INTRODUÇÃO A BOAS PRÁTICAS DE PROGRAMAÇÃO COM JAVA, USANDO PADRÕES DE PROJETO

Guilherme Vilatoro Santos[1]

Luiz Fernando Braga Lopes[2]

RESUMO

Atualmente, a sociedade apresenta uma tendência a ser dependente de softwares para realizar as suas tarefas diárias. Assim, tem aumentado a exigência por qualidade desses produtos. A atividade de programação exige um grande esforço de recursos humanos, sendo que esses são responsáveis pela maior parte dos custos de desenvolvimento de software. Portanto, a atividade de programação merece uma atenção especial para se obter produtividade com qualidade. A programação defensiva tem papel fundamental para atingir esse objetivo, pois permite obter vantagens na geração de software, tais como torná-lo mais bem estruturado, mais legível e confiável. Porém, ela é considerada um desafio para muitas empresas, em especial para as micro e pequenas empresas, cujos profissionais de TI podem não estar habilitados para desenvolvê-la.

Palavras-Chave: Programação Defensiva, Software.

ABSTRACT

Nowadays, society has a tendency to be dependent on software to make their daily tasks. So, has increased the demand for quality of these products. The programming activity requires a great deal of human resources, and these are responsible for the majority of software development costs. Therefore, the programming activity deserves special attention to achieve productivity with quality. The defensive programming has a key role to achieve this goal, because it offers advantages in generating software such as make it better structured, more readable and reliable. However, it is considered a challenge for many companies, especially for micro and small enterprises, whose IT professionals may not be able to develop it.

Keywords: Programming defensively, Software.

---

[1] Acadêmico da pós-graduação de Java da FCV – Faculdade Cidade Verde de Maringá. E-mail: vilatorog@gmail.com

[2] 2 Professor e Orientador da pós-graduação de Java, da FCV – Faculdade Cidade Verde de Maringá. E-mail: prof_braga@fcv.edu.br.

1. INTRODUÇÃO

O desenvolvimento de um software passa por processos na área da Engenharia de software. Esses processos possuem diversas atividades, como comunicação e construção. Na fase de construção o programa é escrito em linhas de código, como também são realizados os testes necessários para validar o código desenvolvido (PRESSMAN, 2006). Para a geração de código é necessário o uso de uma linguagem de programação. A definição desta será feita dependendo do problema que se quer resolver. Essas linguagens possuem aspectos específicos chamados paradigmas de programação, que seria a maneira de abstração de modelo para resolução de determinado problema. Dentre os paradigmas de programação temos o funcional, o lógico, o imperativo e o orientado a objetos (SEBESTA, 2011).

Para que a implementação de um software seja de melhor qualidade, pode-se usar a programação defensiva. Este tipo de programação busca a melhor escrita de código fonte por parte dos desenvolvedores, para isso são usadas boas práticas de programação. Estas práticas têm como propósito diminuir a taxa de manutenção de software, bem como melhorar a qualidade do produto final (SAVIDIS, 2004).

Atualmente, a sociedade é dependente de softwares para realizar suas tarefas diárias, assim tem aumentado a exigência por qualidade desses produtos. Neste contexto, a atividade de programação merece uma atenção especial para obter este resultado esperado. A programação defensiva tem papel fundamental para atingir esse objetivo, pois permite obter vantagens na geração de software, tais como torná-lo melhor estruturado, mais legível e mais confiável. Porém, ela é considerada um desafio para muitas empresas, em especial para as micro e pequenas empresas, cujos profissionais de TI podem não estar habilitados para desenvolvê-la.

O objetivo deste artigo é realizar uma revisão bibliográfica sobre as boas práticas de programação com Java usando padrões de projetos. Para alcançar o objetivo geral, foram utilizados os seguintes objetivos específicos:

a) Pesquisar e exemplificar boas práticas de programação com Java, utilizando padrões de projeto;

b) Evidenciar a importância da aplicabilidade das boas práticas de programação.

A pesquisa foi realizada adotando uma metodologia qualitativa e bibliográfica, realizando o levantamento de boas práticas de programação, usando padrões de projeto, com a justificativa de contribuir para que profissionais de TI possam gerar um código fonte de melhor qualidade tentando diminuir a quantidade de problemas futuros em relação ao código desenvolvido.

Diante do cenário apresentado, esse artigo contribui com um conjunto boas práticas de programação usando padrões de projeto, que auxiliem profissionais a gerar um código fonte de melhor qualidade.

2. REVISÃO DA LITERATURA

Neste capítulo são apresentados os principais conceitos que constituíram as bases teóricas para o desenvolvimento deste trabalho.

2.1 SOFTWARE E SUA ENGENHARIA

De acordo com Pressman (2006), software pode ser definido como um conjunto de instruções que quando executadas desempenha uma determinada tarefa, com o intuito de resolver um determinado problema.

A engenharia de software determina vários aspectos apresentados durante o processo de desenvolvimento de um software, buscando uma padronização e qualidade do produto final. Para Pressman (2006) a engenharia de software é uma aplicação de uma abordagem sistemática, disciplinada e quantificável, para o desenvolvimento, operação e manutenção de software, ou seja, a aplicação da engenharia ao software.

Nesse contexto, temos a representação da engenharia de software como sendo uma tecnologia em camadas, onde a sua constituição seria composta por ferramentas, métodos, processo e qualidade, como mostra a Figura 1. A camada de processo é considerada o alicerce (PRESSMAN, 2006).

Figura 1 - Camadas de engenharia de software (PRESSMAN, 2006).

2.1.1 Processo de Software

O processo de software apresenta uma composição de atividades com a finalidade de obter um produto de software com qualidade (SOMMERVILLE, 2003).

Segundo Pressman (2006), embora existam muitos processos de software diferentes, existem atividades que são comuns e essenciais a todos eles, como:

- Comunicação;
- Planejamento;
- Modelagem;
- Construção;
- Implantação.

A atividade de comunicação tem como finalidade a troca de informações iniciais entre clientes, fornecedores e demais partes envolvidas no processo de software, para que assim possa ser feito o levantamento dos requisitos do projeto, bem como o estabelecimento de um contrato de desenvolvimento.

A atividade de planejamento busca estabelecer um plano a fim de realizar o objetivo esperado, bem como a entrega do produto de acordo com o que foi estipulado juntamente com o cliente. Para que isso possa acontecer são utilizadas algumas técnicas com o intuito de definir um roteiro.

A atividade de modelagem segundo Sommerville (2003) representa a abstração de um sistema como um conjunto de componentes e das suas relações, normalmente ilustrando graficamente em um modelo de arquitetura de sistema, proporcionando ao leitor uma visão geral da organização do sistema.

Em um processo de software as implementações, correções, melhorias ou testes acontecem na atividade de construção do mesmo. Pressman (2006) define

essa atividade como a combinação da geração de código, seja ela manual ou automática, e os testes necessários para identificar os erros no código. Ele também relata que a codificação pode ser:

> "A criação direta de código-fonte em uma linguagem de programação; a geração automática de código-fonte usando uma representação intermediária análoga ao projeto do componente a ser construído; e a geração automática de código executável usando uma linguagem de programação de quarta geração (por exemplo, Visual C++)".

A implementação do software é efetuada, utilizando-se um paradigma de programação que seja mais adequado à solução do problema. Há vários paradigmas de programação tais como, Imperativo e Orientado a objetos.

A outra parte retratada na atividade de construção são os testes. Eles são responsáveis por encontrar erros que durante a programação não foram encontrados, como também tem um papel de aumentar a qualidade do produto final.

A última parte do processo de software, a implantação, é colocada por Pressman (2006) como: "O software (como uma entidade completa ou como um incremento parcialmente efetivado) é entregue ao cliente, que avalia o produto entregue e fornece o *feedback*, baseado na avaliação".

2.2 PARADIGMAS DE PROGRAMAÇÃO

Para Thompson (1999), paradigmas de programação são formas ou técnicas de representarmos situações do mundo real ou imaginário usando um computador, cada uma delas fornecendo diferentes ferramentas para construir modelos e permitir pensar nos problemas de diversas maneiras, resolvendo – os da forma mais adequada.

Robert (2005) reforça a definição citada, afirmando que os paradigmas de programação possibilitam a resolução de vários tipos de problemas utilizando certas abstrações para isso. Um exemplo disto poderia ser a relação do foco de um programador funcional e um de orientação a objetos, o primeiro seria voltado ao relacionamento entre as funções enquanto o outro nos objetos (ROBERT, 2005).

De acordo com Sebesta (2011) as linguagens de programação são

basicamente divididas em quatro paradigmas:

- Imperativas;
- Funcionais;
- Lógicas;
- Orientadas a objetos.

O paradigma imperativo pode ser considerado do ponto de vista coletivo como uma progressão de desenvolvimentos, os quais foram projetados para usar de maneira eficiente a arquitetura de Von Neumann. Esta forma de representação é considerada aceitável pela maioria dos programadores, porém esta forte dependência da arquitetura o torna limitado no desenvolvimento de alguns processos de software. São Exemplos de linguagens imperativas: Java (ORACLE, 2014), C# (MICROSOFT, 2014), Cobol, Delphi (BORLAND, 2014), Pascal, Fortran (ABSOFT, 2014), entre outras (SEBESTA, 2011).

O paradigma funcional é baseado em funções matemáticas. Na sua forma pura, não usam variáveis ou sentenças de atribuição para produzir resultados, para isso são usadas aplicações funcionais, expressões condicionais, recursão e formas funcionais. São exemplos de linguagens funcionais: Racket, Haskell, LISP, COMMON LISP, Scheme, entre outras (SEBESTA, 2011).

No paradigma lógico é usada a lógica simbólica para resolver os problemas, por isso é conhecida como linguagem declarativa. Sua semântica apresenta uma maneira simples de determinar o significado de cada sentença, independentemente de como isso poderia ser usado para resolver um problema. Programas na linguagem lógica não descrevem exatamente como um resultado será obtido, mas a forma do resultado em si. A linguagem mais conhecida deste paradigma é o Prolog (SEBESTA, 2011).

O paradigma orientado a objetos é baseado na composição e interação de diversas unidades de softwares denominados objetos. O funcionamento de um software orientado a objetos se dá através do relacionamento e troca de mensagens entre esses objetos. Esses objetos são instâncias de classes, e nessas classes os comportamentos são chamados de métodos e os estados possíveis da classe são chamados de atributos. Nos métodos e nos atributos também são definidas as formas de relacionamento com outros objetos. São exemplos de linguagens

orientadas a objetos: Java (ORACLE, 2014), Smalltalk, C++, Ruby, C# (MICROSOFT, 2014), Ada95, entre outras (SEBESTA, 2011).

As classes são modelos que definem a forma de um objeto, representam a estrutura de um código orientado a objetos (SCHILDT, 2013).

Na orientação a objetos os métodos equivalem às rotinas presentes nos programas convencionais. Esses métodos são representados por trechos de códigos identificados de um programa, com início e fim bem definidos, que podem ser chamados de outros pontos deste programa (PINHEIRO, 2006).

Os atributos são variáveis declaradas dentro das classes, sendo usados para o armazenamento de valores e caracterizados por definir o estado de uma classe. Cada atributo corresponde a uma variável de instância ou, se o atributo é estático variável de classe (PINHEIRO, 2006). Fazendo uma diferenciação com as variáveis locais de um programa, os atributos são declarados fora de qualquer método, enquanto a variáveis locais são declaradas no corpo de um método (PINHEIRO, 2006).

Segundo Schildt (2013) a orientação a objetos deve apresentar três recursos chave:

- Encapsulamento;
- Herança;
- Polimorfismo.

O encapsulamento é representado por um mecanismo de programação que realiza a vinculação entre o código e os dados que ele trata, mantendo ambos seguros contra a interferência e má utilização externa (SCHILDT, 2013).

A herança é o processo pelo qual um objeto pode adquirir as propriedades de outro objeto. A importância da sua utilização é dar suporte ao conceito de classificação hierárquica. Assim sem o uso de herança, cada objeto teria que definir todas as suas características (SCHILDT, 2013).

O polimorfismo é a qualidade que permite que uma interface acesse uma classe geral de ações. A ação específica é determinada pela natureza exata da situação. A partir do polimorfismo é possível projetar uma interface genérica para um grupo de atividades relacionadas (SCHILDT, 2013).

2.3 PROGRAMAÇÃO DEFENSIVA

A programação defensiva entre outros fatores tem o intuito de permitir aos programadores minimizar a quantidade de erros. Através do emprego de técnicas de programação avançadas, é possível reduzir-se significativamente a quantidade de erros citados anteriormente. A programação defensiva visa à detecção direta de defeitos, ou seja, busca interceptar o defeito exatamente quando é causado (SAVIDIS, 2004).

Basicamente a programação defensiva é um nome genérico atribuído às diversas alternativas de projeto que podem ser implementadas para sistemas, visando à integridade de segurança do software. Pode ser tratada como programar pensando em todas as possibilidades que podem afetar um determinado programa, para que assim, todas as situações possam ser abordadas, antes de partir para a codificação de fato (SECALL, 2007).

A partir do momento que a aplicação da tecnologia de objetos, principalmente a linguagem Java, se tornou usual, surgiu um grande número de programas mal projetados. Tal fato levou os programadores a utilizarem um conjunto crescente de técnicas para melhorar a integridade estrutural e o desempenho de programas de software (FOWLER, 2004). Essas técnicas são chamadas de boas práticas de programação. Elas buscam o desenvolvimento de softwares de maneira mais segura e confiável, visando uma maior qualidade. Entre as diversas práticas existentes podemos citar a utilização de classes reduzidas e métodos com um número pequeno de parâmetros, entre outros.

2.4 BOAS PRÁTICAS DE PROGRAMAÇÃO

No desenvolvimento de softwares é aconselhável realizar o planejamento e projeto antes de realizar a codificação de fato. Partindo deste princípio as boas práticas de programação são utilizadas para auxiliar em um desenvolvimento de um produto com uma maior garantia de qualidade, reduzindo a refatoração de código (FOWLER, 2004).

Assim as boas práticas de programação podem ser consideradas um conjunto

de técnicas e métodos com intuito de obtenção de um código limpo. O código limpo pode ser definido de várias maneiras, porém em síntese seria um código simples, elegante e eficiente, facilitando a manutenção e entendimento deste código (MARTIN, 2009).

2.5 PADRÕES DE PROJETO

Os padrões de projeto são aplicados principalmente em linguagens orientadas a objetos, demonstrando de forma objetiva como utilizar as opções que este paradigma proporciona. Esses padrões buscam solucionar diferentes deficiências que o código ou arquitetura do sistema venha a ter (FERREIRA, 2012).

Padrão de projeto pode ser definido como um conjunto composto por um contexto, um problema e uma solução. Em outras palavras, pode-se descrever um padrão como uma solução para um determinado problema em um contexto. Um padrão não descreve qualquer solução, mas uma solução que já tenha sido utilizada com sucesso em mais de um contexto de forma consolidada (GUERRA, 2013).

Segundo Gof (*Gang of Four*) criadora dos padrões de projeto, são vinte e três, os padrões de projetos. Eles são divididos em padrões de criação, estruturais e comportamentais. Os padrões de projeto são: *Abstract Factory, Factory Method, Singleton, Builder, Prototype, Decorator, Proxy, Adapter, Façade, Bridge, Composite, Flyweight, Strategy, Observer, Chain of Responsibility, State, Template Method, Command, Interpreter, Iterator, Mediator, Memento e Visitor* (GAMMA, 1995).

O uso desses padrões é considerado uma boa prática de programação devido à (RESEARCH, 2009):

- Aumentar a produtividade do desenvolvedor;
- Acelerar o planejamento de soluções;
- Acelerar o desenvolvimento (codificação) de soluções;
- Acelerar o teste de aplicações reduzindo re-testes;
- Acelerar a inclusão de novas funcionalidades;
- Reduzir o custo de manutenção e Atualização;
- Aumentar a performance da aplicação.

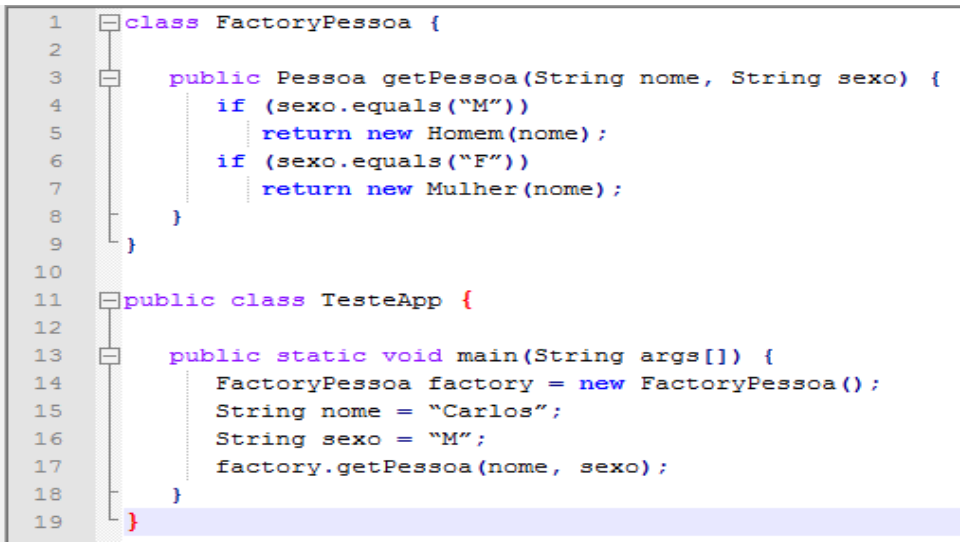
Como existe a divisão dos padrões de projeto em padrões de criação,

estruturais e comportamentais, abaixo serão abordados dois padrões de projeto de cada categoria, considerando como critério de seleção a maior experiência de uso no desenvolvimento de sistemas pelo autor da pesquisa.

2.5.1 Factory Method

Este padrão tem como objetivo a definição uma interface para criação de objetos, permitindo que as subclasses decidam quais classes instanciar. O *Factory Method* permite delegar a instanciação para as subclasses (GUERRA, 2013).

Na figura 2, temos uma classe chamada FactoryPessoa responsável pela criação de uma pessoa, permitindo que a subclasse TesteApp decida quais pessoas instanciar, dependendo dos seus parâmetros de entrada.



```
1  class FactoryPessoa {
2
3      public Pessoa getPessoa(String nome, String sexo) {
4          if (sexo.equals("M"))
5              return new Homem(nome);
6          if (sexo.equals("F"))
7              return new Mulher(nome);
8      }
9  }
10
11 public class TesteApp {
12
13     public static void main(String args[]) {
14         FactoryPessoa factory = new FactoryPessoa();
15         String nome = "Carlos";
16         String sexo = "M";
17         factory.getPessoa(nome, sexo);
18     }
19 }
```

Figura 2 - Exemplo da implementação do padrão de projeto *Factory Method* (MEDEIROS, 2014).

Segundo Guerra (2013) este padrão pode ser usado quando:

- Uma classe não pode antecipar qual o tipo de objeto precisa ser criado;

- Uma classe quer que suas subclasses especifiquem os objetos a serem criados;

- Classes delegam responsabilidade para um ou várias subclasses e assim é necessário encontrar quais subclasses tem o conhecimento necessário para então

delegar.

2.5.2 Singleton

Esta técnica tem como intenção certificar que uma classe possua apenas uma instância. Provendo um ponto global de acesso a ela (FERREIRA, 2012).

Na figura 3, temos a classe Singleton a qual possui a propriedade estática uniqueInstance do tipo da classe ‘Singleton, está por sua vez será instanciada somente uma vez.

```
1  public class Singleton {
2      private static Singleton uniqueInstance;
3
4      private Singleton() {
5      }
6
7      public static synchronized Singleton getInstance() {
8          if (uniqueInstance == null)
9              uniqueInstance = new Singleton();
10
11         return uniqueInstance;
12     }
13 }
```

Figura 3 - Exemplo da implementação do padrão de projeto *Singleton* (MEDEIROS, 2014).

Segundo Ferreira (2012) este padrão pode ser usado quando:

- Deva existir exatamente uma instância de uma classe, e essa deve ser acessível a todos os clientes através de um ponto de acesso conhecido;

- A instância única deva ser estendida por uma subclasse e os clientes devam ter acesso a ela, sem precisar alterar o código.

2.5.3 Proxy

Tem como princípio prover um substituto para que objetos tenham acesso aos

controles de outro objeto (FERREIRA, 2012).

Na figura 4, temos a classe ImageProxy a qual possui um construtor, aonde é alimentada a propriedade imageUrl do tipo da classe URL, permitindo que a classe ImageProxy também tenha acesso a classe URL.

```
1   class ImageProxy implements Icon {
2      ImageIcon imageIcon;
3      URL imageURL;
4      Thread retrievalThread;
5      boolean retrieving = false;
6
7      public ImageProxy(URL url) { imageURL = url; }
8
9      public int getIconWidth() {
15
16      public int getIconHeight() {
22
23     public void paintIcon(final component c, Graphics g, int x, int y) {
24        if (imageIcon != null)
25           imageIcon.paintIcon(c, g, x, y);
26        else {
27           if (!retrieving) {
28              retrieving = true;
29              retrievalThread = new Thread(new Runnable() {
30                 public void run () {
31                    try {
32                       imageIcon = new ImageIcon(imageURL, "CD Cover");
33                       c.repaint();
34                    }
35                    catch (Exception e) {
36                       e.printStackTrace();
37                    }
38                 }
39              });
40              retrievalThread.start();
41           }
42        }
43     }
44  }
```

Figura 4 - Exemplo da implementação do padrão de projeto Proxy (FREEMAN; FREEMAN; SIERRA, 2009).

De acordo com Ferreira (2012) este padrão pode ser usada quando:

- Uma classe remota possui uma representação local de um objeto;

- Uma classe que possa instanciar objetos de forma mais eficiente;

- Uma classe que precise de maiores controles sobre seus métodos, a fim de evitar problemas com utilização errada;

- Uma classe que compense acrescentar ações adicionais, como apenas carregar na memória objetos que foram referenciados.

2.5.4 Façade

A intenção deste padrão é a criação de uma interface unificada para um conjunto de interfaces de um subsistema. Na figura 5, a interface unificada seria a classe ComputadorFacade e as interfaces do subsistema seriam as classes Cpu, Memória e HardDrive. Assim a classe ComputadorFacade para ser construída dependeria de todas as interfaces do subsistema supracitadas. O Façade define uma interface de alto nível que facilita o uso de subsistemas (FERREIRA, 2012).

```
1   public class Cpu {
15
16  public class Memoria {
24
25  public class HardDrive {
33
34  public class ComputadorFacade {
35      private Cpu cpu = null;
36      private Memoria memoria = null;
37      private HardDrive hardDrive = null;
38
39      public ComputadorFacade(Cpu cpu, Memoria memoria, HardDrive hardDrive) {
40          this.cpu = cpu;
41          this.memoria = memoria;
42          this.hardDrive = hardDrive;
43      }
44
45      public void ligarComputador() {
46          cpu.start();
47          String hdBootInfo = hardDrive.read(BOOT_SECTOR, SECTOR_SIZE);
48          memoria.load(BOOT_ADDRESS, hdBootInfo);
49          cpu.execute();
50          memoria.free(BOOT_ADDRESS, hdBootInfo);
51      }
52  }
```

Figura 5 - Exemplo da implementação do padrão de projeto Façade (MEDEIROS, 2014).

De acordo com Ferreira (2012) este padrão pode ser usado:

- Quando se quer usar uma interface simples para utilizar um subsistema complexo;

- Quando existirem muitos clientes diferentes precisando utilizar o mesmo subsistema.

2.5.5 Strategy

Define uma família de algoritmos, encapsula cada um e os faz trocáveis, como mostra a Figura 6. O *Strategy* permite algoritmos variar independentemente dos clientes que o utilizam (FERREIRA, 2012).

```
1  public abstract class AtendenteStrategy {
2      public String getLogin() {
5      public HashMap<Integer, String> getCallCenters(){
8      public HashMap<Integer, String> getIlhas(){
11     public HashMap<String, String> getSegmentos(){
14     public Calendar getInicioVigencia(){
17     public String getOrigemCadastro(){
20     public abstract Object saveParameter();
21 }
22
23 public class AtendenteIlhaStrategy extends AtendenteStrategy {
24     public Object [] saveParameter() {
25         params = new Object[5];
26         params[0] = getCallCenter();
27         params[1] = getLogin();
28         params[2] = getInicioVigencia();
29         params[3] = getOrigemCadastro();
30         params[4] = getIlha();
31         return params;
32     }
33 }
34 public class AtendenteSegmentoStrategy extends AtendenteStrategy {
35     public Object [] saveParameter() {
36         params = new Object[6];
37         params[0] = getCallCenter();
38         params[1] = getLogin();
39         params[2] = getInicioVigencia();
40         params[3] = getOrigemCadastro();
41         params[4] = getIlha();
42         params[5] = getSegmento();
43         return params;
44     }
45 }
```

Figura 6 - Exemplo da implementação do padrão de projeto *Strategy* (MEDEIROS, 2014).

Para Ferreira (2012) esta técnica pode ser utilizada quando:

- Várias classes relacionadas diferem apenas em seus comportamentos;

- São necessários diferentes variantes de um algoritmo. Os Strategies podem ser usados quando essas diferenças são implementadas como uma hierarquia de classes de algoritmos;

- Um algoritmo usa dados que clientes não querem saber. Assim o padrão *Strategy* evita geração de algoritmos complexos para atender estruturas de dados específicas;

- Uma classe define muitos comportamentos, e esses são executados seguindo várias condições de operação.

2.5.6 Template Method

Responsável por definir a sequência de passos de um algoritmo para suas subclasses, como mostra a Figura 7. O *Template Method* permite que as subclasses refinem certos passos sem mudar a estrutura do algoritmo (GUERRA, 2013).

```
1  public abstract class Treinos {
2      final void treinoDiario() {
3          preparoFisico();
4          jogoTreino();
5          treinoTatico();
6      }
7
8      abstract void preparoFisico();
9      abstract void jogoTreino();
10     final void treinoTatico() {
11         System.out.println("Treino Tatico");
12     }
13 }
14
15 class TreinoNoMeioDaTemporada extends Treinos {
16     void preparoFisico() {
17         System.out.println("Preparo Fisico Intenso.");
18     }
19     void jogoTreino() {
20         System.out.println("Jogo Treino com Equipe Reserva.");
21     }
22 }
23 class TreinoNoInicioDaTemporada extends Treinos {
24     void preparoFisico() {
25         System.out.println("Preparo Fisico Leve.");
26     }
27     void jogoTreino() {
28         System.out.println("Jogo Treino com Equipe Junior.");
29     }
30 }
```

Figura 7 - Exemplo da implementação do padrão de projeto *Template Method* (MEDEIROS, 2014).

A princípio Guerra (2013) que o uso deste padrão pode ocorrer:

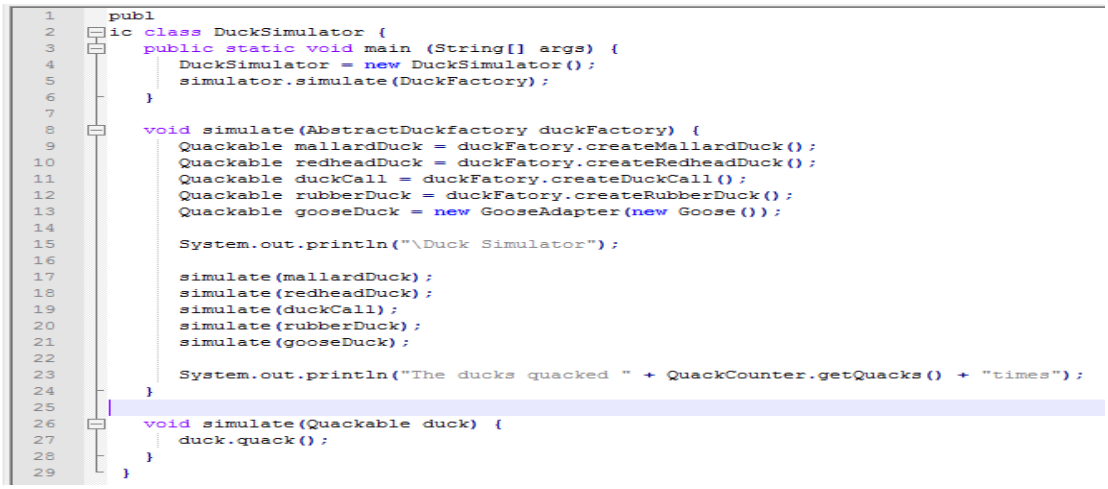
- Para implementar uma única vez partes de um algoritmo que nunca variam, e deixar para subclasses a implementação dos comportamentos que variam;

- Quando comportamentos comuns das subclasses devam ser produzidos e localizados em uma classe comum, evitando duplicação de código;

- Para controlar extensões de subclasses. É possível definir um *Template Method* que chama uma "operação gancho" (*hook operation*) em pontos específicos, controlando os pontos de extensão.

2.5.7 Padrões de projetos compostos

Um conjunto de padrões de projeto trabalhando juntos em um design que pode ser aplicado a diversos problemas é denominado um padrão composto, como mostra a Figura 8. Esses padrões de projeto combinados podem resolver problemas repetitivos ou genéricos. Sendo considerada uma boa alternativa na resolução de certas problemáticas (FREEMAN; FREEMAN; SIERRA, 2009).



```
publ
ic class DuckSimulator {
    public static void main (String[] args) {
        DuckSimulator = new DuckSimulator();
        simulator.simulate(DuckFactory);
    }

    void simulate(AbstractDuckfactory duckFactory) {
        Quackable mallardDuck = duckFatory.createMallardDuck();
        Quackable redheadDuck = duckFatory.createRedheadDuck();
        Quackable duckCall = duckFatory.createDuckCall();
        Quackable rubberDuck = duckFatory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\Duck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " + QuackCounter.getQuacks() + "times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Figura 8 - Exemplificação do uso de dois padrões de projetos (FREEMAN; FREEMAN; SIERRA, 2009).

Ainda que esses padrões possam ser usados em conjunto, nem sempre se deve aplicá-los, às vezes a solução para o problema é mais simples e pode ser utilizada outra resolução como a Orientação a Objetos (FREEMAN; FREEMAN; SIERRA, 2009).

3. CONCLUSÃO E RECOMENDAÇÕES

Neste capítulo são apresentados a conclusão e recomendações do estudo proposto.

3.1. CONCLUSÃO

Considerando a atual dependência da sociedade em relação ao uso dos softwares para realizar as suas tarefas diárias, tem aumentado a exigência pela qualidade desses produtos. Nesse contexto, a atividade de programação merece uma atenção especial, sendo de grande importância a utilização das boas práticas de programação para obter este resultado.

A contribuição que este trabalho proporciona em nível cientifico é o estudo das boas práticas de programação levantadas durante a pesquisa. Em nível prático, os resultados da pesquisa possibilitam subsidiar as decisões de gerentes ou líderes de projetos de software em relação a definição de boas práticas como padrões de programação, ou no auxílio a soluções de problemas existentes relacionados a programação de sistemas.

Assim, com a realização desses estudos e seus resultados, espera-se ter contribuído para destacar a importância da utilização das boas práticas de programação usando os padrões de projetos, afim de proporcionar uma programação defensiva para qualidade de software.

3.2. RECOMENDAÇÕES

Com base no estudo realizado, recomenda-se aos desenvolvedores de software que adotem boas práticas de programação, com intuito de gerir um produto de maior qualidade.

Utilizar as boas práticas de programação é fundamental dentro de uma empresa, para que esta desenvolva um produto de maior qualidade, minimizando os

custos e tempo.

REFERÊNCIAS BIBLIOGRÁFICAS

ABSOFT. **Fortran**. Disponível em: <http://www.sia.com.br/absoft.htm>. Acesso em: 13 de maio 2014.

BORLAND. **Delphi**. Disponível em: <http://www.borland.com>. Acesso em: 13 de maio 2014.

FERREIRA, Isaias Alves. **Análise do Impacto da Aplicação de Padrões de Projeto na Manutenibilidade de um Sistema Orientado a Objetos.** 2012. 78 f. Monografia (Especialização) - Curso de Sistemas de Informação, Universidade Federal de Lavras, Lavras, 2012.

FREEMAN, Eric; FREEMAN, Elisabeth; SIERRA, Kathy. **Use a Cabeça! Padrões de Projetos (Design Patterns).** 2. ed. Rio de Janeiro: Alta Books, 2009.

FOWLER, Martin. **Refatoração: Aperfeiçoando o Projeto de Código Existente.** Porto Alegre: Bookman, 2004.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software.** Portland: Addison-Wesley, 1995.

GUERRA, Eduardo. **Design Patterns com Java: Projeto orientado a objetos guiado por padrões.** São Paulo: Casa do Código, 2013.

MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Agile Software.** São Paulo: Bookman, 2009.

MICROSOFT. **C#**. Disponível em: <http://www.microsoft.com/pt-br/default.aspx>. Acesso em: 13 de maio 2014.

MICROSOFT. Disponível em: <http://racket-lang.org/> Acessado em: 13 de maio 2014.

MEDEIROS, Higor. **Estudo e Aplicação do Padrão de Projeto Strategy**. Disponível em: <http://www.devmedia.com.br/estudo-e-aplicacao-do-padrao-de-projeto-strategy/25856>. Acesso em: 30 out. 2014.

MEDEIROS, Higor. **Padrão de Projeto FactoryMethod em Java**. Disponível em: <http://www.devmedia.com.br/padrao-de-projeto-factory-method-em-java/26348>. Acesso em: 30 out. 2014.

MEDEIROS, Higor. **Padrão de Projeto Facade em Java**. Disponível em: <http://www.devmedia.com.br/padrao-de-projeto-facade-em-java/26476>. Acesso em: 30 out. 2014.

MEDEIROS, Higor. **Padrão de Projeto Singleton em Java**. Disponível em: <http://www.devmedia.com.br/padrao-de-projeto-singleton-em-java/26392>. Acesso em: 30 out. 2014.

MEDEIROS, Higor. **Padrão de Projeto TemplateMethod em Java**. Disponível em: <http://www.devmedia.com.br/padrao-de-projeto-template-method-em-java/26656>. Acesso em: 30 out. 2014.

ORACLE. **Code Conventions for the Java TM Programming Language**.Disponível em: <http://www.oracle.com/ztechnetwork/java/ codeconvtoc-

136057.html> ORACLE. Java SE Specifications. Acesso em: 17 de Set. 2014.

ORACLE. **Java**. Disponível em: <http://www.oracle.com/ index.html> Acesso em: 13 de maio 2014.

PINHEIRO, Francisco A. C. **Fundamentos de Programação e Orientação a Objetos Usando Java.** Rio de Janeiro: Ltc, 2006.

PRESSMAN, Robert S. **Engenharia de Software.** 6. Ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, Robert S. **Engenharia de Software Uma Abordagem Profissional.**7. Ed. São Paulo: McGraw-Hill, 2006.

RESEARCH, Necleus. **Microsoft Patterns and Practices.** Boston: NucleusResearchInc, 2009.

ROBERT, Floyd.**The paradigms of programming**.Resonance, 2005, Vol.10(5), pp.86-98.

SAVIDIS, Anthony. **The Implementation Of Generic Smart Pointers For Advanced Defensive Programming.** Greece, 21 maio 2004.

SCHILDT, Herbert. **Java para iniciantes.** Porto Alegre: Bookman, 2013.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação.**9. Ed. Porto Alegre: Bookman, 2011.

SECALL, Jorge Martins. **Avaliação comparativa do impacto do emprego de técnicas de programação defensiva na segurança de sistemas críticos.** 2007. 201 f. Tese (Doutorado) - Curso de Engenharia Elétrica, Escola Politécnica da Universidade de São Paulo, São Paulo, 2007.

SOMMERVILLE, Ian S. **Engenharia de Software.** 6. Ed. São Paulo: Addilson Wesley, 2003.

THOMPSON, Simon.**Haskell the Craft of Functional Programming**. 2 Ed. Addison-

Wesley.[S.l.], 1999.487p. ISBN 0201342758.