Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

KÊNIA FERREIRA DE JESUS

# OS MANDAMENTOS DA PROGRAMAÇÃO MODULAR EM JAVA

Belo Horizonte
2016

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Engenharia de Software

# OS MANDAMENTOS DA PROGRAMAÇÃO MODULAR EM JAVA

por

KÊNIA FERREIRA DE JESUS

Monografia de Final de Curso
CEI-ES-0352-T6-2007-01

Prof. Roberto da Silva Bigonha

Belo Horizonte
2016

KÊNIA FERREIRA DE JESUS

# OS MANDAMENTOS DA PROGRAMAÇÃO MODULAR EM JAVA

Monografia apresentada ao Curso de Especialização em Informática do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do certificado de Especialista em Informática.

Ênfase: Engenharia de software

Orientador: Prof. Roberto da Silva Bigonha

Belo Horizonte
2016

**Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG**

Jesus, Kênia Ferreira de.

J58m Os mandamentos da programação modular em Java . /
Kênia Ferreira de Jesus. Belo Horizonte, 2016.
xii, 80 f.: il.; 29 cm.

Monografia (especialização) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Roberto da Silva Bigonha.

1. Computação 2. Engenharia de software. 3. Java (Linguagem de programação de computador). 4. Programação modular. I. Orientador. II. Título.

CDU 519.6*32(043)

**UNIVERSIDADE FEDERAL DE MINAS GERAIS**

INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ESPECIALIZAÇÃO EM INFORMÁTICA: ÁREA DE CONCENTRAÇÃO ENGENHARIA DE SOFTWARE

# OS MANDAMENTOS DA PROGRAMAÇÃO MODULAR EM JAVA

KÊNIA FERREIRA DE JESUS

Monografia apresentada aos Senhores:

Prof. Roberto da Silva Bigonha
Orientador
DCC - ICEx - UFMG

Profa. Mariza Andrade da Silva Bigonha
DCC - ICEx - UFMG

Belo Horizonte, 29 de fevereiro de 2016

A Deus,
aos meus pais
e aos meus irmãos.

## AGRADECIMENTOS

Agradeço primeiramente a Deus por me capacitar e renovar a cada dia as minhas forças para que eu possa buscar melhorias.

Agradeço aos meus pais e aos meus irmãos o incentivo que eles sempre me deram para estudar e o apoio mesmo nos momentos difíceis.

Por fim, agradeço ao meu orientador a ajuda desde a escolha do tema até o fim da confecção do trabalho.

## RESUMO

O principal objetivo do presente trabalho é apresentar um conjunto de boas práticas para a programação modular em Java. Um conhecimento sólido dessa linguagem de programação é fundamental, mas, é preciso aplicar boas práticas na estruturação de um código para que o mesmo se torne mais organizado, legível, reaproveitável e seja mais fácil aplicar alguma manutenção quando necessário.

Neste trabalho é apresentado o conceito de módulo e técnicas para a aplicação de boas práticas de programação. A aplicação desses conceitos pode ser feita durante todo o processo de construção ou refatoração do código de um software com o objetivo de torná-lo bem estruturado.

No fim deste trabalho, são apresentados os mandamentos da programação modular em Java como uma abstração de tudo o que foi discutido durante todo o desenvolvimento do documento. Cada mandamento é uma curta definição de uma das principais regras para a aplicação das boas práticas de programação.



**Palavras-chave:** Programação Modular, Módulo, Boas Práticas, Mandamentos da Programação.

## ABSTRACT

The main objective of this paper is to present a set of best practices for modular programming in Java. A solid knowledge of this programming language is extremely profitable, but you need to apply these best practices in structuring a code so that it becomes better organized, readable, reusable and easier to maintain.

This work presents the concept of module and techniques for the application of good programming practices. The application of these concepts can be done through the process of constructing or refactoring code of a software in order to make it well structured.

At the end of this work, the commandments for modular programming in Java as an abstraction of all that was passed throughout the development of the document is presented. Each commandment is a short definition of one of the main rules of good programming practices.



**Keywords*:*** Modular Programming, Module, Practice, Commandments of Programming.

## LISTA DE FIGURAS

## LISTA DE SIGLAS

| | |
|---|---|
| POO | Programação Orientada por Objetos |
| SRP | Single Responsability Principle |
| OCP | Open/Closed Principle |
| LSP | Liskov Substitution Principle |
| ISP | Interface Segregation Principle |
| DIP | Dependency Inversion Principle |
| TAD | Tipo Abstrato de Dados |
| UML | Unified Modeling Language |

**SUMÁRIO**

# CAPÍTULO 1

# INTRODUÇÃO

Boas práticas em desenvolvimento de código em linguagens de programação visam garantir um bom desempenho do aplicativo a ser construído e a organização do código para uma manutenção prática e eficaz.

O próprio título deste trabalho sugere uma boa prática de programação. Programar de forma modular significa dividir um código em partes menores que chamamos de módulos. Trabalhar com módulos contribui com a organização do código e facilita o seu entendimento. Os menores módulos em Java são representados pelas classes.

Além de modularizar um código, é preciso utilizar bons estilos e idiomas de programação. É preciso também evitar o uso de práticas ruins que chamamos de *bad smells* e devem ser corrigidas com refatoração.

É necessário ainda aplicar metodologias de programação. Elas são utilizadas para garantir que o módulo possua qualidades como proteção contra acesso imprevisto de seus dados, pois, essa restrição garante um código mais seguro. Dentre outras qualidades que poderíamos citar temos o uso correto de composição e herança, pois, o uso equivocado de herança pode prejudicar a encapsulação dos dados e provocar alto acoplamento entre módulos.

Para a boa estruturação do código, existem também os estilos de programação. Eles são princípios a serem aplicados durante o desenvolvimento de um software. Dentre os princípios mais utilizados, existe o uso de abstrações corretas, limitação de efeito colateral em função, implementação de objetos como máquinas de estado, consistência de estados e interfaces e uso de polimorfismo de inclusão. A aplicação desses princípios garante códigos mais robustos. Estilos também são peculiaridades que devem ser aplicadas ao escrever o código. No momento da escrita, é necessário ter o cuidado de nomear alguns itens com inicial maiúscula, outros com minúscula, utilizar verbos para algumas nomeações, substantivos e adjetivos para outras e assim por diante. Essas peculiaridades também contribuem com a organização do código e são importantes para que exista um padrão de escrita aplicado durante o desenvolvimento.

Por fim, devemos evitar práticas ruins aplicadas durante a programação de um aplicativo que geram o que chamamos de *bad smells*. Um *bad smell* nada mais é que um defeito no código que deve ser corrigido com a sua refatoração. Podemos citar como exemplos de *bad smells* um método muito longo que deve ser dividido para gerar novos métodos, um método que utiliza somente atributos de outra classe e deve ser movido para essa outra classe, nomeação incoerente de métodos e atributos, dentre outros.

## 1.1 OBJETIVOS

Objetivos Gerais

O principal objetivo do trabalho é o estudo e organização das boas práticas de programação e a contribuição para a disseminação dessas boas práticas.

Objetivos Específicos

Para atingir esse objetivo geral, os seguintes aspectos e metas serão perseguidos:

- apresentar o conceito de módulo;
- mostrar as características de um bom módulo;
- apresentar os princípios fundamentais de programação;
- mostrar estilos de programação;
- descrever processos de refatoração de código;
- mostrar os mandamentos da programação modular.

## 1.2 JUSTIFICATIVA

Para a construção de um bom módulo, não basta que ele tenha ótimas funcionalidades e uma boa interface. De acordo com Sommerville (2007), a análise das oito leis de Manny Lehman sobre manutenção e evolução de sistemas, também chamadas de leis de Lehman, leva a crer que o custo de evolução / manutenção de um software é três vezes maior que o custo de desenvolvimento do mesmo. Sendo assim, é muito importante que um código seja bem estruturado e reutilizável para facilitar a sua manutenção e reduzir os custos da mesma.

O uso de boas práticas de programação modular garante uma melhor manutenibilidade do código, uma vez que um código que tenha sido construído com o uso de boas práticas é mais organizado e mais fácil de aplicar manutenção, além de que códigos mais bem estruturados são mais reutilizáveis.

Possuir códigos bem estruturados é uma necessidade que gera uma busca diária de profissionais da área de desenvolvimento na aplicação de boas práticas de programação para chegar a esse objetivo. No ambiente atual das grandes empresas de software, um

desenvolvedor trabalha com uma equipe de vários outros desenvolvedores que estão constantemente fazendo alterações em códigos de outros colegas e trabalhando juntos para o desenvolvimento de novas funções e aplicativos. O uso das boas práticas garante que o código de um desenvolvedor seja facilmente entendido e manuseado por outro desenvolvedor, uma vez que o produto é de propriedade da empresa que essas pessoas trabalham e essa empresa precisa garantir que o produto produzido por esses profissionais possa ser alterado por outros profissionais quando for preciso.

O reúso também é uma necessidade atual das empresas de software, uma vez que reúso é economia de tempo e consequentemente economia de dinheiro. Ao construir um código reutilizável, é possível poupar tempo durante a evolução daquele mesmo projeto já que novos módulos podem utilizar funções reutilizáveis de outros módulos. É possível diminuir os custos também na criação de um novo projeto, pois, tendo um módulo de um projeto x que possa ser reutilizado em um projeto y, economiza-se todo o tempo que seria gasto para construir aquele módulo caso fosse necessário desenvolvê-lo desde o início. Atingir um bom grau de reúso é uma realidade caso sejam utilizadas boas práticas de desenvolvimento durante a produção de um projeto.

## 1.3 CONCLUSÃO

Estudar boas práticas é importante para que seja levada em consideração a necessidade de utilizar bons recursos na construção de códigos. O estudo e aplicação das boas práticas fazem com que o desenvolvimento seja realizado de acordo com os padrões utilizados para alcançar um bom resultado.

Nesta monografia serão apresentadas as boas práticas para desenvolvimento de software modular. Capítulo 2 define módulo, fala sobre as boas características e boas práticas para se aplicar em um módulo. Capítulo 3 apresenta os princípios fundamentais a serem aplicados durante a construção ou refatoração de módulos. Capítulo 4 apresenta estilos para uma programação eficiente, estruturada e organizada. Capítulo 5 fala sobre refatoração de módulos que apresentam características ruins também chamadas de *bad smells*. Capítulo 6 apresenta os mandamentos da programação modular em Java baseados nas boas práticas estudadas em todos os capítulos anteriores. Capítulo 7 apresenta uma conclusão de todo o texto escrito apontando a necessidade de utilização das boas práticas.

# CAPÍTULO 2

# PROGRAMAÇÃO MODULAR

Programar de forma modular é o ato de dividir um código extenso em partes menores durante a sua construção ou refatoração.

## 2.1 O QUE É UM MÓDULO

Segundo Schach (2009), os primeiros a tentarem descrever um módulo foram Stevens, Myers e Constantine (1974). De acordo com essa primeira descrição, um módulo é um agrupamento de uma ou mais linhas de código. Esse agrupamento possui um nome e pode ser compilado separadamente.

Quando possuímos um software com muitas linhas de código, agrupar todas essas linhas em poucos módulos torna difícil a tarefa de realizar sua manutenção. É interessante que durante a construção ou refatoração do código, ele seja estruturado em vários módulos, cada um com objetivos específicos, facilitando seu entendimento e a sua manutenção.

Na linguagem Java e em outras linguagens orientadas por objetos, o menor módulo possível é representado por uma classe, pois, essa é a menor parte compilável separadamente de um aplicativo construído nesse tipo de linguagem. Uma classe nada mais é que um arquivo do aplicativo que contém atributos e métodos que definem um objeto. Podemos construir, por exemplo, uma classe `Pessoa` que contenha atributos como `nome`, `cpf` e `dataNascimento`. A classe `Pessoa` é um módulo de Java.

```
import sun.util.calendar.BaseCalendar.Date;

public class Pessoa {

	private String nome;
	private String cpf;
	private Date dataNascimento;

	public Pessoa(String nome, String cpf,
	Date dataNascimento){

		this.nome = nome;
		this.cpf = cpf;
		this.dataNascimento = dataNascimento;
	}
}
```

Em outros casos, um módulo pode ser mais que uma simples classe. Conheça os 4 principais tipos de módulos em Java de acordo com Bigonha (2015):

- Tipos-Abstratos-de-Dados (TAD): Arquivo contendo uma ou mais classes que implementam o TAD. Nesse tipo de implementação é criado um contrato de implementação e todas as classes que implementam aquele tipo devem agir de acordo com o contrato.

- Módulo de Rotinas-Relacionadas (RR): Nesse tipo de módulo não existem atributos, ele possui um conjunto de rotinas relacionadas que operam somente em seus parâmetros.

- Módulo de Declarações-de-Dados (DD): Arquivo contendo somente declaração de dados (atributos).

- Módulo de Agrupamento-Geral (AG): Arquivo com diversas declarações de dados e métodos sem qualquer relação entre os mesmos.

## 2.2 CARACTERÍSTICAS DE UM BOM MÓDULO

Segundo estudos de vários autores referenciados ao longo do texto, um bom módulo possui as seguintes características:

1. possui alta coesão de seus componentes internos;
2. possui baixo grau de acoplamento com outros módulos;
3. possui função ou propósito único e bem definido;
4. detém alto grau de *information hiding* e encapsulação;
5. possui interface estreita e bem definida;
6. usa composição e herança apropriadamente;
7. respeita o contrato de sua interface, caso seja um módulo cliente;
8. faz uso certo do reúso;
9. utiliza convenção de nomes e escrita para organizar o código, comentá-lo e nomear seus atributos e métodos.

Características de um bom módulo são obtidas com o uso de boas práticas de programação. Essas boas práticas estão detalhadas uma a uma nos próximos capítulos, mas, para exemplificar inicialmente, podemos citar algumas como: uso dos princípios fundamentais e o uso de bons estilos de programação.

Bons módulos também são construídos evitando o uso de algumas ocorrências como: métodos muito extensos que deveriam ser divididos em vários métodos, existência de uma classe que usa somente atributos de outra classe, classes que possuem muitas atividades quando deveriam ter sido criadas para um único objetivo, etc. Essas ocorrências são chamadas de *bad smells*.

Em um bom módulo Java os atributos devem ser encapsulados evitando fácil acesso externo aos dados, pois, a não restrição ao acesso caracteriza uma falha de segurança e pode elevar o grau de acoplamento entre módulos. Um bom módulo também deve ser extensível, para adaptar-se a novas situações de reúso. Em POO (Programação Orientada por Objetos), o polimorfismo e a herança são constantemente utilizados para estender funcionalidade de módulos. Extensibilidade facilita a reutilização de módulos. Uma vez implementado um módulo para o cumprimento de um objetivo ele pode ser utilizado também em outras partes do aplicativo ou em outros aplicativos para cumprir aquele mesmo objetivo ou então adaptado a objetivos semelhantes.

Caso um módulo não possua os atributos de qualidade mínimos ou desejados, é necessário refatorar esse módulo para ele atenda os padrões necessários para uma boa organização visando a manutenibilidade do código.

## 2.3 ENCAPSULAÇÃO E RESTRIÇÃO AO ACESSO

Encapsular é o ato de esconder detalhes da implementação, dados e estruturas internas de um objeto específico, dessa forma, é restringido o que um módulo tornará disponível aos seus clientes para tornar os módulos da aplicação o mais independentes possível. O simples ato de inserir itens dentro de uma classe é encapsular, mas, para tornar esse módulo mais independente e privilegiar o raciocínio modular, é necessário restringir o acesso aos seus dados. Restrição de acesso é o ato de utilizar modificadores de acesso nos atributos que concedam diferentes permissões sobre a forma como eles podem ser acessados com o intuito de abstrair os dados. Quando os atributos estão utilizando modificadores de acesso, o acesso a eles é restringido de diferentes formas. Se um atributo for `private`, o acesso a ele é feito somente pelos métodos da classe. Se ele for `protected`, seu acesso pode ser feito somente pelos componentes do pacote. É interessante ressaltar que o atributo `private` fornece a forma de encapsulação mais restrita. A encapsulação dos dados e a restrição do acesso a eles estão diretamente relacionados ao quesito de independência entre módulos. Essa prática também é chamada de abstração de dados. Com ela, é possível ter acesso aos métodos sem necessidade de conhecer como os mesmos foram implementados. Dessa forma podemos restringir o acesso aos dados, e privilegiar a modularidade.

Essa proteção é interessante, pois, caso o programador utilize um atributo público `public string x`, ele deixa livre o acesso e a modificação direta daquele atributo por outras classes. Essa prática dificulta o raciocínio modular à medida que fica difícil controlar em quais pontos do aplicativo aquele atributo pode ter sido acessado diretamente. É sempre importante deixar os atributos privados para que eles sejam acessados somente por meio dos métodos que foram criados para a sua manipulação.

Vamos imaginar uma classe `Conta` que pertence a um sistema bancário, conforme a representação da implementação a seguir. Caso o atributo `saldo` fosse `public`, ele poderia ser acessado por qualquer outra classe do sistema. Como em vez de um

modificador `public saldo` ele possui o modificador `private`, ele só pode ser alterado pelos métodos `sacar` e `depositar` presentes na classe.

```
public class Conta {

	private String nome;
	private int conta;
	private double saldo;

	public Conta(String nome, int conta, double saldo){

		this.nome = nome;
		this.conta = conta;
		this.saldo = saldo;
	}

	public void sacar(double valor){

		if(saldo >= valor){
			saldo -= valor;
		}
		else{
			System.out.println("Saldo insuficiente.");
		}
	}

	public void depositar(double valor){

		saldo += valor;
		System.out.println("Depositado: " + valor);
		System.out.println("Novo saldo: " + saldo);

	}

	public void extrato(){

		System.out.println("EXTRATO:");
		System.out.println("Nome: " + nome);
		System.out.println("Conta: " + conta);
		System.out.println("Saldo: " + saldo);
	}
}
```

Restrição de acesso também pode ser imposta a métodos de um módulo. Da mesma forma que ocorre com as variáveis, funções `private` são acessadas somente dentro da classe a qual pertencem e funções `protected` podem ser acessadas de qualquer módulo do pacote e por métodos de subclasses.

Pretende-se obter com a encapsulação dos dados e a restrição ao acesso, módulos mais isolados, que ao serem modificados, não prejudiquem o funcionamento de outros módulos. Assim, quando um módulo necessitar de alteração, minimiza-se a necessidade de alterar outros.

## 2.4 ACOPLAMENTO

Assunto abordado por Myers (1978), podemos definir a interação que ocorre entre dois módulos como conectividade, e a intensidade dessa conectividade, como acoplamento. Quando dizemos que dois módulos estão fortemente acoplados significa que eles possuem alto grau de conectividade, ou seja, que um deles possui um conhecimento indevido da implementação do outro e pode estar com uma forte dependência desse outro módulo. Para ilustrar, vamos exemplificar com a falta de uso dos modificadores corretos na seguinte implementação:

```
public class Aluno {

	public String nome;
	public String sexo;
	public String telefone;
	public String endereco;

	public Aluno(String nome, String sexo,
	String telefone, String endereco){

	this.nome = nome;
	this.sexo = sexo;
	this.telefone = telefone;
	this.endereco = endereco;

	}
}


public class Aplicacao {

	public static String consultarAluno(Aluno aluno){

		String alun = aluno.nome;
		String sex = aluno.sexo;
		String tel = aluno.telefone;
		String end = aluno.endereco;

		return alun + sex + tel + end;
	}
```

```
	public static void main(String[] args){

		Aluno kenia = new Aluno("Kênia", "Feminino",
		"031 986578909", "Rua das estrelas...");

		String imprimeAluno = consultarAluno(kenia);
		System.out.print(imprimeAluno);

	}
}
```

Na implementação mostrada, a classe `Aluno` possui todos os seus atributos púbicos, e a classe `Aplicacao` consegue acessar esses atributos diretamente. Essas classes têm um alto grau de acoplamento, pois, a classe `Aluno` deveria ter atributos privados e disponibilizar ela mesma, métodos para a consulta desses atritutos. Da forma como está, a classe `Aplicacao` conhece detalhes de implementação da classe `Aluno` que não precisa conhecer. Na correção da implementação, teríamos essas classes funcionando da seguinte forma:

```
public class Aluno {

	private String nome;
	private String sexo;
	private String telefone;
	private String endereco;

	public Aluno(String nome, String sexo,
	String telefone, String endereco){

		this.nome = nome;
		this.sexo = sexo;
		this.telefone = telefone;
		this.endereco = endereco;

	}

	public String consultarAluno(Aluno aluno){
		String name = aluno.nome;
		String sex = aluno.sexo;
		String tel = aluno.telefone;
		String end = aluno.endereco;

		return name + sex + tel + end;

	}
}
```

```
public class Aplicacao {

      public static void main(String[] args){

            Aluno kenia = new Aluno("Kênia", "Feminino",
            "031 986578909", "Rua das estrelas...");

            String imprimeAluno = kenia.consultarAluno(kenia);
            System.out.print(imprimeAluno);
      }
}
```

Nessa implementação, `Aluno` tem seus atributos privados e outras classes não conseguirão acessá-lo diretamente. Essa medida aumentou a segurança e reduziu o acoplamento. A classe `Aplicacao` agora precisa acessar o método `consultarAluno` diretamente na classe `Aluno` para imprimir os dados de um aluno. Caso fosse necessária alguma manutenção na implementação anterior no método `consultarAluno`, possivelmente faríamos um número parecido de alterações nas duas classes. Na implementação atual, caso fosse necessária alguma alteração no método `consultarAluno`, a alteração maior seria feita na classe `Aluno` e talvez fosse preciso alterar somente a chamada do método na classe `Aplicacao`.

No primeiro exemplo com a ausência dos modificadores corretos, foi obtido o acoplamento por conteúdo que é o mais alto grau de acoplamento. No segundo exemplo com o emprego do uso dos modificadores corretos, foi obtido o acoplamento por informação que é o mais baixo grau de acoplamento. Essas e outras formas de acoplamento foram propostos por Myers (1978) na seguinte ordem crescente:

1. **Por informação:** conectividade entre módulos por chamada de métodos com passagem de parâmetro por valor.

2. **Por referência:** conectividade entre módulos por chamada de métodos com passagem de parâmetro por referência. No caso de Java, trata-se de passagem de parâmetros que sejam referências a objetos.

3. **Por controle:** módulo que conhece a implementação de métodos de outro e passa a ele parâmetros para controlar sua execução.

4. **Por dado externo:** módulos que dependem de objetos externos.

5. **Por inclusão:** não existe em Java. Trata-se de um módulo que inclui código de outro componente.

6. **Por dado comum:** módulos diferentes que fazem acesso comum a objetos globais.

7. **Por conteúdo:** módulo com acesso direto a objetos de outro módulo.

É importante sempre observar a implementação para reduzir ao máximo o nível de acoplamento. Reduzir o acoplamento é importante, pois, aumenta a coesão e reduz o custo da manutenção. Um módulo funcionando de forma ideal possui baixo acoplamento.

## 2.5 COESÃO

Assunto abordado por Myers (1978), a coesão de um módulo é medida pelo relacionamento entre seus elementos internos. Quando um módulo possui atributos pouco ou nada utilizados por seus métodos, dizemos que ele é pouco coeso. Nesse caso, é necessário revisar os módulos para verificar onde os atributos/módulos deveriam ser realocados para aumentar a coesão dos módulos examinados. Por outro lado, se um módulo possui atributos que estão sendo devidamente utilizado por seus métodos, quanto maior for a intensidade desse relacionamento métodos/atributos, maior será a coesão daquele módulo.

Existem algumas métricas para o cálculo de coesão de um módulo, uma delas é a LCOM (*lack of cohesion in methods*) proposta por Chidamber and Kemerer (1994). Essa métrica mede o uso dos atributos de uma classe por seus próprios métodos para verificar se os métodos estão fazendo uso dos atributos e quantos métodos estão utilizando aqueles atributos.

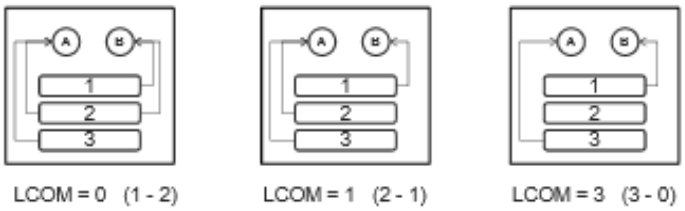


**FIG. 1 Exemplo do uso de LCOM. Fonte: Figueiredo.**

A Figura 1 mostra a medição de coesão de três classes com o uso de LCOM. Na figura podemos verificar o uso que os métodos representados pelos números fazem dos atributos, representados pelas letras. Cada quadrado é uma classe, e a classe mais coesa é a primeira começando pela esquerda. Ela tem todos os seus métodos utilizando seus atributos e cada atributo é acessado por pelo menos dois métodos. A segunda classe tem uma coesão intermediária, pois ela possui todos os métodos acessando atributos internos, mas, o atributo B poderia ser acessado por mais algum método. A terceira classe é a menos coesa das três. Os dois atributos da terceira classe são acessados por um método cada um, isso em uma classe que contém três métodos. Essa classe ainda possui um método que não acessa nenhum de seus atributos, mostrando que esse método pode estar localizado na classe errada.

Coesão é um fator a ser observado para preservar a manutenibilidade do código. Caso uma classe não seja coesa, ela deve ser refatorada para que possua os atributos e métodos corretos e realoque em outras classes os que não devem estar ali. Enfim,

módulos devem ser organizados de forma a facilitar o reúso e a manutenção, além de manter o equilíbrio da coesão. Um módulo funcionando de forma ideal possui alta coesão.

## 2.6 CONCLUSÃO

Introduzir os conceitos base de programação modular foi importante para saber em que estão baseadas todas as demais discussões presentes no restante do texto. Ao saber o que é um módulo e como programar de forma modular, pode-se discutir organização de módulos e práticas que devemos aplicar na construção dos mesmos.

Entender as caracterítiscas de um bom módulo contribui para que se possa buscar sempre desenvolver de acordo com elas para construir módulos mais eficientes. Encapsulação e restrição ao acesso são importantes para facilitar o raciocínio modular. Coesão é algo que deve receber atenção para que um módulo tenha seus atributos e métodos agindo de forma coerente. Por fim, devemos ter cuidado para que o acoplamento nunca seja maior que o necessário.

Ao introduzir esses princípios iniciais, podemos ter ciência da importância de aplicação dos mesmos e partir para o estudo dos demais princípios necessários para construir software com qualidade.

# CAPÍTULO 3

# PRINCÍPIOS FUNDAMENTAIS

Os princípios fundamentais da programação modular são um conjunto de regras básicas de organização e funcionamento a serem utilizados no desenvolvimento de bons módulos. O conjunto desses princípios deve ser utilizado como um modelo que guie o desenvolvimento para que ele seja embasado em boas práticas de programação modular.

Dentre esses princípios, temos o princípio da escolha do uso de composição ou herança, o princípio da programação para a interface e os princípios nomeados pela sigla SOLID por Martin (2006) de acordo com as iniciais de cada um, são eles:

Princípio da responsabilidade única (**S**ingle Responsability Principle - SRP)
Princípio do aberto/fechado (**O**pen/Closed Principle - OCP)
Princípio da substituição de Liskov (**L**iskov Substitution Principle - LPS)
Princípio da segregação de interfaces (**I**nterface Segregation Principle - ISP)
Princípio da inversão da dependência (**D**ependency Inversion Principle - DIP)

Vejamos, na continuação deste capítulo, a definição de cada um desses princípios.

## 3.1 PRINCÍPIO DA ESCOLHA: COMPOSIÇÃO X HERANÇA

Herança e composição são duas formas possíveis que uma classe possui para construir a estrutura de seus elementos a partir de outra classe. No uso da herança, uma classe filha herda todos os atributos e métodos de uma superclasse. O uso da herança é ilustrado na Figura 2:

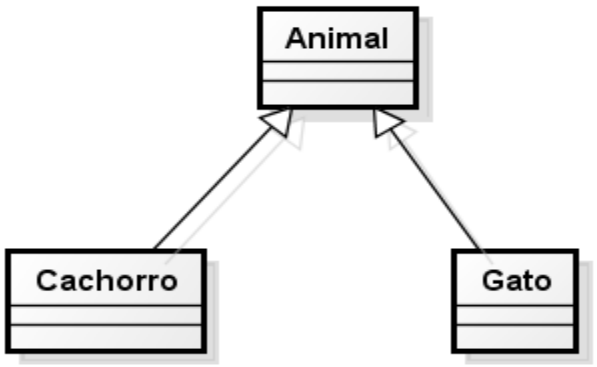


**FIG. 2 Exemplo de herança. Fonte: Medeiros.**

Na implementação da Figura 2, a classe `Animal` é a classe generalizada ou superclasse e pode conter atributos como `raca` e `peso`. Os métodos da classe `Animal` poderiam ser `locomover` e `comer`. A classe `Cachorro`, por sua vez, é uma classe especializada ou

subclasse e herda todos os atributos e métodos da classe `Animal`. Essa classe especializada deve realizar todas as atividades implementadas pela classe `Animal` e direcioná-las de maneira específica para um cachorro. Da mesma forma ocorre com a outra classe especializada `Gato`. Esse tipo de implementação realiza a relação que chamamos de *é-um*, podemos dizer que cachorro é um animal e o gato também.

A implementação da Figura 2:

```
public class Animal { … }
public class Cachorro extends Animal { … }
public class Gato extends Animal { … }
```

Na implementação de uma composição em vez de existir uma relação *é-um*, existe a relação *tem-um*. Nesse tipo de implementação, um componente é incluído na classe compositora que fará reúso da classe que modelou o objeto a ser utilizado. Os métodos reutilizados serão acionados a partir desse componente. O uso da composição é ilustrado na Figura 3:

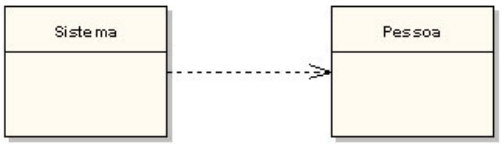


**FIG. 3 Exemplo de composição. Fonte: Medeiros**

Na implementação da Figura 3, `Sistema`, que é a classe compositora, tem um campo do tipo `Pessoa`. A implementação ficaria da seguinte forma:

```
public class Pessoa {... }
public class Sistema { Pessoa pessoa = new Pessoa(); ... }
```

Tanto a herança quanto a composição podem implementar reúso **caixa-preta** (não mostra como são feitas as implementações e protege a encapsulação dos dados). Esse reúso é obtido com o uso do modificador de acesso *private*. Composição e herança também podem implementar reúso **caixa-branca** (mostra como são feitas as implementações e não protege a encapsulação dos dados). Esse outro tipo de reúso é obtido com o uso do modificador de acesso *public* e é uma prática não recomendada.

A conveniência do uso da herança ou da composição deve ser verificada caso a caso. De acordo com Coad (1999), no uso de herança a classe derivada somente deve especializar um papel (ex: papel de pessoa), uma transação (ex: venda ou troca) ou um dispositivo/entidade/coisa (ex: veículo ou casa). Quando a nova classe não é uma especialização de papel, transação ou coisa, é melhor usar composição. Devemos respeitar também a condição *é-um* e não "exerce um papel de". No exemplo acima, `Cachorro` é um `Animal`. Nesse caso o uso de herança foi aplicado corretamente. Veja outro exemplo: em um sistema com os componentes `Carro` e `Veículo`, a classe `Carro` pode ser uma especialização da classe `Veículo`, porque todo carro é um tipo de veículo.

Para o uso de composição, devemos respeitar a condição *tem-um* e não "é um tipo especial de". Por exemplo: um carro tem uma roda, mas, isso não significa que uma

roda é um tipo de carro. Na implementação de um sistema com esses componentes, caso a classe `Carro` seja uma subclasse da classe `Roda`, ela estará ferindo o princípio necessário para a implementação de uma herança uma vez que um carro não é uma roda. O correto seria a classe `Carro` ter um campo do tipo `Roda`, implementação obtida por meio de composição.

Vamos verificar um exemplo de Bigonha (2015) que mostra um caso no qual o uso de composição seria melhor que herança. No modelo da Figura 4, o objeto `Passanger` é um `Person`, mas, em algum momento ele poderia exercer o papel `Agent`. Como o modelo é implementado com herança e possui a hierarquia apresentada, seria inviável realizar essa troca de papéis.

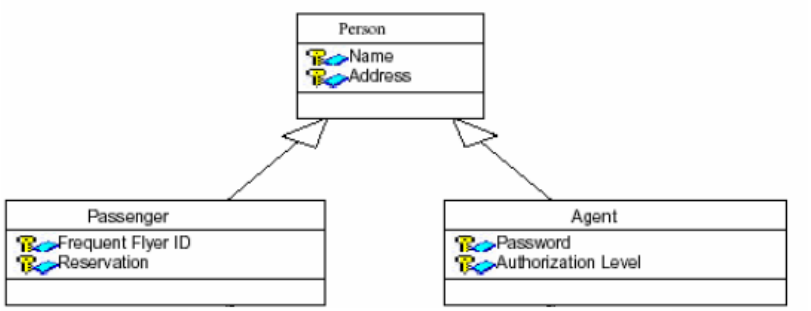


**FIG. 4 Herança x Composição. Fonte: Bigonha.**

Para resolver o problema, poderíamos criar um novo papel e apresentar esse novo modelo com o uso de herança múltipla, como ilustra a Figura 5.

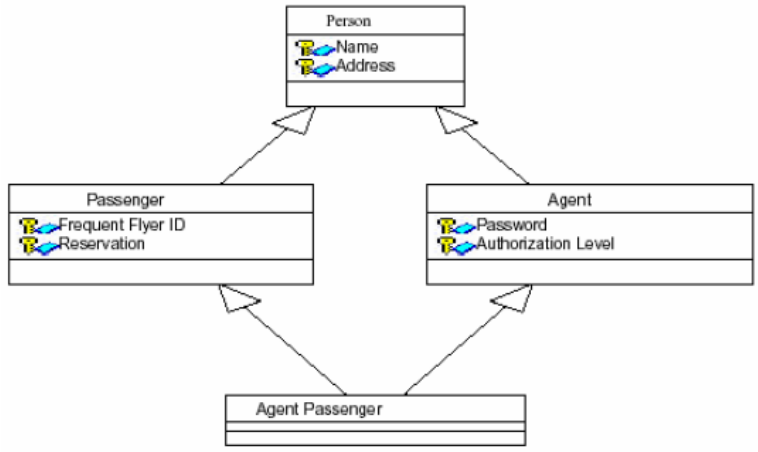


**FIG. 5 Herança x Composição. Fonte: Bigonha.**

Esse novo modelo supostamente resolveria o problema, mas, em herança as subclasses deviam expressar a relação "é um tipo especial de'', e não a de "pode exercer o papel de". Herança não foi a melhor solução para essa implementação. Vejamos então o uso de composição para essa mesma situação, Figura 6.

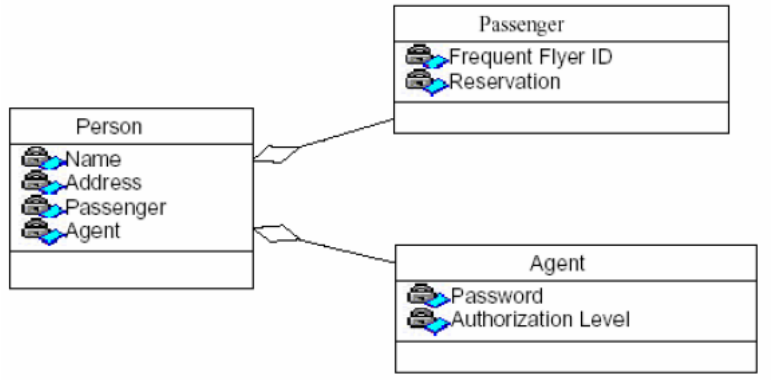


**FIG. 6 Herança x Composição. Fonte: Bigonha.**

Agora ao papel de Person pode mudar sempre que necessário e as condições podem ser satisfeitas. Poderíamos em uma nova solução, utilizar composição e herança para implementar esse caso. Verifique o modelo da implementação, ilustrado na Figura 7.

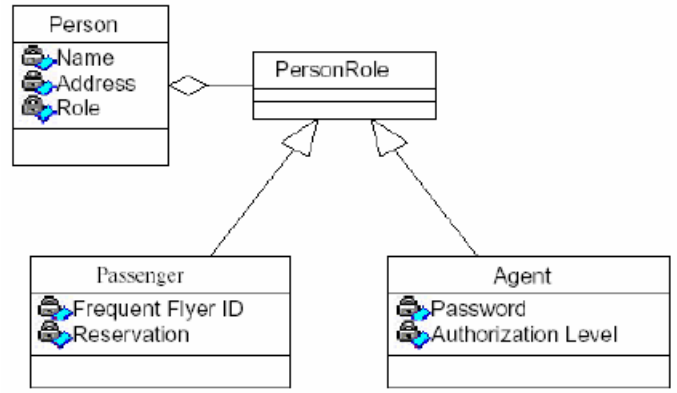


**FIG. 7 Herança x Composição. Fonte: Bigonha.**

No modelo da nova implementação, `Passenger` e `Agent` herdam de `PersonRole` que é um papel exercido por `Person`. Nessas condições, novos papéis podem ser criados para satisfazer os critérios necessários.

Como verificamos com a apresentação de todos os exemplos utilizados, o importante é que as implementações de herança e composição sejam aplicadas de forma coerente em cada caso. Para tal, devemos fazer uma análise conforme indicado nos exemplos acima para verificar as situações nas quais se aplicam a composição e outras nas quais será aplicável a herança.

## 3.2 PRINCÍPIO DA PROGRAMAÇÃO PARA A INTERFACE

Uma interface funciona como um modelo a ser seguido no desenvolvimento de qualquer classe que a implemente. Em uma interface são definidas somente as assinaturas dos métodos que as classes que vão implementá-la devem ter. Na implementação da

interface, a classe que a implementa deve desenvolver o método com a assinatura exata fornecida pela interface implementada.

Dentre os benefícios de se programar para uma interface, temos o fato de que um objeto pode ser substituído facilmente por outro. Verifique o exemplo:

```
interface I { void f( ); void g( ); void h( ); }

public class A1 implements I { ... }
public class A2 implements I { ... }
public class A3 implements I { ... }
public class A implements I { public T d;... }
```

Observe a utilização dessas interfaces pelos seguintes métodos:

```
...
public void s1(I x){ ... x.f( );...; x.g( );...; x.h( ); ... }

public void s2(A x){
... x.f( );...; x.g( );...; x.h( ); ... x.d( ); ...
}
```

O método `s1` recebe como parâmetro um objeto do tipo `I` e realiza uma execução. Esse método aceita objetos de qualquer classe que implementa I, ou seja, aceita objetos das famílias A, A1, A2 e A3. Enquanto o método `s2` só aceita objetos do tipo `A` porque seu parâmetro é do tipo classe. O método `s2` programou para a implementação enquanto o método `s1` programou para a interface, privilegiando o reúso.

Podemos concluir que programar para a interface é aumentar o grau de reúso. Se um método utiliza como parâmetro uma interface, ele se torna aplicável a mais objetos. Programando para as classes, produzimos métodos com menor grau de reúso.

## 3.3 PRINCÍPIO DA RESPONSABILIDADE ÚNICA

De acordo com Tom DeMarco (1979), não deve haver mais de um motivo para uma classe ser alterada. Esse é o SRP, Princípio da Responsabilidade Única. Esse princípio garante que uma classe deve ser implementada para cumprir um único objetivo. Caso essa classe tenha mais de um motivo para mudar, significa que ela está fazendo mais coisas do que deveria. Isso pode indicar que essa classe deve ser dividida para gerar novas classes e dividir esses objetivos. O problema de uma classe assumir mais de uma responsabilidade é que quanto mais responsabilidades ela assume, mais complexa ela se torna no sentido de que, quanto maior a quantia de responsabilidades, maior será a dimensão de possíveis alterações. Nesse caso, quando for preciso realizar alguma alteração nessa classe mais complexa, haverá maior risco de impactar módulos que não necessariamente deveriam ser impactados. Por isso, o projeto se torna mais frágil quando contém módulos que não respeitam o princípio da responsabilidade única.

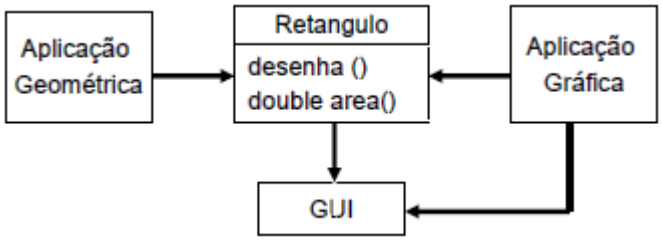


**FIG. 8 Classe Retangulo ferindo SRP. Fonte: Bigonha.**

Na Figura 8, podemos verificar a classe `Retangulo`, que possui o método `desenha`, o qual depende de `GUI`. `Aplicação Geometrica` só necessita utilizar o método `area` mas, possui o `GUI` disponível mesmo sem necessitar utilizá-lo. A `aplicaçao grafica`, por sua vez, teria disponível o método para calcular área, mesmo sem necessitar utilizar esse método. Essa classe deveria ser dividida, pois, possui duas responsabilidades em vez de uma.

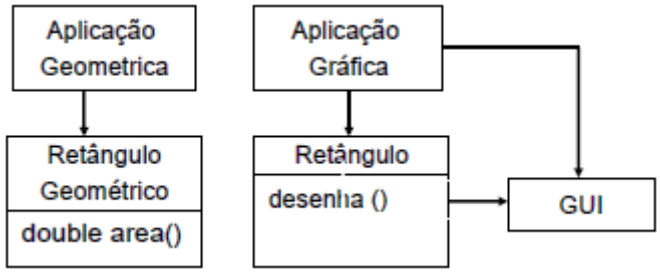


**FIG. 9 Refatoração SRP. Fonte: Bigonha.**

Na Figura 9 o problema ilustrado pela Figura 8 foi corrigido. A classe `Retangulo` foi dividida e criou-se também a classe `RetanguloGeometrico`. Agora as alterações na representação gráfica não vão afetar a aplicação geométrica, e as alterações no cálculo da área não irão afetar a aplicação gráfica.

Manter as classes com objetivos únicos torna o código mais fácil de ser entendido. Quando necessário, será mais rápido realizar também a sua manutenção. “Se uma classe tem mais de uma responsabilidade, então haverá mais de uma razão para alterá-la”. Nesse caso, quanto maior a ocorrência de classes com mais de uma responsabilidade, maior será a fragilidade do sistema uma vez que, se a classe apresentar algum problema ela afetará uma parte maior que ela afetaria se não possuísse mais de uma responsabilidade.

## 3.4 PRINCÍPIO DO ABERTO/FECHADO

De acordo com Meyer (1997), classes deverão ser abertas para extensão, mas fechadas para modificação. Esse é o OCP, Princípio do Aberto/Fechado. Esse princípio estabelece que as classes que estendam alguma outra classe devem utilizar as funcionalidades da classe estendida sem que essa superclasse precise ser alterada. Isso

significa que se o princípio for obedecido, o acréscimo de novas funcionalidades em um software não implica em alteração de suas classes, mas apenas na introdução de novas. Veja o exemplo do método `totalPrice`, desenvolvido para calcular preço de alguns tipos de peças:

```
public class Part {
      public double totalPrice(Part[ ] parts) {
            double total = 0.0;
                  for (int i = 0; i < parts.length; i++) {
                        total += parts[i].getPrice( );
            }
            return total;
      }
}

public class MotherBoard extends Part{
      public double getPrice( ){
            super.getPrice();
      }
}

public class Memory extends Part{
      public double getPrice( ){
            super.getPrice();
      }
}
```

E se devido ao aumento de imposto repentino de alguns produtos fosse necessário cobrar um preço adicional sobre certas peças? Por exemplo, se os preços de memória e de placa mãe tivessem que ser majorado em 30% e 45%, respectivamente, uma solução seria alterar o método `totalPtrice` para:

```
public double totalPrice(Part[ ] parts) {
      double total = 0.0;
      for (int i = 0; i < parts.length; i++) {
            if (parts[i] instanceof Motherboard)
                  total += (1.45* parts[i].getPrice( ));
            else if (parts[i] instanceof Memory)
                  total += (1.30* parts[i].getPrice( ));
                 else total += parts[i].getPrice( );
      }
      return total;
}
```

Imagine que fosse necessário agora diferenciar uma nova peça, seria necessário inserir um novo `if` na condição e ela poderia ficar cada vez maior a medida que a lista de produtos com preços diferenciados aumentasse. Essa nova estrutura funciona como um `switch` e deve ser evitada, pois, prejudica a manutenibilidade do código uma vez que, a necessidade de uma alteração pode desencadear uma série de outras alterações e tornar o código cada vez mais complexo.

Uma melhor solução é evitar o uso do switch, preservar o `totalPrice` original e embutir as variações de preço nos métodos `getPrice` de cada peça, como mostra o código a seguir.

```
public class Part {
      private double basePrice;
      public void setPrice(double price){basePrice=price;}
      public double getPrice( ){ return basePrice;}

      public double totalPrice(Part[ ] parts) {
            double total = 0.0;
                  for (int i = 0; i < parts.length; i++) {
                        total += parts[i].getPrice( );
            }
            return total;
      }

}

public class MotherBoard extends Part{
      public double getPrice( ){
            return 1.45*super.getPrice();
      }
}

public class Memory extends Part{
      public double getPrice( ){
            return 1.30*super.getPrice();
      }
}
```

Na solução acima, o acesso ao atributo que contém o preço base foi restringido e as subclasses passaram a utilizar o método de forma específica. Nesse caso, a classe `Part` ficou aberta para extensão e fechada para modificação e não precisa ser alterada a cada vez que uma peça sofrer alteração de preço ou uma nova peça com um preço diferente começar a ser vendida. Nessa nova implementação, foi utilizando um `switch` implícito também conhecido como *dynamic binding*. Como mostrado nesse código, nesse tipo de implementação, existirão ainda *cases* para a obtenção dos preços, mas, eles serão especificados diretamente nas classes que calculam o preço de cada peça.

Podemos concluir que devemos utilizar o princípio do aberto/fechado para que sempre que precisarmos estender um comportamento de uma classe, seja criado um novo código em vez de alterar o código já existente. Assim, diminuímos o grau de acoplamento, já que quando uma classe estende o comportamento de outra sem alterá-lo, essas classes ficam mais independentes e as alterações realizadas em uma delas não afetarão fortemente a outra. É como mostramos no exemplo com a solução do problema. Caso haja alteração somente no preço de uma `MotherBoard`, não será necessário alterar a classe `Part`. Diminuímos também os custos de manutenção já que, como as classes não estarão fortemente acopladas, ao realizar uma alteração em um método de uma delas, não será necessário alterar a(s) outra(s).

## 3.5 PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

De acordo com Liskov (1987), na implementação de uma herança as funções que usam referências para objetos de uma classe devem ser capazes de referenciar objetos das subclasses desta classe sem o saberem. Esse é o LSP, Princípio da Substituição de Liskov. Esse princípio estabelece que as classes filhas devem implementar a superclasse sem alterar o contrato de implementação, ou seja, as classes filhas devem implementar o que a superclasse propõe em seu escopo.

Para mostrarmos um exemplo de violação do LSP, considere uma implementação parecida com o exemplo utilizado por Martin (2006), trata-se da seguinte classe `Retangulo`:

```
public class Rectangle {
      private double width, height;
      public Rectangle(double w, double h) {
            width = w; height = h;
      }

      public double getWidth( ) {return width;}
      public double getHeight( ) {return height;}
      public void setWidth(double w){width = w;}
      public void setHeight(double h){height = h;}
      public double area( ) {return width*height;}
}
```

Agora vejamos a seguinte classe `Quadrado`, considerando o ponto de vista matemático que todo quadrado é um retângulo:

```
public class Square extends Rectangle {
      public Square(double w) {
            super(w,w);
      }
      public void setWidth(double w) {

            // Violação do princípio aqui
            super.setWidth(w); super.setHeight(w);
      }
      public void setHeight(double h) {

            // Violação do princípio aqui
            super.setHeight(h); super.setWidth(h);
      }
}
```

Conforme a indicação dos locais no código acima onde o princípio é violado, essas linhas da implementação desrespeitam o contrato de implementação da superclasse `Retangulo` na qual quando se modifica a altura, só se modifica a altura e quando se modifica a largura, só se modifica a largura. Na classe `Quadrado` quando a altura é

modificada, a largura também é modificada e vice-versa. Verifique o efeito nas funções de teste abaixo:

```
static void testLSP(Rectangle r) {
      r.setWidth(4.0);
      r.setHeight(8.0);
      System.out print( “4.0X8.0 = " + r.area( ));
}


public static void main(String [ ] args) {
      Rectangle r = new Rectangle(1.0, 1.0);
      Square s = new Square(1.0);
      testLSP(r); // Funciona
      testLSP(s); // Nao Funciona!
}
```

Saída:
4.0X8.0 = 32.0 (Certo)
4.0X8.0 = 64.0 (Errado)

Dizemos que `testLSP(s)` não funciona por que o resultado impresso não corresponde ao esperado pela leitura de seu código.

Vejamos a solução do problema:

```
public class Rectangle {
      private double width, height;
      public Rectangle(double w, double h) {
            width = w; height = h;
      }

      public double getWidth( ) {return width;}
      public double getHeight( ) {return height;}
      public double area( ) {return width * height;}
      }

      public class Square extends Rectangle {
            public Square(double w) { super(w,w); }
}
```

No exemplo acima, são retirados os métodos `setWidth` e `setHeight` das duas classes e após criadas, a altura e a largura não mais podem ser alteradas. Agora a classe `Quadrado` respeita o contrato de implementação de `Retangulo`. Novo método `testLps`:

```
static void testLSP(Rectangle r) {
      double a = r.getWidth( );
      double b = r.getHeight( );
      System.out print(a+ "X "+b+" = " + r.area( ));
}
```

Aplicação do teste:

```
public static void main(String [ ] args) {
      Rectangle r = new Rectangle(4.0, 8.0);
      Square s = new Square(4.0);
      testLSP(r); // Funciona
      testLSP(s); // Funciona!
}
```

Saída:
4.0X8.0 = 32 (Certo)
4.0X4.0 = 16 (Certo)

Respeitar o contrato significa ter cuidado em respeitar as pré-condições e pós-condições definidas no contrato da superclasse. Não é possível que uma classe filha tenha uma restrição maior que a superclasse sobre as pré-condições, mas, é possível ter uma maior restrição sobre as pós-condições.

Como um exemplo de restrição sobre as pré-condições, podemos citar o exemplo de uma função declarada na superclasse que retorne um número inteiro com valores no intervalo de 1 a 15. A subclasse não poderia restringir esse mesmo atributo com os valores de 4 a 6, mas, poderia ampliar o intervalo, por exemplo, com os valores de 1 a 20. Como um exemplo de restrição sobre as pós-condições podemos citar uma função implementada na superclasse que retorne um valor inteiro no intervalo de 5 a 15. A subclasse não poderia ampliar o retorno desse método com um intervalo de 5 a 30, mas, poderia restringir, por exemplo, com um intervalo de 5 a 10.

O LSP é outro princípio importante para facilitar a manutenção e expansão do software. Uma vez que uma subclasse cumpra o contrato de implementação da superclasse, ela terá implementado de maneira organizada os métodos necessários e facilitará o reúso e a compreensão do código.

## 3.6 PRINCÍPIO DA SEGREGAÇÃO DE INTERFACES

O princípio da segregação da interface também conhecido pela sigla ISP é abordado no livro de Martin (2006), ele diz que interfaces específicas são melhores que uma interface geral. A falta a esse princípio pode levar algumas classes a obterem conhecimento de métodos que elas não necessariamente vão utilizar. Isso acontece porque se o código possuir uma única interface geral, as implementações daquela interface conhecerão todos os métodos ali disponíveis mesmo que devam implementar somente alguns. É interessante que em um código, uma classe que implemente uma interface tenha disponível nela somente os métodos que deverá utilizar para evitar sua poluição com a implementação de métodos desnecessários.

Considere por exemplo, um sistema que controla a abertura e fechamento de portas:

```
public class Porta {
      public void fecha( ){ … }
      public void abre( ) { … }
      public boolean portaAberta( ) { … }
}
```

As portas de cofre devem ser abertas somente em determinados horários:

```
public classe PortaCofre extends Porta {
      public void defineHorario ( ){ … }
      public boolean podeAbrir( ) { … }
}
```

As Portas são utilizadas no sistema como um todo:

```
public class Predio {
      private Porta[ ] porta; …
}
```

O sistema funciona de forma conveniente, mas, seria interessante acrescentar a possibilidade de que seja disparado um alarme caso as portas de cofre fiquem abertas após um limite de tempo. O sistema já possui uma classe `Temporizador` implementada, é necessário utilizá-la com o mínimo de alteração no sistema. O `Temporizador` tem um tempo específico para disparar o alarme:

```
public class Temporizador {
      public void registra (int t,Monitorado c){ … }
}

public class Monitorado {
      public disparaAlarme( ){ ... }
}
```

Conforme veremos na implementação a seguir, a classe `Porta` estenderá a classe `Monitorado`. O problema dessa nova implementação é que a classe `Porta` terá acesso ao método `disparaAlarme` sem necessidade, uma vez que somente a classe `PortaCofre` fará o uso do método. A presença de `disparaAlarme` em `Porta` polui sua interface.

```
public class Porta extends Monitorado{
      public void fecha( ){ … }
      public void abre( ) { … }
      public boolean portaAberta( ) { … }

      // Aqui o ISP foi ferido
      public void disparaAlarme() { }
}
public classe PortaCofre extends Porta {
      public void disparaAlarme ( ){ … }
      public void defineHorario ( ){ … }
      public boolean podeAbrir( ) { … }
}
```

Solução do problema:

```
public class Temporizador {
     public void registra (int t, Monitorado c);{ …}
}
public interface Monitorado {
     public disparaAlarme( )
}

public class Porta {
     public void fecha( ){ … }
     public void abre( ) { … }
     public boolean portaAberta( ) { … }
}

public class PortaCofre extends Porta
                  implements Monitorado {
     public void defineHorario ( ){ … }
     public boolean podeAbrir( ) { … }
     public void disparaAlarme( ) { … }
}
```

Agora o ISP passou a ser respeitado, uma vez que as classes estão implementando somente os métodos que realmente irão utilizar. A classe `Monitorado` passou a ser uma interface e `PortaCofre` agora implementa essa interface além de estender `Porta`. Assim, foi eliminada a poluição causada com o problema da implementação, pois, `Porta` não precisa mais ter acesso ao método `disparaAlarme`. As classes não terão que incorporar métodos que não precisam utilizar e o aplicativo fica mais organizado e fácil de entender.

## 3.7 PRINCÍPIO DA INVERSÃO DA DEPENDÊNCIA

O princípio da inversão da dependência, também abordado no livro de Martin (2006) e conhecido pela sigla DIP, sugere abstração de dados. Esse princípio estabelece que o código cliente conheça um objeto somente por sua interface. De acordo com Martin (2006) módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações. No exemplo a seguir, podemos demonstrar uma violação de princípio com a seguinte implementação para efetuar cópia de arquivo:

```
public class A {

      public void copia(byte device) {

            char c; R r = new R(...); W1 w1; W2 w2;

            if (device == D1) w1 = new W1(....);
            else w2 = new W2 (...);
            while ( (c = r.read()) != EOF ) {
                  if (device == D1) w1.write(c);
                  else w2.write(c);
            }
      }
}
```

Nessa implementação, a classe `A` depende das classes `R`, `W1` e `W2`, classes concretas. O problema está ocorrendo porque o método `copia` sempre testa qual o dispositivo para saber onde imprimir. Ele entra nos detalhes de cada dispositivo passado como parâmetro, nesse caso, um método de alto nível está dependendo de detalhes de um módulo de nível mais baixo.

Verifique agora a solução do problema:

```
public interface Reader {public char read( );}

public interface Writer{public void write(char c);}

public class A {
      public void copia(Reader r, Writer w) {
            char c;
            while ( ( c = r.read( )) != EOF ) {
                  w.write(c);
            }
      }
}
```

Na nova implementação foram implementadas interfaces específicas para `Reader` e `Writer`. Agora o método cópia não precisa mais testar o dispositivo antes de realizar a impressão. Essa responsabilidade é de quem chama o método cópia. Estar de acordo com o DIP facilita a manutenção e traz robustez aos códigos, além de contribuir com encapsulação de dados.

## 3.8 CONCLUSÃO

Estudamos aqui a importância de escolher corretamente entre herança e composição e vimos que essa escolha é guiada pela análise da situação. Estudamos que para privilegiar o reúso é sempre melhor programar para a interface em vez de programar para a implementação. Estudamos a importância da responsabilidade única em cada

classe. Aprendemos que devemos construir módulos fechados para modificação e abertos para extensão. Vimos o Princípio da Substituição de Liskov que foi criado para salientar que no reúso é importante cumprir o contrato de implementação. Foi mostrado o Princípio da Segregação da Interface para salientar que interfaces específicas são melhores que uma interface geral. Estudamos por fim, o princípio da Inversão da Dependência para entender a importância de implementar de tal forma que módulos de alto nível não dependam de módulos de baixo nível.

Ao discutir os princípios mais utilizados na construção de software podemos perceber o quanto é importante uma boa estruturação e aplicação de boas práticas de programação. Não utilizar esses modelos que mostram o caminho da programação guiada pelas boas práticas pode trazer sérias consequências como a elevação da complexidade dos módulos e a dificuldade de manutenção.

# CAPÍTULO 4

# ESTILOS DE PROGRAMAÇÃO

Estilos de programação são resultados do estudo das aplicações dos métodos mais adequados para a organização, redução da complexidade de um código e redução do tempo de execução. A adoção de bons estilos torna os aplicativos mais organizados e manuteníveis. No conjunto de estilos apresentados temos: o uso adequado de efeito colateral em funções, os benefícios de implementar objetos como máquinas de estados, a importância de manter a consistência de estados e interfaces, a importância do uso de polimorfismo de inclusão, a necessidade de aplicação da convenção de nomes para nomear itens no código, uso de endentação, comentários e a importância de reduzir o escopo.

## 4.1 EFEITO COLATERAL EM FUNÇÃO

Assunto abordado por Cargill (1992) e Meyer (1997), uma função produz efeito colateral quando além de retornar um valor, ela altera o estado do objeto que está tratando, ou seja, ela muda o valor do objeto em questão. O efeito colateral pode ser necessário em alguns casos, mas, deve ser evitado por ser nocivo de forma geral.

Para ilustrar a ocorrência de efeito colateral, utilizaremos o seguinte exemplo simples:

```
class A{

      int x = 0;

      public int calculoX(){
            return x = x+1;
      }
}
```

Caso a função acima fosse utilizada como o teste de uma condição, qual seria o resultado de `if(a.calculoX() == a.calculoX())`, para `A`? O resultado seria *false*, pois, na primeira vez em que a função é chamada ela retorna um, na segunda ela retorna dois. Nesse caso, as duas chamadas da função não retornarão o mesmo valor. Uma situação como essa pode prejudicar o raciocínio equacional.

Na maior parte das vezes, o efeito colateral nas funções deve ser evitado, pois cada função deve ser executada em prol de um único objetivo e sem surpresas a cada nova execução da mesma. De toda forma, podemos analisar alguns casos onde seria benéfico o uso do efeito colateral em funções. Depende da finalidade de uso da função.

Todo objeto tem um estado abstrato e um estado concreto. O estado abstrato é o estado percebido pelo usuário do objeto. O estado concreto é o estado definido pela

implementação do objeto. Mudanças no estado concreto podem ou não serem percebidas pelo usuário do objeto.

O efeito colateral no estado abstrato do objeto é como ocorre no exemplo mostrado, o objeto sofreu um efeito colateral de forma que a cada vez que a função for chamada, ela retornará valores diferentes. Mas, no caso de efeito colateral no estado concreto do objeto, esse tipo de efeito colateral produz um efeito interno que não é percebido externamente e é constantemente utilizado na implementação de objetos como máquinas de estado. Como exemplo de aplicação de efeito colateral no estado concreto do objeto, podemos citar o uso de variáveis que armazenam os valores dos últimos objetos acessados de uma lista para facilitar a pesquisa e a inserção de itens na lista. Esse tipo de efeito colateral não é nocivo e ajuda a reduzir o custo das operações.

Outro exemplo muito interessante do uso do efeito colateral apenas no estado concreto é o funcionamento de uma memória cache em um navegador de internet. Quando acessamos uma página *web* pela primeira vez, o navegador utilizado carrega todos os itens daquela página para exibi-lo, mas, ao acessar pela segunda vez, podemos notar que a página é carregada de forma mais rápida. Isso ocorre porque o cache trabalha com efeito colateral para armazenar a página visitada e exibi-la mais rápido nas próximas vezes em que ela for solicitada ao navegador. Essa é uma forma de utilizar efeito colateral de forma benéfica ao resultado da execução.

## 4.2 IMPLEMENTAÇÃO DE OBJETOS COMO MÁQUINAS DE ESTADO

Assunto abordado por Meyer (1997), um objeto pode assumir diferentes estados ao longo da execução de um código. Esses estados podem ser alterados em vários momentos de acordo com as funções que tratam esse objeto. A implementação de objetos como uma máquinas de estado, auxilia na redução dos custos de vários tipos de operações que equivale a economia de tempo de execução e melhora o desempenho de aplicativos.

Veja essa implementação de uma lista e suas operações alterar, pesquisar, inserir e apagar itens da lista onde a noção de efeito colateral restrito ao estado concreto não é usado:

```
// Lista
class IntList {
      private static class Node {...}
      private Node firstElement;
      private int nElements;
      public IntList( ) { }

      //Operações de IntList...
}
```

```
// Método para alterar valor de um item da lista com custo O(n)
public void changeValue(int i,int v)throws E1{
      Node e; int j;
      if (i<1 || i>nElements)
            throw new IndexError();
      e = getAddress(i);
      e.info = v;
}

// Método para pesquisar item na lista com custo O(n)
public int search(int v) {
      Node e; int j;
      if (empty( )) then return 0;
      for (j = 1 , e = firstElement ; j<=nElements; j++,e =
      e.right) {
                  if (v == e.value) return j;
      }
      return 0;
}

// Método para inserir item na lista com custo O(n)
public void insert(int i, int v) throws E1 {
      Node previous, node; int j;
      if (i<0 || i>nElements)
            throw new IndexError();
      node = new Node(v);
      if (i == 0) {
            node.right= firstElement;
            firstElement = node;
      } else {
            previous = getAddress(i);
            node.right =previous.right;
            previous.right =node;
      }
      nElements++;
}


// Método para deletar um item da lista com custo O(n)
public void delete(int i) throws IndexError {
      Node previous, toBeDeleted;
      if (i<1 || i>nElements)
            throw new IndexError();
      if (i == 1) {
            firstElement = firstElement.right;
      }
      else {
            previous = getAddresss(i-1);
            toBeDeleted = previous.right;
            previous.right = toBeDeleted.right;
      }
      nb_elements--;
}
```

```
// Uso de IntList
class User {
      public static void main(String[ ] a) {
            LinkedList q;
            int k, v1 = 1, v2 = 2, v3 = 3, v4 = 4, int n = 5;
            ...;
            k = q.search(v1);
            v4 = q.value(k);
            q.insert(n, v2);
            q.changeValue(n,v3);
            q.delete(n);
            ..............
      }
}
```

A lista apresentada possui as operações básicas para o funcionamento de uma lista, porém, suas operações possuem o custo O(n), pois, em todas as operações a lista é percorrida desde o início. Esse tipo de implementação não poupa tempo de execução. Essa lista poderia ser reimplementada de modo que suas operações tivessem um menor custo. A saída seria implementar essa lista como uma máquina de estados.

Na implementação dessa lista como uma máquina de estados, será utilizada a noção de nodo corrente para que as operações sejam mais eficientes. Essa noção será obtida pela construção do nodo ativo que armazena informações de acordo com as últimas operações realizadas. E todas as operações são definidas em função do nodo corrente.

Nas Figuras 10 e 11, podemos verificar o funcionamento da nova lista de acordo com o estado abstrato, ou seja, o estado percebido pelo usuário e de acordo com o estado concreto, ou seja, o estado definido pela implementação do objeto. Na percepção do usuário, o tempo de execução será melhorado e ele notará um retorno mais rápido durante as execuções. Na demonstração do estado concreto, podemos verificar como funcionarão as operações de acordo coma existência do nodo `active`.

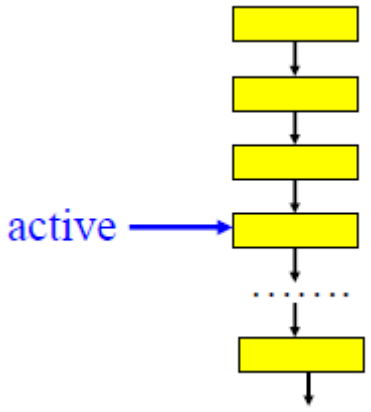


**FIG. 10 Vista do Estado Abstrato do ADT. Fonte: Bigonha.**

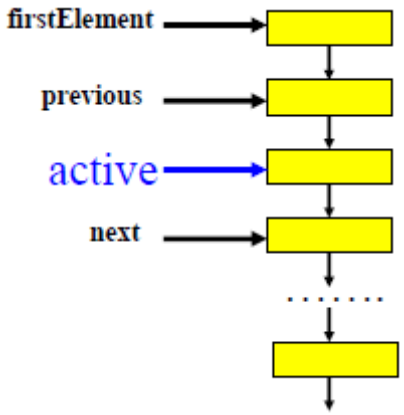


**FIG. 11 Vista do Estado Concreto do ADT. Fonte: Bigonha.**

Vejamos a nova implementação da lista:

```
class IntList {
      private static class Node {
            int value; Node right;
            Node(int value){this.value = value;}
      }
      private int nElements, position;
      private Node firstElement;
      private Node previous, active, next;

      //Operações de IntList ...
}

// É criado o nodo Ativo para definir o nodo corrente da lista
public void start( ) throws ActiveError {
      if (empty( )) throw ActiveError( );
      previous = null;
      active = firstElement;
      next = active.right ;
      position = 1;
}

/* Método para alterar valor do nodo corrente da lista com custo
O(1) */
public void changeValue( int v)
      throws ActiveError {
      if (offLeft( ) || offRight( ))
      throw new ActiveError( );
      active.value = v;
}

// Método para pesquisar item na lista com custo O(n)
public boolean search(int v) {
      try {
            for (start();!offRight() ; forth())
                  if (active.value = v) return true;
      }
      catch(ActiveError e) { }
      return false;
}
```

```
/* Método para inserir item à direita do nodo corrente da lista
com custo O(1) */
public void insertRight(int v)
      throws InsertionError {
            Node node = new Node(v);
      if (empty() ) {
            firstElement = node; active = node;
            previous = null; next = null; position = 1;
      } else {
            if (offLeft( ) || offRight( ))
                  throw new InsertionError( );
            node.right = next; active.right = node;
            previous = active; active = node;
      }
      nElements++;
}

// Método para deletar nodo corrente da lista com custo O(1)
public void delete() throws DeletionError {
      if (offLeft() || offRight() || empty( ))
            throw new DeletionError( );
      active = next;
      if (active != null) next = ative.right;
      if (position == 1) {
            firstElement = active;
            if (firstElement == null) position = 0;
      } else previous.right = active;
      nElements--;
}
```

Note que a função pesquisar produz efeito colateral no objeto no sentido que ela pode alterar o indicador de elemento ativo da lista. Esse efeito está definido na interface do tipo IntList.

```
// Uso da nova classe
class User {
      public static void main(String[ ] a) {
            IntList q; boolean b; int v1=1;
            int v2=2, v3=3, v4=4, n=5;
            b = q.search(v1);
            v4 = q.value();
            q.go(n);
            q.insertRight(v2);
            q.changeValue(v3);
            q.delete();
      }
}
```

Na nova implementação acima, após ser criado o nodo ativo, as operações passaram a ser realizadas com base nos estados armazenados pelo nodo. Nesse caso, as novas operações, exceto `search` passaram a ter custo O(1).

É interessante implementar objetos como máquinas de estado uma vez que armazenando os estados, é possível obter melhorias como o acesso mais rápido dos dados em futuras execuções. O uso dessa prática proporciona economia no tempo de execução, ou seja, mais rapidez e eficiência.

## 4.3 ABSTRAÇÕES CORRETAS

Assunto abordado por Staa (2000), a escolha de abstrações corretas requer um bom planejamento da construção do código para que ele fique de forma mais simples possível e atenda aos requisitos necessários. Muitas vezes poderia ter sido criado um número menor de classes que o número implementado, e o aplicativo ficou mais complexo do que deveria. Um dos benefícios da escolha das abstrações corretas é diminuir a complexidade na medida do possível.

Para diminuir a complexidade do código, é necessário realizar esse planejamento para verificar quais classes realmente deverão ser criadas. Pode, muitas vezes, ser interessante, criar classes que implementem vários objetos ao invés de classes que implementem um único objeto pois, isso evita a criação de várias classes que poderiam ser uma só.

Durante esse planejamento do sistema por meio do projeto de sua arquitetura ou mesmo durante a construção de um simples diagrama UML, os engenheiros de software e desenvolvedores devem se atentar para as classes planejadas que não serão necessárias ou que poderiam ter suas funções encaixadas em outras classes. Eles devem se atentar também para os relacionamentos entre as classes e como devem ser implementados esses relacionamentos para que o programa fique menos complexo possível. Eles devem verificar ainda quais os métodos serão implementados em cada classe para que não haja duplicidade de código.

Ainda durante o planejamento, qualquer estrutura planejada para trabalhar como um componente do aplicativo que possuir um nome (substantivo) será uma classe. Os procedimentos realizados por essas classes e nomeados com verbos serão suas funções. As propriedades de cada um desses componentes que sejam nomeadas com adjetivos ou substantivos serão seus atributos.

Abstração é o ato de conhecer o que um código faz e não como ele faz. Abstrair depende do objetivo da abstração. Por exemplo, uma pessoa na perspectiva de um médico é diferente de uma pessoa na perspectiva de um Analista de Sistemas. Essas duas pessoas possuem características diferentes. As abstrações possíveis das construções de um programa podem ser realizadas por meio de:

- Função: conjunto de comandos que realiza algum tipo de operação, retorna um valor e pode causar efeito colateral.

- Procedimento: conjunto de declarações e comandos que sempre vão causar algum efeito colateral.

- Abstração de dados: estrutura com representação encapsulada que torna conhecidas somente as suas operações.

- Tipo Abstrato de Dados: tipo com representação encapsulada que torna conhecidas somente as suas operações.

- Tipo Abstrato de dados parametrizado: tipo com representação parametrizada e encapsulada que torna conhecidas somente as suas operações.

Funcões e procedimentos devem ser implementados como métodos de classes. Abstração de dados e tipos abstratos de dados são implementados como classes e tipos abstratos parametrizados são realizados por classes genéricas. Assim, encontrar as abstrações corretas equivale a identificar as abstrações de dados e os tipos abstratos de dados necessários à implementação da aplicação. Resumindo, para escolher corretamente as abstrações é necessário conhecer cada uma das abstrações possíveis, conforme a lista mostrada e verificar em cada caso, qual delas se aplica à construção de uma estrutura específica.

## 4.4 CONSISTÊNCIA DE ESTADOS E INTERFACES

Assunto abordado por Cargill (1992), manter a consistência de estados e interfaces é muito importante. Toda operação deve deixar o objeto sobre o qual ela atua em um estado consistente. Para isso, devemos definir os invariantes dos objetos da classe e observar para que todas as construtoras e operações respeitem as condições estabelecidas.

A falta de consistência pode provocar resultados inesperados durante a execução. Imagine o exemplo de implementação de uma pilha com algumas construtoras. Caso o topo seja apontado para um local em uma das construtoras e para outro local em outra construtora, haverá uma inconsistência aí e o algoritmo poderá funcionar de forma inesperada em alguns momentos.

Veja o exemplo de implementação da classe `MyString` com problemas de consistência:

```
class MyString {

      //Campos privados de MyString
      private char[ ] s;
      private int length;
```

```
//Funções auxiliares privadas de MyString
private void copy(char[]d, char[] a){
      int i;
      for(i=0; i<a.length && a[i]!='\0'; i++){
            d[i] = a[i];
      }
      if (i < d.length)
            d[i] ='\0';
}

private void join(char[]d,char[]a,char[]b){
      int i,j;
      for (i=0; i<a.length && a[i]!='\0'; i++){
            d[i]= a[i];
      }
      for (j=0; j<b.length; j++)
            d[i+j] = b[j];
      d[i+j] = '\0';
}

private int strlen(char[] a){
      int i = 0;
      while ((i < a.length) && (a[i]!='\0')) i++;
      return i;
}

//Construtoras de MyString
public MyString() {
      s = new char[80]; length = 0;
}
public MyString(int n) {
      s = new char[n]; length = n;
}
public MyString(char[ ] p) {
      length = strlen(p); s = new char[length + 1];
      copy(s,p);
}



public MyString(MyString str){
      length = strlen(str); this.s = new char[length];
      copy(this.s, str.s);
}

//Operações de String
public void assign(char[ ] f) {
      copy(this.s, f);
      length = strlen(f);
}
public void print() {
      int k = strlen(this.s);
      for(int i=0; i< k; i++)
      System.out.print(s[i]);
      System.out.println();
}
```

```
      //Definição de String
      public void concat(MYString a, MyString b){
            length = strlen(a) + strlen(b);
            this.s = new char[length];
            join(this.s, a.s, b.s);
      }
      public int length( ) {
            return length;
      }

}
```

Observe que existe um problema com consistência de estados nas duas primeiras construtoras do exemplo dado:

```
public MyString() {
      s = new char[80]; length = 0;
}
public MyString(int n) {
      s = new char[n]; length = n;
}
```

O campo `length` recebe `0` na primeira implementação e `n` na segunda. Na primeira `length` é o tamanho da cadeia de caractere dentro da área alocada, enquanto na segunda `length` é o tamanho da área alocada. Essa implementação está incorreta, pois, ocorreu uma inconsistência na definição no estado do objeto e isso pode causar conflitos durante a execução do código. O correto seria que o campo `length` estivesse com um tamanho definido de forma igual nas duas construtoras.

Uma solução do problema seria unificar essas duas construtoras para que mantenham a consistência no estado do campo `length`:

```
public MyString(int n) throws Erro{
      if (n < 0) throw new Erro() ;
            s = new char[n]; length = 0;
}

public MyString( ) throws Erro {this(80);}
```

Mesmo com a solução apresentada para esse problema, estamos com um problema na definição da semântica do campo `length`, pois, não sabemos ao certo o que ele representa. Verifique nas partes do código original examinadas separadamente a seguir:

```
// Aqui, length é o tamanho do string armazenado:
public MyString(char[ ] p) {
      length = strlen(p); s = new char[len + 1];
      copy(s, p);
}
```

```
// Aqui, length é o tamanho da área reservada (array):
public MyString(int n) throws Erro{
      if (n < 1) new Erro();
            s = new char[n]; length = n;
}

// Aqui, length é o tamanho do array ou do string:
public MyString(MyString str){
      length = str.length;
      s = new char[length + 1];
      copy(s, str.s);
}

// Aqui, length é o tamanho do string armazenado:
public void assign(char[ ] c) {
      copy(s, c);
      length = strlen(c);
}

// Aqui, length é o tamanho do string acrescido de uma unidade
public void concat(String a, String b){
      length = a.length + b.length;
      s = new char[length+1];
      join(s, a.s, b.s);
}
```

Precisamos definir uma única semântica para o `length` em todas as operações, então foi registrado na definição da classe o invariante que estabelece `length` como o tamanho do `String`. Verifique a solução do problema, na nova versão de `MyString`:

```
class MyString {

      //Campos Privados de MyString
      private char[] s;

      /*Com a definição do campo length seguida de seu comentário
      a seguir, ele é registrado como invariante na definição da
      classe*/
      private int length; // length == strlen(s)

      //Funções auxiliaries privadas de MyString
      private void copy(char[]d,char[]a) throws Erro{
            int i;
            if (d==null || a==null) throw new Erro();
            for(i=0; i<a.length && a[i]!='\0'; i++){
                  d[i] = a[i];
            }
            if (i < d.length) d[i] ='\0';
      }
```

```
private void join(char[]d,char[]a,char[]b) throws Erro{
      int i,j;
      if (d==null) throw new Erro();
      if (a==null || b == null) throw new Erro();
      int nc = strlen(a) + strlen(b) + 1;
      if (d.length < nc) throw new Erro();
      for (i=0; i<strlen(a); i++) d[i] = a[i];
      }
      for (j=0; j<strlen(b); j++) d[i+j] = b[j];
      d[i+j] = '\0';
}

private int strlen(char[] a){
      int i = 0;
      if ( a == null) return 0;
      while ((i < a.length) && (a[i]!='\0')) i++;
      return i;
}

//Construtoras de MyString
public MyString(int n) throws Erro{
      if (n < 0) throw new Erro();
      s = new char[n];
      length = 0;
}
public MyString( ){ this(80);}

public MyString(char[] p) throws Erro {
      length = strlen(p);
      s = new char[length + 1];
      copy(s, p);
}

public MyString(Mystring str){
      length = str.length;
      s = new char[length+1];
      try {copy(s, str.s);} catch Erro { };
}

//Operações de string
public void assign(char[ ] c) throws Erro {
      copy(s,c); length = strlen(c);
}

public void print() {
      int k = strlen(s);
      for (int i=0; i< k; i++)
            System.out.print(s[i]);
      System.out.println();
}
```

```
      //Definição de String
      public void concat(MyString a, MyString b){
            length = a.length + b.length;
            s = new char[length+1];
            join(s, a.s, b.s);
      }

      public int length(){
            return length;
      }
}
```

Resumindo em quatro passos o que é preciso fazer para manter a consistência de estados e interfaces. Primeiro é necessário identificar quais devem ser os invariantes de uma classe, ou seja, quais condições devem ser respeitadas durante a representação ou alteração de estado de um objeto. Segundo, os estados dos objetos devem ser definidos de forma consistente. Terceiro, a semântica de um campo deve ser a mesma em todas as operações nas quais ele esteja envolvido. Quarto, o invariante deve ser registrado na definição da classe para facilitar a manutenção do código.

## 4.5 USO DE POLIMORFISMO DE INCLUSÃO

Assunto abordado por Cardelli (1985), o polimorfismo de inclusão é o polimorfismo que decorre do mecanismo de herança também conhecido como *é-um* e subtipos. A questão ser observada é a implementação correta do contrato da classe em todas as suas subclasses e se os campos estão sendo tratados de forma consistente. É necessário verificar ainda se as informações estão distribuídas corretamente ao longo da hierarquia. Itens usados de forma geral devem estar na superclasse e itens particulares, nas subclasses. Os subtipos devem ser aceitos onde os tipos forem esperados.

Veja o exemplo da implementação a seguir de um sistema de guarda de veículos em uma garagem. Nele observamos problemas como inconsistência no uso do campo `plate` por algumas operações e a operação `identify` que poderia ter sua implementação realizada de uma forma mais geral:

```
//Classe Veículo
class Vehicle {
      protected String plate;
      public vehicle(){ };
      public Vehicle(String p){
            plate = p;
      }
      public String identify(){
            return "generic vehicle";
      }
}
```

```
//Classe Carro, subclasse de Veículo
class Car extends Vehicle {
      public Car(){super();}
      public Car(String p){super(p);}
      public String identify {
            return "car with plate" + plate;
      }
}

//Classe Caminhão, sublcasse de veículo
class Truck extends Vehicle {
      public Truck() { super(); }
      public Truck(String p) {super(p);}
      public String identify( ){
            return "truck with plate " + plate;
      }
}

//Classe Garagem
class Garage {
      private int maxVehicles;
      private Vehicle[ ] parked;
      public Garage(int max) {
            maxVehicles = (max < 1)? 1 : max;
            parked = new Vehicle[maxVehicles];
      }

      //Preencher vaga
      public int accept(Vehicle v) {
            for(int bay = 0; bay<maxVehicles; ++bay )
                  if (parked[bay] = null) {
                        parked[bay] = v; return bay;
                  }
            return -1; // no bay available;
      }
      //Devolver o veículo e liberar a vaga
      public Vehicle release(int bay){
            if (bay<0 || bay>=maxVehicles)return null;
            Vehicle v = parked[bay];
            parked[bay] = null;
            return v;
      }


      //Listar veículos estacionados na garagem
      public void listVehicles(){
            for(int bay = 0; bay < maxVehicles; ++bay)
            if( parked[bay] != null ) {
                  System.out.println ("Vehicle in bay " + bay +
                  "is "
                  parked[bay].identify());
            }
      }
```

```
      //Um dia na garagem
      public static void main(String[ ] args){
            Garage park(15);

            //Veículos em operação
            Vehicle c1 = new Car("RVR 101");
            Vehicle c2 = new Car("SPT 202");
            Vehicle c3 = new Car("CHP 303");
            Vehicle c4 = new Car("BDY 404");
            Vehicle c5 = new Car("BCH 505");

            Vehicle t1 = new Truck("TBL 606");
            Vehicle t2 = new Truck("IKY 707");
            Vehicle t3 = new Truck("FFY 808");
            Vehicle t4 = new Truck("PCS 909");
            Vehicle t5 = new Truck("SLY 000");

            park.accept(c1);
            int t2bay = park.accept(t2);
            Park.accept(c3); park.accept(t1);
            int c4bay = park.accept(c4);
            park.accept(c5); park.accept(t5);
            park.release(t2bay);
            park.accept(t4); park.accept(t3);
            park.release(c4bay); park.accept(c2);
            park.listVehicles();
      }

}
```

Ao analisar separadamente as seguintes partes do código dado, vamos identificar o problema da implementação:

```
//As construtoras iniciam plate com null:
public Vehicle( ) { }
public Car( ) extends Vehicle( ) { }
public Truck( ) extends Vehicle( ) { }

//As seguintes operações supõem que plate não poderia ser null
public String identify() { //identify de Car
      return “car with plate " + plate;
}
public String identify( ){ //identify de Truck
      return "truck with plate " + plate;
}
```

Verifique a solução do problema com as operações corrigidas. Agora elas inicializam o campo antes de imprimir:

```
public String identify() { //identify de Car
      String p = plate? plate : “<none>”;
      return "car with plate " + p);
}
```

```
public String identify( ){ //identify de Truck
      String p = plate? plate : “<none>”;
      return "truck with plate “ + p);
}
```

Ainda verificando problemas nessa implementação de polimorfismo. Podemos ver na organização da hierarquia que a operação `idenfify` é utilizada de forma quase idêntica pelas duas subclasses. A única alteração nelas é que uma imprime “`car`” e a outra “`truck`” em seus textos de identificação. Esta operação então deve ser feita na superclasse e simplificada nas subclasses.

Vamos então, mover identify para superclasse e verificar a nova versão das classes `Vehicle`, `Car` e `Truck` no sistema de garagem:

```
//Classe veículo
class Vehicle {
      private String plate;
      private String group;
      public Vehicle(string g, String p){
            group = g; p = plate;
      }
      public Vehicle(string g){this(g,null);}
      public String identify( ){
            String p = plate? plate : "<none>";
            return group + " with plate " + p;
      }
}

//Classe Carro, subclasse de Veículo
class Car extends Vehicle {
      public Car(String p){super("car", p);}
      public Car( ){this(null);}
}

//Classe Caminhão, subclasse de Veículo
class Truck extends Vehicle {
      public Truck(String p){super(“truck",p);}
      public Truck( ){this(null);}
}
```

## 4.6 O MÉTODO MAIN

Quando clicamos em compilar para gerar o aplicativo no qual estamos trabalhando, muitas vezes construímos várias classes em arquivos diferentes e a JVM precisa iniciar a execução desse código de algum ponto de nosso sistema. Ela inicia a execução pelo método main da classe ativada.

Cada classe pode possuir um método `main` para fins de teste, mas, é importante especificar qual é a classe cujo `main` iniciará a execução para que o código seja iniciado no local desejado.

O método `main` possui a seguinte assinatura:

```
public static void main(String[] args) { ... }
```

## 4.7 CONVENÇÃO DE NOMES

Precisamos obedecer alguns padrões de escrita de acordo com a *Code Conventions* da Oracle (2016). Na nomeação das classes, métodos e atributos não devem ser incluídos números ou caracteres especiais como _, #, @, * $, etc. As classes devem ser nomeadas com substantivos e ter sua inicial maiúscula. O nome da classe deve possuir o mesmo nome de seu arquivo, caso contrário, haverá erro durante a execução do aplicativo. Se essa classe possuir um nome composto, por convenção, esse nome deverá ter todas as suas iniciais maiúsculas. Ex: `NomeDaClasse`, não é interessante utilizar hífen ou *underline* para separar os nomes de uma classe com nome composto.

Cada método/função deve ser nomeado com verbo e ter sua inicial minúscula. O nome dos métodos não deve ser iniciado ou composto por caracteres especiais. Caso o nome seja composto, a primeira inicial é minúscula e as demais são letras maiúsculas. Ex: `nomeDoMetodo`.

Um atributo pode ser nomeado com um adjetivo ou um substantivo e ter sua inicial minúscula. De igual forma, o nome do atributo não deve conter caracteres especiais e caso seja composto, deverá possuir uma inicial minúscula e todas as demais iniciais das palavras que o compõe em letra maiúscula. Ex: `nomeDoAtributo`. De toda forma, é interessante escrever nome do atributo da forma mais simples possível, então, em vez de `nomeDoAtributo`, seria interessante nomeá-lo apenas como nome.

O nome de uma constante deve ser escrito com todas as letras maiúsculas. Caso a constante possua um nome composto, ele deve ser separado pelo caractere *underline*. Ex: `NOME_DA_CONSTANTE`. Essa é a única circunstância na qual o caractere *underline* deve ser usado na formação de nomes.

Todas essas regras de nomenclatura são utilizadas para deixar o código da forma mais organizada possível. Não devemos declarar variáveis locais com o mesmo nome de variáveis globais. É interessante utilizar sempre identificadores ainda não utilizados para funções, variáveis, métodos e classes. Dessa forma, a manutenção é facilitada já que poderão ser evitadas situações como uma variável comum ser confundida com uma constante ou que uma variável local seja confundida com uma variável global.

## 4.8 ENDENTAÇÃO

Endentar também é uma necessidade de acordo com a *Code Conventions* da Oracle (2016). Procedimento também conhecido como Indentação ou Identação. A endentação possui o propósito de trazer estética e organização ao código. Endentar é organizar o código de maneira hierárquica e o código em sua totalidade deve possuir endentação.

```
public class Endentacao {
      String a;
      String b = "Como endentar um código.";

      public void mostrarEndentacao(){
            ...
            if(a == b){
                  System.out.println(b);
            }
      }
}
```

Note que no código acima, todas as linhas de código construídas após a definição da classe têm um espaço parecido com o espaço que forma um parágrafo nas linhas de um texto. O código desenvolvido dentro da função possui um novo espaçamento para mostrar que ele se encontra dentro do método. Após a criação o `if`, o código que segue para a execução do if possui mais espaçamento para mostrar que ele se encontra dentro do `if`. Fica mais fácil identificar com essa organização que a operação `a=b` é realizada dentro do método `mostrarEndentacao()` a operação `System.out.println(b)` é realizada dentro do `if`.

No código abaixo, podemos verificar a falta da prática de endentação. Nesse código, todas as demais linhas de código estão no mesmo nível da linha da definição da classe. Podemos notar que fica mais confusa a verificação da hierarquia do código a não ser pela visualização das chaves. Ocorre que em um código cada vez maior e com várias estruturas de repetição aninhadas, ficará muito mais complicada o entendimento para a realização de manutenção no código, caso necessário. Os editores mais utilizados para Java como o eclipse e o netbeans fazem uma endentação automática do código à medida que o desenvolvedor vai apertando enter e escrevendo novas linhas. Mas, é necessário ficar atento a endentação, pois, de acordo com a refatoração do código ou mesmo no início do desenvolvimento quando um bloco de código é copiado de algum lugar para ser colado em outro, por exemplo, a endentação pode ser prejudicada e precisa ser reorganizada.

```
public class FaltaEndentacao {
String a, b = "Aqui não se pratica a endentação.";

public void mostrarFaltaDeEndentacao(){
...
if(a == b){
System.out.println(b);}
}
}
```

## 4.9 COMENTÁRIOS

De acordo com a *Code Conventions* da Oracle (2016), é importante colocar comentários ao longo do desenvolvimento do código que servirão como uma documentação de consulta para futuras manutenções. Quanto maior um código, maior é a necessidade de sua documentação para ser mais fácil encontrar um método específico, por exemplo.

Quando um bloco com um comentário inicial for inserido, é necessário separar a última linha do código anterior e o comentário com uma linha em branco. O comentário é sempre referente ao código que segue.

```
public class Comentario {

	String a;
	String b = "Como comentar um código.";

	//Método que mostra uma mensagem
	public void mostrarMensagem(){
		...
		if(a == b){
			System.out.println(b);
		}
	}
}
```

No código acima podemos verificar a inserção correta de um comentário. Existe uma linha em branco entre o comentário e a última linha de código acima dele. Além disso, o comentário se refere ao método que inicia logo abaixo dele.

## 4.10 REDUZIR ESCOPO

Assunto abordado por Buschmann (1996). É interessante utilizar blocos aninhados para diminuir o escopo de algumas variáveis que vão ser utilizadas somente ali, declarando-as dentro dos blocos. É ainda interessante declarar e iniciar as variáveis em um mesmo comando.

```
public class Carro {

	public static final int LIMITE = 150;
	protected int velocidadeAtual = 0;

	public void acelerar() {

		velocidadeAtual++;
		{
		String alerta = "Bib.";
```

```
            if (velocidadeAtual == LIMITE)
                  System.out.println(alerta);
            }

            // Testar o limite de velocidade do carro.
            if (velocidadeAtual > LIMITE) {
                  velocidadeAtual = LIMITE;
            }
      }

      //...
}
```

No exemplo mostrado, a classe Carro foi implementada com os atributos `velocidadeAtual` e `LIMITE`. O método `acelerar` utiliza esses dois atributos e cria a *string* alerta já a iniciando imediatamente. Essa *string* foi criada dentro do método e de um bloco aninhado, dessa forma ficou definido que ela poderá ser usada somente no limite desse bloco. É interessante delimitar esse espaço por questões de organização. Durante a manutenção desse código, fica fácil saber onde essa variável está sendo utilizada. Ainda neste método, foi utilizado um bloco aninhado com um comentário, em vez de utilizar o comentário antes do início do método. Essa disposição do comentário também facilita a manutenção do código uma vez que, não será necessário verificar todo o método para saber onde está sendo testado o limite de velocidade.

## 4.11 REPETIÇÕES E EXPRESSÕES

Assunto abordado por Staa (2000). Os laços de repetição são estruturas muito utilizadas para testar condições ou repetir procedimentos. Os comandos `for` e `if` precisam ser utilizados de forma consciente para que a estrutura implementada desses comandos não fique muito complexa.

Evite expressões complexas ou muito longas para condicionar um `if`.

Ex:

```
if(a>3 && b==5 || c>1 && d==7 || e>2 && f==9 || g>4 && h==10)
```

Caso seja necessária uma condição maior, é interessante criar outros *ifs* aninhados, mas, evite muitos laços de repetições aninhados. De toda forma, situações nas quais um `if` possua uma condição muito complexa ou haja ocorrência de muitos comandos *ifs* e *fors* aninhados devem ser estudadas para seja verificada a viabilidade de criar novos métodos. Estruturas com laços muito complexos se tornam difíceis de entender no momento de realizar manutenção.

Cada bloco de código vazio ter um comentário com a explicação de o porquê estar vazio. Não é interessante deixar no código somente uma abertura e fechamento de chaves `{}` sem explicar porque essa estrutura foi posta ali. Essa ação pode confundir

desenvolvedores que façam futuras manutenções no código e pode levar a crer que a existência da estrutura é desnecessária.

## 4.12 CONCLUSÃO

Foram apresentados vários estilos para a construção adequada de módulos. Aprendemos que efeito colateral é o que se pode observar externamente em uma função e que uma função sem efeito colateral pode ser executada várias vezes sem prejudicar o racioncínio modular. Se for uma função com parâmetros, sempre que ela receber as mesmas entradas, ela terá as mesmas saídas. Vimos também a implementação de objetos como máquinas de estado utilizando efeito colateral restrito ao estado concreto para implementar objetos que têm seus estados armazenados para facilitar as operações. Analisamos como escolher as abstrações corretas para construir módulos. Entendemos a importância de manter a consistência de estados e interfaces deixando os objetos sempre num estado consistente. Aprendemos como utilizar polimorfismo de inclusão, nomear cada item de um módulo, endentar o código, realizar comentários, a importância de reduzir o escopo de atributos utilizados em áreas específicas e como utilizar repetições e expressões.

Ao conhecer cada estilo, notamos a importância de estruturar o código da melhor forma possível aplicando os principais padrões já existentes. O uso dos estilos é muito importante para saber qual a melhor forma de planejar e organizar o código. Saber como utilizar os recursos é essencial para realizar o desenvolvimento em conformidade com os principais padrões utilizados.

# CAPÍTULO 5

# REFATORAÇÃO E BAD SMELL

A prática de refatoração de código consiste em reestruturar um código que possa remover certos problemas de organização. Esses problemas também são chamados de *bad smells*. Esse termo foi proposto por Fowler (1999) e significa mau cheiro. A ocorrência de *bad smells* em um código pode ser vista de várias formas, como classes que contenham muitos métodos executando funções muito diferentes uma das outras, métodos muito longos, métodos e atributos alocados em classes erradas, nomeação inconveniente de atributos e métodos, dentre outros. A refatoração do código consiste em deixá-lo mais organizado, compreensível e aumentar sua manutenibilidade. Vejamos algumas técnicas de refatoração propostas por Fowler (1999).

## 5.1 DIVIDIR O MÉTODO EM DOIS

O *extract method* é uma forma de refatoração aplicada para tratar casos nos quais um método possui acúmulo de tarefas muito distintas ou quando possuímos um método muito longo e de difícil entendimento. Neste caso, é interessante verificar a necessidade de refatoração do método.

Em alguns casos, pode ser melhor que o método seja dividido em dois ou mais métodos. Esses casos podem ocorrer, por exemplo, quando um método está realizando muitas tarefas e não deixa claro qual é a sua real finalidade.

Veja a seguir, no exemplo de Fowler, um método utilizado para realizar a impressão de um item e exibir alguns detalhes do mesmo:

Antes:

```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + getOutstanding());
}
```

Na refatoração, o método foi dividido para gerar dois métodos e tornar o código mais coerente. Imprimir e exibir detalhes são tarefas distintas e é interessante que onde elas coexistam, sejam criados métodos diferentes para cada uma delas.

Depois:

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);
}
```

## 5.2 TRANSFORMAR UM MÉTODO EM UMA CLASSE

O *replace method with method object* é um método de refatoração utilizado para tratar métodos muito longos que possuam muitas variáveis locais. Nesses casos, é interessante verificar a viabilidade de esse método se tornar uma nova classe, pois, métodos muito longos tornam o código menos legível.

Nas situações em que seja interessante aplicar essa refatoração, podemos verificar que em alguns casos uma classe possui um ou mais métodos com essas características e a própria classe se torna muito longa por esse motivo. É importante verificar também se o método está utilizando outros atributos gerais da classe ou se utiliza somente sua grande variedade de atributos locais. Se ele estiver de acordo com a segunda opção, é um método candidato a se tornar uma classe.

Veja a seguir um exemplo no qual o método `calculaPreco` de uma classe de `Venda` possui muitas variáveis locais.

Antes da refatoração:

```
public class Venda {

      private String produto;
      private String vendedor;
      private String cpfCliente;
      double precoBase;
      int quantidade;

      public void processarVenda(String produto,
      String vendedor, String cpfCliente, double precoBase,
      int quantidade){
            this.produto = produto;
            this.vendedor = vendedor;
            this.cpfCliente = cpfCliente;
            this.quantidade = quantidade;
            this.precoBase = precoBase;
            ...
```

```
            calculaPreco();
      }

      ...

      public double calculaPreco(){

            double imposto;
            double desconto;
            double precoFinal;

            if (quantidade == 1){
                  precoFinal = precoBase;
            }

            else if(quantidade >= 2){
                  desconto = 0.1 * precoBase;
                  precoFinal = precoBase - desconto;

            }
                  return precoFinal;
      }

}
```

Na refatoração, o método `calculaPreco` foi transformado em uma nova classe para melhorar a legibilidade do código uma vez que o mesmo possuía muitas variáveis locais.

Depois da refatoração:

```
public class Venda {

      private String produto;
      private String vendedor;
      private String cpfCliente;
      double precoBase;
      int quantidade;

      public void processarVenda(String produto,
      String vendedor, String cpfCliente, double precoBase,
      int quantidade){

            Preco preco = new Preco();

            this.produto = produto;
            this.vendedor = vendedor;
            this.cpfCliente = cpfCliente;
            this.precoBase = precoBase;
            this.quantidade = quantidade;
            ...
            preco.calculaPreco(produto, quantidade);

      }
...
```

```
}
public class Preco {

      double imposto;
      double desconto;
      double precoFinal;

      public double calculaPreco(String _produto, int quant){


            if (quant == 1){

                  precoFinal = precoBase;
            }

            else if(quant >= 2){

                  desconto = 0.1 * precoBase;
                  precoFinal = precoBase - desconto;

            }

                  return precoFinal;

      }

}
```

## 5.3 MOVER MÉTODOS E ATRIBUTOS ENTRE CLASSES

O *move method and move field* é um método de refatoração utilizado para mover métodos e atributos que possam estar em classe errada. A necessidade de realização desse tipo de refatoração pode ser um indício de que um método está utilizando atributos de outra classe e esteja interagindo pouco ou em nenhum momento com os atributos de sua própria classe, nesse caso, talvez esse método deva ser movido para essa outra classe. De igual forma, se um atributo estiver sendo muito utilizado por métodos de outra classe, é necessário verificar a viabilidade de mover esse atributo para a outra classe que o esteja utilizando com maior frequência.

A necessidade de uso dessa refatoração pode indicar que o método ou atributo a ser movido esteja atrapalhando a medida de coesão de certa classe. Se um atributo está alocado em uma classe `A`, `mas`, é utilizando em sua maior parte pela classe `B` a coesão das duas classes pode ficar prejudicada. Isso ocorre porque os métodos de `B` podem estar utilizando aquele atributo de `A` e não utilizem adequadamente os atributos de sua própria classe. Pode ocorrer ainda nessa situação, o fato de que os métodos de `A` não estejam utilizando em nenhum momento um atributo que foi alocado ali que esteja sendo utilizado somente pela classe `B`.

No exemplo abaixo utilizado por Fowler (1999), o método `participate` foi movido para a classe `Project` que estava utilizando método.

Antes da refatoração:

```
class Project {
  Person[] participants;
  ...
}

class Person {
  int id;
  ...
  boolean participate(Project p) {
    for(int i=0; i<p.participants.length; i++) {
       if (p.participants[i].id == id) return(true);
    }
    return(false);
  }
}

... if (x.participate(p)) ...
```

Verificamos acima que a classe `Person` não utiliza o método `participate`, mas, o possui por um possível erro de estruturação.

Depois da refatoração:

```
class Project {
  Person[] participants;
  ...
  boolean participate(Person x) {
    for(int i=0; i<participants.length; i++) {
       if (participants[i].id == x.id) return(true);
    }
    return(false);
  }
}

class Person {
  int id;
  ...
}

... if (p.participate(x)) ...
```

Na refatoração o método foi movido para a classe `Project` que faz o uso do mesmo. Essa refatoração deixou o código mais coeso, uma vez que, antes o atributo `Person` estava sendo acessado por um método alocado fora de sua classe.

## 5.4 UNIR DUAS CLASSES EM UMA

O *inline class* é um método de refatoração utilizado para unir duas classes que sejam muito pequenas e possuam atividades em comum. Esta é mais uma forma de simplificar o sistema e diminuir o número de arquivos. É sempre importante modularizar o máximo possível um aplicativo, mas, se duas classes são muito pequenas e uma utiliza a outra, elas podem ser unidas em apenas uma.

O *inline class* pode ser útil para diminuir a complexidade, pois, se for notada a existência de duas ou mais classes muito pequenas e que se pareçam em suas tarefas, pode ser um indício de que o número de arquivos no aplicativo é maior do que poderia ser. Com a existência de um número de arquivos desnecessário, pode se tornar complicada a tarefa de encontrar um procedimento ou outro item e dificultar a manutenção. Isso ocorre porque, se você tem classes muito pequenas com tarefas parecidas, quando for necessário encontrar um atributo `x`, você pode imaginar que ele esteja em várias daquelas classes com tarefas parecidas. Mas, se o aplicativo é bem organizado e não possui esse tipo de classe, é fácil encontrar, por exemplo, o atributo `telefone` acessando diretamente a classe `Pessoa` como vamos verificar no exemplo a seguir.

Veja a classe `Pessoa` que após a refatoração foi unida a classe `Telefone`:

Antes da refatoração:

```
public class Pessoa {

	private String nome;
	private String cpf;
	private String endereco;
	private String estadoCivil;
	private String sexo;
	private String telefone1;
	private String telefone2;

	public Pessoa(String nome, String cpf, String endereco,
	String estadoCivil, String sexo){
		this.nome = nome;
		this.cpf = cpf;
		this.endereco = endereco;
		this.estadoCivil = estadoCivil;
		this.sexo = sexo;
	}


	public void telefones(){

		Telefone telefone = new Telefone();
		telefone1 = telefone.retornarTelefone("telefoneFixo");
		telefone2 = telefone.retornarTelefone("telefoneCelular");
	}
}
```

```
public class Telefone {

	private String telefoneFixo;
	private String telefoneCelular;

	public Telefone(String telefoneFixo,
	String telefoneCelular){
		this. telefoneFixo = telefoneFixo;
		this. telefoneCelular = telefoneCelular;
	}


	public String retornarTelefone(String telefone){

		String tel;

		if(telefone == "telefoneFixo")
			tel =  telefoneFixo;

		else if(telefone == "telefoneCelular")
			tel =  telefoneCelular;

		return tel;

	}

}
```

No exemplo dado, a classe `Telefone` existia fora da classe `Pessoa` de forma incoerente, pois, geralmente no mesmo ato de cadastrar uma pessoa são incluídos seus telefones. Nesse caso, a classe pessoa pode possuir um ou mais atributos de telefone para que não seja necessário criar uma classe somente para incluí-los.

Após a refatoração:

```
public class Pessoa {

	private String nome;
	private String cpf;
	private String endereco;
	private String estadoCivil;
	private String sexo;
	private String telefoneFixo;
	private String telefoneCelular;
```

```
    public Pessoa(String nome, String cpf, String endereco,
    String estadoCivil, String sexo, String telefoneFixo,
    String telefoneCelular){
        this.nome = nome;
        this.cpf = cpf;
        this.endereco = endereco;
        this.estadoCivil = estadoCivil;
        this.sexo = sexo;
        this.telefoneFixo = telefoneFixo;
        this.telefoneCelular = telefoneCelular;
    }

}
```

Agora a classe pessoa possui dois campos de telefone e não mais necessita recorrer à classe `Telefone` para obter os telefones de uma pessoa específica. Essa refatoração diminiu a complexidade do código simplesmente criando dois atributos a mais na classe `Pessoa`.

## 5.5 DIVIDIR UMA CLASSE EM DUAS

O *extract class* é um método de refatoração utilizado para dividir classes muito longas e que com essa característica provavelmente possuem mais de uma responsabilidade. Se uma classe possui mais de uma responsabilidade, ela está ferindo o SRP e deve ser refatorada para garantir um código mais legível.

Classes com mais de uma responsabilidade podem provocar uma complexidade desnecessária no código conforme já abordamos ao falar do SRP. Quando uma classe tem mais de uma responsabilidade, a necessidade de sua alteração pode causar um maior impacto no aplicativo. Nesses casos, é sempre importante aplicar o *extract class* para atribuir uma responsabilidade única para cada uma das novas classes geradas.

Podemos identificar mais de uma responsabilidade em uma classe quando observamos que essa classe possui uma ou mais tarefas distintas/sem ligação. Esse é o caso do exemplo mostrado a seguir de uma classe `Pessoa` que possui um método para cálculo de IMC totalmente distinto das tarefas básicas da classe `Pessoa` que seriam inserção e consulta de uma pessoa. Esse método deveria ser inserido em uma classe destinada somente para o cálculo do IMC.

Antes da refatoração:

```
public class Pessoa {

	String nome;
	String sexo;
	String cpf;
	String endereco;
	String telefone;
	String estadoCivil;
	float peso;
	float altura;

	public Pessoa(String nome, String sexo, String cpf,
	String endereco, String telefone, String estadoCivil,
	float peso, float altura) {

		this.nome = nome;
		this.sexo = sexo;
		this.cpf = cpf;
		this.endereco = endereco;
		this.telefone = telefone;
		this.estadoCivil = estadoCivil;
		this.peso = peso;
		this.altura = altura;

	}

	public String consultarPessoa(String _cpf){

		...
	}

	public float calcularImcPessoa(){

		float imc = peso/(altura * altura);

		return imc;
	}

}
```

Na refatoração, as responsabilidades da classe `Pessoa` foram divididas com a criação da nova classe `Imc`. Agora a classe `Pessoa` possui as tarefas de sua responsabilidade e a classe `Imc` é responsável pelo cálculo.

Após a refatoração:

```
public class Pessoa {

	String nome;
	String sexo;
	String cpf;
	String endereco;
	String telefone;
	String estadoCivil;

	public Pessoa(String nome, String sexo, String cpf,
	String endereco, String telefone, String estadoCivil) {
		this.nome = nome;
		this.sexo = sexo;
		this.cpf = cpf;
		this.endereco = endereco;
		this.telefone = telefone;
		this.estadoCivil = estadoCivil;

	}

	public String consultarPessoa(String _cpf){

		/...
	}


}

public class Imc {

	float peso;
	float altura;

	public Imc(float peso, float altura) {

		this.peso = peso;
		this.altura = altura;

	}


	public float calcularImcPessoa(){

		float imc = peso/(altura * altura);

		return imc;

	}

}
```

## 5.6 MOVER O MÉTODO OU ATRIBUTO DA SUPERCLASSE PARA A SUBCLASSE

O *push down method/field* é um método de refatoração utilizado quando a implementação de um método da superclasse não está sendo utilizado por todas as suas subclasses. Nesse caso, é interessante mover o método ou atributo da superclasse para a subclasse que faz o uso do mesmo.

A necessidade de aplicação dessa refatoração pode sugerir que algumas subclasses da hierarquia tenham conhecimento desnecessário de um método específico. Nesse caso, elas estão implementando o método de forma irregular e aumentando a complexidade do código como um todo.

Veja no exemplo a seguir uma superclasse `Funcionario` que é estendida pelas classes `Vendedor` e `Engenheiro`. O método `calculoVendas` para verificar quantidade de vendas é implementado pela superclasse, mas, utilizado somente pela classe `Vendedor`. Após a refatoração, o método foi movido para a classe `Vendedor`.

Antes da refatoração:

```
public class Funcionario {

      private String nome;
      private String cpf;
      private String rg;
      private String endereco;
      private String estadoCivil;
      private String numFilhos;
      private String telefoneFixo;
      private String telefoneCelular;
      private double salario;

      ...

      public double calculoVendas(){
            ...
            return vendas;
      }

}

public class Engenheiro extends Funcionario {
      ...
}

public class Vendedor extends Funcionario {

      ...

}
```

Após a refatoração:

```
public class Funcionario {

      private String nome;
      private String cpf;
      private String rg;
      private String endereco;
      private String estadoCivil;
      private String numFilhos;
      private String telefoneFixo;
      private String telefoneCelular;
      private double salario;
      ...
}

public class Engenheiro extends Funcionario {
      ...
}

public class Vendedor extends Funcionario {
      ...
      public double calculoVendas(){
                  ...
                  return vendas;
            }
}
```

## 5.7 RENOMEAR MÉTODO OU ATRIBUTO

O rename *method /field* é um método de refatoração utilizado para renomear métodos e atributos que possuem nomes que não revelam a sua real funcionalidade. A importância de realizar uma nomeação coerente dos métodos e atributos muitas vezes só é percebida durante a manutenção do código quando muitas vezes fica difícil saber qual é a real finalidade de um método ou atributo mal nomeado.

A nomeação adequada dos itens poupa custos no momento da manutenção já que o tempo com certeza será reduzido. Quando ela não é realizada no momento da produção é necessário aplicar o rename *method/ field* para melhorar a organização e a legibilidade do código.

No exemplo a seguir, a classe `Cliente` possui um conjunto de atributos difíceis de identificar e um método cuja nomeação não expressa com clareza o que ele realiza.

Antes da refatoração:

```
public class Cliente {

      private String nmCliente;
      private String cpfCliente;
      private String tel1;
      private String tel2;
      private String numCt;

      public double obtlmtcred(){

            ...

      }

}
```

Os atributos e o método continham palavras abreviadas e não era possível entender somente na verificação do nome qual o real objetivo do atributo/método.
Verifique agora a implementação após a refatoração:

```
public class Cliente {

      private String nome;
      private String cpf;
      private String telefoneFixo;
      private String telefoneCelular;
      private String numeroCartao;

      public double obterLimiteDeCredito(){

            ...

      }
}
```

## 5.8 CONCLUSÃO

Com a apresentação dos principais métodos de refatoração vimos que é interessante dividir métodos muito longos para evitar um código muito longo e o acúmulo de tarefas em um único método. Caso o método seja muito longo e com muitas variávies locais ele pode ser transformado em uma classe. Se verificarmos a existência de métodos ou atributos que estão alocados em classes erradas, devemos movê-los para o local correto. Existe também a possibilidade de unir duas classes muito pequenas com tarefas parecidas em apenas uma. Podemos ainda, dividir uma classe muito longa com mais de uma responsabilidade em duas ou mais classes. Na implementação de reúso, caso um método da superclasse seja utilizado somente por uma de suas subclasses podemos

movê-lo para essa subclasse específica. Por fim, devemos renomear métodos e atributos que possuam nomes que não facilitem o entendimento de sua utilidade.

Podemos concluir que refatorar, em alguns casos, é muito importante para reorganizar o código de forma correta. Muitas vezes o desenvolvimento de um aplicativo/módulo foi realizado e durante seu andamento ou após a sua conclusão, foi verificado que algumas partes do código poderiam ter sido organizadas de uma melhor forma para facilitar o entendimento e a manutenção. Nesses casos, é sempre importante verificar o melhor meio para refatorar e aplicar as medidas necessárias.

# CAPÍTULO 6

# OS MANDAMENTOS DA PROGRAMAÇÃO MODULAR

Vimos até aqui uma série de recomendações para desenvolver códigos de programação de forma coerente com o intuito de alcançar organização, eficiência e manuteniblidade. Após estudar a programação modular, princípios, estilos e formas de refatorar, é possível criar uma espécie de *checklist* para descrever resumidamente o que não pode ser esquecido para se programar de forma modular e aplicar boas práticas de programação.

Os principais conceitos vistos até aqui se provaram necessários para a programação com o uso de boas práticas. Baseando-se em todos esses conceitos conheça os 21 mandamentos utilizados para aplicar boas práticas na programação modular em Java:

1. Procure realizar o desenvolvimento com um alto grau de encapsulação e restrição ao acesso para desenvolver de acordo com uma das boas práticas básicas que abordamos no Capítulo 2.
2. Um módulo deve possuir baixo grau de acoplamento com outros módulos, de acordo com uma das boas práticas básicas que abordamos ainda no Capítulo 2.
3. Um módulo deve possuir alta coesão entre seus componentes internos de acordo com uma das boas práticas básicas que abordamos por último no Capítulo 2.
4. Utilize composição e herança de forma adequada para respeitar o Princípio da Escolha: Composição x Herança.
5. Prefira programar para uma interface em vez de programar para uma implementação para respeitar o Princípio da Programação para a Interface.
6. Ao construir uma classe garanta que ela será utilizada para um único objetivo para respeitar o Princípio da Responsabilidade Única.
7. Construa módulos abertos para extensão e fechados para modificação para respeitar o Princípio do Aberto/Fechado.
8. Na implementação de reúso, garanta que o código cliente cumpra o contrato de implementação para respeitar o Princípio da Substituição de Liskov.
9. Prefira implementar interfaces específicas em vez de uma única interface geral para respeitar o Princípio da Segregação de Interfaces.
10. Tenha cuidado para que módulos de alto nível não dependam de módulos de baixo nível para respeitar o Princípio da Inversão da Dependência. Os módulos devem depender de abstrações.
11. Tenha cuidado com a existência de efeito colateral em uma função que possa prejudicar o raciocínio equacional para desenvolver de acordo com o estilo estudado: Efeito Colateral em Função.
12. Em caso de operações mais complexas, é interessante implementar objetos como máquinas de estado para desenvolver de acordo com o estilo estudado: Implementação de Objetos como Máquinas de Estado.
13. Escolha as abstrações corretas para construir um módulo ou um conjunto de módulos para desenvolver de acordo com o estilo estudado: Abstrações Corretas.

14. Mantenha a consistência de estados e interfaces para desenvolver de acordo com o estilo estudado: Consistência de Estados e Interfaces.
15. Utilize o polimorfismo de forma correta cumprindo o contrato de implementação e tratando os campos de forma consistente para desenvolver de acordo com o estilo estudado: Uso de Polimorfismo de Inclusão.
16. Nomeie de acordo com a convenção de nomes utilizada os atributos, métodos, classes e demais itens do código para desenvolver de acordo com o estilo estudado: Convenção de Nomes.
17. Mantenha o código sempre organizado e corretamente endentado para desenvolver de acordo com o estilo estudado: Endentação.
18. Comente as partes mais complexas do código e obedeça aos critérios utilizados para comentar para desenvolver de acordo com o estilo estudado: Comentários.
19. Reduza ao máximo possível o escopo de variáveis utilizadas somente em pequenas partes do código para desenvolver de acordo com o estilo estudado: Reduzir Escopo.
20. Estruture corretamente os laços de repetição para desenvolver de acordo com o estilo estudado: Repetições e Expressões.
21. Verifique sempre durante ou após o desenvolvimento de um módulo se a ele é aplicável alguma atividade de refatoração necessária, de acordo com os procedimentos estudados no Capítulo 5, Refatoração e Bad Smell.

## 6.1 CONCLUSÃO

Os 21 mandamentos apresentados são aplicáveis em Java e em diversas outras linguagens de programação orientadas por objetos. Verificar se a produção está em conformidade com esse *checklist* é importante para saber se estão sendo aplicadas as boas práticas necessárias ao código criado. O uso de cada um desses mandamentos é aplicável em diferentes situações e pode ser determinante para um produto final de qualidade.

Cada um dos mandamentos foi retirado das boas práticas estudadas e resume em poucas palavras cada uma dessas boas práticas. Ao longo do estudo de todas essas boas práticas, vimos o quanto é importante programar de forma modular e aplicar bons princípios e estilos para construir um bom código. Verificamos ainda, a necessidade de realizar refatoração sempre que necessário. Os 21 mandamentos, são a síntese de todas essas boas práticas estudadas.

# CAPÍTULO 7

# CONCLUSÃO

Este trabalho teve como principal objetivo, contribuir com a disseminação de boas práticas de programação modular em Java. Muitas das práticas aqui abordadas podem ser utilizadas inclusive por outras linguagens orientadas por objetos. O estudo dessas práticas mostra que elas tornam os códigos mais organizados, robustos e manuteníveis.

O custo de manutenção dos aplicativos é sempre maior que o custo de produção, mas, para reduzir cada vez mais esses custos, é importante um cuidado durante a produção para que eles sejam estruturados da melhor forma possível. Muitas vezes, ao rever o processo de produção, nota-se que esses custos poderiam ser menores se durante o desenvolvimento, algumas práticas tivessem sido aplicadas para melhorar a organização, o entendimento e a eficiência do código.

Estudamos todas as melhores práticas detalhadamente e podemos salientar que elas reduzem os custos de manutenção além de melhorar a eficiência de um código. Aprendemos que modularizar é importante para facilitar o entendimento e a organização. Aprendemos ainda que as boas características de um bom módulo são obtidas com a aplicação das boas práticas.

Dentre as várias boas práticas utilizadas para caracterizar um bom módulo temos a importância de encapsular os dados e restringir a eles o acesso para privilegiar o raciocínio modular, os benefícios de possuir baixo grau de acoplamento entre módulos, a importância de se ter coesão nos módulos, a necessidade de aplicação dos principais princípios de programação modular, os benefícios de se aplicar bons estilos de programação e a importância de refatorar códigos mal escritos.

Todas as boas práticas estudadas são de grande importância. Elas devem ser tratadas com atenção durante o desenvolvimento para que o resultado seja sempre o melhor possível. Muitas vezes o desenvolvedor se preocupa em entregar um produto final funcionando, mas, nem sempre o “estar funcionando” é sinônimo de qualidade. Após a entrega do produto, com certeza, haverá necessidade de alteração e manutenção. É nesse momento que é perceptível se o produto entregue funcionando perfeitamente também foi produzido com um padrão de qualidade e boas práticas. Códigos ilegíveis, mal estruturados, com partes duplicadas, contendo itens mal nomeados e que não seguiram princípios para desenvolvimento de suas estruturas complicam o trabalho de quem precisa realizar alguma alteração ali.

Para ter um alto grau de legibilidade, facilitar a manutenção, economizar tempo de execução e melhorar a organização é preciso ter conhecimento de boas práticas e aplicá-las durante o desenvolvimento. Sem conhecimento de boas práticas é praticamente impossível programar de acordo com os principais padrões utilizados para a estruturação dos aplicativos.

## REFERÊNCIAS


ALMEIDA, H.M. *Interface vs. Implementação Herança vs. Composição*. Disponível em: <http://www2.lsd.ufcg.edu.br/~nazareno/ensino/si1/06.InterfacePolimorfismoHerancaComposicao.pdf> Acesso em: 01 nov. 2015.

BIGONHA, R.S.; BIGONHA, M. *Programação Modular*. Notas de Aula – DCC UFMG. Belo Horizonte: 2015.

BUSCHMANN, F. et al. *Pattern-Oriented Software Architecture*: A System of Patterns. 1. ed. New Jersey: John Wiley & Sons, 1996.

CADU. *Entendendo Coesão e Acoplamento*. Disponível em: <http://www.devmedia.com.br/entendendo-coesao-e-acoplamento/18538> Acesso em: 17 out. 2015.

CADU. *Open Close Principle*. Disponível em: < http://www.devmedia.com.br/open-close-principle/18970> Acesso em: 17 out. 2015.

CARDELLI, L.; WEGNER, P. *On understanding types, data abstractions and polimorphism*, ACM Computing Surveys, 17, 451-522. 1985. Disponível em < http://lucacardelli.name/papers/onunderstanding.a4.pdf>. Acesso em: 24 jan.2016.

CARGILL, T. *C++ Programming Style, Reading*. 1. ed. Reading, MA: Addison-Welsey, 1992.

CASTILHO, R. *Princípios SOLID*: Princípio da Inversão de Dependência (DIP), 2013. Disponível em: <http://robsoncastilho.com.br/2013/05/01/principios-solid-principio-da-inversao-de-dependencia-dip/> Acesso em: 18 out. 2015.

CHAIM, M.L. *Refatoração de Software*. Disponível em <http://www.ic.unicamp.br/~eliane/Cursos/Transparencias/Manutencao/Aula25-Refatoring.pdf>. Acesso em: 24 jan.2016.

CHIDAMBER, S. R. and KEMERER, C. F. (1994). *A metrics suite for object oriented design*. IEEE Trans. Softw. Eng., 20:476–493.

COAD, P.; MAYFIELD, M.; KERN, J. *Java Design*: Building Better Apps and Applets. Yourdon Press Computing Series. 2. ed. New Jersey: Prentice Hall, 1999.

DEMARCO, T. *Structured Analysis and System Specification*. 1. ed. New Jersey: Prentice Hall, 1979.

FERREIRA, K.A.M. et al. *Métricas de Coesão de Responsabilidade*: A Utilidade de Métrica de Coesão na Identificação de Classes com Problemas Estruturais. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbqs/2011/SBQS2011-TT01_82947_1.pdf> Acesso em: 01 fev. 2015.

FIGUEIREDO, E. *Idiomas de Programação*. Disponível em <http://homepages.dcc.ufmg.br/~figueiredo/disciplinas/aulas/idiomas-java_v02.pdf>. Acesso em 20 set 2015.

FIGUEIREDO, E. Metrics for Object-Oriented Programss. Disponível em < http://homepages.dcc.ufmg.br/~figueiredo/disciplinas/lectures/oo-metrics_v01.pdf>. Acesso em: 15 nov 2015.

FIGUEIREDO, E. *Reengenharia, Refatoração e Bad Smell*. Disponível em <http://homepages.dcc.ufmg.br/~figueiredo/disciplinas/aulas/bad-smell-refactoring_v01.pdf>. Acesso em 20 set 2015.

FOWLER, M. *Refactoring*. Disponível em < http://refactoring.com/>. Acesso em 05 set 2015.

FOWLER, M. *Refactoring*: Improving the Design of Existing Code. 1. ed. Boston: Addison- Wesley Professional, 1999.

LEMOS, H.D. *Encapsulamento, Polimorfismo, Herança em Java*. Disponível em: <http://www.devmedia.com.br/encapsulamento-polimorfismo-heranca-em-java/12991> Acesso em: 17 out. 2015.

LISKOV, B. *Data Abstraction and Hierarchy*, 1987. Laboratory for Computer Science, Cambridge University, Cambridge, 1987.

MARTIN, R.C.; MARTIN, M. *Agile Principles, Patterns, and Practices in C#*. 1. ed. New Jersey: Prentice Hall, 2006.

MEDEIROS, H. *Herança versus Composição*: qual utilizar? Disponível em: <http://www.devmedia.com.br/heranca-versus-composicao-qual-utilizar/26145>. Acesso em: 17 out. 2015.

MEDINA, R.D. *Flash utilizando Orientação a Objetos e a linguagem XML*: Capítulo 1 - Introdução a Programação Orientação a Objetos. Disponível em <http://www-usr.inf.ufsm.br/~rose/curso3/cafe/cap1_Introducao.pdf> Acesso em: 17 out. 2015.

MEYER, B. *Object-Oriented Software Construction*. 2. ed. New Jersey: Prentice Hall, 1997.

MYERS, G. J. *Composite/Structured Design*. 1. ed. New York: Van Nostrand Reinhold, 1978.

ORACLE. *Code Conventions for the Java Programming Language*: Contents, 2016. Disponível em: <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>. Acesso em: 23 jan.2016.

PRIBERAM. *Significado/definição de -logia no Dicionário Priberam da Língua Portuguesa*. Disponível em <http://www.priberam.pt/dlpo/-logia> Acesso em: 17 out. 2015.

SCHACH, S.R. *Engenharia de Software*: Os Paradigmas Clássico e Orientado a Objetos. 7. ed. São Paulo: McGraw-Hill, 2009.

SIGNIFICADOS. *Significado de Metodologia*. Disponível em <http://www.significados.com.br/metodologia/> Acesso em: 17 out. 2015.

SOMMERVILLE, I. *Engenharia de Software*, 8. ed. Boston: Pearson Education, 2007.

STAA, A.D. *Programação Modular:* Desenvolvendo Programas Complexos de Forma Organizada e Segura. 1. ed. MA: Rio de Janeiro: Campus, 2000.

STEVENS, W.; MYERS, G.; CONSTANTINE, L. *Structured Design*. IBM Systems Journal, 13 (2), 115-139, 1974.