão "sair" é idêntico em todas as telas
- Se um sistema usa tabelas para apresentar resultados, todas as tabelas têm o mesmo leiaute
- Todos os resultados são mostrados com 2 casas decimais

Integridade Conceitual vale também para o projeto e código de um sistema

Exemplos (em nível de projeto/código)

- Todas as variáveis seguem o mesmo padrão de nomes
 - **contra-exemplo:** `nota_total` vs `notaMedia`
- Todas as páginas usam o mesmo framework (mesma versão)
- Se um problema é resolvido com uma estrutura de dados X, todos os problemas parecidos também usam X

Integridade Conceitual = coerência e padronização de funcionalidades, projeto e implementação

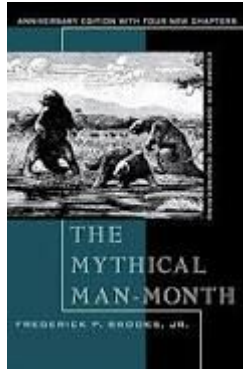


Exemplo



Contra-Exemplo

Integridade conceitual é a consideração mais importante no projeto de sistemas -- Fred Brooks



Motivo: integridade conceitual facilita uso e entendimento de um sistema

Ocultamento de Informação

(Information Hiding)

Origem do conceito (David Parnas, 1972)

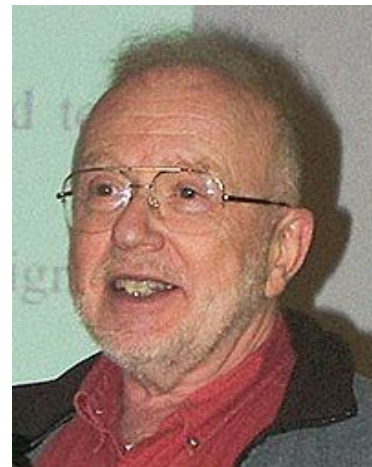
On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a “modularization” is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:¹



```
import java.util.Hashtable;

public class Estacionamento {

    public Hashtable<String, String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

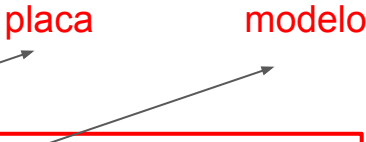
    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.veiculos.put("TCP-7030", "Uno");
        e.veiculos.put("BNF-4501", "Gol");
        e.veiculos.put("JKL-3481", "Corsa");
    }
}
```

```
import java.util.Hashtable;

public class Estacionamento {
    public Hashtable<String, String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.veiculos.put("TCP-7030", "Uno");
        e.veiculos.put("BNF-4501", "Gol");
        e.veiculos.put("JKL-3481", "Corsa");
    }
}
```



```
import java.util.Hashtable;

public class Estacionamento {

    public Hashtable<String, String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.veiculos.put("TCP-7030", "Uno");
        e.veiculos.put("BNF-4501", "Gol");
        e.veiculos.put("JKL-3481", "Corsa");
    }
}
```

Construtora, cria a Hashtable

```
import java.util.Hashtable;

public class Estacionamento {

    public Hashtable<String, String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.veiculos.put("TCP-7030", "Uno");
        e.veiculos.put("BNF-4501", "Gol");
        e.veiculos.put("JKL-3481", "Corsa");
    }
}
```

Problema: clientes precisam manipular uma estrutura de dados interna da classe, para estacionar um veículo

Problema

- Classes precisam de um pouco de "privacidade"
- Até para evoluir de forma independente dos clientes
- Código anterior: clientes manipulam a hashtable
- Comparação: clientes não podem entrar na cabine do estacionamento e eles mesmo anotar os dados do seu carro no "livro" do estacionamento

Agora uma versão com ocultamento de
informação

1

```
import java.util.Hashtable;

public class Estacionamento {

    private Hashtable<String,String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public void estaciona(String placa, String veiculo) {
        veiculos.put(placa, veiculo);
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.estaciona("TCP-7030", "Uno");
        e.estaciona("BNF-4501", "Gol");
        e.estaciona("JKL-3481", "Corsa");
    }
}
```

2

```
import java.util.Hashtable;

public class Estacionamento {

    private Hashtable<String,String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public void estaciona(String placa, String veiculo) {
        veiculos.put(placa, veiculo);
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.estaciona("TCP-7030", "Uno");
        e.estaciona("BNF-4501", "Gol");
        e.estaciona("JKL-3481", "Corsa");
    }
}
```

3

```
import java.util.Hashtable;

public class Estacionamento {

    private Hashtable<String,String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public void estaciona(String placa, String veiculo) {
        veiculos.put(placa, veiculo);
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.estaciona("TCP-7030", "Uno");
        e.estaciona("BNF-4501", "Gol");
        e.estaciona("JKL-3481", "Corsa");
    }
}
```

Resultado: classe Estacionamento fica livre para alterar a sua estrutura de dados interna

Ocultamento de Informação

- Classes devem ocultar detalhes internos de sua implementação (usando modificador **private**)
- Principalmente aqueles sujeitos a mudanças
- Adicionalmente, interface da classe deve ser estável
- **Interface** = conjunto de métodos públicos de uma classe

Getters e Setters

- Métodos get e set — muitas vezes chamados apenas de getters e setters — são muito usados em linguagens orientadas a objetos, como Java e C++.
- Recomendação é: todos os dados de uma classe devem ser privados e o acesso a eles — se necessário — deve ocorrer por meio de getters (acesso de leitura) e setters (acesso de escrita).

Getters e Setters

```
class Aluno {  
  
    private int matricula;  
    ...  
    public int getMatricula() {  
        return matricula;  
    }  
  
    public setMatricula(int matricula) {  
        this.matricula = matricula;  
    }  
    ...  
}
```


Getters e Setters

- No nosso exemplo, vamos assumir que é **imprescindível** que os clientes possam **ler e alterar a matrícula de alunos**. É melhor que o acesso a esse atributo seja feito por meio de métodos get e set, pois eles constituem uma interface mais estável para tal acesso, pelos seguintes motivos:

Getters e Setters

- No futuro, podemos precisar de recuperar a matrícula de um banco de dados, ou seja, ela não estará mais em memória. Essa nova lógica poderá, então, ser implementada no método get, **sem impactar nenhum cliente da classe**.
- No futuro, podemos precisar de adicionar um dígito verificador nas matrículas. Essa **lógica** — cálculo e incorporação do dígito verificador — poderá ser implementada **no método set, sem impactar os seus clientes**.

Coesão

Coesão

- Uma classe deve ter uma única função, isto é, oferecer um único serviço
- Vale também para outras unidades: funções, métodos, pacotes, etc.

Contra-exemplo 1

```
float sin_or_cos(double x, int op) {  
    if (op == 1)  
        "calcula e retorna seno de x"  
    else  
        "calcula e retorna cosseno de x"  
}
```

Contra-exemplo 1

```
float sin_or_cos(double x, int op) {  
    if (op == 1)  
        "calcula e retorna seno de x"  
    else  
        "calcula e retorna cosseno de x"  
}
```

Deveria ser quebrada em duas funções: sin e cos

Contra-exemplo 2

```
class Estacionamento {  
    ...  
    private String nome_gerente;  
    private String fone_gerente;  
    private String cpf_gerente;  
    private String endereco_gerente;  
    ...  
}
```

Contra-exemplo 2

```
class Estacionamento {  
    ...  
    private String nome_gerente;  
    private String fone_gerente;  
    private String cpf_gerente;  
    private String endereco_gerente;  
    ...  
}
```

Deveria ser quebrada em duas classes:
Estacionamento e Gerente

Exemplo

```
class Stack<T> {  
    boolean empty() { ... }  
    T pop() { ... }  
    push (T) { ... }  
    int size() { ... }  
}
```

Todos esses métodos manipulam os elementos da Pilha

Acoplamento

Acoplamento

- Nenhuma classe é uma ilha ...
- Classes dependem uma das outras (chamam métodos de outras classes, estendem outras classes, etc)
- A questão principal é a **qualidade** desse acoplamento
- Dois tipos:
 - Acoplamento aceitável ("bom")
 - Acoplamento ruim

Acoplamento Aceitável

- Classe A depende de uma classe B:
 - Mas a classe B possui uma interface estável
 - Classe A somente chama métodos da interface de B

```
import java.util.Hashtable;
```

```
public class Estacionamento {
```

Classe Estacionamento depende (está acoplada) à classe Hashtable, mas esse acoplamento é aceitável

```
    private Hashtable<String,String> veiculos;
```

```
    public Estacionamento() {  
        veiculos = new Hashtable<String, String>();  
    }
```

```
    public void estaciona(String veiculo, String placa) {  
        veiculos.put(veiculo, placa);  
    }
```

```
    public static void main(String[] args) {  
        Estacionamento e = new Estacionamento();  
        e.estaciona("TCP-7030", "Uno");  
        e.estaciona("BNF-4501", "Gol");  
        e.estaciona("JKL-3481", "Corsa");  
    }
```

```
}
```

Acoplamento Ruim

- Classe A depende de uma classe B:
 - a. Mas interface da classe B é instável
 - b. Ou então a dependência não ocorre via interface de B

Como uma classe A pode depender de uma classe B sem ser via a interface de B?

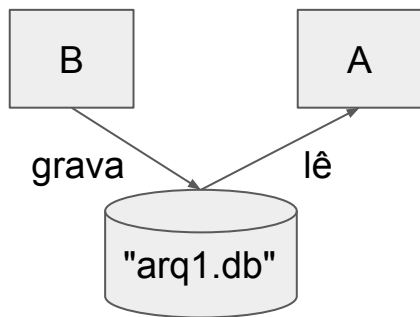
```
class A {  
  
    private void f() {  
        int total;  
        ...  
        File f = File.open("arq1.db");  
        total = f.readInt();  
        ...  
    }  
  
}
```

```
class B {  
    private void g() {  
        int total;  
        // computa valor de total  
        File f = File.open("arq1.db");  
        f.writeInt(total);  
        ...  
        f.close();  
    }  
}
```



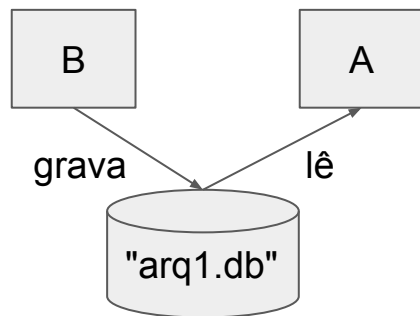
```
class A {  
  
    private void f() {  
        int total;  
        ...  
        File f = File.open("arq1.db");  
        total = f.readInt();  
        ...  
    }  
  
}
```

```
class B {  
    private void g() {  
        int total;  
        // computa valor de total  
        File f = File.open("arq1.db");  
        f.writeInt(total);  
        ...  
        f.close();  
    }  
}
```



```
class A {  
  
    private void f() {  
        int total;  
        ...  
        File f = File.open("arq1.db");  
        total = f.readInt();  
        ...  
    }  
  
}
```

```
class B {  
    private void g() {  
        int total;  
        // computa valor de total  
        File f = File.open("arq1.db");  
        f.writeInt(total);  
        ...  
        f.close();  
    }  
}
```



```

class A {

    private void f() {
        int total;
        ...
        File f = File.open("arq1.db");
        total = f.readInt();
        ...
    }

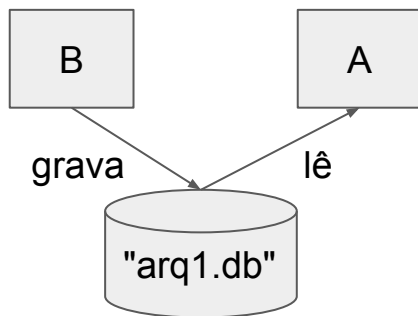
}

```

```

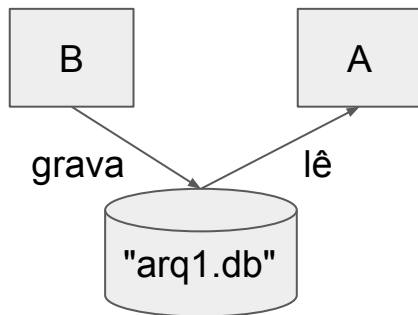
class B {
    private void g() {
        int total;
        // computa valor de total
        File f = File.open("arq1.db");
        f.writeInt(total);
        ...
        f.close();
    }
}

```



Problema

- Classe B não sabe que a classe A é sua "cliente"
- Logo, B pode mudar o formato do arquivo ou mesmo deixar de salvar o dado que é lido por A



Tornando acoplamento ruim em bom

```
class B {  
  
    int total;  
  
    public int getTotal() {  
        return total;  
    }  
  
    private void g() {  
        // computa valor de total  
        File f = File.open("arq1");  
        f.writeInt(total);  
        ...  
    }  
}
```

```
class A {  
  
    private void f(B b) {  
        int total;  
        total = b.getTotal();  
        ...  
    }  
}
```

Tornando acoplamento ruim em bom

```
class B {  
  
    int total;  
  
    public int getTotal() {  
        return total;  
    }  
  
    private void g() {  
        // computa valor de total  
        File f = File.open("arq1");  
        f.writeInt(total);  
        ...  
    }  
}
```

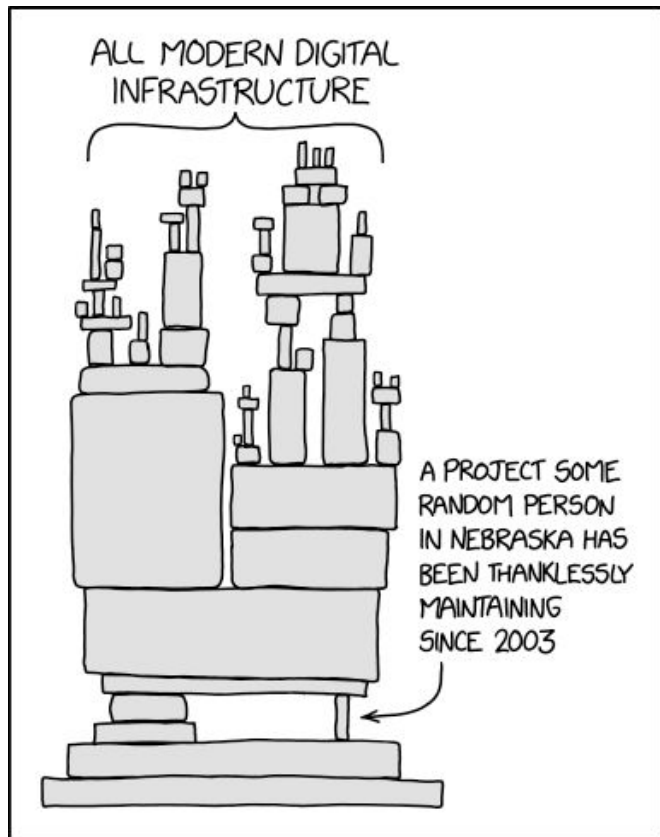
```
class A {  
  
    private void f(B b) {  
        int total;  
        total = b.getTotal();  
        ...  
    }  
}
```

Frase muito comum

Maximize a coesão, minimize o acoplamento

Mas cuidado: minimize (ou elimine)
principalmente o acoplamento ruim

Riscos de Acoplamento



Principalmente, quando envolve dependências para bibliotecas de terceiros

Revisão dos conceitos




Propriedades de “bons projetos” de software

- Integridade Conceitual
- Ocultamento de Informação
- Coesão
- Acoplamento

Princípio de Projeto	Propriedade de Projeto
Responsabilidade Única	Coesão
Segregação de Interfaces	Coesão
Inversão de Dependências	Acoplamento
Prefira Composição a Herança	Acoplamento
Demeter	Ocultamento de Informação
Aberto/Fechado	Extensibilidade
Substituição de Liskov	Extensibilidade



"Diretriz"



Consequência (o que
vamos ganhar
seguindo o princípio)

Princípios SOLID

- **S**ingle Responsibility Principle
- **O**pen Closed/Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle



Robert Martin

Referências

- VALENTE, M. Engenharia de Software Moderna. 2020. cap. 5 - <https://engsoftmoderna.info/cap5.html>
- PRESSMAN, R. S . Engenharia de Software. [S . I] : McGraw- Hill, 2011. cap. 12

Créditos

- CC-BY: Slides adaptados de Marco Tulio Valente, ESM

FIM