Comunicando-se com o Computador

Utilizando a Linguagem Java

Comunicando-se com o Computador

Análise, Modelagem e Desenvolvimento de Sistema Orientado a Objetos

Prof. Dr. Eduardo Manuel de Freitas Jorge, PMP
Prof. Dr. Hugo Saba Pereira Cardoso
Prof. MSc. Marcio Luis Valença Araújo
Prof. MSc. Uedson Santos Reis

Programação Avançada em Java

Editora do Instituto Federal da Bahia – Edifba

Salvador-BA
2018

Comunicando-se com o computador utilizando a linguagem Java

Eduardo Manuel de Freitas Jorge, PMP
Hugo Saba Pereira Cardoso
Marcio Luis Valença Araújo
Uedson Santos Reis
(Organizadores)

Capa: Marlon Xavier

Revisão: Claudio Reynaldo B. de Souza
Marcio Luis Valença Araújo

Impressão e Acabamento: EGBA – Empresa Gráfica da Bahia

Ficha catalográfica elaborada pela Biblioteca
do IFBA Campus Santo Amaro.
Reginaldo Pereira Pascoal Junior – Bibliotecário CRB-5/1470.

C741 Comunicando-se com o computador utilizando a linguagem Java: análise, modelagem e desenvolvimento de sistema orientado a objetos / Eduardo Manuel de Freitas Jorge... [et. al.]. – Salvador: EDIFBA, 2018.

136 p. : il. color.

ISBN 978-85-67562-22-3

1. Java (Linguagem de programação de computador). 2. Linguagem de programação (Computadores). 3. Programação orientada a objetos (Computação). I. Jorge, Eduardo Manuel de Freitas. II. Cardoso, Hugo Saba Pereira. III. Araújo, Marcio Luis Valença. IV. Reis, Uedson Santos. V. Título.

CDU 004.43JAVA

SUMÁRIO

ÍNDICE DE ILUSTRAÇÕES

ÍNDICE DE TABELAS

ÍNDICE DE CÓDIGOS

INTRODUÇÃO AO JAVA

INDICAÇÃO DE LEITURA

Estes três primeiros capítulos deste material usam como base, o livro "Java, Como Programar – 4ª Edição (2003)", Deitel, H. M. e o material produzido pelo site www.t2ti.com "Java Starter (2003)". Assim recomenda-se que você complemente os seus conhecimentos com a leitura destas literaturas indicadas.

1.1 Histórico do Java

Segundo Deitel (Deitel, 2003) centenas de linguagens de computadores estão atualmente em uso. Essas linguagens podem ser divididas em três tipos gerais:

a) Linguagens de máquina;
b) Linguagens *assembler*;
c) Linguagens de alto nível.

Os computadores entendem as linguagens de máquina que são traduções binárias de comandos solicitados para a realização de alguma operação. Estas linguagens são de difícil assimilação para os seres humanos. Com o passar dos tempos, por serem linguagem tediosas e complexas, estas linguagens de máquina foram se tornando mais acessíveis à medida que a maioria dos programadores começaram a utilizar abreviações semelhantes ao inglês para codificar operações elementares do computador. Surgiram então, as linguagens assembler que se utilizavam de programas tradutores chamados de montadores para converter os comandos estruturados na língua inglesa para a binária.

Mesmo sendo mais próxima ao entendimento humano, a linguagem *assembler*, exigia muitas instruções (linhas de código) para realização de pequenas tarefas. Como a programação de computadores (softwa-

re) crescia paralela ao amadurecimento da tecnologia física dos computadores (hardware), a aceleração do processo de desenvolvimento de softwares se tornava inevitável, haja visto que, os componentes surgiam cada vez mais integrados e com grandes capacidades de processamento. Foram então criadas as linguagens de alto nível, que com poucas instruções realizavam tarefas com grande substância. Aliado ao processo de tradução da linguagem de alto nível para a linguagem de máquina surge o compilador, que libera a escrita de instruções em um inglês mais cotidiano e com notações matemáticas. Pascal, C, C++ e Java estão entre as linguagens de programação de alto nível amplamente utilizadas e difundidas.

Segundo o curso Java Starter no site t2ti, a linguagem de programação Java foi criada em 1991 por James Gosling, ela iniciou-se como parte do projeto Green da Sun Microsystems. Inicialmente a linguagem iria chamar-se Oak (Carvalho) em referência a árvore que era visível pela janela de James Gosling. A mudança de nome ocorreu pois já existia uma linguagem de programação com este nome, então a linguagem foi rebatizada para Java.

O termo Java é utilizado, geralmente, quando nos referimos a:

- Linguagem de programação orientada a objetos;
- Ambiente de programação composto pelo compilador, interpretador, gerador de documentação e etc.;
- Ambiente de execução que possua Java Runtime Environment (JRE) instalado. Podendo ser em qualquer máquina capaz de receber a instalação;

Por ser uma linguagem de programação de alto-nível, o Java possui as seguintes características:

- De Simples aprendizado: O aprendizado pode ser feito em um curto período de tempo;
- Orientada a objetos: Projetada para ser orientada a objetos;
- Com Robustez: Desenvolvida para a criação de programas confiáveis, provendo verificações tanto em tempo de compilação quanto execução.

– Segurança: Impossibilita a invasão por códigos maliciosos, visto que, é executado em ambiente próprio (JRE);
– Portabilidade: Podem ser executados em praticamente qualquer máquina desde que esta possua o JRE instalado.

1.2 Máquina Virtual Java (JVM)

A máquina virtual java (JVM) é uma máquina fictícia que roda uma aplicação em um equipamento real. A responsável pela portabilidade do código Java é a JVM, porque todo código Java é compilado para um, *bytecode* (formato transitório) e este formato, então, é traduzido pela JVM.

Há diversas máquinas virtuais Java. Cada uma delas é responsável por um tipo de sistema operacional (Unix, Windows, Linux, etc.). Então, como o código da aplicação Java é um *bytecode*, ou seja, linhas de código interpretadas pela JVM, pode-se criar um programa sem a preocupação com a portabilidade, isto é, onde o programa será executado, já que, se a JVM existir, a aplicação será executável.

1.3 Coletor de Lixo – *Garbage Collection*

É comum que linguagens de programação permitam reservar espaços na memória de um computador em tempo de execução. Também é comum e uma boa prática que uma vez finalizada a execução do programa, deve-se criar uma forma de liberação desta memória alocada, para que outras aplicações possam reutilizá-la. O programador, na maioria das linguagens, é o responsável pelo papel de desalocação de memória utilizada durante a execução do código. Mesmo assim, não é um trabalho simples controlar o que foi utilizado ou o que está em uso. Desta forma, o manejo desordenado e má gestão da memória alocada, pode causar estouro de memória e outros problemas.

O Java traz consigo uma função importante, que é a do Coletor de Lixo ou em inglês *Garbage Collector*. Este se responsabiliza pela gestão da memória, e assim, retira dos desenvolvedores está tarefa complexa.

Com isso, os programadores por exemplo, não precisam ficar criando os chamados em C++, destrutores.

A execução do Coletor de Lixo é em segundo plano e é responsável pela desalocação de memória reservada por variáveis que não mais serão usadas pela aplicação em execução.

1.4 Etapas de um programa Java

Abaixo estão relacionadas as etapas que constituem o processo desde a criação do código.java até o momento da criação do código em linguagem de máquina.

Figura 1 – Etapas do processo de criação do código Java.
Fonte: Autor.

1) Criação do Programa.java;
2) Compilação do código .java e geração do .class;
3) Interpretação pela máquina virtual java;
4) Conversão do código interpretado pela Máquina Virtual Java (JVM) em linguagem de máquina.

1.5 Hotspot

Hotspot é uma técnica de otimização dinâmica, para aumentar o desempenho da JVM. No Java as classes são compiladas em arquivos distintos. Desta forma, foram criados os compiladores JIT (just-in--time), que selecionam o *bytecode* e transforma de uma só vez para a linguagem de máquina antes da execução. A questão deste paradigma é o tempo de espera para iniciar a aplicação, ou seja, ficar

esperando por um longo período o compilador trabalhar é um problema. Então, questiona-se a completude desta otimização.

Inicialmente pode-se pensar que o fato de um programa não precisar passar por um passo a mais, ou seja, a interpretação, irá torná-lo mais eficiente, mas muitas vezes a compilação estática não consegue prenunciar situações que acontecerão durante a execução do programa: partes da aplicação mais usadas, carga do sistema, quantidade de usuários utilizando a aplicação ao mesmo tempo, memória ociosa etc.

Todas as informações citadas com relação ao ambiente da aplicação em execução são usadas em otimizações realizadas pela JVM. Caso seja necessário, o programa que está sendo interpretado é adaptado utilizando linhas de código nativas do sistema operacional em um processamento de compilação dinâmica.

1.6 JRE e JDK

JRE: Em inglês, Java Runtime Environment é o ambiente de execução Java. É composto pela JVM e bibliotecas;

JDK: Em inglês, Java Development Kit é composto pelo ambiente de execução Java (JRE) e por um conjunto de ferramentas para os desenvolvedores Java.

1.7 Ferramentas do JDK

Segundo o curso Java Starter no site t2ti, abaixo estão algumas das principais ferramentas que compõem o JDK:

a) javac: Compilador da linguagem Java;
b) java: Interpretador Java;
c) jdb: Debug Java;
d) java – prof: Interpretador com opção para gerar estatísticas sobre o uso dos métodos;
e) javadoc: Gerador de documentação;
f) jar: Ferramenta que comprime, lista e expande;

g) appletviewer: Permite a execução e debug de applets sem browser;
h) javap: Permite ler a interface pública das classes;
i) extcheck: Detecta conflitos em arquivos Jar.

1.8 Ambiente de Desenvolvimento Integrado (IDE)

O Java é um tipo de linguagem de alto nível que pode ser compilada e interpretada. O compilador Java (javac), responsabiliza-se pelo o código-fonte do Java, transformando-o em *bytecod* (código de nível intermediário). O *bytecod* não é diretamente executável em qualquer plataforma de hardware existente.

Um ambiente de desenvolvimento integrado (IDE) é, conforme o nome já diz, um ambiente de desenvolvimento completo, isto é, um local onde o desenvolvedor encontra todas as ferramentas para desenvolvimento necessárias. Uma IDE é composta quase sempre por: ,editor de código-fonte, um compilador, depurador (*debug*), gerador de documentação entre outras ferramentas de programação. Também pode ser encontrado nestas IDEs, flexibilidade de personalização de ambiente de codificação, que muitas vezes proporcionará uma melhor interação com os desenvolvedores.

Um dos primeiros ambientes de desenvolvimento para computadores pessoais da IBM e outros compatíveis, foi o Turbo Pascal da Borland. Atualmente existem diversos compiladores disponíveis.

Historicamente, vários IDEs já estavam disponíveis para o Java, antes da versão oficial 1.0 ser lançada. No primeiro semestre de 1996, os mais importantes fabricantes de compiladores já tinham lançado um IDE pronto para o Java.

Abaixo alguns ambientes de desenvolvimento integrado disponíveis no mercado:

- Eclipse;
- NetBeans;
- JavaMaker;

- Diva for Java;
- Java WebIDE;
- Kalimantan;
- Roaster;
- Symantec Café;
- Borland C++ Versão 5.0.

1.9 Nomenclaturas Java

Existem diversas palavras que iremos encontrar no Java, e que é importante, neste momento, defini-las já que a maioria dos ambientes de desenvolvimento integrado utilizam. São elas:

Workspace: define o diretório em que as suas configurações pessoais e seus projetos serão gravados;

Projeto: projeto é o nome dado ao diretório que armazenará o pacote e classes do sistema em desenvolvimento;

Pacote: como o próprio nome sugere, são estruturas criadas para dividir o código fonte, assim como promover a separação de classes e responsabilidades;

Classe: são os códigos fontes do sistema em desenvolvimento. Também tem o seguinte significado para programação orientada a objetos: agrupamento de objetos com a mesma estrutura de dados (definida pelos atributos ou propriedades) e comportamento (operações), ou seja, classe são as descrições dos objetos;

Atributos: conjunto de propriedades da classe. Podem ser constantes ou variáveis;

Métodos: conjunto de funcionalidades da classe. Para cada método, especifica-se sua assinatura, composta por:

- Nome: um identificador para o método;
- Tipo: quando o método tem um valor de retorno, o tipo desse valor;

- Lista de argumentos: quando o método recebe parâmetros para sua execução, o tipo e um identificador para cada parâmetro.;
- Visibilidade: como para atributos, define o quão visível é um método a partir de objetos de outras classes.

Comentários: são detalhes do código em desenvolvimento. Utilizado pelos programadores para explicar o que cada rotina faz. É de grande importância para o entendimento do programa. Utiliza-se "//" ou "/* */";

Sintaxe: é a forma de escrita do código Java. Utiliza palavras reservadas. Quando o compilador lê uma palavra reservada que corresponde a um comando, ele precisa executar uma ação como resposta. Veja alguns exemplos:

Comandos

- Comandos em Java são separados por um ponto-vírgula. Pode-se colocar vários comandos em uma mesma linha.
- Comandos são estruturados em blocos definidos por chaves {...}. Principalmente, classes e métodos que são blocos. Blocos podem englobar outros blocos.

Variáveis

- Nomes de variáveis devem iniciar com uma letra, um caractere de sublinhado ou um cifrão $. Não podem iniciar por um número.
- É normal (mas não obrigatório) usar uma letra minúscula para a primeira letra do nome de uma variável.
- Todas as variáveis de tipos primitivos necessitam ser declaradas e inicializadas, através de uma instrução da forma

```
<tipo> NomeDaVariavel = <valor>;
```

onde <tipo> identifica um dos tipos primitivos e <valor> um valor correto para este tipo. Observe a notação <...> utilizada nestas notas

para indicar um elemento de código que deve ser modificado por uma palavra chave, ou um valor real. Existem valores *default* para os tipos primitivos.

- As variáveis utilizadas em objetos, precisam ser declaradas. Os objetos também precisam ser criados. Tanto a declaração, como a criação, poderá ser realizada por comandos separados.

Palavras Reservadas: como qualquer linguagem de programação de alto nível, a linguagem Java possui certas palavras que são especiais para o compilador, desta forma, não é permitido utilizá-las para dar nomes às construções em Java. A lista de palavras reservadas é muito pequena:

PALAVRAS RESERVADAS JAVA					
abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	while
throws	transient	try	void	volatile	

Tabela 1 – Lista de Palavras Reservadas.
Fonte: Autor adaptado do Curso Java Starter.

Programa Principal (Main): esse é o método que inicia a execução do aplicativo Java. É o método que mostrará o resultado do que foi codificado. Quando solicitamos ao interpretador que execute uma determinada classe compilada ele procura o método *main*, se este método não existir irá ser gerada uma exceção informando que o método não foi localizado.

A máquina virtual Java só irá identificar o método *main* se ele possuir as seguintes características:

1. Ser público (public);
2) Estático (static);
3) Não retornar nenhum valor (void);
4) O nome deve ser “main”;
5) Receber como parâmetro um array de String.

1.10 Antes de utilizar o Java – Pré-requisitos

1.10.1 Instalar o JDK

1) Fazer o download da última versão do JDK no sítio: *http://java.sun.com/javase/downloads/index.jsp*
2) Instalar o JDK (É Autoexplicativo);
3) O diretório padrão onde é instalado o JDK é: C:\arquivos de programas\java\jdk[versão]
4) O principal subdiretório é o bin. Neste diretório contém os aplicativos necessários para compilar e rodar os aplicativos JAVA.

1.10.2 Instalar a IDE

Dentre todas as IDEs, será utilizada como exemplo o Eclipse.

Pré-Requisitos para instalação:

a) Antes de tudo o desenvolvedor terá de possuir no mínimo o JRE e JDK do Java. Para maiores informações acesse esse sítio: *http://www.devmedia.com.br/preparacao-do-ambiente-para-desenvolvimento-em-java/25188*;
b) Baixar exatamente essa versão “Eclipse IDE for Java EE Developers” e deverá ser escolhida a versão 32bit/64bit no sítio: *http://www.eclipse.org/downloads*;

c) Após a instalação da IDE Eclipse o desenvolvedor poderá começar a fazer o seu primeiro programa. Para isso, precisará criar uma workspace, projeto, um pacote dentro

do projeto e, por fim, a classe que representa o programa que se quer desenvolver.

1.11 Primeiro Programa Java

```
//Comentários: Este é o meu primeiro programa

public class Exemplo1  {
/*
Comentário: Começa com public class e o nome do programa
e depois a chave
* /

public static void main (String[]  args)  {

//Comentários: Inicia-se o programa principal "main" com public static void main(String[] args)

e chave

    System.out.println("Olá Mundo! ! Meu primeiro programa java");

//Comentários: O comando "System.out.println("")" emite ao monitor informações.

}
}
```

Código 1 – Primeiro programa Java que escreve na tela uma mensagem.
Fonte: Autor

1.12 Exercícios do Capítulo

1) Cite algumas palavras principais do Java?

2) Cite algumas ferramentas do JDK?

3) Crie seu primeiro programa em Java, escrevendo "Meu primeiro programa Java". Para isso precisará instalar uma IDE.

4) Compile e execute a classe desenvolvida no exercício anterior e veja o resultado na aba de console da IDE.

5) Altere o nome do método "main" para "start", compile e execute. O que será que irá acontecer?

6) O que é a máquina virtual Java e o que é um coletor de lixo?

EXPRESSÃO E OPERADORES, TIPOS PRIMITIVOS E CONTROLE DE FLUXO

1.13 Tipos de Dados

As variáveis em Java podem ser utilizadas para guardar valores numéricos, lógicos e caracteres. Existem alguns tipos, que são usados dependendo da aplicação e do valor à ser armazenado.

Os tipos que são mais utilizados são: int, double, boolean e char. Com esses quatros tipos, pode-se fazer quase tudo o que se precisa. Mas, em alguns casos existe a necessidade do uso de outros, por exemplo, se precisar economizar memória. Nesse caso, pode-se trocar double por float e int por short. Mas essas necessidades quase nunca acontecem, fazendo com que quase todos os programas usem esses 4 tipos, inicialmente citados.

Para a armazenagem de palavras e textos, o Java não possui tipo primitivo, mas uma classe chamada de *String*. Em algumas linguagens de alto nível, *String* é tratada como tipo primitivo, isto é, como um tipo de dado de uma variável.

Tipo	Descrição
boolean	Pode ser contido em um bit. Assume valores verdadeiro ou falso.
char	Armazena dados alfanuméricos.
byte	Inteiro de 8 bits em notação de complemento de dois. Pode assumir valores entre – 128 a 127.
short	Inteiro de 16 bits. Os valores possíveis: – 32.768 a 32.767.
int	Inteiro de 32 bits. Pode assumir valores entre: – 2.147.483.648 a 2.147.483.647.
long	Inteiro de 64 bits.

Tipo	Descrição
float	Ponto flutuante com precisão dupla de 32 bits.
double	Ponto flutuante com precisão dupla de 64 bits.

Tabela 2 – Tipos de dados primitivos.
Fonte: Autor adaptado do Curso Java Starter, módulo 2, pg. 3 e 4.

Veja exemplo de um código abaixo aplicando alguns dos tipos primitivos. Este código foi produzido utilizando a IDE Eclipse.

```
package br.rp;

        public class Teste {

        public static void main(String[] args) {

            boolean boo;
            char c;
            byte x;
            short s;
            int i;
            long l;
            Float f;
            double d;

            c = 33;
            System.out.println("============================");
            System.out.println("char");


System.out.println("============================");
System.out.println("valor char = " + c);
System.out.printf("valor numero = %d \n",(int)c);
System.out.println("----------------------------");


x = 9;
s = x;
i = s;
System.out.println("============================");
System.out.println("valores inteiros");
System.out.println("============================");
System.out.println("i = s = x = "+i);
```

```
36 i *= 10;
37 System.out.println("novo valor de i = "+i);
38
39
40 x = (byte)i;
41 System.out.println("novo valor de x = "+x);
42
43
44 l = i;
45 System.out.println("valor de l = "+l);
46 System.out.println("-----------------------------");
47
48
49 System.out.println("=============================");
50 System.out.println("valores ponto flutuante");
51 System.out.println("=============================");
52 d = 125.32;
53 System.out.println("valor de d = " + d);
54 d = 125.32d;
55 System.out.println("valor de d = " + d);
56 d = 125.32f;
57 System.out.println("valor de d = " + d);
58
59
60 f = (float)125.32;
61 System.out.println("valor de f = " + d);
62 f = 125.32f;
63 System.out.println("valor de f = " + d);
64 f = (float)125.32d;
65 System.out.println("valor de f = " + d);
66 System.out.println("-----------------------------");
67
68 System.out.println("=============================");
69 System.out.println("valores booleano");
70 System.out.println("=============================");
71 boo = true;
72 System.out.println("valor de boo = " + boo);
73 boo = (1 > 2);
74 System.out.println("valor de boo = " + boo);
75 boo = (f == d);
76 System.out.println("valor de boo = " + boo);
77 System.out.println("-----------------------------");
78
79
80 System.out.println("=============================");
81 System.out.println("exercitando os tipos");
82 System.out.println("=============================");
83 System.out.println("podemos converter int para float?");
84 System.out.println("inteiro antes da conversao = " + i);
85 System.out.println("float antes da conversao = " + f);
86 f = i;
87 System.out.println("inteiro depois da conversao = " + i);
88 System.out.println("float depois da conversao = " + f);
89 System.out.println("--");
```

```
90
91
92  f = (float)d;
93  System.out.println("podemos converter float para int?");
94  System.out.println("inteiro antes da conversao = " + i);
95  System.out.println("float antes da conversao = " + f);
96  i = (int)f;
97  System.out.println("inteiro depois da conversao = " + i);
98  System.out.println("float depois da conversao = " + f);
99  System.out.println("----------------------------");
100 }
101 }
```

Código 2 – Programa Java que Testa os Tipos Primitivos de Dados.
Fonte: Autor

1.14 Expressão e Operadores

Uma expressão é uma instrução para efetivar uma operação que produzirá um valor. Operadores são símbolos específicos usados para as operações matemáticas, atribuições, comparações e operações lógicas.

Os operadores são divididos em:

a) Operadores aritméticos:

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da Divisão (módulo)

Tabela 3 – Tipos de operadores aritméticos.
Fonte: Autor.

Para a operação de potência o Java não possui um operador próprio, assim, deve-se utilizar o método *pow da classeMath* do pacote *lang*.

b) Operadores de atribuição:

Os operadores de atribuição caracterizam-se pela notação compacta que desencadeia em uma operação aritmética. Significa que, é realizada a atribuição do valor de retorno à variável que continha o primeiro termo da operação.

Operador	Expressão	Descrição
+ =	A += B	A = A+B
– =	A-=B	A = A-B
* =	A*=B	A = A*B
/ =	A/=B	A = A/B
% =	A%=B	A = A%B

Tabela 4 – Tipos de operadores de atribuição.
Fonte: Autor.

c) Operadores de incremento e decremento

São operadores que atuam sobre uma mesma variável numérica, incrementando ou decrementando o seu valor de uma unidade:

Operador	Exemplo	Descrição
++	++x	Somar 1 à variável x e depois calcular a expressão que x participa
	x++	Calcular a expressão que a participa e depois somar 1 à variável x
--	--x	Subtrair 1 da variável x e depois calcular a expressão que x participa
	x--	Calcular a expressão que x participa e depois subtrair 1 da variável x

Tabela 5 – Tipos de operadores de incremento e decremento.
Fonte: Autor

Exemplo:

```
int z = 6, w = 8; // z vale 6 e w vale 8
w += ++z; // z agora vale 6 + 1 = 7 e w vale 8 + 7 = 15
z – = w--; // z agora vale 7 – 15 = – 8 e w vale 15 – 1 = 14
```

d) Operadores de Comparação:

Os operadores de comparação são utilizados com valores numéricos e retornam valores do tipo: verdadeiro ou falso:

Operador	Descrição
==	é igual a
!=	é diferente de
<	é menor que
>	é maior que
<=	é menor ou igual a
>=	é maior ou igual a

Tabela 6 – Lista de Operadores de Comparação.
Fonte: Autor.

e) Operadores Lógicos

Os operadores lógicos trabalham sobre o retorno dos valores booleanos. A saída destes pode ser verdadeiro ou falso.

Operador	Descrição	Utilização	Funcionamento
&&	E lógico (AND)	x && y	retorna verdadeiro se x e y forem ambos verdadeiro, senão retorna falso. Se x for falso, y não é avaliado.
&	E lógico (boolean AND)	x & y	retorna verdadeiro se x e y forem ambos verdadeiro, senão retorna falso. As expressões x e y são sempre avaliadas
||	OU lógico (OR)	x || y	retorna verdadeiro se x ou y forem ambos verdadeiro, senão retorna falso. Se x for verdadeiro, y não é avaliado.
|	OU lógico (boolean inclusivo OR)	x | y	retorna verdadeiro se x ou y forem ambos verdadeiro, senão retorna falso. As expressões x e y são sempre avaliadas
^	OU Exclusivo lógico (boolean exclusivo OR)	x ^ y	retorna verdadeiro, se x for verdadeiro e y for falso (vice-versa), senão retorna falso.
!	NÃO lógico (NOT)	!x	retorna verdadeiro se x for falso, senão retorna falso.

Tabela 7 – Lista de Operadores Lógicos.
Fonte: Autor.

1.15 Entrada de Dados

Como ler algo que o usuário digita no teclado? Para isso, deve-se utilizar a classe Scanner. Esta classe tem como princípio separar a entrada dos textos em conjuntos gerando *tokens*, que são fluxos de caracteres separados por delimitadores que por padrão correspondem aos espaços em branco, tabulações e mudança de linha.

Para utilizar esta classe, deverá ser importada no início do código <import java.util.Scanner> para que o compilador entenda o seu uso. Segue a sintaxe abaixo para sua utilização:

```
import java.util.Scanner;
//Comentário: Deve-se colocar o import antes da abertura da classe
que irá utilizar o Scanner

Scanner <nome qualquer> = new Scanner(System.in);
/* Comentário: O "nome qualquer" significa que o programador po-
derá utilizar qualquer nome, por exemplo "s" */

String str = s.nextLine();
int i = s.nextInt();

/*Comentário: Para cada tipo de variável o comando "next" deverá
ser adaptado. No caso de inteiro será nextInt(), no caso de String será
nextLine() e no caso de Double nextDouble().*/
```

Exemplo:

```
import java.util.Scanner;
public class TesteScanner {

    public static void main(String[] args) {

        Scanner str = new Scanner(System.in);

        System.out.print("digite uma string: ");
        String string = str.nextLine();
        System.out.print("digite um inteiro: ");
        int i = str.nextInt();
        System.out.print("digite um numero ponto flutuante: ");
        double x = str.nextDouble();
        }
}
```

Código 3 – Programa Java para Leitura de Dados.
Fonte: Autor

1.16 Controle de Fluxo

É comum que as linguagens de programação de alto nível possuam várias estruturas de controle de fluxo da aplicação para simplificar a codificação de programas e propiciar um melhor estilo de programação. O controle de fluxo é realizado utilizando condicionais, que são instruções que modificam o fluxo de execução padrão do programa.

1.16.1 If-Else

O comando " if " executa um trecho de código apenas uma vez, dependendo de sua condição. É também chamado de estrutura de seleção simples. Utiliza-se a estrutura de seleção para escolher entre rotinas de ação alternativas dentro de um programa. A estrutura de seleção if/else, também chamada de estrutura de seleção dupla, permite a especificação de condições e execuções distintas para resultados verdadeiros ou falsos. Segue a sintaxe abaixo:

```
if (expressão) {
    instruções
} else if (expressão) {
    instruções
} else {
    instruções
}
```

Exemplo:

```
33          if ((media <= 10) && (media >= 9)){
34              System.out.println(" ");
35              System.out.println("Nome do aluno: " + nome);
36              System.out.println("Disciplina: " + disciplina);
37              System.out.println("Primeira nota: " + nota1);
38              System.out.println("Segunda nota: " + nota2);
39              System.out.println("Media: " + media);
40              System.out.println("Conceito: A");
41              System.out.println("APROVADO!");
```

```
42          }
43
44          else if ((media >= 7.5) && (media <= 9)){
45              System.out.println(" ");
46              System.out.println("Nome do aluno: " + nome);
47              System.out.println("Disciplina: " + disciplina);
48              System.out.println("Primeira nota: " + nota1);
49              System.out.println("Segunda nota: " + nota2);
50              System.out.println("Media: " + media);
51              System.out.println("Conceito: B");
52              System.out.println("APROVADO!");
53          }
54
55          else {
56              System.out.println(" ");
57              System.out.println("Nome do aluno: " + nome);
58              System.out.println("Disciplina: " + disciplina);
59              System.out.println("Primeira nota: " + nota1);
60              System.out.println("Segunda nota: " + nota2);
```

Código 4 – Programa Java Utilizando if-else.
Fonte: Autor

1.16.2 Switch-Case

Estrutura de seleção que testa uma série de decisões utilizando uma variável ou expressão separadamente para cada um dos vários valores. O switch é também chamado de estrutura de seleção múltipla. Esse tipo de estrutura somente testa valores dos seguintes tipos: byte, short, int e char. Segue abaixo a sintaxe:

```
switch(letra) {
case 'A' :
    System.out.println("A");
    break;
case 'B' :
    System.out.println("B");
    break;
...
default:
    System.out.println("?");
}
```

Exemplo:

```
        switch (operador) {
        case '+':
            System.out.println(n1+n2);
            break;

        case '-':
            System.out.println(n1-n2);
            break;
        case '*':
            System.out.println(n1*n2);
            break;
        case '/':
            System.out.println(n1/n2);
            break;

            default:
                System.out.println("O operador digitado não é válido");
        }
```

Código 5 – Programa Java Utilizando switch-case.
Fonte: Autor

1.16.3 While e do-while

A estrutura de repetição "while" (enquanto) permite identificar que uma ação deve ser repetida enquanto alguma condição continuar verdadeira. Para o "while" as instruções serão executadas enquanto a expressão *boolean* for verdadeira. Já para o "do-while" (faça/enquanto) será executada a instrução pelo menos uma vez e continua executando enquanto a expressão *boolean* for verdadeira. Segue abaixo a sintaxe:

while (expresão)
{
    instruções;
}
ou
do
{
    instruções;
} **while** (expresão);

Exemplo:

```
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        double a=0, b=0, c=0;
        double delta, x1, x2, draiz, divisa, menosb, delta2, delta3;
        int opt=0;
        System.out.println("Este programa calcula equações de 2 gra
        System.out.println("ENTER para continuar");
        String sdasd = sc.nextLine();
        System.out.println("MENU PRINCIPAL");
        System.out.println("(1) COMEÇAR");
        System.out.println("(2) TUTORIAL");
        System.out.println("(0) SAIR");
        System.out.println(" ");
        System.out.println("Criado por Wallace Nery");
        opt = sc.nextInt();
        while(opt!=0){
            if(opt==2){Scanner fs = new Scanner(System.in);
                System.out.println("Uma equação é assim: aX²+bX+C");
                System.out.println("Exemplo: a=2,   b=3,   c=4");
                System.out.println("A equação ficará assim = 2X²+3X+
                System.out.println("Isto é tudo que você precisa col
                System.out.println("O Programa calcula todo o resto.
                System.out.println(" ");
                System.out.println("Pressione ENTER para voltar ao m
                String asiedhaid = fs.nextLine();}
            if(opt<0||opt>2){System.out.println(" ");
                System.out.println("Por favor, digite alternativas v
```

Código 6 – Programa Java Utilizando while.
Fonte: Autor

1.16.4 For

Em algumas situações, precisamos de laços de repetições nos quais certa variável é usada para contar o número de execuções. A estrutura de repetição "for" se responsabiliza pelos detalhes da repetição controlada por um contador.

O "for" pode ter apenas uma instrução na sua composição. Assim, não é necessário abrir um bloco. O "for" já possui alguns comandos na sua composição, ou seja, no seu cabeçalho, como a inicialização da variável e o passo da repetição, ou seja, o incremento/decremento da variável. Segue a sintaxe abaixo:

for ([expressão 1]; [condição]; [expressão 2])
[comando].

Exemplo:

```
public class Questão22 {
        public class for1 {
        public static void main(String[] args) {
            for(int count=1 ; count <= 10 ; count++){
                System.out.println(count);
            }
        }

    }
```

Código 7 – Programa Java Utilizando for.
Fonte: Autor

1.17 Exercícios do Capítulo

1) Utilizando o Java instalado no seu computador, desenvolva um programa que imprima o resultado das expressões abaixo:
   a) 5 – 3 – 1 + 6 + 2 + 9
   b) 5 x 3 – 2 x 5
   c) 9 + 6 – 3 / 8 x 8
   d) 6 % 2 –8

2) Escreva um programa que recebe do teclado um número de 1 a 7 e imprima o dia da semana correspondente (ex: 1-Domingo, 2-Segunda, etc).

3) Escreva um programa que recebe repetidamente do teclado um número de 1 a 12, enquanto esse número for diferente de 0 (zero), e imprime o mês do ano correspondente (Ex: 1-Janeiro, 2-Fevereiro, etc). Se o número for diferente do intervalo de 1 a 12, imprimir “Mês Inválido”.

4) Implemente um código que utilize dois laços de 0 a 5, um interno ao outro, imprimindo os contadores e mensagem “laço 1” para o primeiro e “laço 2” para o segundo laço. Quando estes forem iguais, o programa deve passar à próxima interação do laço mais externo, caso contrário, deve imprimir os valores dos contadores dos dois laços.

VETORES, MATRIZES E INTERFACE GRÁFICA

1.1 Introdução

A linguagem Java também possui a funcionalidade de vetores e matrizes (*arrays unidimensionais e multidimensionais*).

Arrays são variáveis que servem para guardar vários valores do mesmo tipo de forma uniforme na memória. Como um array pode guardar vários valores temos que definir quantos valores ele deve guardar para que seja reservado o espaço necessário em memória.

1.2 Vetores ou Arrays Unidimensionais

Os vetores são uma forma muito conveniente de organizar informações em fileira. Por exemplo, podemos formar um vetor com valores de números crescentes a partir do 1 até o 10 da seguinte forma:

```
int numeros[] = {1,2,3,4,5,6,7,8,9,10}; // declara um vetor de 10 posições com valores estáticos
```

Para utilizar vetores ou matrizes em Java deve se seguir três passos:

a) Declarar o vetor ou matriz:

   Para realizar a declaração, deverá acrescentar um par de colchetes antes ou depois do nome da variável. Veja exemplo:

```
int vet[];
int [ ]valor;
```

b) Reservar espaço de memória e definir o tamanho:

   Inicialmente deverá definir o tamanho do vetor, ou seja, a quantidade total de elementos à armazenar. Depois é necessário alocar espaço de memória para armazenar os valores. Deverá ser utilizada a palavra *new*. Veja exemplo:

```
vet = new int[10]; // declara um vetor de 10 posições
valor = new int[70]; // declara um vetor de 70 posições
```

c) Armazenar elementos no vetor ou matriz:

Para guardar uma informação em alguma posição de um vetor ou matriz, é preciso prover um índice que indique a localização desse elemento. Veja exemplo abaixo:

vet[5]=6; // inseri o número 6 na posição 5 do vetor

OBSERVAÇÃO

É importante destacar que os índices começam em zero e vão até o número de posições reservadas, menos um. No caso de tentar atribuir um valor a um elemento cujo índice esteja fora desse intervalo, ocorrerá um erro que impedirá a execução do programa. Então, deve-se ter cuidado ao trabalhar com esses limites, garantindo que o programa funcione corretamente.

Exemplo:

```
5⊖      public static void main(String[] args) {
6           int i;
7           int Vetor[];
8           for(i=0; i<5; i++)
9                   Vetor[i] = i+1;
```

Código 8 – Programa Java Utilizando Vetor.
Fonte: Autor

Para conhecer o tamanho total de um *array* deve-se utilizar o atributo *length. E*ste atributo retorna um valor inteiro (int) que aponta qual o espaço máximo de armazenamento do array.

ATENÇÃO

– A primeira posição de qualquer array é sempre 0;
– Última posição é sempre o seu tamanho – 1 (length – 1).

1.3 Matrizes ou Arrays Multidimensionais

Uma matriz é semelhante a um vetor onde cada elemento é por sua vez um vetor, ou seja, um vetor de vetores e todos de mesmo tamanho. No Java, existe a possibilidade de criar a matriz com tamanhos diferente dos vetores. Para isso, cada elemento deverá ser criado independentemente.

Conforme já discutido em vetores deverão ser seguidos os três passos demonstrados anteriormente para utilizar uma matriz. Segue abaixo a sintaxe:

```
double Mat1[ ][ ],Mat3[ ][ ][ ];
Mat1 = new double[1][2];
Mat1[1][1]=2;
```

Exemplo:

```
3  public class Exemplomatriz {
4
5      public static void main(String[] args){
6          int[][] x = new int[10][];
7          for(int i = 0; i < 9; i++)
8              x[i] = new int[i+1];
9          for(int i = 0; i < 9; i++){
10             x[i][0] = 1;
11             x[i][0] = 1;
12             for(int j = 1; j < i; j++)
13                 x[i][j] = x[i-1][j-1]+x[i-1][j];
14         }
15 }
```

Código 9 – Programa Java Utilizando Matriz.
Fonte: Autor

1.4 Manipulando Vetores Utilizando a Classe Arrays

A classe Arrays, disponível no pacote java.util, fornece uma grande quantidade de métodos utilitários, como por exemplo métodos para ordenação, procura, comparação e etc.. Estes métodos são muito úteis quando manipulamos arrays. A seguir serão apresentados os principais métodos e a as respectivas funcionalidades oferecidas.

- Ordenação: Realizada utilizando-se o método "sort" cujo parâmetro é o vetor a ser ordenado;
- Pesquisa: É realizada utilizando-se o método "binarySearch" que retorna a posição que o elemento foi encontrado no array. Se o elemento não for encontrado será retornado um valor negativo;
- Preenchimento: Utilizando-se a classe utilitária Arrays é possível preencher um determinado array com o elemento desejado;
- Comparar elemento desejado: O método "equals" compara valor a valor (dado dois arrays) e retorna verdadeiro se os vetores são iguais em valores e índices.

1.5 Entrada de Dados – Interface Gráfica

Nesta seção será utilizada a biblioteca Swing que introduzirá o conceito de interface gráfica com o usuário. Esta biblioteca fornece a possibilidade de geração dos objetos gráficos em Java.

Quando se pensa em troca de dados com o usuário, pensa-se logo em algo mais amigável, isto é, que gere facilidade de utilização. O uso do "console" (*prompt*) não é a melhor opção neste caso. Como intuito de melhorar esta troca de informações com o usuário, pode-se utilizar a classe JOptionPane. Neste tópico não será aprofundado o assunto, mas cabe buscar mais informações.

A classe *JOptionPane* cria uma simplicidade no desenvolvimento de uma interface com o usuário, facilitando diálogos, na solicitação de

dados e na troca de mensagens. Abaixo estão relacionadas as formas de diálogo desta classe:

Método	Descrição
showConfirmDialog	Comando utilizado para confirmações e diálogos. Resposta esperadas: Sim, Não ou Cancela.
showInputDialog	Comando para entrada de dados. Por exemplo, pelo teclado.
showMessageDialog	Comando utilizado para exposição de mensagem.
showOptionDialog	Comando com algumas funções: Informativo, Entrada de Dados e Confirmações.

Tabela 8 – Comandos da Classe *JOptionPane*.
Fonte: Autor adaptado do Curso Java Starter.

Exemplo:

```
1  package br.terceira;
2
3  import javax.swing.JOptionPane;
4
5
6
7  public class JoptionQ {
8
9  public static void main(String[] args) {
10
11  String nome = null;
12
13  nome = JOptionPane.showInputDialog("Poderia me dizer seu nome?");
14
15  JOptionPane.showConfirmDialog(null, "Você digitou " + nome + "?");
16
17 }
18
19 }
```

Código 10 – Programa Java Utilizando a Classe *JOptionPane*.
Fonte: Autor.

1.6 Exercícios do Capítulo

1. Desenvolva um programa para a realização de saques em um caixa eletrônico considerando que o mesmo armazena cédulas de R$50,00, R$20,00, R$10,00 e devem ser entregues ao cliente de forma que gere o menor número possível de cédulas.

2) Crie um programa que ordene um vetor com 8 números inteiros. O código deve fazer uma comparação de cada número com o seu sucessor e se a ordem não for crescente os elementos devem ter suas posições trocadas. O vetor deverá estar totalmente ordenado.

3) Crie um programa que construa um vetor de inteiros (int) de tamanho 1.000 com valores atribuídos da seguinte forma: cada posição conterá o resultado da operação 1.000 – índice da posição, isto é, a posição 0 terá o valor 1.000, a posição 1 terá o valor 999 e assim por diante. Ordene utilizando a classe utilitária Arrays (pesquise como fazer).

APLICANDO CONCEITOS RELACIONADOS COM COMPOSIÇÃO

2.1 Introdução

O objetivo deste capítulo é desenvolver a capacidade de resolução de problemas através do paradigma da Orientação a Objetos (OO). Será apresentado um breve histórico que contextualiza o surgimento e evolução do paradigma OO. Além disso, os fundamentos básicos, tais como: Classes, Objetos, Métodos, Atributos, Encapsulamento, Herança de Classe, Generalização/Especialização, Classe Abstrata/Método Abstrato, Método Construtor e Polimorfismo serão trabalhados objetivando a compreensão de como é possível fazer representações de "mundo" em OO. Em um segundo momento, será exposto como colocar na prática o paradigma OO através de uma linguagem de programação. Lembrando que a grande fórmula de estudo, neste caso, é a resolução de muitos exercícios. Será utilizada a linguagem de programação Java para isso, mas sem o objetivo de trabalhar completamente todos os recursos da mesma. A estratégia da utilização de níveis para cada exemplo e exercícios, ampliará paulatinamente a complexidade, e assim ao final do aprendizado, o leitor-estudante estará capacitado e confiante neste paradigma de programação.

Codificar um problema corriqueiro em uma linguagem que o computador entenda é desafiador e empolgante. O segredo é simplificar e reduzir o mesmo, focando na resolução em etapas e lidando primeiramente com suas questões essenciais. Além disso, deve-se ter conhecimento da linguagem a ser utilizada, conforme dito anteriormente. Logo, visa-se transformar o problema em um algoritmo que pode ser executado em uma máquina computacional. Um dos desafios inicias é criar uma excelente ambientação com a linguagem escolhida. Para isso, a resolução de vários exercícios irá proporcionar a expertise necessária para a criação de soluções diversas e criativas.

O desenvolvimento de projetos de softwares não é uma tarefa fácil. Os projetos, geralmente, têm que ser robustos, atuar sobre problemas complexos e estar implantados em curto prazo (*time-to--market*). Entretanto, é necessário produzir softwares genéricos e extensíveis que possam ser aplicados a uma gama de aplicações. Ao longo dos anos, muitas metodologias surgiram na Engenharia de Software com o intuito de reduzir a complexidade e aumentar a produtividade no desenvolvimento de software. A Orientação a Objetos é reconhecida hoje como o principal paradigma para atender a esses requisitos. Assim, conhecimentos em fundamentos nesta área estão na essência da formação de um profissional que deseja atuar na área de Informática.

Este livro aborda os conceitos da Orientação a Objetos através da interação entre a teoria e a prática. Está no escopo do conteúdo uma introdução à Engenharia de Software abordando técnicas como Programação Orientada a Objetos, Padrões de Projeto, Framework e Testes de Software.

Observa-se que desenvolvedores e interessados na produção de software devem estruturar os seus conhecimentos, mesmo que sejam iniciantes na área da Ciência da Computação, em princípios e boas práticas da Engenharia de Software. Questões como reutilização, arquitetura em camadas, delegação clara de responsabilidades, modularização, redução de acoplamento e complexidade, teste de software são alguns dos princípios que irão nortear toda a apresentação do conteúdo deste livro.

Apesar de demonstrar alguns conceitos e exemplos de programação na linguagem Java, esta obra não tem como objetivo a apresentação desta linguagem em detalhes. Assim, aborda-se a linguagem Java somente para apresentar o conteúdo relevante para o entendimento dos exemplos que facilitam o entendimento dos fundamentos da Orientação a Objetos. Conhecimentos mais específicos da linguagem Java sobre tipos de dados e comandos devem ser obtidos através de literaturas complementares.

O método de apresentação do conteúdo do livro consiste primeiramente em tratar dos fundamentos teóricos básicos e, posteriormente, serão apresentados exemplos práticos para facilitar e reforçar os conceitos inicialmente vistos. Em seguida, será apresentado o conteúdo mesclando a teoria e a prática. Para facilitar a tarefa de programação Orientada a Objetos, os exemplos de algoritmos estão em uma ordem crescente de complexidade, ou seja, cada exemplo acrescenta novos aspectos ligados a este paradigma de desenvolvimento.

A motivação para o estudo do conteúdo deste livro é entender a necessidade para a uma boa formação dos conceitos aqui apresentados, pois os mesmos são a base para a obtenção de diversos outros conhecimentos imprescindíveis para a realização de projetos em desenvolvimento de software.

Existe uma separação clara dos profissionais que possuem esta base e, assim, estão aptos para assimilar e construir, até de forma autodidata, softwares bem estruturados. No contexto da Internet e dos dispositivos móveis é imprescindível ter conhecimento sobre tecnologias de desenvolvimento Orientado a Objetos para a construção de Sistemas Web, Sistemas Embarcados, Sistemas Distribuídos, Programação para Dispositivos Móveis, etc.

Neste cenário, o profissional reutiliza componentes e técnicas que reduzem o ciclo de vida de construção de software e aumentam a qualidade dos projetos, questões fundamentais para o sucesso do mesmo.

A transposição da fase inicial de entendimento dos conceitos sobre Orientação a Objetos pode parecer difícil no seu começo devido a mudanças de paradigma e da necessidade de bons fundamentos para resolver problemas e codificá-los em uma linguagem. Para os que já programam no paradigma funcional existe a necessidade de reestruturar o pensamento da forma de como é feita a construção de software, ou seja, é necessária uma quebra de paradigma em relação à forma de como se projeta software convencional.

Percebe-se que com a assimilação dos conceitos de Orientação a Objetos dificilmente volta-se a pensar na resolução de problemas usando o paradigma funcional. Ressalta-se que alguns algoritmos com poucas linhas de código podem ficar mais difíceis de serem resolvidos na programação Orientada a Objetos. Porém como os softwares nas corporações são normalmente de médio ou grande porte, a Orientação a Objetos é fundamental para futuras manutenções corretivas e evolutivas. Bons estudos e aproveitem bastante o conteúdo deste material.

INDICAÇÃO DE LEITURA

Este material usa como base nos seus exemplos práticos, em todas as suas seções, o livro "Core Java 2: Fundamentos (vol.1.)". Este é um clássico de referência em Programação OO dos autores Gary Cornell. Assim recomenda-se que você complemente os seus conhecimentos com a leitura deste livro.

2.2 Questões Históricas da Orientação a Objetos

Métodos de modelagem Orientados a Objetos começaram a aparecer entre meados da década 70 e início dos anos 80. Os primeiros registros da concepção dos conceitos de Orientação a Objetos surgiram com a linguagem Simula na década de 60, na Noruega, no centro de Norweigan Computin Center (NCC), sobre a responsabilidade de Kristen Nygaard e Ole-Johan Dahl. O cerne dessa linguagem era a criação de um modelo computacional que fosse similar às estruturas do mundo real, facilitando assim, o projeto de sistemas. Após a linguagem Simula um outro projeto a implementar os conceitos de Orientação a Objetos, muito conhecido, foi a linguagem Smalltalk criada pelo núcleo de pesquisa da Xerox PARC e desenvolvida pelos pesquisadores Adele Goldberg, Alan Key e Dan Ingalls. Estes métodos e linguagens inicialmente permearam o meio acadêmico em pesquisas pouco aplicáveis.

Outra linguagem importante na história da Orientação a Objetos foi a linguagem C++. Esta é uma evolução da linguagem C que está sobre o paradigma Estruturado Funcional. A primeira versão foi desenvolvida em 1980 com o nome de 'C with Classes'. Em 1983 é lançada a primeira versão do C++ com diversas características de Orientação a Objetos. O pesquisador responsável por essa evolução da linguagem C foi Bjarne Stoustrup. Apesar da linguagem C++ ser poderosa, alguns pesquisadores e professores têm restrições quanto ao primeiro contato do aluno, com os conceitos de Orientação a Objetos, ser com a linguagem C++. Um dos principais motivos dessa restrição deve-se ao fato do C++ ser uma linguagem hibrida. Como foi dito, o C++ é uma evolução da linguagem C que está sobre o paradigma Estruturado Funcional, permitindo que o aluno programe desrespeitando os conceitos da Orientação a Objetos.

Já a linguagem Java surgiu no ano de 1991, projetada por James Gosling, Bill Joy e Guy Steele na Sun Microsytems. O seu projeto inicial visava ser uma linguagem para programação de sistemas embarcados. Estes programas rodariam em dispositivos eletrônicos da Sun Microsystem. Com o surgimento da Internet e da WWW (*World Wide Web*) a linguagem Java começou a ser amplamente utilizada. O sucesso da linguagem está associado com o recurso de independência de plataforma e por ser uma linguagem parecida e mais simples que C e C++, visto que a alocação e a liberação de memória não são feitos diretamente pelo programador. Portanto, Java possui um mecanismo de gerência de memória e coleta de lixo que evitam erros de manipulação de memória. Existem outras linguagens de programação, porém menos conhecidas.

Na década de 90 a aplicação comercial dos conceitos da Orientação a Objetos se fortaleceu, saindo do meio acadêmico e sendo aplicados nas organizações no desenvolvimento de suas soluções de software comerciais. Uma das causas dessa popularização foi o aparecimento das ferramentas RAD (*Rapid Aplication Development*), que objetivavam aumentar a produtividade na construção de software. Ferramentas estas que eram aplicadas, principalmente, na construção de

elementos da camada de interface, tais como menus, botões, telas, tabelas, combos, etc. Os mecanismos de construção de aplicações para o desenvolvedor na ferramenta RAD se dá através da reutilização de componentes visuais, previamente construídos, que são manipulados e customizados para atender aos requisitos específicos da aplicação que está sendo construída. Um exemplo clássico de ferramentas RAD são as IDE Visual Basic , Delphi, Centura, etc. A figura abaixo ilustra a IDE Visual Basic com a barra de componentes a esquerda e a direita a lista de propriedades que podem ser configuradas para os componentes.

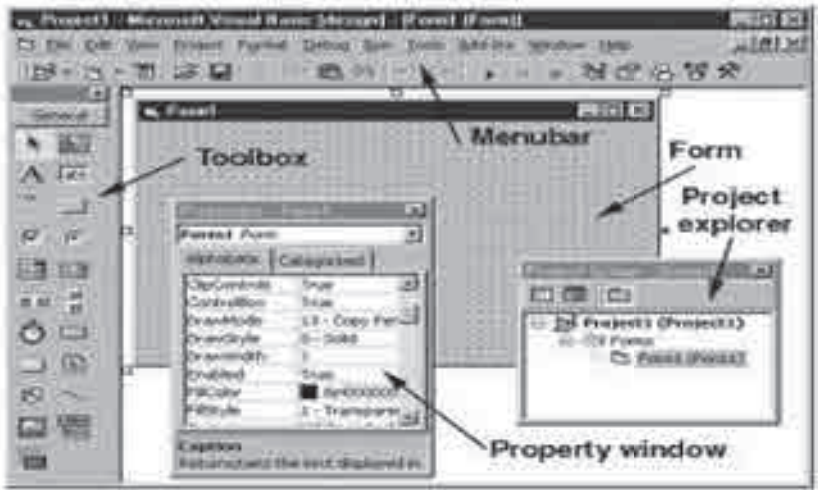


Figura 2 – IDE Visual Basic.
Fonte: Autor

Apesar dos benefícios alcançados com a produtividade e facilidade no desenvolvimento de software, diversos problemas relacionados com a construção de aplicações baseados em RAD são identificados. Uma primeira questão a ser analisada é que um software normalmente possui um custo muito mais alto de manutenção do que de desenvolvimento. Como as ferramentas RAD e as técnicas de Engenharia de Software difundidas na década de 90 os desenvolvedores acabavam construindo softwares com um forte acoplamento entre

a camada de aplicação e a camada de interface (ver figura abaixo). Este tipo de solução gerava graves problemas de acoplamento e baixa reutilização dificultando a manutenção corretiva e evolutiva das aplicações. Por exemplo, o espalhamento de regras de negócio nas telas exige uma mudança em vários pontos do software.

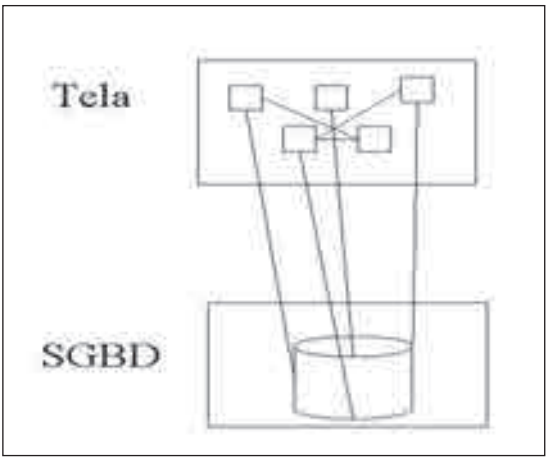


Figura 3 – Exemplo de desenvolvimento com forte acoplamento entre a camada de interface e camada de persistência.
Fonte: Autor

Na Ciência da Computação a modelagem Orientada a Objeto se popularizou na década de 90 com o sucesso da linguagem de notação UML (*The Unified Modeleing Language*) que unificou as melhores práticas da OMT (*Object Modeling Technique*) proposta por Booch, Rumbough e modelo OOSE (*Object-Oriented Software Engineering*) proposta por Jacobson. A UML definida como uma linguagem unificada para modelagem gráfica de objetos foi um marco na Engenharia de Software, pois antes do seu surgimento cada autor de referência na área tinha a sua forma de modelagem específica com alguns poucos elementos em comum. A UML tem uma grande aceitação na Engenharia de Software e já extrapola o seu uso em outras áreas. Com isso, é uma estratégia interessante para modelagem de conceitos.

A Orientação a Objetos está ainda em plena evolução, apesar de ser um paradigma de desenvolvimento da década 1970. Exemplo são as técnicas de desenvolvimento baseadas em: Framework, Padrões de Projeto, Orientação a Aspectos, TDD (*Test Driven Development*), BDD *Behavior Driven Development*) entre outras que agregam novas possibilidades a Orientação a Objetos.

INDICAÇÃO DE LEITURA

Vamos entender melhor dessa história tão rica do surgimento do paradigma OO. Para aprofundar esse conteúdo acesse http://www.carto.eng.uerj.br/fgeorj/segeo1996/129/index.htm

2.3 Conceitos Básicos OO

Esta seção apresenta os conceitos iniciais sobre Orientação a Objetos. Estes conhecimentos são de suma importância, pois representam a base para a compreensão dos demais assuntos apresentados ao longo deste livro.

2.3.1 Objeto

No paradigma da Orientação a Objetos a principal estrutura de representação é o Objeto. Semanticamente é uma forma de mapear coisas de mudo real com estruturas internas de um programa de computador. Logo, o conceito Objeto é definido como qualquer elemento ou 'coisa' do mundo real. Uma definição mais formal pode ser conceituada da seguinte forma: um objeto é uma abstração de estruturas do mundo real.

Por exemplo, uma pessoa, um relógio, um produto, etc são exemplos de objetos. Todo objeto é composto pelo seu estado que é um conjunto de dados que o caracterizam. Imaginemos um objeto Carro com o seguinte estado: uma placa XPT 36789, a informação da

quantidade de 40 litros no seu tanque e o seu consumo de 10 km/litros.

Um objeto, também, possui ciclo de vida, ou seja, ele nasce modifica o seu estado ao longo do tempo e morre. A modificação de um objeto no decorrer do seu ciclo de vida representa uma mudança no seu estado ou transição de estado. Similar a um elemento do mundo real, um objeto é único por definição. Portanto, um objeto Carro pode até ter o mesmo estado de outro objeto Carro, mas são objetos diferentes por definição. Cada um possui o seu ciclo de vida próprio, portanto nascem, mudam de estado e morrem de forma particular. o Objeto é uma unidade semântica que agrupa dados, além disso possui um conjunto de comportamentos.

Comportamentos são mecanismos que normalmente baseados no estado permitem que o objeto faça algo e ou responda sobre uma determinada ação. A mudança de estado de um objeto é normalmente realizada através de um comportamento. Por exemplo, a redução da quantidade de litros no tanque do Carro é uma mudança de estado efetuada por um comportamento do próprio objeto. Em Programação Orientada a Objetos o comportamento é escrito através de funções que são denominadas de métodos do objeto.

VOCÊ SABIA?

Por exemplo, um objeto Carro pode possuir o método double getAutonomiaKm() para responder sobre a sua autonomia. Esta ação é processada sobre a responsabilidade do objeto Carro que através do seu estado pode responder a esse questionamento. Assim, com base no exemplo anterior, para saber a autonomia seria feito o cálculo da (Quantidade de litros x Consumo), então (40 litros x 10km/litros) = 400km.

Ainda sobre Objeto, o mesmo é um modulo indivisível ou atômico com métodos (comportamento) e o seu estado que são variáveis que mapeiam informações compartilhadas por todos os métodos.

Os métodos e o estado de um objeto são agrupados no mesmo local sobre uma semântica única, ou seja, em um objeto.

2.3.2 Classe

Um conjunto de objetos do mesmo tipo é denominado de Classe. Define-se como uma forma natural de agrupar os objetos que possuem as mesmas características. Diferente do objeto a Classe não possui estado, assim pode-se conceituar Classe como uma estrutura estática utilizada para a construção de objetos (ver figura abaixo).

Quando um Objeto é gerado a partir de uma Classe este herda todas as características da Classe, adicionando o seu estado particular. Assim, uma Classe determina os padrões estruturais do objeto. Uma Classe é constituída de atributos (variáveis compartilhadas) e comportamento representados pelas funções denominadas na concepção de um Objeto como métodos.

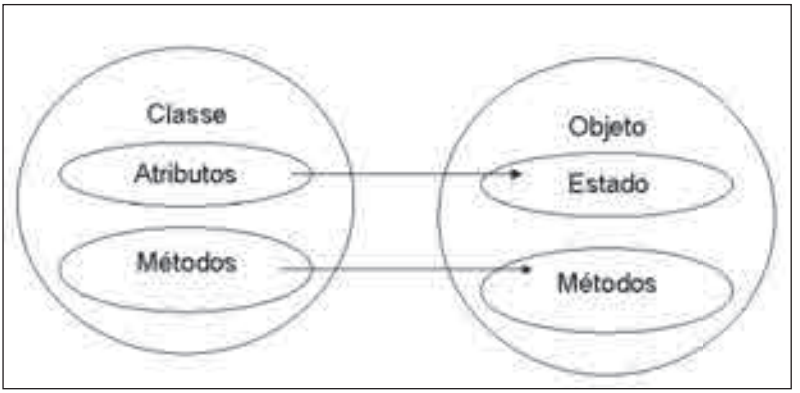


Figura 4 – Representação de Classe e Objeto.
Fonte: Autor

No exemplo do objeto Carro, pode-se definir a classe Carro com os atributos placa, combustível, consumo e o comportamento representado através do método double getAutonomiaKm(). Sobre o questionamento de quem nasce primeiro um objeto ou uma classe,

ressalta-se que é necessário primeiro a definição da Classe para posteriormente, se efetuar a geração dos objetos com base na classe. Assim, entende-se também o conceito classe como uma fábrica de objetos do mesmo tipo.

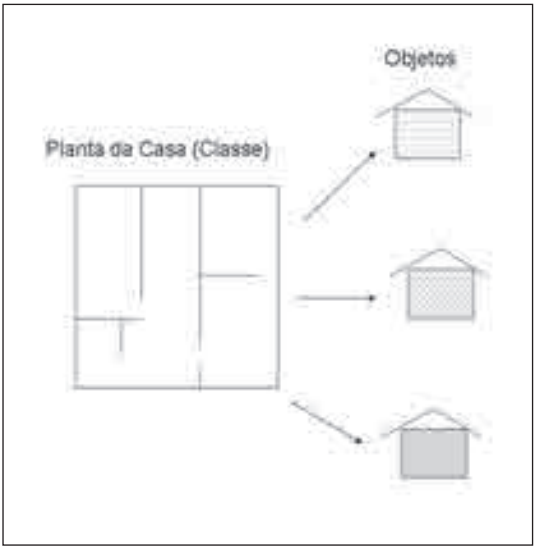


Figura 5 – Representação Lúdica para o Conceito de Classe.
Fonte: Autor

A criação de um objeto com base em uma Classe também é conhecida com a nomenclatura instância de Classe. Para ilustrar o conceito de classe pode-se fazer uma analogia com a planta de uma casa (ver figura acima). Com base nesta planta são construídas n casas (objetos) com as mesmas características básicas, diferenciando-as através do seu estado particular, por exemplo cada casa pode ter uma cor diferenciada.

SAIBA MAIS

Através da análise comparativa entre Classe e Objeto concluem-se os seguintes pontos:

- Classe é uma estrutura estática composta por atributos e métodos.
- Uma Classe tem como funcionalidade principal ser uma estrutura genérica para a construção de objetos, ou seja, uma fábrica de objetos.
- Um objeto é gerado sempre a partir de uma classe (instância de classe), herdando todas as suas características atributos e métodos, adicionando mais o estado.
- O estado do objeto é entendido como os valores assumidos ao longo do tempo pelos atributos, sendo utilizados como os dados para os métodos.

Fortemente relacionado com o Conceito Classe e Objeto temos o encapsulamento ou encapsulação que significa esconder informação. Objetiva-se com o encapsulamento proteger o estado e o comportamento interno do objeto, disponibilizando somente uma camada de interface com o ambiente externo para que seja efetuada a comunicação com o objeto, como ilustrado na figura abaixo.

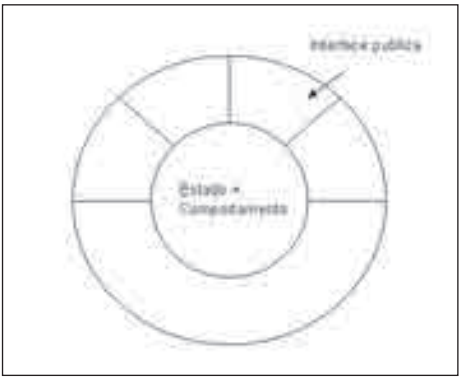


Figura 6 – Representação Gráfica de um Objeto.
Fonte: Autor

O reflexo do encapsulamento na Orientação a Objeto é que todo o objeto na sua estrutura elementar possui uma membrana de proteção ao longo do seu núcleo, conforme figura anterior. O acesso ao estado e comportamento só é permitido através da passagem pela camada de comunicação representada pela membrana.

Em programação somente a assinatura do método, ou seja, a chamada da função está publicada na interface. O conjunto da assinatura de todos os métodos também é conhecida como API (*Application Programming Interface*). Para um desenvolvedor utilizar objetos é necessário que ele conheça somente API dos mesmos. A complexidade de como um Objeto foi implementado está encapsulada, não sendo relevante neste processo.

Pode-se ilustrar esta situação através da seguinte análise: quando se dirige um carro não é necessário compreender como o motor funciona com todas as suas 'milhares' de peças. O motorista abstrai a complexidade interna do motor e fixa-se na interface disponibilizada por quem construiu o Carro, portanto as interfaces seriam os pedais de aceleração, freio e embreagem, painel com medidor de velocidade, etc.

A forma de garantir o encapsulamento de atributos e métodos é determinar a sua visibilidade privada através dos modificadores de acesso disponíveis nas linguagens de programação (private ou protected). Por exemplo, normalmente um atributo na linguagem java é declarado private String nome, sendo private uma palavra-chave indicando que a visibilidade do atributo nome só é permitida para métodos da própria classe.

Figura 7 – Abstração de um Objeto do Mundo Real.
Fonte: Autor

A definição básica sobre a abstração é supressão de detalhe, visando controlar a complexidade pela ênfase em características essenciais. Na Orientação Objeto pode-se aplicar este conceito no processo de especificação, construção e uso de classes.

Observa-se que os objetos identificados no mundo real normalmente são muito complexos. Por exemplo, imagine o que seria modelar uma classe Pessoa com todos os atributos e comportamentos! Para a redução da complexidade da Classe Pessoa é necessário delimitar o escopo com base no domínio do sistema que esta Pessoa será modelada, considerando somente os aspectos relevantes (ver figura acima).

Por exemplo, para um sistema acadêmico uma Pessoa pode representar um aluno, sendo nesse escopo somente necessário os atributos nome, matrícula e idade. As demais características que não são essenciais nesse contexto serão abstraídas. Como os detalhes ficam internos e escondidos no objeto Pessoa pode-se subentender que este objeto possui quaisquer características de uma Pessoa do mundo real.

A abstração também se aplica na construção de uma classe. Uma primeira etapa é definir toda a interface da classe, ou seja, como os

componentes externos irão interagir com a classe e qual o comportamento esperado pela mesma. O objetivo nesta etapa é somente de definir a interface (protocolo de comunicação), focando no que a classe deve fazer e não como a interface será implementada. Portanto, abstrai-se a implementação interna e concentra-se na especificação da interface que representa o protocolo de comunicação com o meio externo. Um bom critério de programação para a redução do acoplamento é programar para uma interface e não para uma implementação. Com a interface bem definida, pode-se abstrair o mundo externo e focar-se em como realizar o que foi especificado na interface.

2.3.3 Herança de Classe

Uma Classe pode representar genericamente uma ou mais Classes. Assim, Classes podem ser definidas a partir de uma Classe pai. O recurso de herança estabelece que uma Classe que herda de outra Classe possui todas as suas características (atributos e comportamentos).

O recurso de herança é semelhante a herança genética do mundo real. Logo, os filhos mantêm as características dos pais. Uma Classe pai, também denominada com superclasse, passa todas as suas características para a classe filha. A Classe filha ou subclasse, além de conter os atributos e comportamentos da superclasse, pode acrescentar algo especifico a sua funcionalidade ou reescrever métodos da superclasse. O termo *overridden* é utilizado quando a subclasse reescreve ou redefine um método herdado da superclasse.

Uma superclasse pode ser ainda definida como uma Classe concreta (como qualquer classe convencional) ou uma Classe Abstrata. Classes abstratas por definição não permite a criação de objetos, portanto não é possível a criação de instâncias a partir desta classe.

A definição de uma Classe como abstrata pode parecer algo estranho, visto que uma das funções principais de uma classe é a geração de objetos. A declaração de Classes abstratas é algo comum e deve ser utilizado quando não faz sentido a geração de instâncias da super-

classe. Por exemplo, em uma estrutura hierárquica com a superclasse Pessoa e as subclasses PessoaFisica e PessoaJuridica seria um erro a criação de instâncias a partir da classe Pessoa, já que existe uma disjunção entre as subclasses, ou seja, uma Pessoa ou é do tipo Física ou Jurídica. Na classe abstrata além de atributos e métodos, como uma classe concreta, têm-se os métodos abstratos. Métodos Abstratos não possuem implementação e são assinados na classe Abstrata. As classes que herdam da classe abstrata são obrigadas a implementar os métodos abstratos herdados da superclasse.

Associados ao conceito de herança, apresentam-se os conceitos de especialização e generalização. A especialização ocorre quando uma subclasse, além de herdar da superclasse, acrescenta características especificas diferentes da superclasse. Já o conceito de generalização ocorre quando existem Classes que possuem características em comum e cria-se uma Classe genérica, ou seja, uma superclasse que englobe essas características.

A figura abaixo apresenta duas subclasses PessoFisica e PessoaJuridica que possuem atributos e um método em comum. Neste caso aplica-se a generalização sobre os elementos em comum encapsulando-os em uma superclasse Pessoa que é uma abstração das subclasses. Os atributos e métodos particulares das subclasses PessoaFisica e PessoaJuridica especialização a classe Pessoa.

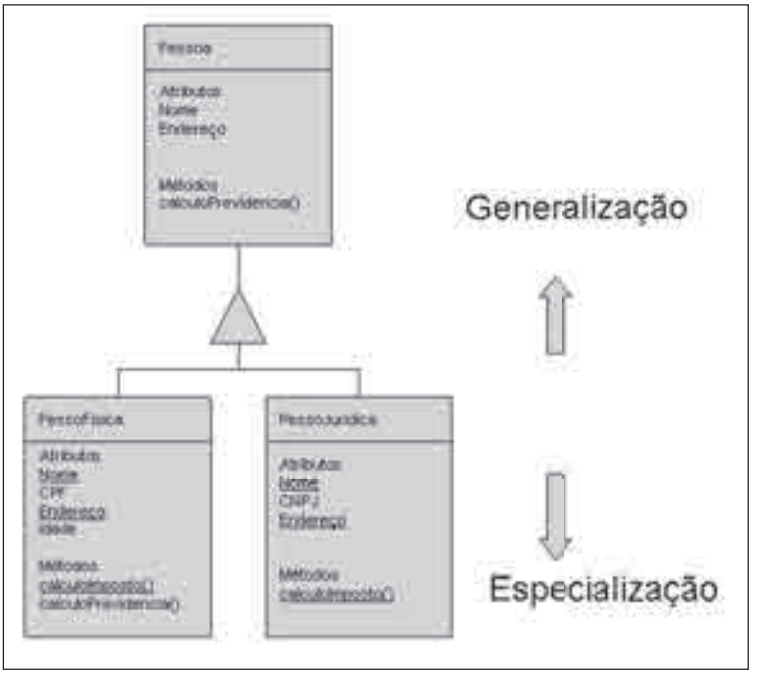


Figura 8 – Representação Gráfica de Especialização e Generalização.
Fonte: Autor

O uso de herança é um recurso poderoso, mas mal aplicado gera graves problemas nos projetos Orientados a Objetos. Muitos professores já apresentaram ou ainda apresentam a herança como o principal recurso da Orientação a Objetos e a forma ideal para alcançar a reutilização. Ao longo desta obra irá se demonstrar que Orientação a Objetos vai muito além da herança e que a reutilização pode ser alcançada através de outras formas (ver seção sobre Herança versus Composição).

2.3.4 Métodos, Método Construtor e Sobrecarga

A forma como os objetos colaboram em um sistema Orientado a Objetos é feito basicamente através da troca de mensagens. As mensagens são utilizadas para a comunicação entre objetos para a troca

de dados e realização de ações. Ações são implementadas através métodos que são publicados na camada de interface do objeto.

Em programação os métodos são funções que possuem um parâmetro de retorno ou não, sendo neste caso do tipo *void*. Um método possui também um nome e seus parâmetros, por exemplo, a figura abaixo apresenta o método public double getDistancia(int x, int y) enumerando os seus elementos estruturais. O conjunto nome mais parâmetros também é conhecido como a assinatura do método. Em uma classe não é possível a ocorrência de métodos com a mesma assinatura, porém podem existir métodos com o **mesmo nome, mas parâmetros diferentes**.

Figura 9 – Estrutura de Composição de um Método.
Fonte: Autor

O método construtor é um método especial que está presente em todas as Classes. Este método deve possuir o mesmo nome da classe, por exemplo, a classe Ponto poderia ter um método construtor public Ponto(). Além disso, diferente dos outros métodos, o método construtor não possui parâmetro de retorno, nem mesmo a declaração *void*.

Em uma classe podem existir vários métodos construtores, desde que possuam parâmetros diferentes. Outra característica importante é em relação a sua utilização. Quando um objeto nasce, ou seja, é instanciado a partir de uma Classe o primeiro método a ser executado é o método construtor. Como foi mencionado, uma classe obrigatoriamente tem pelo menos um método construtor.

O conceito sobrecarga ou *overloading* é definido como métodos em uma classe que possuem o mesmo nome, mas parâmetros diferentes.

Ressaltando que pode-se ter vários métodos com o mesmo nome, pois o que identifica um método é o seu nome associado com os seus parâmetros.

Portanto, para que em uma classe Ponto existe a ocorrência de sobrecarga de construtores devemos ter, por exemplo os métodos *public Ponto()* e *public Ponto(int x, int y)*. Observa-se que a ocorrência sobrecarga existe em métodos construtores ou não.

2.3.5 Polimorfismo

Objetos no mundo real que estão sobre uma mesma hierarquia podem possuir comportamentos definidos pelos seus pais, mas realizá-los de forma particular. Por exemplo, uma superclasse FiguraGeométrica com um método public double calculoArea(). Notoriamente esta superclasse não tem como responder a este método, pois a função desta classe é ser um conceito genérico. Logo, é uma classe que tem o objetivo de definir como uma FiguraGeométrica deve funcionar genericamente. O objetivo desta classe é definir que toda a FiguraGeométrica tem o comportamento de calculo de área. Já subclasses Quadrado e Triângulo vão possuir o método de calculo de área, sendo que a implementação para cada Classe será feita de forma particular.

A importância de definir um conceito padronizado é a facilidade de absorver futuras mudanças. Por exemplo, uma nova classe Retângulo que herda de FiguraGeométrica tem também o método public double calculoArea(). Apesar de o método possuir a mesma assinatura a sua realização seguira as características da classe Retângulo. Conclui-se, que um comportamento pode ter várias formas de ser respondido a depender do tipo de objeto, este recursos é definido como Polimórfico.

A definição para o conceito Polimorfismo estabelece que para um método genérico pode-se especializá-lo e defini-lo em classes de tipos diferentes que responderão a este método de forma particular. O método genérico deve ser assinado em uma superclasse, determi-

nando um padrão para as classes que irão herdar suas características. Diz-se assinado, pois o método da superclasse pode ser um método concreto ou abstrato, normalmente é um método abstrato.

A vantagem do Polimorfismo reflete-se no baixo acoplamento no projeto. Um objeto pode estar acoplado a um outro objeto sem necessariamente saber o seu tipo. Por exemplo, um objeto do tipo Retângulo pode ser manipulado como uma FiguraGeometrica. Caso o programa seja modificado para utilização de um Triangulo ao invés de um Retângulo o impacto com as mudanças no programa serão mínimas.

2.3.6 Coesão

Para garantir a qualidade do projeto e código de um programa Orientado a Objetos não basta apenas adotar conceitos tais como: Classe, Herança e Polimorfismo. É preciso garantir que cada objeto esteja assumindo o papel adequado e contribuindo da melhor forma para a finalidade do domínio do problema. Para tanto, os componentes desta unidade devem colaborar da melhor forma.

Todo objeto no sistema deve ter uma finalidade bem definida. Representar um Cliente, um Produto ou uma Empresa são exemplos de responsabilidades que um objeto pode carregar. Definimos por exemplo que um objeto Cliente deve ter atributos como o nome, cpf, endereco e limiteDeCRedito.

A coesão avalia o quanto bem definido está a finalidade de um objeto e o quanto cada parte deste objeto colabora para atingi-la. Uma alta coesão indica que o objeto possui uma finalidade bem definida e os elementos deste objeto colaboram para tanto. Em contra partida, uma baixa coesão indica que o objeto tem finalidades mal definidas ou em excesso ou ainda que existam partes do objeto que não colaboram da melhor forma.

Para demonstrar o que é coesão, sua importância e como atingir um alto nível, demonstra-se exemplos que desrespeitam a coesão, fazendo analogia a uma pequena empresa, onde os funcionários represen-

tam nossos objetos. Caso um funcionário trabalhe com ferramentas inadequadas, uma postura incorreta, um ambiente em que não há uma comunicação coerente entre seus colaboradores, certamente não desempenhará corretamente suas funções. Suas partes não estão colaborando com a finalidade do objeto funcionário, que por sua vez não irá conseguir colaborar plenamente com a finalidade da empresa (sistema). Caso suas partes (ferramentas, postura e ambiente) estejam colaborando perfeitamente, contudo sua função não seja bem definida, certamente este funcionário irá representar um desperdício de recursos e de tempo. Ele poderá ficar ocioso ou desempenhando uma função que lhe seja incompatível. Se atribuirmos demasiadas funções, ocorrerá que ele ficará sobrecarregado, e não conseguirá desempenhar todas as funções perfeitamente. Desta forma não irá colaborar plenamente com a finalidade da empresa. Nestes dois casos estaríamos faltando com a coesão.

Devemos observar também que um altíssimo grau de coesão também é prejudicial para o projeto. Imagine se tivéssemos um funcionário para cada micro-função dentro da empresa. Seria um gasto desnecessário e subestimaríamos a capacidade destes funcionários. Portanto devemos ponderar na mediada da coesão, garantindo que cada objeto colabore com a finalidade do sistema mantendo sua própria finalidade bem definida, porém sem comprometer o desenvolvimento do projeto. É interessante também sempre deixar explícito (através de comentários e/ou documentação) os objetivos do objeto, principalmente quando este assumir responsabilidades complexas.

2.4 Exercícios do Capítulo

1. Defina Objeto, explicando estado, comportamento e ciclo de vida.

2) Defina Classe, fazendo uma correlação com Objeto.

3) Explique os seguintes conceitos:
   a) Encapsulamento
   b) Sobrecarga
   c) Método Construtor
   d) Polimorfismo

4) Sobre herança de classe explique os conceitos:
   a) Classe Abstrata e Método Abstrato
   b) Generalização/Especialização
   c) Subclasse x Superclasse
   d) Overridden (reescrever)

5) Qual a importância na definição primeiramente da interface da classe para depois efetuar a sua realização, ou seja, implementação do conteúdo interno da classe?

6) Qual a relação entre abstração e encapsulamento?

7) No projeto de uma classe o que deve ser especificado e em que ordem?

8) Qual o objetivo, no projeto de uma classe, de defini-la como abstrata? E como seria o uso desta classe no contexto do programa OO?

9) Como você identificaria se um método é polimórfico?

10) O que define a assinatura de um método? E em que situação acontece a sobrecarga de métodos?

11) Explique como os objetos colaboram em um programa OO.

12) Sobre abstração indique o V ou F. Caso F explique.
    ( ) É uma técnica para controlar a complexidade pela ênfase em características essenciais e pela supressão de detalhes.
    ( ) Para construção de modelos é importante desconsiderar alguns aspectos do mundo real, visando controlar a complexidade.
    ( ) Com a abstração vamos modelar somente o que é relevante para o contexto do sistema.
    ( ) Por exemplo, como os detalhes ficam internos e escondidos no objeto Pessoa podemos entender que este objeto pode possuir qualquer característica de uma Pessoa do mundo real.
    ( ) No projeto de uma Classe não existe a necessidade do uso da técnica de abstração. Assim, devemos projetá-la com os seus atributos, métodos e interface.

13) Quais as consequências de um projeto que possui uma boa delegação de responsabilidades para suas unidades e seus objetos?

APLICANDO OS CONCEITOS DE OBJETO, CLASSE, ENCAPSULAMENTO E SOBRECARGA NA LINGUAGEM JAVA

3.1 Introdução

Esta seção apresenta três exemplos simplificados na linguagem Java com o objetivo de ilustrar uma visão prática dos conceitos teóricos de OO. Portanto, este é o nosso primeiro contato com a linguagem Java.

Antes de iniciarmos a apresentação dos exemplos, determinaremos alguns padrões de nomenclatura apresentados na tabela a seguir para nome de classe, atributos, métodos, variáveis locais e constantes.

INDICAÇÃO DE LEITURA

O padrão aqui utilizado baseia-se na forma convencional de escrita de classes Java utilizada pela Oracle. Observa-se que este padrão é amplamente utilizado por desenvolvedores ao redor do mundo.
Já para entender mais sobre o padrão de codificação acesse (http://www.oracle.com/technetwork/java/codeconventions-150003.pdf).

A definição de um padrão de nomenclatura é útil, pois ajuda nas futuras manutenções do código, normalmente efetuadas por pessoas diferentes. Lembre-se que o foco dos conceitos apresentados aqui não se resumem a construção de um bom algoritmo, mas o de criar um projeto robusto de fácil manutenção.

Somado a isto, o padrão de nomenclatura ajudará aos programadores iniciantes a compreenderem melhor as estruturas de programação. Como Java é uma linguagem *case-sensitive* muitos erros são

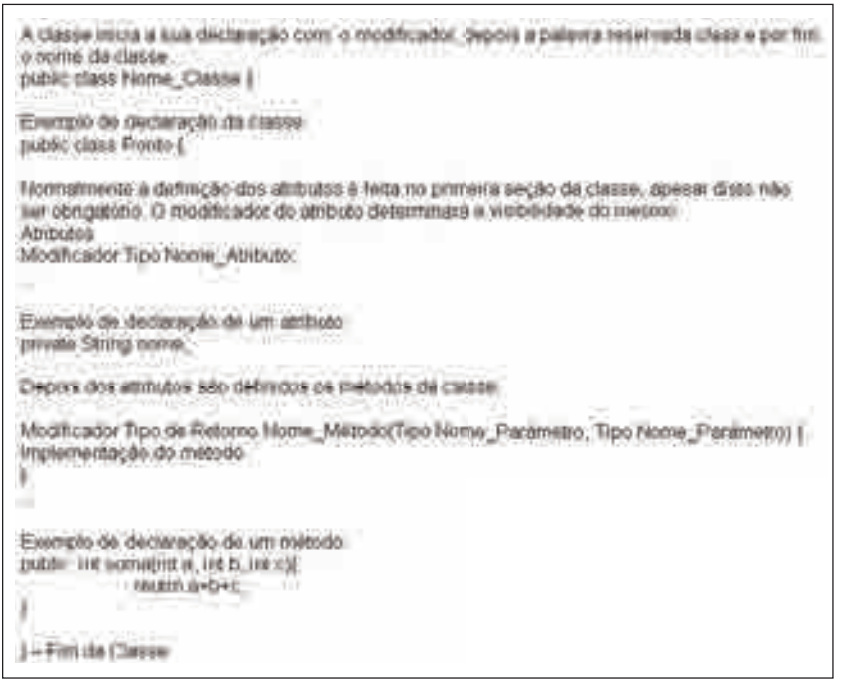
gerados, por quem está iniciando nesta área, somente pela falta da adoção de um padrão de nomenclatura.

Agora que já conhecemos o padrão de escrita de um programa em Java vamos iniciar os nossos estudos de introdutórios de programação. Como dito anteriormente, antes da instanciação dos objetos deve-se criar a classe. Uma classe Java é escrita em um arquivo fonte com o nome da classe e a extensão ".java", por exemplo a classe de nome Ponto será escrita em um arquivo Ponto.java.

Estruturas da Linguagem	**Padrão de Nomenclatura (Regra geral é não colocar acento ou caracteres especiais nos nomes)**
Nome da Classe	Primeira letra maiúscula o resto do nome em Minúscula. Caso seja a composição de dois nomes aplicar a regra anterior para o Segundo nome. Exemplo: Pessoa, PessoaFisica, PessoaJuridica
Atributos e variáveis	A primeira parte do nome da variável deve ser em minúsculo. A segunda parte do nome, caso ocorra, deve possui a primeira letra maiúscula e o resto do nome em minúscula. Exemplo: Nome, preco, qtdLitros
Constantes	Todas as letras do nome em maiúscula. Exemplo: ORDEM_NOME, ABERTO, VENCIDO
Métodos	A primeira parte do nome do método deve ser em minúsculo. A segunda parte do nome, caso ocorra, deve possui a primeira letra maiúscula e o resto do nome em minúscula. Uso do prefixo get e set para indicar métodos de recuperação de valores ou definição de valores respectivamente. Exemplo: getNome(), setNome(String nome), listaProduto()

Tabela 9 – Padrões de nomenclatura.
Fonte: Autor

Toda a classe em Java segue um padrão de formação na sua escrita. No código exemplo abaixo, de forma simplificada, apresenta-se o padrão para a definição de uma classe em Java.



Código 11 – Padrão de definição de uma classe em Java.
Fonte: Autor

Nas próximas seções apresentaremos três exemplos em Java para você entender como os conceitos de OO se aplicam nesta linguagem. Remoendo que você faça os exemplos em um computador para entender melhor o funcionamento e possa alterar trechos de código para ter suas próprias experiências. Um questão importante também é sabre ler os erros gerados pelo compilador do Java. Isto irá ajudar na solução da maioria dos problemas.

3.2 Exemplo 1: Primeiro Programa OO

Tem-se as classes Ponto e DoisPontos. A classe Ponto (utilizada como primeiro exemplo) é uma forma simplificada de escrita de uma classe, pois só possui atributos. Normalmente, classes possuem atributos e métodos. Na linha 5 do Código 12 da classe Ponto são declarados os atributos x e y, também denominados como variáveis de instância. A classe DoisPontos, também possui uma estrutura simplificada, visto que só será utilizada para criação de objetos. Está classe não possui atributos e nem métodos de instancia. O método public static void main(String args[ ]) é um método estático que será iniciado quando a classe DoisPontos for executada (ver linha 3 do Código 13). Os métodos estáticos são métodos que podem ser executados como funções da classe sem a necessidade de criação de instancias. O método *main* é um método que existe em todas as classes que se deseja rodar o programa. Esta possibilidade pode gerar algumas dúvidas sobre qual classe deve representar o programa principal. Assim, deve-se definir uma classe que possui um método main, semelhante à classe DoisPontos, que representa o programa principal. As outras classes que possuírem métodos *main* servirão como métodos de testes.

```
1 // Exemplo 1 (Classe Ponto)
2 // Início de Arquivo
3
4 public class Ponto {
5     int x,y;
6 }
```

Código 12 – Listagem do Código da Classe Ponto.
Fonte: Autor

As linhas 4 e 5 na classe DoisPontos (Código 13) utilizam o operador new para instanciação de dois objetos Ponto. Como a classe Ponto é uma fabrica de objetos pode-se ter a criação de n instâncias a partir desta classe. As variáveis p1 e p2 são do tipo Ponto, em progra-

mação Orientada a Objetos as classes são novos tipos definidos pelo desenvolvedor. A palavra new Ponto() indica a criação de um objeto da classe Ponto na memória.

```
1  public class DoisPontos {
2
3  public void main(String args[]){
4      Ponto p1 = new Ponto();
5      Ponto p2 = new Ponto();
6      p1.x=10; p1.y=20; p2.x=43; p2.y=78;//Não desejável
7      System.out.println("Ponto 1-> x=" + p1.x + "y=" + p1.y);// Imprime
8      System.out.println("Ponto 2-> x=" + p2.x + "y=" + p2.y);// Imprime
9  }
10 }
```

Código 13 – Listagem do Código classe DoisPontos.
Fonte: Autor

As variáveis p1 e p2 são do tipo Ponto porque apontam para os endereços de memória que estão os objetos Pontos instanciados. As variáveis p1 e p2 podem acessar atributos e métodos públicos através do operador ponto (“.”), por exemplo p1.x. O acesso direto de atributos é algo pouco recomendado, visto que fere a propriedade do encapsulamento (linha 6 do código acima). O próximo exemplo ilustra como corrigir o problema de desrespeitar ou “furar” o encapsulamento. Um detalhe ainda neste exemplo é que em Java pode-se concatenar automaticamente um valor int com uma string (ver linha 7 do Código acima).

3.3 Exemplo 2: Evoluindo o Exemplo 1 Aplicando os Conceitos de Encapsulamento

Este exemplo é um aprimoramento do exemplo 1. O problema de desrespeitar o encapsulamento, identificado no exemplo1, é corrigido através da colocação do modificador privado na declaração dos atributos x e y (linha 3 da classe Ponto do Código 14). Um teste a ser feito, para verificar o bloqueio do acesso direto aos atributos, tentar executar a classe DoisPontos do exemplo anterior com a classe

Ponto deste exemplo. Observe que não se consegue compilar a classe DoisPontos, pois é gerado um erro em tempo de compilação.

```
1  package exemplo2;
2  public class Ponto {
3      private int x,y;//Dados encapsulados
4  public void setX(int x){
5      this.x=x;//this é um valor de referencia, que refere-se ao objeto corrente
6  }
7  public int getX(){
8      return this.x;
9  }
10 public void setY(int y){
11     this.y=y;
12 }
13 public int getY(){
14     return this.y;
15 }
16 }
```

Código 14 – Listagem do Código da Classe Ponto com Get/Set.
Fonte: Autor

A principal diferença entre os dois exemplos é que neste do Código 14 o acesso aos atributos da classe Ponto serão feitos através de métodos de acesso *get/set*. Outro detalhe diferente é a ocorrência da palavra reservada *this* (ver linha 5 da classe Ponto atual). A palavra *this* é utilizada internamente na classe para fazer referência a atributos e métodos da classe. Os atributos da classe são variáveis com escopo de visibilidade em qualquer método da classe que não seja estático. Assim, as variáveis que representam os atributos podem ser utilizadas (obtenção e alteração dos seus valores) em qualquer método.

Um erro comum entre os programadores iniciantes é querem utilizar *this* no método *main*. Este é um erro grave, pois o método *main* não é um método de instância e sim uma função da classe (um método estático – static), como mencionado no exemplo anterior.

3.4 Exemplo 3: Exemplificando Método Construtor e Sobrecarga

Este exemplo apresenta dois métodos importantes: o método construtor e o método toString. O método construtor é um método es-

pecial e diferente, pois não possui nenhum retorno (nem mesmo *void*). O método construtor deve ter o mesmo nome da classe, e será o primeiro método a ser chamado quando o objeto for instanciado (ver linha 24 no próximo Código).

```
1  package exemplo3;
2  public class Ponto {
3      private int x,y;//Dados encapsulados
4      public Ponto(int x_, int y_){
5          this.x=x_;
6          this.y=y_;
7      }
8      public Ponto(int x_){
9          this(x_,40);//O atributo y é fixo em 40. Esta linha fará a chamada ao construtor
10                     //que recebe dois parametros.
11     }
12     public Ponto(){
13         this(-1);//O atributo x e y são fixos em -1 e 40 respectivamente.
14     }
15     public String toString(){
16         return "Ponto X=" + this.x + "Y=" +this.y;
17     }
18     public static void main(String args[]){
19         Ponto p=new Ponto (10,20);
20         System.out.println(p);
21         Ponto p1=new Ponto (10);
22         System.out.println(p1);
23         Ponto p2=new Ponto ();
24         System.out.println(p2);
25     }
26 }
```

Código 15 – Listagem do Código da Classe Ponto – Construtores Distintos.
Fonte: Autor

Em uma classe podem ser definidos *n* métodos construtores, contanto que estes possuam parâmetros diferentes (conceito de sobrecarga) ver as linhas 19 até a 23 do código acima. Em Java caso não seja definido um método construtor por padrão tem-se um construtor vazio (situação apresentada no dois exemplos anteriores). A palavra reservada *this*, apresentada no exemplo anterior, está sendo usada para a reutilização e a chamada de um método construtor por outro método construtor (ver as linhas 4 até 14 do código acima). O método toString é um método padronizado em java para informar normalmente o estado do objeto. Assim quando se imprime a referência de um objeto automaticamente o método toString é chamado (ver linha 15 do código acima). O comando System.out.println(p) é o mesmo que System.out.println(p.toString()).(faça o teste)

O método toString() e outros métodos como *equals* estão definidos em uma classe Object. Esta classe é a classe pai de todas as classes em Java. Assim, qualquer classe que seja criada em Java por padrão herda da classe Object (sem a necessidade de nenhuma diretiva como *extends* que veremos no próximo exemplo). Um teste a ser feito para a comprovação disto é comentar o método toString() da classe Ponto e executa – lá. Observe que o programa funciona apesar da impressão ser algo inesperado. O método toString() da classe Object, herdado pela classe Ponto, redefiniu o método toString(), mantendo a mesma assinatura, mas mudando a sua implementação (overridden).

3.5 Herança, Polimorfismo e *Cast*

Nesta seção vamos abordar sobre o uso da Herança, Polimorfismo e *Cast*. Estes são conceitos poderosos favorecem a abstração e se complementam. A seguir explicamos e exemplificamos cada um deles.

3.5.1 Exemplo 4: Exemplificando Herança

Este exemplo mostra como pode ser implementado o conceito de herança em Java. No Código a seguir é exibida a classe Ponto3D que é uma especialização da classe-pai (superclasse) Ponto (Código 15). Na linha 2 do Código abaixo nota-se a presença da palavra reservada *extends* do Java para indicar que a classe Ponto3D herda da classe Ponto.

Dessa forma, um objeto do tipo Ponto3D tem todas as características da classe Ponto, e ainda sua especialização, que corresponde a 3ª dimensão (Z). Pode ser observado ainda que na linha 6 tem-se a palavra, *super*, reservada do Java, utilizada para fazer referência à super classe que no exemplo super(x,y) está fazendo referência ao construtor da classe Ponto(int x_,int y_).

```
1  package exemplo5;
2  public class Ponto3D extends Ponto {
3      int z;// Dados encapsulados
4
5      Ponto3D(int x, int y,int z) {
6          super(x,y);
7          this.z=z;
8      }
9
10
11     void setZ(int z){
12         this.z = z;// this é um valor de referênci
13         //que refere-se ao objeto corrente
14     }
15
16     int getZ(){
17         return this.z;
18     }
19
20
21     public String toString(){
22         return "Ponto3D X=" + this.getX() +
23                 " Y="+this.getY()+" Z="+this.getZ()
24     }
25 }
```

Código 16 – Listagem do Código da Classe Ponto3D.
Fonte: Autor

O Código abaixo mostra um exemplo de como utilizar a estrutura de herança definida para Ponto e Ponto3D. Na linha 6 da é definido um objeto do tipo Ponto e logo em seguida na linha 7, o objeto do tipo Ponto é instanciado do tipo da sub-classe Ponto3D. Nesse caso para acessar métodos que existem na super-classe deverão ser invocados normalmente como é feito na linha 9 – getX(). Entretanto, caso se deseje acessar atributos da sub-classe Ponto3D deverá ser feito uma conversão explícita de objetos que denomina-se de *Cast*, para a classe filha Ponto3D, como mostrado na linha 10.

```
package exemplo5;

public class Principal {

    public static void main(String args[ ]){
        Ponto p;
        p = new Ponto3D(10, 20,30);
        System.out.println(p);
        System.out.println(p.getX());
        System.out.println("Z ="+ ( (Ponto3D) p).getZ() );//cast

    }
}
```

Código 17 – Listagem do Código da Classe Principal.
Fonte: Autor

3.5.2 Exemplo 5: Exemplificando Herança – 2

O Código abaixo apresenta a classe Horario que corresponde à abstração do horário em um determinado ponto na linha do tempo. A classe Horario possui o método construtor public Horario(int hora,int minuto) e também o método public int tempoTranscorrido() que retorna o tempo em minutos.

```
package exemplo5;
public class Horario{
        protected int hora,minuto;

    public Horario(int hora, int minuto){
        this.hora = hora;
        this.minuto = minuto;
    }

    public int getHora(){
        return this.hora;
    }

    public int getMinuto(){
        return this.minuto;
    }

    public int tempoTranscorrido(){
        return this.hora*60 + this.minuto;
    }
}
```

Código 18 – Listagem do Código da Classe Horário.
Fonte: Autor

Abaixo temos um exemplo de herança, pois a classe HorarioS estende de Horário, herdando atributos e comportamentos da classe Horário. Na classe HororioS e incluindo os seus próprios, podendo, como já foi abordado, reescrever métodos da super classe, *overridden*. O Código 19 mostra que a subclasse HorarioS reescreve o método public int tempoTranscorrido() da superclasse, Horário, para retornar o tempo em segundos.

Na linha 5, encontra-se a invocação do construtor da superclasse, como já abordado *super* é uma palavra-chave utilizada para invocar explicitamente atributos (esquecendo-se o encapsulamento) ou métodos da superclasse.

Salientando que torna-se necessário, para invocar o construtor do super classe que esse comando seja o primeiro do construtor da subclasse. Na linha 10 temos outro exemplo da utilização desta palavra-chave.

```
1  package exemplo8;
2  public class HorarioS extends Horario {
3          private int segundo;
4      public HorarioS(int newHora, int newMinuto, int newSegundo){
5          super(newHora,newMinuto);
6          this.segundo = newSegundo;
7      }
8
9      public int tempoTranscorrido(){
10         return super.hora*60*60 + super.minuto*60 + this.segundo;
11     }
12
13
14 }
```

Código 19 – Listagem do Código da Classe HorárioS.
Fonte: Autor

No Código 20 nota-se um exemplo de como utilizar a herança definida entre a classe Horario e a classe HorarioS. Na linha 6 e 7 são definidos duas variáveis h1 e h2 do tipo Horario e instanciadas. Na linha 8 são invocados os métodos tempo transcorridos dos dois objetos, h1 e h2, e calculada a diferença em minutos entre eles. Nas

linhas 11 e 12 são definidos dois objetos do tipo Horario, h3 e h4, entretanto instanciado a partir da subclasse HorarioS, o que é possível pois HorarioS é uma subclasse de Horario. Sendo feito assim (sem uso explicito) ao ser invocado métodos nesses objetos serão acessados os métodos que existem na classe Horario, para o caso de overridden, especialmente, será invocado o método da subclasse, devido a instancia do objeto. Já para se ter acesso a um método especifico da subclasse torna-se necessário fazer um explicito para o tipo HorarioS, no exemplo para se invocar os segundos definidos em HorarioS será necessário invocar o método getSegundos().

```
1  package exemplo6;
2
3  public class Principal {
4
5      public static void main(String args[ ]){
6          Horario h1 = new Horario(12,30);
7          Horario h2 = new Horario(11,40);
8          int diference = h1.tempoTranscorrido() - h2.tempoTranscorrido();
9          System.out.println(diferenca);
10
11         Horario h3 = new HorarioS(12,30,30);
12         Horario h4 = new HorarioS(11,40,30);
13         diference = h3.tempoTranscorrido() - h4.tempoTranscorrido();
14         System.out.println(diferenca);
15     }
16 }
```

Código 20 – Listagem do Código da Classe Principal – Herança entre Horário e Horários.
Fonte: Autor

3.5.3 Exemplo 6: Exemplificando Classe Abstrata e Polimorfismo

Neste exemplo é mostrado herança utilizando classe abstrata e polimorfismo. Abaixo temos a classe abstrata Letra, que será a super classe do exemplo. Percebe-se que ela possui o método abstract String Imprime() que devera ser, obrigatoriamente, implementado pelas subclasses. A linha 6, apresenta a assinatura do método estático (permite ser acessado sem instanciar um objeto do tipo da classe, o que é impossível instanciar para o exemplo fornecido pois é uma classe

abstrata) imprimeLetra que recebe como parâmetro um objeto do tipo Letra e imprime no console o seu conteúdo.

```
1 package exemplo6;
2 public abstract class Letra {
3
4     abstract String Imprime();
5
6     public static void imprimeLetra(Letra l){
7         System.out.println(l.Imprime());
8     }
9 }
```

Código 21 – Listagem do Código da Classe Letra.
Fonte: Autor

No Código a seguir tem-se a sub-classe Consoante que herda da classe Letra. Como já dito toda subclasse de Letra tem que implementar o método abstrato Imprime(), o que é feito no Código mencionado.

```
1 package exemplo6;
2 public class Consoante extends Letra{
3
4     public String Imprime(){
5         return "Consoante";
6     }
7
8 }
```

Código 22 – Listagem do Código Consoante.
Fonte: Autor

No próximo Código tem-se a subclasse Vogal que também implementa o método Imprime().

```
1 package exemplo6;
2 public class Vogal extends Letra{
3
4     public String Imprime(){
5         return "Vogal";
6     }
7
8 }
```

Código 23 – Listagem do Código da Classe Vogal.
Fonte: Autor

No exemplo a seguir, nas linha 7 e 8 são instanciados (criados) dois objetos , c e v, como Consoante e Vogal respectivamente. Na linha 10 e 11 é invocado o método estático imprimeLetra da classe Letra passando como parâmetro os objetos instanciados. Na linha 10 será retornado "Consoante" e na linha 11 será retornado "Vogal". Que corresponde a um exemplo de polimorfismo que a depender do objeto será invocado o mesmo método que retornara um valor diferente a dependente do objeto passado como parâmetro.

```
1  package exemplo6;
2
3  public class Principal {
4
5      public static void main(String args[ ]){
6
7          Letra c = new Consoante();
8          Letra v = new Vogal();
9
10         Letra.imprimeLetra(c);
11         Letra.imprimeLetra(v);
12
13
14     }
15 }
```

Código 24 – Listagem do Código da Classe Principal – Herança Letra.
Fonte: Autor

3.6 Exercício do Capítulo

1) Indique a sequência de etapas que você seguiria para a especificação e construção de um programa orientado a objetos até a sua execução.

2) Indique as falhas na implementação da classe Ponto escrita a baixo.

```
1    public abstract class Ponto {
2        public int x, y;
3
4        public Ponto(int x, int y) {
5            this.x = x;
6            this.y = y;
7        }
8        public Ponto() {
9            super(1, 3);
10       }
11       public String toString() {
12           return "Ponto X= " + this.x + " Y= " + this.y;
13       }
14       public static main(String args[]) {
15           this.x = 100;
16           this.y = 200;
17           Ponto p = new Ponto(20, 30, 10);

     2   18       }
19   }
```

3) Considere a classe a seguir cuja instância (objeto) representa um Login(nome, senha). Seu método verificaLogin retorna se um nome e senha passado como parâmetro são iguais ao estado do objeto.

```
1   public class Login {
2       protected String nome;
3       protected String senha;
4       public Login(String nome, String senha) {
5           this.nome = nome;
6           this.senha = senha;
7       }
8       public String getNome() {
9           return this.nome;
10      }
11      public String getSenha() {
12          return this.senha;
13      }
14      public boolean verificaLogin(String nome, String
senha) {
17      return ((this.nome.equals(nome)&&this.senha.
        equals(senha)) ?
18              true : false);
19      }
20  }
```

Baseando-se na classe a cima implemente um método *main* que crie uma instância da classe Login com o seguinte estado nome = eduardo e senha = 123. Em seguida imprima o valor de retorno do

método verificaLogin com os parâmetros nome = carlos e senha =123.

OBSERVAÇÃO

comando para impressão System.out.println();

4) Crie uma classe Pessoa com os atributos *String nome* e *int idade*, métodos *Get* para os seus atributos, o método construtor *public Pessoa(String nome_, int idade_)*, um segundo método construtor *public Pessoa(String nome_)* que reutilize o método construtor anterior fixando o valor da idade em 20 e implemente também o método *public String toString()*. Em seguida crie um método *main*, instancie dois objetos Pessoa, utilizando o primeiro e o segundo método construtor, e imprima seus valores, para o primeiro objeto utilize seu toString implicitamente, para o segundo utilize os métodos *get*.

VANTAGENS E DIFERENÇAS ENTRE A PROGRAMAÇÃO FUNCIONAL E A PROGRAMAÇÃO ORIENTADA A OBJETOS

4.1 Introdução

A programação F. Estruturada e a programação Orientada a Objetos estão em paradigmas bem diferentes. Por isso, os primeiros passos na Orientação a Objetos sejam, às vezes, tão complicados. Na programação F. Estruturada o desenvolvedor possui uma liberdade maior na escrita do seu código, sendo mais fácil à criação de algoritmos pontuais. Por exemplo, a construção de algoritmos de ordenação ou que implementam funções matemáticas. Os programas no paradigma F. Estruturada possuem um fluxo principal que é organizado de forma modular através de um conjunto de funções que podem ser chamadas ao longo do fluxo ou por outras funções.

A programação Orientada a Objetos não foi projetada para substituir a programação F. Estruturada. Essas duas formas já convivem por muitos anos e certamente ainda coexistiram ao longo do tempo. Muitos dos conceitos apresentados na programação F. Estruturada são aplicados na programação Orientada a Objetos. Toda a estrutura de linguagem como os comandos de *loop* (*while, for*), condicionais (*if*, operações condicionais), variáveis locais entre outros são mantidos na programação OO. Funções e a técnica de modularização características marcante da programação F. Estruturada também estão presentes na programação Orientada a Objetos, agora com um novo enfoque. Assim, pode-se afirmar que a programação Orientada a Objetos é na verdade uma evolução da programação F. Estruturada, visando solucionar problemas mais complexos.

Muitos projetos de software foram e continuaram sendo construídos na programação F. Estruturada. Todavia, existem algumas dificuldades notórias neste modelo. Uma das principais dificuldades

encontra-se no contexto da modelagem de sistemas. A modelagem do mundo real e a transformação deste modelo em um programa de computador no paradigma F. Estruturado não é um processo intuitivo, além de não existir um esquema formal que apoie esse processo. Observa-se que a notação para construção de um projeto de software no paradigma F. Estruturado não tem uma correlação direta entre os conceitos modelados e os algoritmos a serem construídos. É importante ressaltar que o mundo real é composto por entidade e não por funções. Além da problemática supracitada, questões como a baixa reutilização de código (veremos um exemplo prático sobres esta questão), dificuldade na delegação semântica das responsabilidades dentro do programa, ou seja, qual parte do programa deve ser responsável por conter determinada função e códigos muitas vezes com alto grau de acoplamento estão presentes na programação F. Estruturada. Conclui-se, então a existência de problemas tanto na modelagem como na construção de projetos de software e dificuldades na manutenção, seja esta evolutiva ou corretiva. Não existe uma forma de fazer uma correlação direta entre a Programação F. Estruturada e a Programação Orientada a Objetos. Subtendendo a aproximação dos conceitos, verifica-se que os dados utilizados nas funções podem representados pelo estado do objeto e que as funções são os métodos do objeto (ver Figura abaixo). Ressalta-se que a principal diferença entre esses dois paradigmas consiste no aspecto que na programação F. Estruturada os dados e funções não estão ligados diretamente, dificultado a reutilização das funções e semântica da modelagem do mundo real. Já na orientação a objetos os dados e funções estão estritamente relacionados, sendo conjugados através de um módulo semântico representado pelo objeto. Outro ponto importante a ser ressaltado é que os dados referentes ao objeto estão armazenados no próprio objeto, mantendo o seu estado ao longo do seu ciclo de vida.

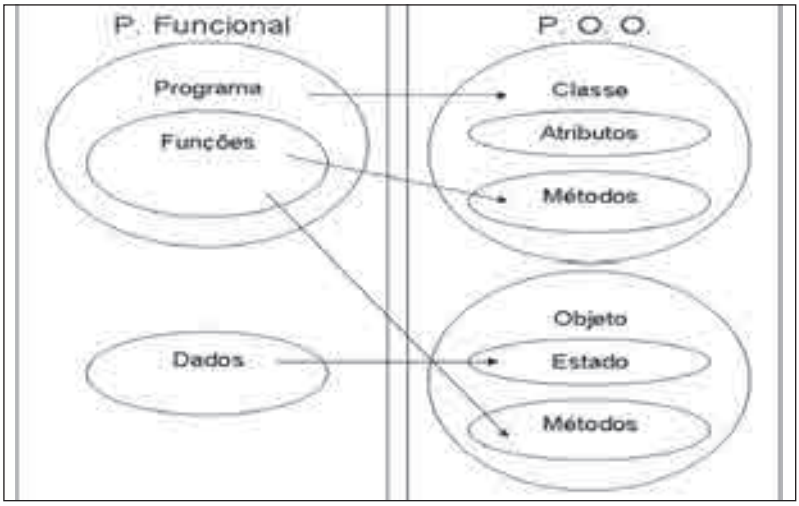


Figura 10 – Correlação entre a Programação F. Estruturada e a Programação Orientada a Objetos.
Fonte: Autor

4.2 Benefícios Alcançados com a OO: Acoplamento e Complexidade, Quanto Menor Melhor

A meta de um desenvolvedor no projeto e implementação de um software deve ser sempre obter o máximo de redução de acoplamento e complexidade. É dito redução, pois um software sempre terá algum nível de acoplamento visto os objetos estão relacionados e trocam mensagens entre si. Um projeto de software Orientado a Objeto que respeita a propriedade do encapsulamento, mesmo não usando nenhuma outra técnica (como padrões de projeto que será abordado adiante) já está em um bom caminho para alcançar a redução de acoplamento e complexidade.

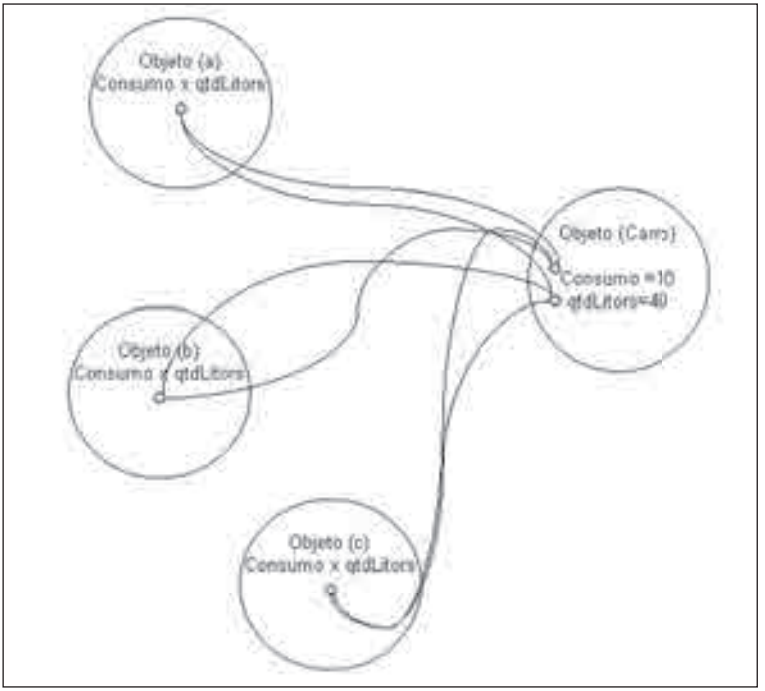


Figura 11 – Exemplo de implementação como forte acoplamento.
Fonte: Autor

Portanto, uma vantagem básica da encapsulação é facilitar mudanças isolando pedaços uns dos outros (reduzindo o acoplamento). Para melhor entendimento de como o encapsulamento pode beneficiar o redução do acoplamento ilustra-se com um exemplo prático.

No contexto do calculo da autonomia de um carro supondo que exista uma regra de negócio mapeada pela formula R(1) = (consumo * qtdLitros). Neste exemplo têm-se três objetos a, b e c que necessitam da informação da autonomia. Para isso, esses três objetos acessam um quarto objeto do tipo Carro, obtendo seus dados de consumo e qtdLitros. Como ilustrado na figura anterior, o algoritmo para o calculo da autonomia está implementado nos objetos a, b, e c. Este exemplo está totalmente desalinhado como os conceitos da Orientação a Objetos, pois desrespeitando o encapsulamento, visto

que o calculo da autonomia é um comportamento que deve estar encapsulado no objeto Carro.

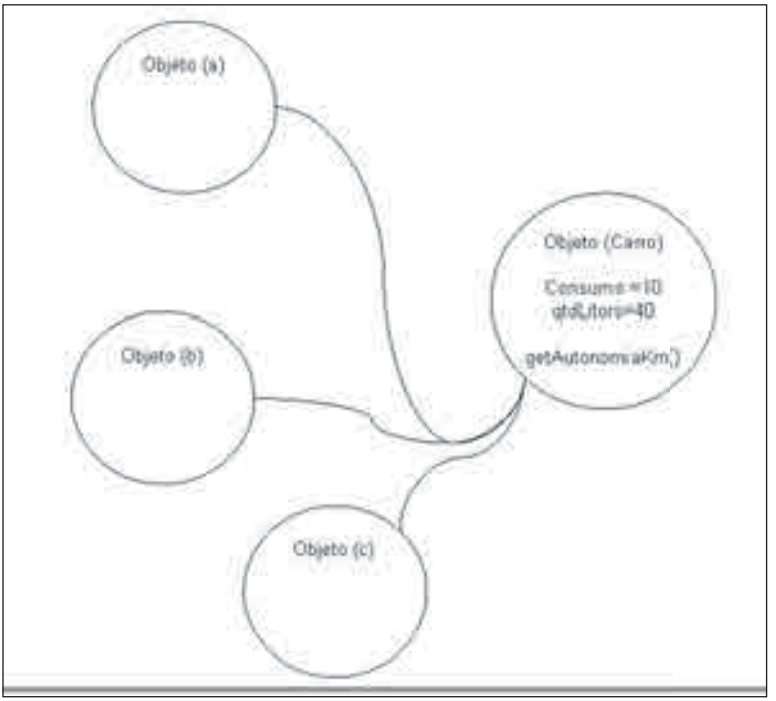


Figura 12 – Exemplo de implementação com um fraco acoplamento.
Fonte: Autor

O acoplamento entre os objetos pode fazer com que uma mudança tenha repercussão em vários pontos do software, dificultando a o processo de manutenção. Para o exemplo anterior, suponha, agora, que haja uma alteração na regra de negócio referente a autonomia e a formula seja mudada para R(1) = (consumo * qtdLitos * percentualPerda). O resultado desta simples mudança tem impacto significativo, é um alteração nos objetos a,b, c. A forma correta para esta situação é encapsular o algoritmo referente à regra de negócio R(1) no objeto Carro e disponibilizar na sua interface um método double getAutonomiaKm() como apresentado na Figura a cima. Assim,

caso haja alguma mudança na formula da autonomia do Carro está fica restrita ao objeto Carro.

Outro critério que o desenvolvedor deve buscar ao longo do projeto de um software é a redução da complexidade do mesmo. A técnica de modularização amplamente utilizada no paradigma funcional. A técnica estabelece que para problemas complexos deve-se dividi-lo em partes menores, criando unidades funcionais menores de mais fácil compreensão e desenvolvimento. Caso seja aplicável devido ao tamanho de um projeto, deve-se decompor o sistema em subsistemas menores para reduzir a complexidade da situação original. Ao se quebrar o problema em partes menores e mais simples, é possível identificar e estabelecer as camadas e os módulos do sistema, e suas interdependências. A divisão em camadas é uma estrutura desejável, pois reduz sensivelmente o acoplamento dos subsistemas, visto que, uma camada é um subsistema que interage apenas com o subsistema imediatamente inferior ou superior. Os módulos (chamados de pacotes, em Java) são unidades menores que compõem os subsistemas e agrupam as unidades funcionais – as classes. Um critério que deve ser adotado para encapsular classes em um módulo é que as classes que pertencem a um mesmo módulo devem ter um maior nível de acoplamento entre si, ao contrário das classes pertencentes a módulos distintos, que possuem um baixo nível de acoplamento.

4.3 Exercícios do Capítulo

Vamos neste ponto fazer uma atividade de revisão condensando pontos de assunto que já foram apresentados e sobre a redução de acoplamento e complexidade vistos nesta seção. Bons estudos!

1) Marque a alternativa correta sobre as definições de Objeto e Classe.
   a) Objeto é uma estrutura para representar elementos do mundo real composta por estado, comportamento, ciclo de vida e identidade
   b) Classe é composta por estado e comportamento

c) Todas estão corretas
d) Classe é uma forma de tipificar Objetos do mesmo tipo e é criada com base em um Objeto
e) Classe é um estrutura dinâmica utilizada para criar Objetos

2) Complete a frase corretamente: A encapsulação facilita as mudanças.....
   a) pois aumenta a velocidade do software
   b) pois isola pedaços reduzindo complexidade e acoplamento
   c) pois garante que o software foi feito corretamente
   d) pois isola pedaços garantido que não existe complexidade e acoplamento

3) Sobre encapsulamento marque a alternativa correta.
   a) É uma forma de comunicação entre objetos
   b) É uma forma de isolar partes de um software
   c) É uma forma de proteger dados
   d) É uma forma de proteger comportamentos

4) Sobre Generalização e Especialização marque a alternativa correta
   a) O que existe de comum em duas Subclasses pode ser especializado e empacotado na Superclasse
   b) Para existir a especialização não é necessário ocorrer generalização em uma estrutura hierárquica de Classes
   c) Especialização é o que existe de diferente em um Subclasse em relação a sua Superclasse
   d) Pode ocorrer quando não existe um estrutura hierárquica de Classes

5) Sobre Classe Abstrata, Polimorfismo marque a alternativa correta
   a) Polimorfismo só existe quando temos duas classes com uma Superclasse Abstrata ou Não
   b) Polimorfismo só existe quando temos três classes com uma Superclasse Abstrata
   c) Polimorfismo só existe quando temos três classes com uma Superclasse Abstrata ou Não e obrigatoriamente um método abstrato assinado na Superclasse
   d) Polimorfismo só existe quando temos três classes com uma Superclasse Abstrata ou Não e obrigatoriamente um método assinado na Superclasse abstrato ou não
   e) Nenhuma alternativa está correta

6) Marque a alternativa incorreta
   a) Uma classe Abstrata só tem atributos e métodos abstratos
   b) Classe Abstrata não pode criar instâncias
   c) Método abstrato só existe em Classe Abstrata
   d) Classe Abstrata permite definir um tipo genérico
   e) Classe Abstrata é sempre uma Superclasse

7) Marque a alternativa correta sobre o conceito Sobrecarga
   a) Métodos iguais em Classes diferentes
   b) Classes da mesma hierarquia com métodos de nomes diferentes
   c) Métodos com parâmetros iguais e o nome diferente
   d) Na mesma Classe métodos com mesmo nome, mas com parâmetros diferentes

8) Escreva sobre as principais diferenças entre a Programação OO versus a Programação Funcional.

9) Explique abstração apresentado as possibilidades de aplicação em um projeto OO.

Responda as próximas questões com base na figura:

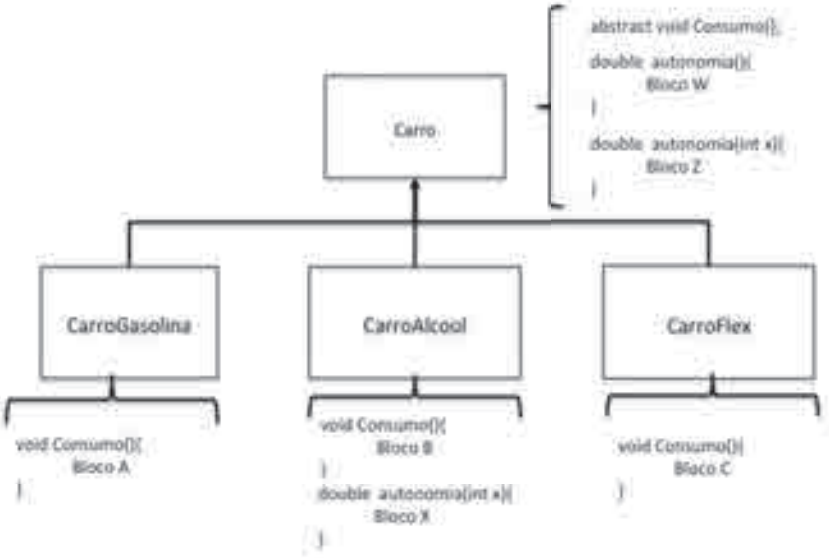


Figura 13 – Exercício do Capítulo: Classe Carro.
Fonte: Autor.

10) Marque a alternativa correta para os métodos na Classe CarroAlcool com os seguintes recursos respectivamente polimorfismo, sobrecarga e oveeridden (sobrescrever)
    a) Consumo ( ), autonomia ( ) e Consumo ( )
    b) autonomia ( ), autonomia ( ) e Consumo ( )
    c) autonomia ( ), autonomia ( ) e autonomia(int x)
    d) Consumo ( ), autonomia ( ) e autonomia(int x)
    e) Todas estão corretas

11) Marque a alternativa incorreta
    a) Existe Polimorfismo em dois métodos
    b) Existe sobrecarga em mais de uma Classe
    c) Existe sobrecarga em uma subclasse
    d) Existe Oveeridden (sobrescrever) em mais de um método
    e) Existe classe Abstrata e Método Abstrato

12) Marque v ou F (uma alternativa errada zera a questão)

N	Comando	V ou F
1	Carro c = new CarroGasolina();	
2	Carro c1 = new CarroAlcool();	
3	CarroGasolina c2= new CarroFlex();	
4	CarroFlex c3 = new CarroFlex();	
5	CarroFlex c4 = new CarroFlex(1);	
6	c.autonomia(10);	
7	c1.Consumo();	
8	c3. autonomia(20);	
9	c4.autonomia();	
10	((CarroGasolina) c).autonomia(10);	

4.4 Exemplo Prático em Java de Comparativo de Programação OO Versus Programação F. Estruturada

Está seção será norteada por um problema para que através da solução você faça uma compreensão ampliada da Programação OO Versus Programação F. Estruturada. Suponha o seguinte problema:

Um algoritmo para calcular a autonomia do carro (função Km) e para calcular quanto esse carro pode rodar antes de atingir o combustível que está no tanque de reserva (função KmParaReserva) de um

Carro. Suponha que o tanque desse Carro é de 8 litros e o consumo e a quantidade de combustível no tanque são variáveis de entrada.

Exemplo:

Informações de Entrada do Programa

1) Placa: JPK 3306
2) Consumo: 10 Km por Litro
3) Qtd Litros: 40 Litros

Informações de Saída do Programa

1) Carro Placa: JPK3306
2) Km: 400 Km
3) KmParaReserva: 320 Km

Os Códigos seguintes apresentam a solução do problema usando as técnicas de Orientação a Objetos (Código 25) e os conceitos do paradigma F. Estruturado (Código 26). A classe Console é uma classe utilitário.

```
1       Exemplo (Classe Carro)
2 public class Carro {
3
4 // Declaração dos Atributos
5       private String placa;
6       private double consumo;
7       private double qtdLitros;

8 // Método Construtor
9 public Carro( String placa_,double consumo_, double qtdLitros_){

10 // Definição dos valores dos atributos através dos parâmetros
11.     this.placa = placa_;
12.     this.qtdLitros = qtdLitros_;
13.     this.consumo = consumo_;
14.}

15.// Método que retorna a autonomia do Carro
16.public double getKm(){

17.     return this.consumo*qtdLitros;
18.}
19.// Método que retorna a autonomia Carro antes de entrar na reserva
20.public double getKmParaReserva(){

21.     return this.getKm()-5*consumo;
22.}

23.// Método que retorna o valor referente a Placa do Carro
24.public String getPlaca(){
25.             return this.placa;
26.}
27.// Método que será executado quando a classe for rodada
28.public static void main(String[] args) {

29.         while (true){
30.           //Obtém a placa do teclado
31.           String placa = Console.readString("Indique a Placa do Carro:");
32.           //Obtém o consumo do teclado
33.           double consumo = Console.readDouble("Indique o consumo do Carro:");
34.           //Obtém a quantidade de litros no tanque do teclado
35.           double qtdLitros = Console.readDouble("Indique a Quantidade de Litros do Carro:");
36.           // Criação de uma instância de carro
37.           Carro carro = new Carro(placa,consumo,qtdLitros);
38.           //Impressão da Placa do Carro
39.           System.out.println("Carro Placa:"+carro.getPlaca());
40.           // Impressão do Autonomia e da Autonomia para atingir a reserva
41.           System.out.println(" KM: " + carro.getKm()+
42.           " KM para Reserva: " + carro.getKmParaReserva());
43.         }
44.
45. }
46.}
```

Código 25 – Exemplo da Solução do Algoritmo na Programação Orientada a Objetos.

Fonte: Autor

```
1.  Exemplo (Classe CarroF)

2.  class CarroF {

3.    //Função que retorna a autonomia do Carro
4.    public static double getKm(double consumo, double qtdLitros){

5.      return consumo*qtdLitros;
6.    }

7.    //Função que retorna a autonomia do Carro antes de entrar na reserva
8.      public static double getKmReserva(double consumo, double qtdLitros){
9.
10.     return CarroF.getKm(consumo,qtdLitros) - 5*qtdLitros;
11.   }

12.   // Função executado quanto a classe for rodada
13.   public static void main(String args[]) {

14.   while (true){
15.
16.    String placa = Console.readString("Indique a Placa do Carro:");
17.    double consumo =     Console.readDouble("Indique o consumo do Carro:");
18.    double qtdLitros = Console.readDouble("Indique a Quantidade de Litros do Carro:");

19.    System.out.println("Carro Placa:"+placa);
20.
21.   System.out.println(" KM: " + CarroF.getKm(consumo,qtdLitros)+
22.   " KM para Reserva: " + CarroF.getKmReserva(consumo,qtdLitros));

23.   }
24.
25.   }
26.   }
```

Código 26 – Exemplo da Solução do Algoritmo na Programação F. Estruturada.
Fonte: Autor

A conclusão aferida sobre estes dois paradigmas é que são complementares e que o paradigma Orientado a Objetos é uma evolução do paradigma F. Estruturado. No processo de assimilação pode-se afirmar que o inicio do aprendizado na Orientação a Objetos possui uma curva de aprendizado bem maior, visto que se tem uma quebra de paradigma na forma convencional de programação e especificação. Além disso, existe o envolvimento de um maior número de definições e conceitos. Em contra partida uma vez feita a assimilação dos conceitos sobre Orientação a Objetos certamente será difícil pensar ou projetar/implementar software de outra forma.

4.5 Exercícios Adicionais do Capítulo

Questionário de Comparativo OO e Programação Funcional

Vamos neste ponto fazer mais uma atividade de revisão juntado com o assunto anteriormente apresentado. A realização do exercício te ajudará a reter mias ainda o que está sendo apresentado ao longo da disciplina. Bons estudos!

1) Explique a frase: Acoplamento e Complexidade Quanto Menor Melhor

2) Porque o empacotamento de dados e funções na Orientação a Objetos é vantajoso.
3) Faça um comparativo entre a Programação Funcional e a Programação Orientada a Objetos.
4) Complete o conteúdo da cada frase abaixo:
   a) A abstração controla a complexidade...
   b) A encapsulação facilita as mudanças...
   c) A hierarquia (grafo de herança) permite a construção de software...
   d) O software orientado a objetos é mais fácil de manter porque sua estrutura....
5) Para a situação ***(a)*** temos um programa com um alto acoplamento e um fraco encapsulamento. Explique como seria possível corrigir esse problema estrutural.

   Um programa com as Classes ***Carro***, ***Vendedor*** e ***Consumidor***. Na implementação atual, cada classe usa diretamente os atributos de ***Carro*** consumo e quantidade de litros para efetuar o calculo da *Km* (Regra1: Km = consumo * quantidade de litros). Suponha que seja necessário fazer a manutenção da Regra1: Km = consumo * quantidade de litros*0.01.

ASSOCIAÇÕES E COMPOSIÇÕES

5.1 Introdução

Naturalmente objetos em uma aplicação estão relacionados. Uma forma de associação é a hierarquia de objetos que é representada através da herança entre Classes. Assim, se uma classe Poento3D herda de uma classe Ponto, pode-se afirmar que um objeto de Ponto3D está abaixo na hierarquia em relação a um objeto do tipo Ponto.

Outra forma de associação são as referências ou relacionamentos. Os relacionamentos determinam a forma de associação entre os objetos. O relacionamento entre uma classe Empregado uma classe Departamento indicará como os objetos dessas duas classes se relacionarão. Por exemplo, pode-se enunciar que um Empregado trabalha em um somente um Departamento e que em um Departamento podem trabalhar vários Empregados(ilustar). Existem alguns tipos de associação que são estruturais também conhecidos de composição. Quando um objeto é responsável por outros objetos ou é composto por outros objetos tem-se um tipo de associação denominada de agregação. Por exemplo, um Circulo é composto por dois Pontos que representam o seu centro. Assim, quando se enuncia que um objeto é "parte-de" tem-se a ocorrência de composição ou agregação. Existem dois tipos de agregação simples ou composta. A agregação simples mantém a semântica de composição "parte-de", mas o objeto pode ser compartilhado, ou seja, fazer "parte-de" mais de um objeto. A agregação composta somente um objeto possui os objetos agregados. Uma forma fácil de fazer a diferença entre agregação simples e composta é que na agregação composta os objetos agregados estão no mesmo ciclo de vida do objeto que os agrega. Por exemplo, um objeto Pedido é composto por Itens, caso o Pedido encerre o seu ciclo de vida os Itens também serão encerrados. A composição de objetos é um tipo de relação muito utilizada em projetos des-

sa natureza. Como no mundo real onde os objetos são compostos por outros objetos, na orientação a objetos isto não é diferente. Na prática observa-se que em projetos OO o uso de composição ocorre quase quem 100% dos projetos. Este livro traz na seção X diversos exemplos práticos de composição devido a sua importância.

5.2 Componente

O conceito de componente de software é entendido como uma composição de classes que juntas formam uma unidade para cumprir certas responsabilidades. Componentes devem possuir uma interface pública para a comunicação com os blocos de código externos (não confundir interface homem máquinas, ou seja, telas, menus com interface de um componente que é a forma padronizada de comunicação com o mesmo). Eles têm como principal característica o funcionamento como caixas pretas, onde o desenvolvedor não precisa, necessariamente, conhecer os detalhes de sua implementação. Esta diretiva permite a redução da complexidade na produção de um software visto o desenvolvedor se abstrai da complexidade do algoritmo interno do componente. Outro ponto relaciona-se com a qualidade do código encapsulado no componente, pois como o bloco de código é usado em diversos projetos por principio este código já esta homologado e bem testado. Assim, através da "componentização" e dos conceitos de Programação Orientada a Objetos é possível a construção de projetos com alto grau de reutilização e compartilhamento de código, gerando uma redução no ciclo de desenvolvimento e facilitando as futuras manutenções corretivas e evolutivas.

5.3 Aplicando Conceitos Relacionados com Composição

Para a melhor absorção da prática associada à composição veremos uma introdução sobre coleções em Java. Ressalta-se que o conteúdo que será apresentado nesta seção não esgota o assunto, assim para um maior grau de aprofundamento no tema recomendam-se leituras complementares.

Antes de detalhar a temática de coleções é importante entender porque este assunto está tão associado como composição. Para compormos objetos com outros é necessário saber como constituir as relações entre eles. Quando um objeto é composto por somente um objeto de uma outra classe o mapeamento é feito através de uma associação ou referencia. De forma aplicada é necessário que a classe que é composta por outra possua um atributo do seu tipo. Por exemplo, se um cliente tem um endereço à forma de mapeamento é na classe Cliente ter um atributo do tipo endereço. Quando temos uma relação de uma classe com n objetos de outra classe a forma de mapeamento é através de um atributo multivalorado. Por exemplo, se uma Agenda está associada com n objetos contatos da classe Contato a forma de mapeamento é a criação na classe Agenda de um atributo multivalorado com uma coleção de contatos, ou seja, este atributo terá n referencias para objetos do tipo contato. A figura abaixo apresenta graficamente os dois tipos de composição entre objetos.

Figura 14 – Representação Gráfica da Relação de Composição entre Objetos.
Fonte: Autor.

Para a implementação de atributos multivalorados pode-se utilizar vetores como, por exemplo, ilustra o Código a seguir. Neste caso o nosso atributo multivalorado é a variável *c* que pode referenciar até três objetos do tipo contato.

```
public class Contato {
    private String nome;
    private String telefone;

    /**
     * @return Returns the nome.
     */
    public String getNome() {
        return nome;
    }
    /**
     * @param nome The nome to set.
     */
    public void setNome(String nome) {
        this.nome = nome;
    }
    /**
     * @return Returns the telefone.
     */
    public String getTelefone() {
        return telefone;
    }
    /**
     * @param telefone The telefone to set.
     */
    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
}
```

```
public static void main(String args[]){
    Contato[] c = new Contato[3];
    c[0] = new Contato();
    c[0].setNome("Eduardo");
    c[0].setTelefone("3351-9878");

    c[1] = new Contato();
    c[1].setNome("Pedro");
    c[1].setTelefone("3345-6789");

    c[2] = new Contato();
    c[2].setNome("Maria");
    c[2].setTelefone("8789-7688");

    for (int i=0;i<c.length;i++){
        System.out.println(c[i].getNome()+"/"+c[i].getTelefone());
    }

}
```

Código 27 – Exemplificação de um Vetor para a Classe Contato.
Fonte: Autor

5.4 Coleções (Collection)

Apesar deste tipo de implementação nos possibilitar a criação de atributos multivalorados, o esforço para desenvolver todos os algoritmos de inserção, remoção, alteração, recuperação e ordenação de objetos é muito grande. Assim, vamos usar coleções que é um conjunto de bibliotecas de classe e interfaces disponíveis no java que nos permite reutilizar algoritmos para a composição de grupos de objetos. De forma sucinta coleções (também chamado de *container*) é um objeto que agrupa múltiplos objetos. As coleções são usadas para armazenar, recuperar e manipular dados na memória primária. Para isso, existem vários tipos de coleções que devem ser usadas de acordo com o problema a ser resolvido. (por exemplo *ArrayList*, *HashSet*, *HashMap*, etc)

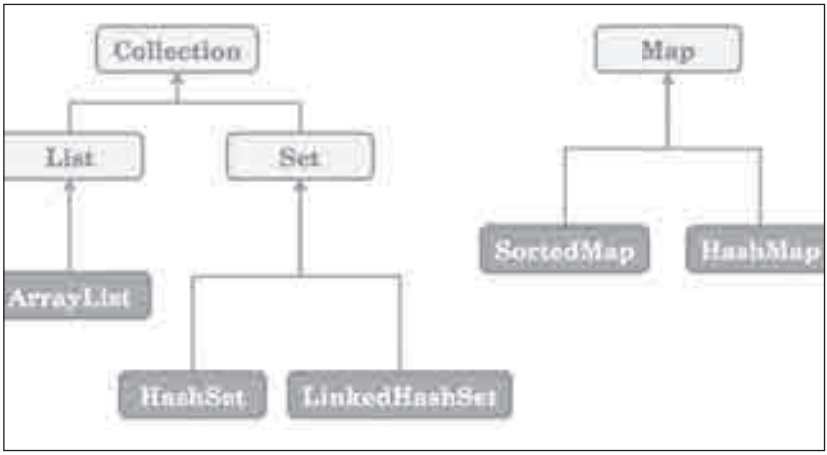


Figura 15 – Diagrama de Hierarquia de Interface de Coleções em Java.
Fonte: Autor

A classe *Collection* é a superclasse mais genérica para as estruturas de armazenamento no Java. Sua interface deve ser implementada por todas as suas classes filhas e seus métodos mais utilizados são os seguintes:

- **boolean add(Object)** – O método *add* pode adicionar qualquer objeto em sua estrutura, retornando um tipo *boolean* para indicar se o mesmo foi adicionado com sucesso;
- **boolean addAll(Collection)** – O método *addAll* pode adicionar qualquer coleção de objetos em sua estrutura, retornando um tipo *boolean* para indicar se a mesma foi adicionada com sucesso;
- **boolean remove(Object)** – O método *remove* pode remover qualquer objeto que esteja armazenado em sua estrutura, retornando um tipo *boolean* para indicar se o procedimento foi realizado com sucesso;
- **boolean removeAll(Collection)** – O método *removeAll* pode remover uma coleção de objetos que esteja armazenado em sua estrutura, retornando um tipo *boolean* para indicar se a mesma foi removida com sucesso;

- **boolean contains(Object)** – O método *contains* verifica se o objeto passado está armazenado em sua coleção e retorna um tipo *boolean* para indicar a resposta;
- **boolean containsAll(Collection)** – O método *containsAll* verifica se a coleção de objetos passada está armazenada em sua coleção e retorna um tipo *boolean* para indicar a resposta;
- **boolean isEmpty()** – O método *isEmpty* indica através de um retorno do tipo *boolean* se a coleção está vazia;
- **void clear()** – O método *clear* limpa a coleção, removendo todos os objetos armazenados.

Dentre as filhas da interface *Collection* podemos citar *List* e *Set*, pois são duas das coleções mais utilizadas em Java. A interface List visa a criação de estruturas de armazenamento que implementem a ideia de uma lista encadeada, como por exemplo a classe concreta ArrayList.

As estruturas que herdam a classe List devem implementar, entre outros e além dos já citados da classe Collection, os seguintes métodos:

- **boolean add(int, Object)** – Esta sobrecarga do método *add* pode adicionar qualquer objeto em sua estrutura a partir de um índice (número inteiro) que servirá para inserir o objeto na posição desejada, tendo 0 como sua primeira posição. Também haverá um retorno do tipo *boolean* para indicar se o mesmo foi adicionado com sucesso;
- **Object remove(int)** – Esta sobrecarga do método *remove* pode remover a partir de uma posição, informada no índice que é recebido por parâmetro. Neste caso o retorno será o objeto (tipo *Object*) removido;
- **Object get(int)** – O método *get* pode localizar um objeto, também, de acordo a sua posição. Neste caso o retorno será o objeto (tipo *Object*) desejado.
- **int indexOf(Object)** – O método indexOf serve para identificar o índice de um objeto, ou posição na qual o mesmo foi armazenado. O retorno deste método é o índice do objeto informado no parâmetro.

Já a interface *Set* visa a criação de estruturas que não permitam a duplicidade de seus elementos, como por exemplo a classe concreta *HashSet*. As estruturas que herdam a classe *Set* devem implementar, entre outros, os métodos já citados da classe pai *Collection*.

Utilizando uma coleção podemos implementar o conceito de composição de elementos em orientação a objetos. Quando identificamos, por exemplo, que uma turma é composta por vários alunos, a implementação correta da classe Turma deve prever uma variável do tipo *Set* que armazene vários tipos da classe Aluno, já que o mesmo aluno não pode estar presente na sala duas vezes. Utilizando uma variável do tipo *Set* para armazenar a coleção de alunos estamos garantindo a unicidade da relação. Abaixo um exemplo de implementação desse possível problema:

```
public class Turma {
      private String disciplina;
      private Set<Aluno> alunos;

      public String getDisciplina() {
            return disciplina;
      }
      public void setDisciplina(String disciplina) {
            this.disciplina = disciplina;
      }

      public Set<Aluno> getAlunos() {
            return alunos;
      }
      public void setAlunos(Set<Aluno> alunos) {
            this.alunos = alunos;
      }
}

public static void main(String[] args) {
      Turma turmaA = new Turma();
      turmaA.setAlunos(new HashSet<Aluno>());

      Aluno jose = new Aluno();
      turmaA.getAlunos().add(jose);

      Aluno pedro = new Aluno();
      turmaA.getAlunos().add(pedro);
}
```

Código 28 – Exemplo de Implementação Turma.
Fonte: Autor

No caso de um problema que envolva a modelagem de uma linha de produção, onde a montagem de um produto envolve a organização em fila dos itens que compõem o produto, deve-se utilizar a classe *List*, pois um mesmo item pode ser utilizado mais de uma vez durante o processo de montagem do mesmo produto, tendo a necessidade de estar presente duas ou mais vezes na mesma lista. Abaixo uma exemplo de implementação desse problema utilizando a classe *List*:

```
public class Produto {
      private String partNumber;
      private List<Item> itens;

      public String getPartNumber() {
            return partNumber;
      }
      public void setPartNumber(String partNumber) {
            this.partNumber = partNumber;
      }

      public List<Item> getItens() {
            return itens;
      }
      public void setItens(List<Item> itens) {
            this.itens = itens;
      }
}

public static void main(String[] args) {
      Produto produtoA = new Produto();
      produtoA.setItens(new ArrayList<Item>());

      Item porca = new Item();
      produtoA.getItens().add(0, porca);

      Item parafuso = new Item();
      produtoA.getItens().add(1, parafuso);
}
```

Código 29 – Exemplo de Implementação Produto.
Fonte: Autor

As coleções podem ser percorridas facilmente no Java através da estrutura de repetição For. Abaixo temos um exemplo de implementação que trata disso:

```
public static void main(String[] args) {
      Produto produtoA = new Produto();
      produtoA.setItens(new ArrayList<Item>());

      Item porca = new Item();
      produtoA.getItens().add(0, porca);

      Item parafuso = new Item();
      produtoA.getItens().add(1, parafuso);

      for (Item it : produtoA.getItens()) {
            int alt = it.getAltura();
            int larg = it.getLargura();
      }
}
```

Código 30 – Exemplo de Implementação *for*.
Fonte: Autor

Além de *List* e *Set*, outra estrutura de armazenamento utilizada no Java pode ser a da classe *Map*. Esta trabalha de forma diferente de uma coleção, porém utiliza uma dentro de sua estrutura. Ao invés de adicionar o objeto solto dentro da classe um *Map* sempre irá vincular esse objeto com uma chave, que deverá ser outro objeto.

Abaixo temos os principais métodos de uma classe *Map*:

- **Object put(Object, Object)** – Este método é responsável por adicionar dois elementos na estrutura de armazenamento, o 1º é a chave que servirá de índice para o 2º, que é o valor que se deseja armazenar;
- **Object remove(Object)** – O método *remove* pode remover a partir da chave, que é recebida por parâmetro. Neste caso o retorno será o objeto (tipo *Object*) removido;
- **Object get(Object)** – O método *get* pode localizar um objeto de acordo a chave passada. Neste caso o retorno será o objeto (tipo *Object*) desejado;
- **boolean containsKey(Object)** – Este método permite ao programador verificar se uma determinada chave (passada

no parâmetro) já está sendo utilizada para indexar algum valor. Para isso é utilizado um retorno do tipo *boolean*;

- **boolean containsValue(Object)** – O método *containsValue* é utilizada para verificar se um objeto já foi adicionado ao *Map* e retornar um *boolean* para indicar a resposta;
- **Set<Object> keySet()** – O método *keySet* retorna a coleção que armazenas todas as chaves que estão sendo utilizadas;
- Collection<Object> values() – Este método retorna em uma coleção todos os objetos que estão sendo armazenados no *Map*.

Abaixo temos um exemplo de utilização da estrutura *Map*:

```
public class Turma {
      private String disciplina;
      private Map<String, Aluno> alunos;

      public String getDisciplina() {
            return disciplina;
      }
      public void setDisciplina(String disciplina) {
            this.disciplina = disciplina;
      }

      public Map<String,Aluno> getAlunos() {
            return alunos;
      }
      public void setAlunos(Map<String, Aluno> alunos) {
            this.alunos = alunos;
      }
}

public static void main(String[] args) {
      Turma turmaA = new Turma();
      turmaA.setAlunos(new HashMap<String, Aluno>());

      Aluno jose = new Aluno();
      jose.setMatrituca("ABC123");
      turmaA.getAlunos().add(jose.getMatricula(), jose);

      Aluno pedro = new Aluno();
      pedro.setMatrituca("EDF456");
      turmaA.getAlunos().put(pedro.getMatricula(), pedro);
```

Código 31 – Exemplo de Estrutura MAP.
Fonte: Autor

As estruturas de armazenamentos também podem ser utilizadas para outros fins que não sejam a composição de classes. Tudo vai depender do problema em questão e de como a solução deve ser implementada.

CONSIDERAÇÕES SOBRE REUTILIZAÇÃO DE SOFTWARE

6.1 Introdução

Nesta seção, aborda-se o tema reutilização. O objetivo é discutir os vários níveis de análise, projeto ou código de reutilização. Em seguida, faz-se o estudo de dois tipos de reutilização: Framework Orientado a Objeto e Padrões de Projeto.

6.2 Motivação para a Reutilização de Software

O desenvolvimento de projetos de software não é uma tarefa fácil. Os projetos, geralmente, têm que ser robustos, atuar sobre problemas complexos e estar implantados em curto prazo ("time-to-market").

A reutilização é reconhecida como um importante modo para se alcançar um aumento na produtividade em projetos de software, pois possibilita agregar funcionalidades pré-existentes na produção de novos software. Ressalte-se também a possibilidade de programadores iniciantes criarem software complexos através da utilização de padrões de projetos descritos por programadores mais experientes.

Além do mais, a reutilização de pedaços de software garante uma maior qualidade ao projeto, visto que os blocos a serem reutilizados já estão testados e validados por uma ou mais aplicações. Portanto, o direcionamento no desenvolvimento de um projeto de software é para que não se construa nada que já exista e que possa ser reutilizado.

Entretanto, é necessário produzir software genéricos e extensíveis que possam ser aplicados a uma gama de aplicações. Ao longo dos anos, muitas metodologias surgiram na engenharia de software com o intuito de reduzir a complexidade e aumentar a produtividade no desenvolvimento. A Orientação a Objetos é reconhecida hoje como o principal paradigma para atender a essa redução da complexidade.

6.3 Reutilização e Orientação a Objetos

Reutilização de software é o grande desafio dos projetistas de software. A engenharia de software vem evoluindo substancialmente para ir ao encontro a essa necessidade. Dentre as diversas metodologias existentes, Orientação a Objetos (OO) é o principal paradigma para a construção de software reutilizável. Apesar do grande sucesso do paradigma OO, nem sempre os resultados, em relação a reutilização, são obtidos de forma fácil.

As vantagens da Orientação a Objetos, como paradigma para a reutilização de software, são bem conhecidas. As abstrações podem corresponder às coisas do domínio do problema. O nível é mais natural. É mais fácil comunicar-se com o usuário ou *domain expert* na linguagem dele.

Os mesmos objetos existem em todas as fases e uma notação única (objetos) facilita, portanto, a integração entre fases de desenvolvimento (passar de uma fase para a outra).

É mais fácil entender o domínio do problema quando este é quebrado em pedaços: gerenciamento da complexidade através da modularização.

O mesmo pode ser dito no domínio do computador (projetando e programando com objetos). A abstração controla a complexidade (escondendo algo através da separação da interface e da implementação). A encapsulação facilita as mudanças (através do isolamento). A hierarquia (grafo de herança) permite uma melhor reutilização.

Um dos mecanismos da Orientação a Objetos para alcançar a reutilização de código é o uso de biblioteca de classes que encapsulam determinadas funcionalidades que possam ser reutilizadas. Embora classes sejam unidades de código reutilizáveis, uma classe para cumprir sua funcionalidade, geralmente, tem dependências com outras classes. Para a realização de tarefas complexas é necessário que um grupo de classes trabalhem conjuntamente, caracterizando assim, um componente de software.

O conceito de componente de software pode ser entendido como uma composição de classes que juntas formam uma unidade para cumprir certas responsabilidades. Componentes devem possuir uma interface pública para a comunicação com os blocos de código externos. Eles têm como principal característica o funcionamento como `caixas pretas', onde o desenvolvedor não precisa, necessariamente, conhecer os detalhes de sua implementação.

Através do uso de componentes, a reutilização popularizou-se. Um exemplo clássico são as ferramentas RAD ("Rapid Application Development") gráficas (Visual Basic, Delphi, Centura, etc), que fornecem componentes visuais aplicáveis nas construções de interfaces gráficas. Os componentes visuais (VBX, OCX, etc) disponíveis nessas ferramentas proporcionam um aumento na produtividade de desenvolvimento de sistemas comerciais.

Além da reutilização de código, existem outros tipos de reutilização. Por exemplo, a reutilização de projeto atua não somente no patamar de código, como também nos projetos conceitual e lógico.

A questão que se coloca é: Como encapsular, em bibliotecas ou em componentes, comportamentos genéricos para um domínio de aplicação (Contexto em que uma aplicação está inserida), que possam ser usados em software de mesmo domínio (Uma fronteira formal que define um particular assunto ou área de interesse [LAR00]), só que com características específicas?

A próxima seção trata de Framework, uma técnica de desenvolvimento de software que permite a construção de projetos reutilizáveis em todos os níveis.

6.4 Framework Orientado a Objetos

A reutilização em um projeto de sistemas deve contemplar todos os níveis de um projeto e não se restringir somente à codificação, ou seja, deve englobar análise, projeto, codificação e testes.

Framework provê reutilização em alto nível. Portanto, através da técnica de projeto de software framework, é possível construir projetos extensíveis, em que a reutilização está focada desde o domínio da aplicação.

É importante destacar que framework não é alternativa à Orientação a Objetos ou a componentes, mas sim, trata-se de um recurso adicional de projeto para reutilizar algo mais do que código.

Na verdade, é difícil imaginar a construção de um framework que não seja OO. A Orientação a Objetos possui características como abstração, encapsulamento, herança e colaboração de classes que viabilizam a sua criação. Da mesma forma, um framework é construído com base em componentes.

Na sequencia, tratamos o tema Framework com mais detalhes, envolvendo o conceito, as diferenças entre framework e bibliotecas de classes, o projeto de um framework, e a construção de aplicações baseadas em frameworks. Por fim, discutimos a importância do tema.

6.5 Conceito de Framework

Na literatura existente são várias as definições de framework. Nesta seção, vamos enfatizar as duas que, na nossa opinião, representam melhor o conceito de framework.

Iniciamos com uma definição genérica: “Framework na sua essência pode ser definido como um conjunto de blocos de software que os programadores podem usar, estender, ou customizar, com pouco esforço, para um domínio específico de problema.” [IBM 99].

Já uma definição mais técnica determina que “Um framework é um conjunto de classes colaborativas abstratas e concretas, que pode ser usado como um template (gabarito) para resolver uma família de problemas relacionados. Ele é usualmente estendido através de subclasses, para obter-se o comportamento específico de uma aplicação.” [LAR00]

É importante enfatizar que existe uma diferença substancial entre framework e biblioteca de classes.

6.6 Framework Versus Biblioteca de Classes

A compreensão do conceito de framework relaciona-se com o entendimento de sua diferença com biblioteca de classes. A confusão entre os dois conceitos é uma incompreensão clássica entre desenvolvedores de software.

Framework ainda pode ser visto como uma espécie de biblioteca, só que o controle do fluxo de chamadas é bidirecional, ou seja, tanto uma aplicação pode chamar métodos do framework como o framework pode invocar métodos da aplicação. Isto é diferente de uma biblioteca de classes stricto sensu, em que o fluxo de chamadas é unidirecional, da aplicação para a biblioteca. A possibilidade de o framework chamar métodos da aplicação se dá através de chamadas *dynamic binding*, implementadas nas subclasses da extensão do framework para a aplicação.

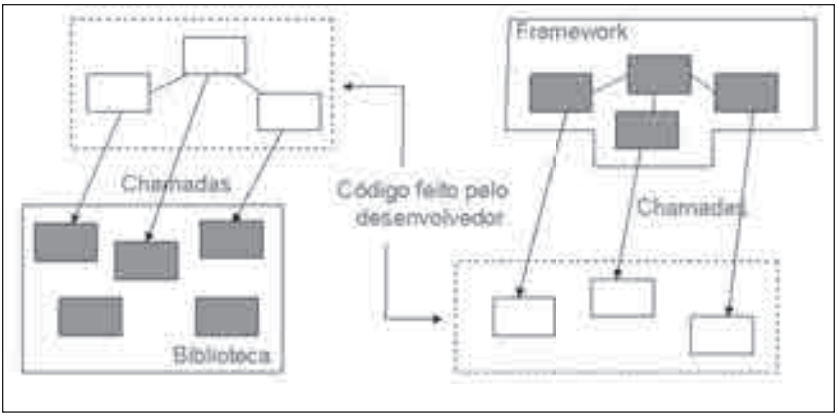


Figura 16 – Diferença no Fluxo de Controle entre Framework e Bibliotecas de Classes
Fonte: [LAN95].

A Figura anterior ilustra duas aplicações. A primeira aplicação possui classes que fazem chamadas às classes pertencentes a uma biblioteca de classes. A segunda aplicação foi criada a partir de um Framework. Neste caso, são as classes do Framework que fazem as chamadas às classes da aplicação. A construção de um framework não é uma tarefa trivial. Passemos a discuti-la.

6.7 Framework e Reutilização de Software

Ao projetar um Framework, é imprescindível levar em conta que sua estrutura deve servir para a construção de diversos sistemas, que pertencem a um mesmo domínio de aplicação. Este projeto genérico deve capturar e encapsular todas as funcionalidades comuns às futuras aplicações que serão desenvolvidas, como ilustra a Figura abaixo.

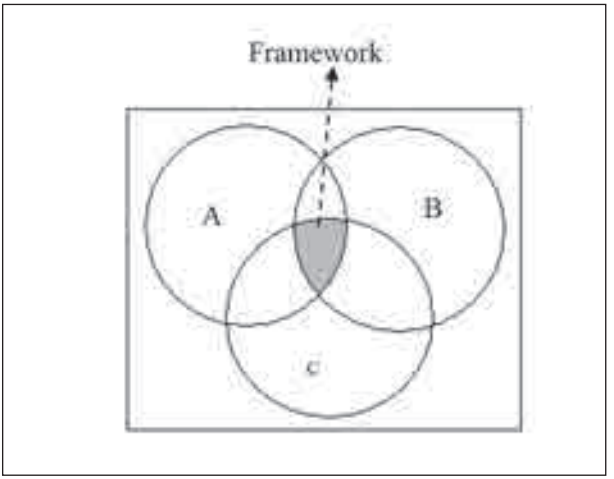


Figura 17 – Interseção de Três Aplicações.
Fonte: Autor

A Figura anterior sugere três aplicações A, B e C, representadas por círculos. Estas aplicações possuem funcionalidades específicas e funcionalidade comuns. As funcionalidades comuns do framework estão representadas no gráfico pela área hachurada que é a interseção

dos três círculos. Quanto maior a interseção entre as aplicações, mais funcionalidades poderão estar encapsuladas no framework.

Um framework deve ser projetado para ser usado através do princípio *Hollyood Principle*, isto é, "Não nos chame, nós iremos chamar você". As classes desenvolvidas nas aplicações irão receber mensagens das classes do framework (Figura 17). Assim, é possível que as classes definidas nas aplicações façam o resto do trabalho específico que o framework deixou em aberto.

Ainda ao projetar-se um framework, deve-se construir uma aplicação quase que completamente, faltando apenas "pedaços" para serem completados com as funcionalidades específicas das futuras aplicações. Mais precisamente, a arquitetura do framework deve ser tal que possibilitará às aplicações que vierem a ser construídas utilizar a sua infraestrutura. A Figura abaixo dá uma ideia disto. A aplicação é a soma do framework e do bloco com as extensões que encapsulam as funcionalidades específicas.

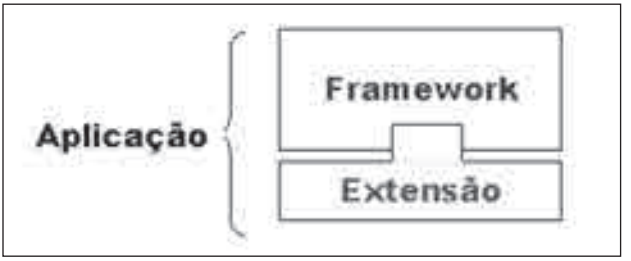


Figura 18 – Aplicação Construída Utilizando a Infraestrutura de um Framework.
Fonte: Autor

Pode-se observar também na figura anterior que o bloco do framework é maior que o bloco da extensão, sugerindo que foi alcançado um alto grau de reutilização, o que é desejável.

Por fim, um framework deve ser tratado como um produto que necessita de documentação e suporte. Como produto ele deve ser planejando para ser distribuído e sofrer manutenções constantes.

6.8 Construindo Aplicações a partir de um Framework

Para construir aplicações a partir de um framework o desenvolvedor, preliminarmente, deve: Verificar se o framework atende às necessidades da aplicação. Analisar o grau de complexidade para incorporá-lo à aplicação que será gerada. Distinguir qual é a camada (subsistema) que é fixa, ou seja, o framework, e qual é a camada variável (o subsistema que será desenvolvido).

Fica óbvio então que, para começar, o framework a ser reutilizado deve ser muito bem projetado, e dispor de uma excelente documentação. A documentação deve descrever em detalhes os passos a ser seguidos para a construção de uma aplicação através do framework, incluindo:

que classes devem ser estendidas; que métodos devem ser implementados; que novas classes devem ser criadas.

Na ótica do usuário-implementador (aquele que vai estender o framework) o desenvolvimento da aplicação, na prática, implica numa "instanciação" do framework. Por conseguinte, o desenvolvedor terá de estender por herança algumas classes abstratas ou utilizar composição em outras. Como foi dito antes, ele deve utilizar-se da documentação que detalha o que é necessário para estender o Framework. É como utilizar "ganchos" existentes no framework, que permitem "plugar" funcionalidades. Tratam-se de blocos de código que serão adicionados ao framework, para que o sistema fique habilitado a executar funcionalidades específicas da aplicação.

Ressalta-se que, ao desenvolver uma aplicação a partir de um framework, está-se na verdade reutilizando o projeto do framework como um todo, incluindo os níveis conceitual, lógico e físico.

6.9 Framework "A Bola da Vez"

O conceito de Framework não é algo recente e desconhecido na engenharia de software. Desde do final da década de 80, muitas pesquisas nesta área já tinham sido realizadas [WEI88] [TAL93].

Então, porque só mais recentemente o conceito de framework tem sido largamente aceito? Isto se deve: À maturidade das linguagens orientadas a objeto (Java, C++, etc); À demanda por projetos de software mais complexos e reutilizáveis; Aos novos recursos da engenharia de software como padrões de projeto [GAM94], processo interativo incremental [JAC98], UML [LAR00], etc.

Dentre os recursos citados, padrões de projeto têm uma importância significativa e podem ser de grande valia para os desenvolvedores de frameworks. A próxima seção trata de padrões de projeto, com o foco em framework.

PADRÕES DE PROJETO

7.1 Introdução

O bom desenvolvimento de um software orientado a objeto depende de alguns fatores:

Uma boa decomposição do sistema em classes e objetos e o conhecimento pelos desenvolvedores dos diversos aspectos ligados à orientação a Objeto, tais como: Encapsulação; Reutilização; Flexibilidade; Portabilidade; Desempenho.

Um meio para contemplar todos esses requisitos é a utilização de padrões de projeto. Alguns motivos para isto são: Desenvolvedores iniciantes podem utilizar padrões de projetos catalogados por *experts* para obter conhecimento e experiência na construção de aplicações orientadas a objeto.

Com a boa utilização de padrões de projeto a comunicação entre desenvolvedores, e a manutenção de sistemas, tornam-se menos complexas. Padrões de projeto podem ajudar a encontrar abstrações que tornem o software mais flexível e reutilizável. Cada padrão descreve um problema que ocorre várias vezes em um ambiente de desenvolvimento de software, podendo ser utilizado milhões de vezes em situações diferentes, constituindo-se assim num dos cernes da solução [GAM94].

Um padrão de projeto orientado a objeto descreve uma situação em que várias classes cooperam para realizar uma determinada tarefa, criando uma organização específica, ou seja, micro-arquiteturas de classe e objetos com seus papéis e colaborações. Essa micro-arquitetura é uma solução genérica a um problema que pode ocorrer em diversas aplicações. Dizemos que um padrão de projeto resolve um problema recorrente.

A definição de um padrão de projeto é ela própria padronizada da seguinte forma (tabela abaixo) [GAM94]:

Documentação de Padrões de Projeto	
Nome	Identificador usado para descrever o problema, suas soluções e consequências, em uma ou duas palavras;
Descrição	Apresenta o problema e seu contexto; Indica quando devemos aplicar o padrão; Pode apresentar problemas específicos de projeto, ou descrever estruturas de classes ou objetos que geralmente tornam o projeto inflexível (estruturas a evitar, portanto).
Solução	Descreve as relações, as responsabilidades, as colaborações e os elementos que compõem a solução do projeto; Serve como um gabarito que pode ser utilizado em várias situações.
Consequências	Apresentam os resultados e "trade-offs" da utilização do padrão; Ajudam a avaliar se os resultados obtidos justificam a adoção do padrão; Apresentam os impactos em termos de flexibilidade, extensibilidade e transportabilidade.

Tabela 10 – Documentação de Padrões de Projeto.
Fonte: Autor

Padrões de Projeto têm tomado dimensões que sobrepõem a parte de projeto físico. Já existem padrões de análise, testes, etc. Na próxima seção, tratamos do relacionamento entre padrões de projeto e framework e exemplificaremos o padrão *Template Method*.

7.2 Padrões de Projeto e Framework

Num projeto de um framework, padrões de projeto ajudam a encontrar abstrações que tornam o software mais flexível e reutilizável.

Por exemplo, num framework temos o problema de como instanciar objetos de classes que foram "plugadas" no framework. Neste caso, poder-se-ia utilizar o padrão de projeto *Abstract Factory*, que provê uma solução genérica para este tipo de problema. Os padrões de projeto mais comuns utilizados em projetos de framework são: *Façade*, *Command*, *Template Method*, e *Abstract Factory*.

Dentre os padrões citados, *Template Method* é a base para a construção de um framework.

7.3 Template Method

Template Method define o esqueleto de um algoritmo para uma operação, transferindo alguns passos desta operação para as subclasses. Este padrão permite que uma subclasse redefina certos passos de um algoritmo, sem alterar a estrutura do mesmo.

Seja um problema prático "Criar uma solução genérica de *Login* que possa ser reutilizada em qualquer projeto".

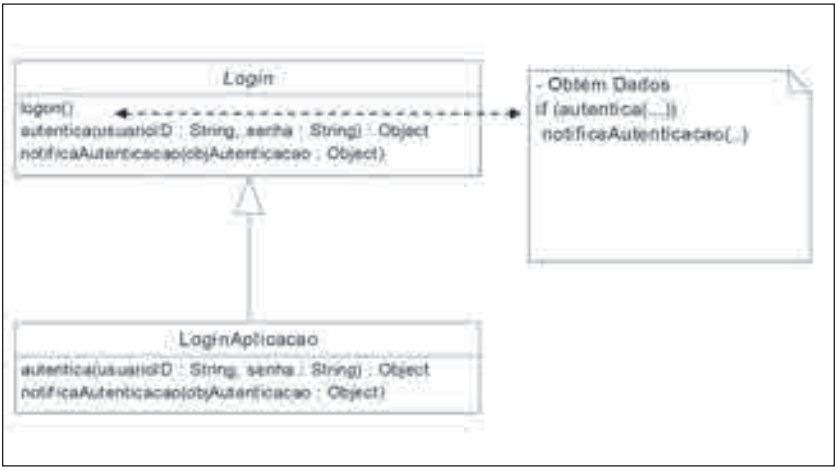


Figura 19 – Classe e Subclasse de *Login*.
Fonte: Autor

A figura acima ilustra um digrama de classes com a solução do problema enunciado. Define-se uma classe abstrata *Login* com estrutura genérica preparada para ser "customizada" pelas futuras aplicações. Esta classe contém um método *logon* (*Template Method*) que encapsula a lógica do processo genérico de *login*. Este método fará chamadas aos métodos abstratos "autentica" e "notificaAutenticacao". Os métodos abstratos serão implementados pela subclasse concreta LoginAplicacao que estende a classe *Login*. A classe LoginAlicação é responsável pela lógica específica da aplicação que está usando o *template* de *Login*.

Adota-se uma solução padrão para problemas do tipo exemplificado. A figura abaixo apresenta uma micro-arquitetura genérica, ou seja, servindo para qualquer aplicação que necessite do *Template Method*. Esta micro-arquitetura é base para construção de um Framework.

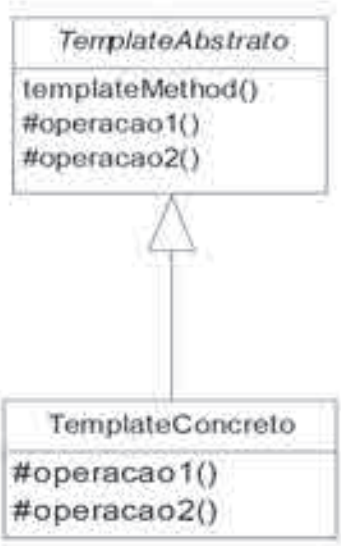


Figura 20 – Padrão Template Method.
Fonte: Autor

A classe TemplateAbstrato é a classe abstrata que contém o comportamento genérico. Na aplicação de *Login* equivale a classe *Login*. Esta classe possui um método "templateMethod" que encapsula a lógica da solução e faz chamadas a operações abstratas (#operacao1, #operacao2). Por fim, temos a subclasse TemplateConcreto que estende a classe TemplateAbstrato para definir as operações, operação1 e operação2, com funcionalidades específicas.

A próxima seção trata de um processo de desenvolvimento voltado para a construção de um framework.

VOCÊ SABIA?

O Hibernate é um framework para o mapeamento objeto-relacional escrito na linguagem Java, mas também é disponível em .Net com o nome NHibernate. O mesmo facilita o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação, mediante o uso de arquivos (XML) ou anotações Java.
Outros exemplos de frameworks são: Rich Faces, MyFaces, Struts, Spring, JDBC.

O PROCESSO UNIFICADO

8.1 Introdução

A tendência é de desenvolvimento de sistemas de software cada vez maiores e mais complexos. Isto se deve à evolução dos computadores, que estão se tornando cada vez mais poderosos, fazendo com que os usuários alimentem mais expectativas em relação a eles. Esta tendência também tem sido influenciada pela expansão do uso da Internet e da multimídia.

Esse contexto leva à necessidade de utilização de processos de desenvolvimento que sejam adequados ao dinamismo e às expectativas do mercado.Em nível de implementação, já existe um consenso de que OO é a chave para o desenvolvimento rápido de aplicações complexas. Em relação a análise e projeto, faltava uma notação unificada realmente eficaz. Esta lacuna foi preenchida com o surgimento da linguagem UML ("Unified Modeling Language" [JAC98]). UML é uma linguagem de modelagem, mas ela não define um processo de desenvolvimento. Como complemento à UML, surgiu uma especificação para um processo de desenvolvimento de software, chamada de Processo Unificado [JAC98].

O Processo Unificado contempla todo o ciclo de vida de um projeto, e guia as equipes de desenvolvimento nas atividades práticas de engenharia de software. Ele tem sua base em três conceitos chaves: Desenvolvimento orientado por caso de uso; Ênfase em uma arquitetura, desde o inicio; Abordagem iterativa e incremental.

8.2 Desenvolvimento Orientado por Caso de Uso

A construção de um software tem por objetivo atender às requisições dos usuários. As requisições ditas funcionais no Processo Unificado,

desembocam nos casos de uso, isto é, pedaços de funcionalidades de sistemas que retornam valores em que os usuários estão interessados. O conjunto de casos de uso forma o diagrama de casos de uso, que descreve o contexto do sistema, com todas as suas entidades externas (Atores), funcionalidades (Casos de Uso) e os seus relacionamentos.

Na fase de análise, os casos de uso captam e validam as funcionalidades com os usuários. As fases seguintes, projeto, implementação e testes, serão norteadas pelos casos de uso.

8.3 Ênfase na Arquitetura (desde a fase de elaboração do projeto).

A arquitetura de um software é o mais importante aspecto do sistema, e refere-se à estrutura de subsistemas e de componentes. É um mapa do sistema, com as diferentes partes, suas interações e mecanismos de interconexões. Devido à importância da arquitetura no desenvolvimento de software, o Processo Unificado propõe que ela seja definida o mais cedo possível, quase que em paralelo com a coleta dos requisitos.

Ao projetar-se a arquitetura, vários fatores são importantes: A plataforma que o software vai rodar – a arquitetura do computador, o sistema operacional, o gerenciador de Banco de Dados, etc; Considerações de desenvolvimento – linguagem de programação, internacionalização, etc; Integração com sistemas legados e requisitos não-funcionais – performance, confiabilidade, tipo de interface, etc;

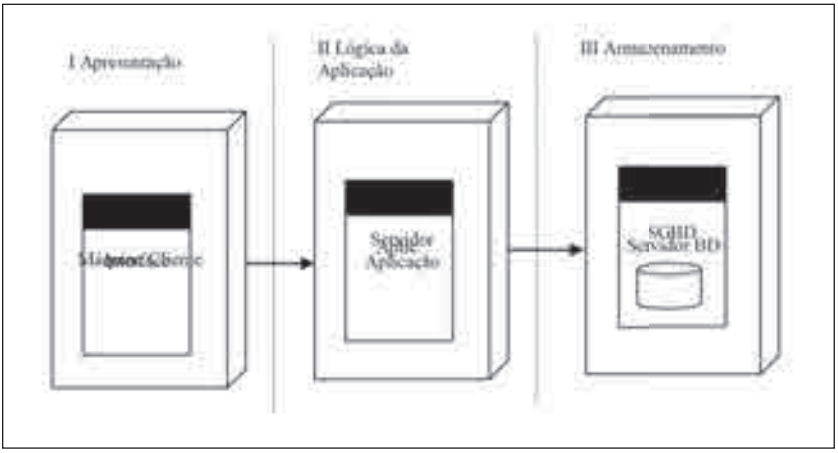


Figura 21 – Visão Clássica de um Arquitetura em Três Camadas.
Fonte: Autor

A figura anterior apresenta um arquitetura comum para sistemas de informação. A arquitetura está dividida em três camadas: apresentação, lógica da aplicação e armazenamento. Na figura em questão cada camada está distribuída em máquinas diferentes (representadas pelas caixas). Listamos alguns elementos das camadas.

I Apresentação – menus, botões, janelas, etc;

II Lógica da Aplicação – objetos e regras de negócio;

III Persistência – mecanismos de armazenamento persistente (XML, tabelas em um SGBD, etc);

8.4 Abordagem Iterativa e Incremental

Os processos de desenvolvimento convencionais têm por premissa ciclos de desenvolvimento em cascata, ou seja, faz-se necessário o término de uma fase para o início de outra. Neste processo, o software só é "lançado" quando as fases de análise, projeto, implementação e testes (do software como um todo) já foram concluídas. Para sistemas que têm um ciclo de desenvolvimento extenso, temos o problema da primeira versão ser disponibilizada muito tempo após

a coleta dos requisitos, que podem ter sofrido mudanças ao longo desse período.

O Processo Unificado iterativo e incremental difere dos processos convencionais, pois baseia-se num ciclo de desenvolvimento curto, em que as versões (incrementos) são disponibilizadas ao longo do ciclo de desenvolvimento do software.

Para a construção das versões, são feitas várias iterações. As iterações são uma sequência de etapas e análise , projeto, implementação, testes e implantação que detalham e ampliam o modelo. Ao final de um ciclo de interações, temos uma versão do sistema.

A vantagem desse processo é que ao longo de todo o processo podem ser feitas avaliações a respeito do cumprimento dos requisitos funcionais, desempenho, confiabilidade, cronograma, recursos utilizados, entre outros. Esta forma de trabalhar diminui riscos que são inerentes ao processo de desenvolvimento de um software. A figura abaixo exemplifica graficamente o Processo Unificado iterativo e incremental [JAC98].

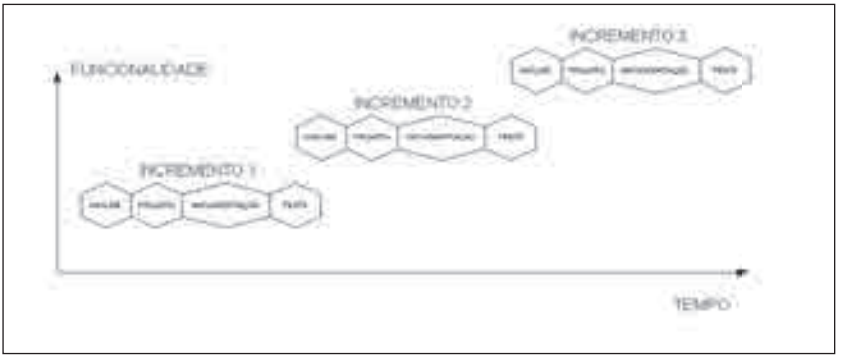


Figura 22 – Processo Iterativo e Incremental.
Fonte: Autor.

O gráfico da figura anterior apresenta a evolução dos incrementos no tempo. Cada incremento para ser construído passa por todo o ciclo de iterações. Ao final de um ciclo de iterações, temos um incremento com mais funcionalidades.

Durante o processo de desenvolvimento dos incrementos, deve-se utilizar técnicas de reutilização de software, para aumentar a produtividade dos desenvolvedores. A obtenção de bons resultados é dependente da experiência de profissionais que desenvolvem projetos (com o pensamento voltado para a reutilização), e do emprego de técnicas de reutilização, como Framework e Padrões de Projeto.

REFERÊNCIAS

Deitel, H.M., Java, Como Programar – 4ª Edição, Bookman, 2003, [GAM94] GAMMA E. & HELM R. & JOHNSON R. & VLISSIDES J., Design Patterns Elements of Reusable Object-Oriented Software,. Addison-Wesley, 1994.

[IBM 99] IBM, Building Object-Oriented Frameworks, White Paper, 1999.

[JAC98] JACOBSON I. & BOOCH G. & RUMBAUGH J., The Unified Software Development Processs, Addison-Wesley,1998.

[LAN95] LANDIN N & NIKLASSON A., Development of Object-Oriented Framework, Conden:Lutedx,1995.

[LAR00] LARMAN C., Utilizando UML e padrões: Uma Introdução à Análise e ao Projeto Orientados a Objetos, Bookman, 2000.

[TAL93] TALIGENT, Inc, Leveraging Object-Oriented Frameworks, A Taligent White Paper, 1993.

[WEI88] WEINAND, E. GAMMA, R Marty, ET++ – An Object+Oriented Application Framework in C++, Proceedings of the OOPSLA'88, 1988.

Sítio: www.t2ti.com, Curso Java Starter (2003). Acessado em 15/02/2015.

Sítio: http://www.if.ufrgs.br/~betz/jaulas/aula2.htm. Acessado em 15/02/2015.

Editora do Instituto Federal da Bahia – Edifba

Formato: 150x210
Fonte: Adobe Garamond Pro, Optimum, 12/14
Papel miolo: Offset 90g.
Papel capa: Supremo 250g.
Tiragem: 300
Impressão: março 2018