PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

MICHAEL SANTAGUIDA

ARQUITETURA EVOLUTIVA E DESIGN EMERGENTE

São Paulo, dezembro de 2010

MICHAEL SANTAGUIDA

ARQUITETURA EVOLUTIVA E DESIGN EMERGENTE

Monografia apresentada ao Curso de Especialização em Engenharia de Software da Pontifícia Universidade Católica de São Paulo, como requisito parcial para obtenção do título de Especialista em Engenharia de Software, orientado pelo Prof. MSc. José Paulo Papo.

São Paulo, dezembro de 2010

ARQUITETURA EVOLUTIVA E DESIGN EMERGENTE

MICHAEL SANTAGUIDA

Monografia aprovada em sua versão final pelos abaixo assinados:

Prof. MSc. José Paulo Papo
Orientador

Prof. Dr. Carlos Eduardo de Barros Paes
Coordenador do Curso de Especialização em Engenharia de Software

Dedico este trabalho a todos os profissionais da área de Tecnologia de Informação que atuaram comigo no decorrer desses seis anos e que de alguma maneira contribuiram para o meu crescimento profissional.

Agradecimentos

Agradeço a todos que de alguma maneira me deram forças para que eu conseguisse concluir este trabalho com sucesso.

Agradeço primeiramente a Deus.

Agradeço ao professor José Papo por ter sido meu orientador e me apoiado sempre que eu precisei.

Agradeço ao meu chefe Wagner Caseiro pelo apoio e por ter permitido que eu fizesse este trabalho nos horários vagos do meu expediente.

Agradeço a todos os familiares e amigos pelo apoio e pela compreensão.

"Não acredite em algo simplesmente porque ouviu. Não acredite em algo simplesmente porque todos falam a respeito. Não acredite em algo simplesmente porque está escrito em seus livros religiosos. Não acredite em algo só porque seus professores e mestres dizem que é verdade. Não acredite em tradições só porque foram passadas de geração em geração. Mas depois de muita análise e observação, se você vê que algo concorda com a razão, e que conduz ao bem e beneficio de todos, aceite-o e viva-o."

Buda

Resumo

O desenvolvimento de uma arquitetura de sistema de software que seja flexível a mudanças, a princípio pode parecer uma tarefa complicada, pois as decisões relacionadas à arquitetura estão entre as mais cruciais em um projeto de sistema de software, onde uma mudança de requisitos não-funcionais durante o projeto pode gerar grandes impactos.

O objetivo desta monografia é mostrar que é possível desenvolver uma arquitetura evolutiva, capaz de atender as mudanças que surgem no decorrer do projeto, através da elaboração de um design de qualidade, que faça com que o impacto das mudanças seja mínimo.

A monografia está divida da seguinte maneira: no capítulo 1 será abordada arquitetura de sistema, o que é arquitetura de sistema, e a evolução da arquitetura de sistema de como é tradicionalmente para uma arquitetura evolutiva.

No capítulo 2, será explicado melhor o que é um design emergente com alguns exemplos.

No capítulo 3, serão explicadas as qualidades de código: encapsulamento, coesão, acoplamento, redundância, legibilidade e testabilidade; será explicada a importância destas qualidades no código do sistema, juntamente com alguns exemplos.

No capítulo 4, serão mostrados alguns princípios de design de software, que permitem melhorar a qualidade do design do sistema, juntamente com alguns exemplos.

No capítulo 5, serão mostrados os design patterns, o que são, quais os seus benefícios, como implementá-los, juntamente com alguns exemplos.

No capítulo 6, será mostrado à técnica de Test-Driven Development, como utilizá-la passo-a-passo com exemplo, e seus benefícios para um design emergente.

E para finalizar, no capítulo 7, serão mostradas as técnicas de refatoração, que permitem modificar um código com problemas de qualidade para um código de qualidade sem afetar o comportamento externo do sistema.

Abstract

The development of a software system architecture that is flexible to changes, at first, might seem like a complicated task. The reason is because decisions related to architecture are amongst the most crucial in a system software project, where a change of non-functional requirements during the project can generate large impacts.

The purpose of this monograph is to show that you can develop an evolutionary architecture, able to meet the changes that arise during the project, by developing a design quality that makes the impact of change minimal.

This monograph is divided as follows: In chapter 1, System Architecture will be addressed by showing what it is and its evolution from traditional to evolutionary architecture.

In chapter 2, a better explanation of what an emerging design is with some examples will be given.

In chapter 3, the following code qualities will be explained; encapsulation, cohesion, coupling, redundancy, readability and testability. Its importance to the system code will also be explained along with some examples.

In chapter 4, some principles of software design and how it improves its quality will be shown along with some examples.

In chapter 5, design patterns will be analyzed in terms of what they are, its benefits and how to implement them together along with some examples.

In chapter 6, the techniques for Test-Driven Development will be shown along with a step-by-step explanation of how to use it and its benefits for an emergent design.

To finalize, in chapter 7, the techniques of refactoring, which allows changes to the code with quality problems to a good quality code without affecting the external behavior system, will be shown and explained.

Sumário

Introdução

Em projetos de sistemas de software, as mudanças têm aparecido com mais freqüência, e isso tem sido um dos motivos de fracasso de muitos projetos, o que ocasiona grandes prejuízos para as empresas. É importante estar preparado para as mudanças que possam ocorrer, pois nos dias modernos estas são inevitáveis e isso se deve a inúmeros fatores, como: concorrência, time-to-market, novas legislações e etc.

Os profissionais que seguem as metodologias ágeis têm levado em conta essa preocupação, de trabalhar de uma maneira que possam conduzir o trabalho de desenvolvimento de um sistema de software com mais dinamismo e mais preparado para atender as mudanças que possivelmente irão aparecer no decorrer do projeto, trabalhando de uma maneira iterativa e incremental, dividindo o projeto em projetos menores com escopos menores.

Em se tratando de um projeto de sistema de software dinâmico e aberto para novas mudanças, é importante que o sistema a ser desenvolvido tenha uma arquitetura mais flexível, capaz de atender as mudanças e sem gerar impactos negativos para o projeto, ou seja, uma arquitetura evolutiva. É nesse ponto que começa o problema.

Como fazer uma arquitetura evolutiva, se as decisões arquiteturais são as decisões mais cruciais em um projeto de sistema de software, e as mais difíceis de modificar, devido ao tamanho do impacto que pode gerar?

A resposta é elaborar um bom design, um design emergente, em outras palavras, um design de qualidade, onde se é possível realizar mudanças tendo um mínimo de impacto. Nesta monografia, será mostrado o que se deve fazer para se ter uma arquitetura evolutiva baseada em um design emergente.

1. Arquitetura de Sistema

1.1. O que é a Arquitetura de Sistema?

Antes de tudo é importante esclarecer o conceito de Arquitetura de Sistemas. Sempre houve diversas definições e entendimentos sobre o que é arquitetura. Fowler (2002) define: “o termo arquitetura envolve a noção dos principais elementos do sistema, as peças que são difíceis de mudar. Uma fundação na qual o resto precisa ser construído”. Já Carnegie (2010) apresenta diversas definições, dentre as clássicas: “Arquitetura significa uma estrutura de sistema que consiste de módulos ativos, um mecanismo que permite interação entre esses módulos, e uma série de regras que governam a interação”. Johnson (Gang of Four) (1994) define: “o conjunto de decisões que devem ser tomadas no começo do projeto” e ele finaliza dizendo que “arquitetura é tudo aquilo que é importante, o que quer que seja”.

Outro assunto que também gera diferentes definições e que vem junto com arquitetura é o design, aonde muitas vezes chega a ser difícil definir a fronteira entre a arquitetura e o design do sistema de software.

> “Alguns dos padrões descritos neste livro podem razoavelmente serem considerados arquiteturais, na medida em que representam uma importante decisão; outros estão mais relacionados ao design e nos ajudam a atender aquela determinada arquitetura. Eu não faço esforços em tentar separar o que é cada um, pois o que é arquitetural ou não é, é algo muito subjetivo” (FOWLER, 2002).

> “A arquitetura de um sistema de software é o alicerce sobre o qual todo o resto fica, [...]. Os elementos de design ficam a arquitetura, [...] elementos arquiteturais são difíceis de serem movidos e substituídos, porque você terá que mover todas que estão sobre estes elementos para acomodar as mudanças.”. (FORD, 2010)

> “o framework web que você utiliza é uma decisão arquitetural porque é difícil de substituí-lo. Dentro de um framework web, ainda que, você possa utilizar diferentes design patterns para representar diferentes objetivos, o que sugere que a

> maioria dos design patterns formais são de fato parte do design e não da arquitetura.”. (FORD, 2010)

1.2. Da abordagem original para uma Arquitetura Evolutiva

Como foi dito anteriormente, a arquitetura está relacionada às decisões cruciais do projeto que serão difíceis de modificar em uma situação posterior, mas a partir daí surgem diversas questões e dúvidas, por exemplo: Como lhe dar com as mudanças que possivelmente aparecerão no decorrer do projeto?

De acordo com Bain (2008), além de outros problemas que levam um projeto a não ser concluído, um dos principais são as constantes mudanças que ocorrem no decorrer do projeto. Bain (2008) cita alguns motivos de fracasso de projetos justificados por seus respectivos alunos:

- “O cliente não sabe o que quer.”
- “O cliente sabe o que ele quer, mas não sabe como explicar.”
- “O cliente sabe o que ele quer, e sabe como explicar, mas nós o entendemos errado.”
- “O cliente sabe o que ele quer, sabe como explicar, nós o entendemos, porém perdemos o que é mais importante.”
- “Tudo dá certo, porém há mudanças no mercado que exigem mudanças no sistema.”

E quando se fala em mudanças, sabe-se que em projetos de sistema de software é impossível levantar todos os requisitos nas fases iniciais, no caso antes de se dar início ao desenvolvimento, requisitos novos sempre surgem, e uma parte dos que já existem acabam sofrendo mudanças, sendo assim, acaba sendo um erro grande erro adotar a metodologia da Cascata para a grande maioria dos projetos de software, exceto para pequenos projetos com chances mínimas de mudanças, portanto, é necessário adotar uma metodologia que trabalhe de maneira incremental e iterativa, para que se consiga atender melhor as mudanças.

Outro fator importante a ser levado em consideração, que Bain (2008) menciona, é que de acordo com um estudo do Standish Group com relação às funcionalidades de software desenvolvidas: 45% nunca são utilizadas, 19% raramente são utilizadas, 16% às vezes são utilizadas, 13% frequentemente são utilizadas e 7% são sempre utilizadas.

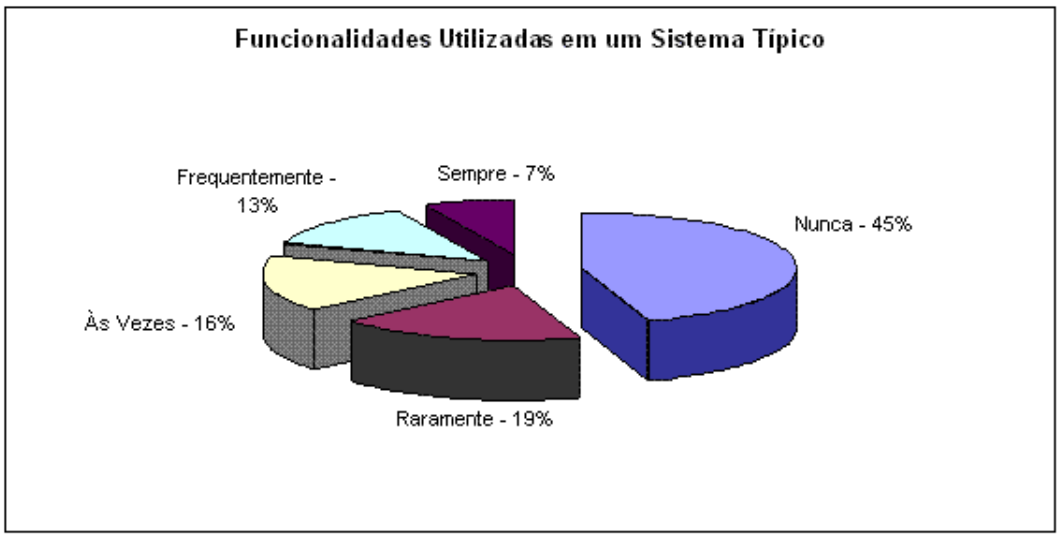

Figura 1 – Estatísticas levantadas pelo Standish Group referentes à utilização das funcionalidades de um sistema típico. (BAIN, 2008)

Isso tem sido levado em consideração pelos seguidores das metodologias ágeis, como *Scrum* e *XP*, que trabalham de maneira incremental e iterativa. E de acordo com este estudo, se verifica que é possível criar uma versão inicial entregável, e depois disso ir desenvolvendo iterativamente e incrementalmente novas versões entregáveis, fazendo com que o cliente fique satisfeito e as mudanças sejam abraçadas e adaptadas ao projeto. Para isto se tornar realidade, é necessário definir uma arquitetura flexível, que atenda os requisitos iniciais do projeto, tanto os funcionais como os não funcionais, e que possa atender mudanças gradativamente, de acordo com as necessidades do projeto.

Outra teoria antiga, que tem sido levado em consideração por alguns agilistas, e que Waters (2007) faz uma visão geral, é a Lei de Pareto, ou seja, a lei do 80/20, onde a idéia principal é que se é possível obter 80% do resultado esperado com 20% dos esforços, isto na prática seria assim, em um cenário onde haja um speed-to-market grande, se criaria um produto com algumas das funcionalidades, as mais importantes que vão agregar bastante resultado, evitando assim aquela situação de querer concluir 100% do produto para colocá-lo no

mercado, onde de fato isso acaba nunca acontecendo, e como conseqüência ou o projeto acaba fracassando ou se perde espaço no mercado para a concorrência. Waters (2007) comenta um caso bastante curioso sobre uma visita feita à Microsoft, onde os pesquisadores da Microsoft descobriram que a grande maioria dos usuários do Microsoft Word utiliza somente 8% de suas funcionalidades, claro que isto não quer dizer que se eles lançassem uma primeira versão com somente 8% das funcionalidades eles teriam o mesmo retorno ou um retorno melhor, isso acaba sendo sempre uma incógnita, porém permite se chegar à conclusão da importância de se criar uma primeira versão com algumas das funcionalidades de importância maior, e incrementalmente ir adicionando novas funcionalidades.

Ambler (2002) enumera os principais problemas existentes nas abordagens tradicionais:

- “Não existe um esforço de arquitetura de sistema”.
- “Foco destorcido”.
- “As equipes de projeto não sabem que a arquitetura de sistema existe”.
- “As equipes de projeto não seguem a arquitetura de sistema”.
- “As equipes de projeto não trabalham com o arquiteto de sistema”.
- “Arquiteturas desatualizadas”.
- “Modelos de arquiteturas estritamente focadas”.
- “Dysfunctional ‘charge back’ schemes”.
- “Atitudes do tipo ‘faça todo trabalho extra porque é bom para a empresa’”.

Baseado no ponto de vista de Ambler (2002), percebe se que as abordagens tradicionais geram muitas conseqüências negativas, tanto para a equipe quanto para o projeto, principalmente por falta de acompanhamento constante do arquiteto, que acaba não vendo as dificuldades que os programadores estão tendo, e nem as possíveis soluções que os mesmos têm a propor. Conseqüentemente acarretará também outros problemas como: falta de comunicação, falta de sincronia entre a equipe, descontentamento dos desenvolvedores, e até mesmo o fracasso do projeto.

Os profissionais seguidores de metodologias ágeis se preocupam bastante em mudar esse tipo de situação, adotando algumas posturas e boas práticas para que isto aconteça. Ambler (2002) cita algumas premissas a serem adotadas, que permite que as equipes trabalhem de maneira

ágil, tendo maior sinergia, evitando problemas como os que foram citados, e conseguindo atender melhor as expectativas do cliente:

1. "**Foco nas pessoas e não nas tecnologias e nem nas técnicas**"
   Valorizar o capital humano, o bom relacionamento, a auto-organização das equipes, de maneira que aumente a produtividade, a qualidade na escolha das tecnologias e técnicas e como conseqüência um bom resultado final, pois de nada adianta criar uma boa arquitetura, se a equipe não souber ou não conseguir tirar um bom proveito, ou se algum membro da equipe seguir do seu jeito sem nenhuma sincronia.

2. "**Mantenha as coisas simples**"
   Evitar desperdiçar tempo e verba do projeto com documentações desnecessárias, ou seja, procurar documentar o necessário que vá agregar valor para o projeto e que a equipe irá utilizar, pois isso tem sido um grande problema que vêm ocorrendo em muitos projetos de Tecnologia da Informação (TI), onde se desperdiça esforços para elaborar documentações que acabam nem sendo utilizadas, ou por não agregar, ou por ser tão elaborada que com o tempo acaba ficando desatualizada e sendo descartada.

   Ambler (2006) dá um foco maior para este tipo de problema, onde ele diz que ser agilista não é parar de escrever documentação, mas sim documentar o necessário que agregue valor e não algo que irá somente desperdiçar tempo e verba do projeto, ele cita casos como documentações feitas para ajudarem os programadores que irão dar manutenção no sistema e acabaram sendo descartadas por não terem as informações necessárias de que eles precisavam, e eles acabaram confiando mais em debulhar os códigos para entender melhor. Para cada documentação que for adotar no projeto, é ideal sempre levar em consideração algumas questões, o próprio Ambler (2006) sugere algumas:

   a. " Quem irá ler este documento?"
   b. " Como eles irão fazer uso deste documento?"
   c. " O que eles esperam ler neste documento? Em que formato?"
   d. " Quem vai pagar por este documento?"
   e. "O que realmente precisa ser dito?"
   f. " Qual é a maneira mais resumida de dizer isto?"

g. “ O que irá acontecer se o leitor precisar de mais esclarecimentos?”
h. “ Será que já existe algum documento do qual podemos fazer referência total ou parcial?”

3. “**Trabalhe iterativamente e incrementalmente**”

   Como foi visto no início deste capítulo, onde é dito para se desenvolver a princípio o mínimo necessário para uma versão entregável e que agregue valor para o negócio, e em seguida interativamente e incrementalmente ir desenvolvendo novas funcionalidades, sempre priorizando as que trazem mais retorno para o negócio.

4. **“Arregace as suas mangas”**

   Esta premissa diz respeito ao arquiteto adotar uma postura de estar junto com a equipe, programando e acompanhando o andamento do projeto, ver os beneficios e dificuldades que a equipe de projeto está tendo e poder tomar as devidas providências o quanto antes para garantir o bom andamento do desenvolvimento do projeto. Ambler (2006) cita os inúmeros benefícios, dentre eles:

   a. “Você descobre rapidamente se as suas idéias funcionam ou não.”
   b. “Você aumenta as chances do time de projeto entender a arquitetura, pois você está presente.”
   c. “ Você compartilha suas idéias e estratégias com a equipe.”
   d. “Você aumenta as chances de uma infra-estrutura, tanto técnica como de negócio ser reutilizada o tempo todo, pois o time estará trabalhando no contexto de arquitetura corporativa.”
   e. “Você adquire experiência nas ferramentas e tecnologias que o time de projeto trabalha, assim como do domínio do negócio, melhorando o seu entendimento sobre a arquitetura que você está definindo.”
   f. “Você tem a chance de receber feedbacks mais concretos para melhorar a arquitetura.”
   g. “Você ganha respeito dos seus clientes e da equipe de desenvolvimento.”
   h. “Você ativamente ajuda a desenvolver sistemas de software, o objetivo prioritário dos profissionais de TI.”

i. “Você pode ser o mentor dos desenvolvedores e dos agile Data Base Administrator (DBA) na equipe de projeto em modelagem e arquitetura, melhorando o skill deles.”
j. “Você fornece um benefício claro para a equipe, pois você está os ajudando a realizar os seus objetivos imediatos, substituindo o comportamento ‘faça todo o trabalho extra porque é bom para a empresa’ por algo mais atrativo como ‘deixe me te ajudar a alcançar os seus objetivos, para que possamos fazer algo bom para a empresa’.”

5. **“Analise o cenário como um todo”**

   Esta premissa diz que o arquiteto não deve se limitar somente aos diagramas de dados, componentes ou qualquer outro diagrama tradicional, mas também naqueles diagramas que fornecerão um bom entendimento das partes mais críticas da arquitetura, mas sem deixar de lado a premissa anterior de manter as coisas simples.

6. **“Faça a arquitetura do sistema atrativa para os seus clientes”**

   Esta premissa diz para o arquiteto fazer uma arquitetura que agrade o cliente, e que ele perceba que é algo que realmente agregue valor para ele, pois do contrário ele simplesmente achará que você estará tomando o tempo dele e passará a evitar de agendar reuniões com você.

2. Design Emergente

O Design Emergente é um conceito que está relacionado com a Arquitetura Evolutiva, da qual foi discutida anteriormente, e tem sido cada vez mais adotado pelas empresas nos projetos de software devido à adoção de metodologias ágeis, pois como as metodologias ágeis trabalham iterativamente e incrementalmente, sempre abraçando as mudanças que ocorrem no decorrer do projeto, necessita-se ter uma arquitetura mais flexível para mudanças e que cause o mínimo de impacto possível, e o intuito principal do design emergente é este, a cada mudança arquitetural que ocorrer no decorrer do projeto, gerar o mínimo possível de impacto no projeto.

Para que isso ocorra, diversas providências, que serão vistas com mais detalhes nos próximos capítulos, deverão ser tomadas. Bain (2008) mostra uma pirâmide, onde o design emergente está no topo, e nos níveis abaixo estão as disciplinas necessárias para se chegar até um design emergente:

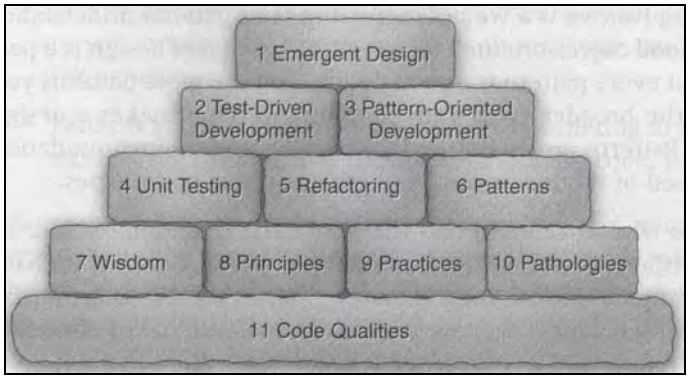

Figura 2 – Pirâmide que ilustra as disciplinas de um design emergente. (BAIN, 2008)

Em outras palavras, para se ter um design emergente, é necessário primeiramente ter um código com boa qualidade, e as principais qualidades serão vista com mais detalhes no capítulo 3 que são: encapsulamento, alta coesão, baixo acoplamento, sem redundância, legível e testável.

Além disso, existem diversas boas práticas e princípios de design a serem adotadas que garante uma boa qualidade de software, no capítulo 4 estes princípios serão vistos com mais detalhes.

Aplicar corretamente os design patterns, é fundamental para ter um design flexível, porém é importante ter um bom entendimento do domínio do negócio além de ter um bom entendimento dos requisitos do sistema, tanto os funcionais quanto os não funcionais. Os design patterns serão explicados com mais detalhes no capítulo 5.

Durante o desenvolvimento de um sistema com design emergente, é de grande valia desenvolver seguindo a abordagem Test-Driven Development, que será explicada com mais detalhes no capítulo 6, pois é uma abordagem que faz com que o desenvolvedor desenvolva códigos com mais qualidade.

Conhecer e saber aplicar as técnicas de refatoração é importante durante o desenvolvimento de um sistema com design emergente, pois permite alterar de maneira mais eficaz um código com problemas de qualidade, sem alterar o comportamento externo do sistema. No capítulo 7 será explicado com mais detalhes algumas técnicas de refatoração e como aplicá-las.

Ford (2010) mostra um exemplo de design emergente, tanto a forma incorreta, quanto a forma correta. Neste exemplo, ele mostra sobre um problema em que ocorre em muitos sistemas, que é acoplar classes de negócio às classes de um determinado framework, e isso não é nada bom, pois o sistema ficará dependente daquele framework, E caso se decida trocar de framework, haverá impactos imensuráveis. Neste caso em específico, é mostrado o que acontece normalmente quando se adota o framework java *Struts*, onde as classes de domínio herdam a classe *ActionForm* permitindo que muitas tarefas aconteçam automaticamente, como população das variáveis com os valores dos parâmetros, validações e etc. Segue abaixo uma figura com o design típico de aplicações que utilizam *Struts*:

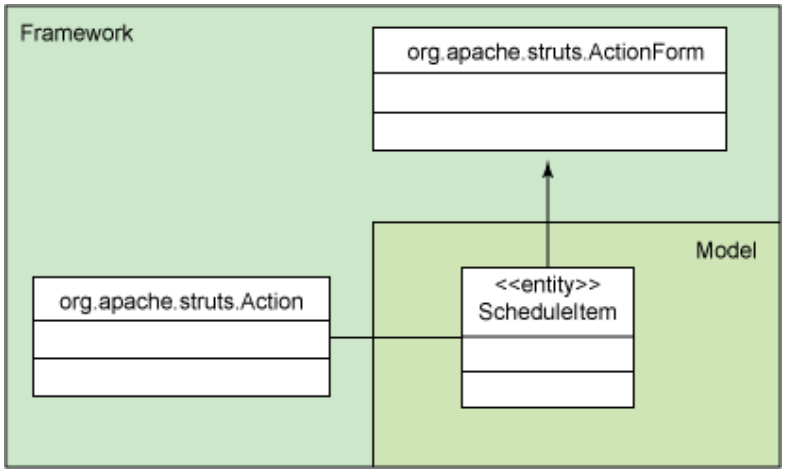

Figura 3 – Exemplo de design não-emergente em um sistema utilizando o framework *Struts* (FORD, 2010)

Para evitar futuros problemas, Ford (2010) sugere outro modelo de design, como pode ser visto na Figura 4, onde segundo ele:

> "A classe de domínio inclui uma interface que define a semântica de um item do cronograma. A *ScheduleItem* original implementa esta interface, que também é implementada pela interface *ScheduleItemForm*, forçando a semântica dessas duas classes estarem de acordo. O *ScheduleItemForm* por sua vez, possui uma instância do objeto de domínio *ScheduleItem*, e todos os acessores e modificadores de *ScheduleItemForm* passam através dos modificadores e acessores do *ScheduleItem* encapsulado. Isso permite tirar vantagens de recursos interessantes do *Struts* enquanto se mantém desacoplado do framework." (FORD, 2010)

Resumindo, a idéia é que o framework possa saber sobre o sistema, mas que o sistema não sabia sobre o framework, permitindo assim ter um design mais flexível e fazendo com que qualquer mudança arquitetural gere o mínimo possível de impacto.

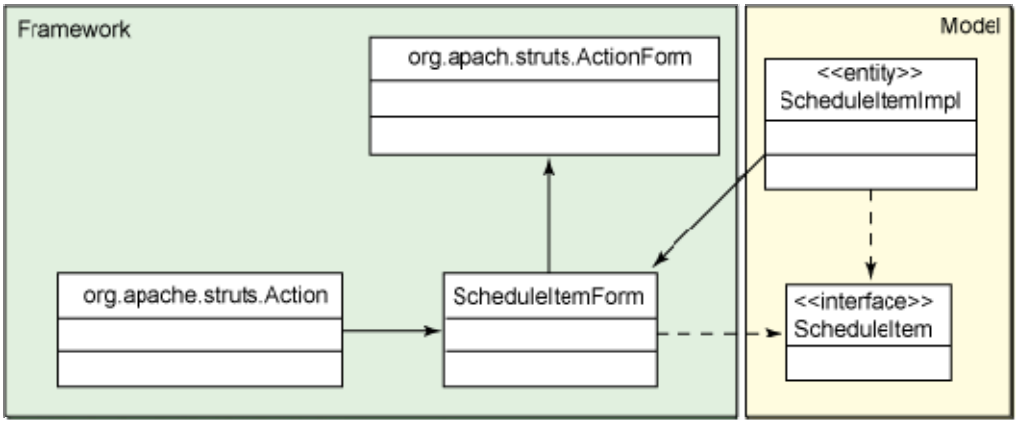

Figura 4 – Exemplo de design emergente em um sistema utilizando o framework Struts.

(FORD, 2010)

3. Qualidades de Código

Desde os primeiros projetos de software até os dias de hoje, os problemas relacionados à qualidade de códigos tem gerado muito desconforto para os programadores, principalmente na hora de dar manutenção em códigos já existentes e feitos por outros programadores, pois para se implementar uma lógica em uma determinada linguagem de programação, existem inúmeras maneiras, isso pode não ser tão ruim quando só há único programador, o projeto é pequeno e sem probabilidade de sofrer futuras mudanças, porém no mundo real isso é quase impossível de se acontecer, pois sempre existe mais de uma pessoa ali trabalhando, e em alguns casos, existem equipes muito grandes, algumas vezes chegando a ter mais de cem profissionais geograficamente distribuídos, que é o que acontece em grandes projetos de multinacionais, e os negócios são bem dinâmicos, as empresas estão sempre cada vez em busca de aumentarem seus lucros, e isso requer mudanças em suas estratégias de negócio e que consequentemente acarretará mudanças em mudanças constantes nos sistemas que estão ali para atender as necessidades dos negócios, eis que aí vem a necessidade de se desenvolver código adotando alguns padrões que garantam maior manutenibilidade, flexibilidade e extensibilidade.

O paradigma de Orientação a Objetos surgiu com o intuito de resolver diversos problemas e dificuldades existentes nas linguagens estruturadas. A adoção dos conceitos propostos pela Orientação a Objetos são de grande importância para a criação um design emergente, alguns deles serão abordados neste capítulo, como encapsulamento, coesão e acoplamento, juntamente com outros conceitos que também são de grande importância como legibilidade, testabilidade e redundância; já outros conceitos de Orientação a Objetos como herança e polimorfismo serão visto com mais detalhes nos capítulos seguintes.

3.1. Encapsulamento

Esse princípio é fundamental para se desenvolver um código que permita ter manutenibilidade e flexibilidade. O seu objetivo é esconder trechos de código de quem o utiliza, pois caso seja necessária uma manutenção, esta será feita somente em um lugar e não em dezenas de lugares, sem afetar as outras partes do sistema que fazem acesso deste trecho de código. Uma maneira de encapsulamento é através do uso de interfaces, pois se acessa os métodos de uma

interface e os detalhes de implementação destes métodos estão escondidos na classe que implementa esta interface, e caso seja necessário alterar essa implementação, a classe que acessa a interface não será afetada.

Será mostrado abaixo um exemplo utilizando uma classe Java, nas duas situações, sem e com encapsulamento:

```
public class Pessoa {
      public Integer idade;
}
```

Listagem 1 – Exemplo de código sem encapsulamento

O exemplo acima, a princípio pode estar correto, porém ao imaginar a situação seguinte, onde alguém atribui um valor de idade inválido, percebe-se o problema:

```
public class Teste {
      public static void main(String[] args) {
            Pessoa pessoa = new Pessoa();
            pessoa.idade   = -50;
      }
}
```

Listagem 2 – Exemplo de problema ocasionado pela falta de encapsulamento

Agora imagina que em todo o lugar que utilize esta classe *Pessoa*, seja necessário colocar uma regra de validação para impedir que fosse atribuída uma idade inválida. Realmente ficaria muito complicado de dar manutenção, então para evitar este tipo de problema, encapsulam-se todas as variáveis de instância de uma classe, declarando as variáveis para o tipo de acesso *private*, onde ela só pode ser acessada diretamente dentro da classe, e declara os métodos de acessos a essas variáveis como *public*, utilizando as convenções de nomenclatura *JavaBean* (*getNomeDaVariavel* e *setNomeDaVariavel*), como será visto no exemplo abaixo:

```
public class Pessoa {
      private Integer idade;

      public void setIdade(Integer idade) {
            this.idade = idade;
      }
      public Integer getIdade() {
            return this.idade;
      }
}
```

Listagem 3 – Exemplo de código encapsulado

Fazendo assim, será possível realizar eventuais mudanças, como eventuais regras de validação, dentros dos métodos de acesso, tornando a manutenção mais simples e sem afetar quem faz uso da classe *Pessoa*.

3.2. Coesão

O princípio de coesão está relacionado ao quão bem focado são os propósitos de uma classe e os seus respectivos métodos, e a idéia é ter sempre classes altamente coesas para garantir uma fácil manutenção nos códigos, reusabilidade e tornar as classes mais testáveis, pois o número de combinações possíveis a serem testadas será menor, e sendo mais testável, maior será a confiabilidade e qualidade do sistema.

```
class BudgetReport {
      void connectToRDBMS(){ }
      void generateBudgetReport() { }
      void saveToFile() { }
      void print() { }
}
```

Listagem 4 – Exemplo de código com baixa coesão. (SIERRA; BATES, 2005)

Ao invés do exemplo anterior em que uma única classe é responsável pela conexão com o banco de dados, geração de report, salvar arquivo e impressão; é possível se criar um exemplo mais coeso, onde será dividida em classes mais específicas, que terão propósitos mais focados:

```
class BudgetReport {
      Options getReportingOptions() { }
      void generateBudgetReport(Options o) { }
}
class ConnectToRDBMS {
      DBconnection getRDBMS() { }
}
class PrintStuff {
      PrintOptions getPrintOptions() { }
}
class FileSaver {
      SaveOptions getFileSaveOptions() { }
}
```

Listagem 5 – Exemplo de código com alta coesão. (SIERRA; BATES, 2005)

Deve se sempre prestar atenção nos principais sinais que indicam baixa coesão, e evitá-los ao máximo, dentre os principais podemos citar: dificuldades de denominar uma classe ou um método (o nome é grande ou é muito vago); quantidade grande de casos de testes; e classes e métodos muito grandes (com muitas linhas). Essas três situações mencionadas anteriormente indicam baixa coesão e que é necessário quebrar essas classes e/ou métodos em partes mais específicas, fazendo com que estas se tornem mais fáceis de dar manutenção.

3.3. Acoplamento

O acoplamento está relacionando ao quanto uma classe sabe sobre outra classe, sabe se que é necessário existir certo acoplamento entre as classes do sistema, mas isso tem um limite, pois se existe um alto-acoplamento, a sistema fica difícil de dar manutenção.

Bain (2008) cita alguns tipos de acoplamentos:

- **Acoplamento de identidade**: Esse tipo de acoplamento existe quando uma entidade *A* sabe que uma entidade *B* existe, porém ela não sabe como utilizar esta entidade, ou seja, não acessa os seus membros públicos.
- **Acoplamento representacional**: É quando uma entidade *A* acessa os membros públicos da classe *B*.
- **Acoplamento de sub-classe**: É quando um cliente tem acesso à uma sub-classe que vai além do que é permitido pelo contrato de sua interface.
- **Acomplamento de herança**: É o tipo de acoplamento que uma sub-classe tem com a sua superclasse através dos elementos que são herdados.

O acoplamento de herança é um ótimo exemplo e deve ser utilizado, o acoplamento de sub-classe deve ser evitado pois ele vai contra os princípios da Orientação a Objetos, já os acoplamentos de identidade e representacional podem ser utilizados mas com uma certa responsabilidade, ou seja, tomando alguns cuidados, como: evitar acoplamentos bi-direcionais, limitar somente o acoplamento da classe que utiliza para a classe utilizada por exemplo.

Para saber se está ocorrendo este tipo de problema, basta se atentar a alguns sinais como: efeitos colaterais nas manutenções, ou seja, modificações em uma classe costumam causar necessidade de manutenção em outras classes; hesitação dos desenvolvedores em fazer uma manutenção prevendo problemas; comentários excessivos explicando relacionamentos das classes; e por fim, casos de testes que necessitem da criação de uma grande quantidade de instâncias.

3.4. Redundância

Em um design emergente, um aspecto que apesar de óbvio precisa ser levado em consideração são as redundâncias, deve-se evitar ter trechos de códigos iguais, pois quanto mais código redundante, maior será a manutenção e também as chances de problemas, pois muitas vezes alguns pontos do sistema que existem códigos redundantes passam despercebidos. Bug Milenio (2010) explica um grande problema nesse sentido, que foi o famoso Bug do Milênio, onde muitos sistemas estavam configurados com o ano no formato de dois dígitos, o que acarretou prejuízos bilionários que poderiam ser reduzidos se os sistemas tivessem a configuração das datas feitas em somente um local.

Para evitar códigos redundantes, precisa se atentar à maneira que estão sendo modeladas as classes e aplicados os design patterns, que serão vistos com mais detalhes adiante, deve-se substituir o uso de interfaces por classes abstratas no caso das classes que tiverem implementações comuns. Bain (2008) dá dicas nesse sentido, onde em sistemas que estão tendo mudanças repetitivas ou trechos de códigos desenvolvidos estão freqüentemente sendo copiados e colados, nesses casos é preciso dar uma revisada nas classes e nos patterns aplicados para fazer uma modelagem correta do sistema.

3.5. Legibilidade

A legibilidade de um código é de extrema importância, além das outras práticas a serem adotadas para garantir a manutenibilidade do código, o código tem que estar legível, ou seja, quando um desenvolvedor pegar um código para dar manutenção ele tem que conseguir entender o código sem dificuldades, e isso se torna difícil em equipes numerosas que não tem um padrão de codificação a ser seguido. No caso de Java, a própria Oracle recomenda seguir o Code Conventions (1999), pois segundo a própria Oracle:

- "80% do custo de vida útil de uma peça de software vai para manutenção."
- "Dificilmente um software é mantido durante toda sua vida útil pelo autor original."
- "Code Conventions melhora a legibilidade do software, permitindo os engenheiros entender um novo código mais rapidamente e completamente."
- "Se você enviar seu código-fonte como um produto, você precisa ter certeza que ele está bem empacotado e claro como qualquer outro produto que você cria."

E nessa especificação são definidos detalhes de código como: nome de arquivo, organização de arquivo, identação, comentários, declarações, instruções, espaços em brancos, convenções de nomenclatura e práticas de programação.

Muitos desenvolvedores não seguem a risca o Code Conventions (1999), porém é de extrema importância que a equipe de desenvolvimento siga um mesmo padrão, uma dica boa para ajudar nesse sentido é o plug-in Checkstyle (2010). Este plug-in é utilizado junto com a Integrated Development Environment (IDE) de desenvolvimento em Java, e por default vem configurado de acordo com a Code Conventions (1999), mas pode ser configurado de outras maneiras, e durante o desenvolvimento de código, ele sinaliza os trechos de códigos que estão fora do padrão.

Alguns exemplos de padrão de código sugeridos pela Code Conventions (1999) são:

- Evitar declarações de várias variáveis em uma linha.

```
private int dia, mes, ano;    // MÁ PRÁTICA

private int dia;              // BOA PRÁTICA
private int mes;
private int ano;
```

Listagem 6 – Exemplo de boa e de má prática na declaração de variáveis

- Apesar de uma instrução condicional *if* que tenha uma linha de instrução funcionar sem a necessidade de isolá-la com chaves "*{}*", é melhor colocar para uma melhor legibilidade.

```
if (condicao)     // MÁ PRÁTICA
      instrucao;

if (condicao) {   // BOA PRÁTICA
      instrucao;
}
```

Listagem 7 – Exemplo de boa e de má prática na condicional *if*.

(CODE CONVETIONS, 1999)

- Utilizar espaço em branco entre uma palavra-chave e um parêntesis, após o ponto-e-vírgula e entre o parêntesis e a chave.

```
If (condicao) {
      ...
}

while (true) {
      ...
}

for (expressao1; expressao2; expr3) {
      ...
}
```

Listagem 8 – Exemplo de boa prática com o uso de espaço em branco.

(CODE CONVETIONS, 1999)

3.6. Testabilidade

Apesar de ser algo que gera certo desconforto nas equipes de projetos, é de grande importancia que em sistemas com design emergente, sejam feitos testes unitários das classes

desenvolvidas, para garantir uma melhor qualidade da aplicação, tanto no sentido das classes estarem funcionando corretamente, quanto no design correto, pois para um sistema ser testável, ele tem que ser bem desenhado, e Bain (2008) cita os principais motivos pelo qual seus clientes se recusam a fazerem testes unitários:

- "A classe que eu quero testar não pode ser testada isoladamente. Possui diversas dependências que precisam ser instanciadas para poder testar os seus respectivos comportamentos."
- "A classe tem muitas permutações de comportamentos para serem testados, a fim de ser abrangente, o caso de teste ficará imenso."
- "Este assunto de ser testado é atualmente repetido em muitos lugares do sistema, e então os testes serão muito redundantes e difíceis de manterem."

Em outras palavras, para o sistema ser testável, é preciso ter alta coesão, baixo acoplamento e sem redundâncias. No capítulo 8, será vista com mais detalhes a disciplina de test-driven development.

4. Princípios

Neste capítulo, serão abordados alguns princípios de design orientado a objetos que são de grande importância para garantir um design emergente, que permita realizar mudanças sem gerar impactos desta mudança pela aplicação. Os princípios são na verdade um determinado conhecimento que foi adquirido no dia-dia de trabalho dos profissionais como solução para evitar algum tipo de problema. São diversos os princípios que podem ser citados, porém iremos falar somente de alguns, tendo como principais referências: Martin Fowler, Bob Martins e Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides).

4.1. Bob Martins

Martin (2000) explica os principais sintomas de um design ruim, que segundo ele são:

- **Rigidez**: "Rigidez é a tendência de um software se tornar difícil de mudar, mesmo em situações mais simples."
- **Fragilidade**: "Fragilidade é a tendência do software se corromper em muitos lugares a cada vez que uma mudança é realizada."
- **Imobilidade**: "Imobilidade é a incapacidade de reutilizar um software de outros projetos ou partes do mesmo projeto."
- **Viscosidade**: "Viscosidade acontece de duas formas: viscosidade de design, e viscosidade de ambiente. Quando confrontados com uma mudança, os engenheiros geralmente encontram mais de uma maneira de realizar a mudança. Algumas das maneiras preservam o design, as demais não."

Como solução para problemas de design, Martin (2000) sugere alguns princípios de design de classes orientadas a objeto, dentre estes temos:

- **Open-Closed**: "Um módulo deve ser aberto para extensão, porém fechado para modificação."
- **Liskov Substitution**: "Sub-classes devem ser substituídas por suas classes base."
- **Inversão de Dependência**: "Dependa de algo abstrato e não de algo concreto."

- **Segregação de Interface**: “Muitas interfaces client específicas são melhor do que uma interface de propósito geral.”

4.1.1. Princípio Open-Closed

De acordo com Martin (2000): “De todos os princípios de design orientado a objetos, é o mais importante. Foi criado por Bertrand Meyer”. O objetivo deste princípio é desenvolver código que possa ser extendido nas manutenções, sem ter que fazer modificações no código-fonte original, no caso de uma manutenção, se você tem mais de uma maneira de fazer essa modificação, o ideal é sempre optar pela mais open-closed, utilizando como principal estratégia a abstração.

Segue abaixo um exemplo sem a utilização do princípio Open-Closed, onde uma classe *Client* utiliza-se classe *ConverterProcessor*, que por sua vez se utiliza da classe *ImageConverter* que faz a conversão de imagens, como mostra abaixo a figura 5.1:

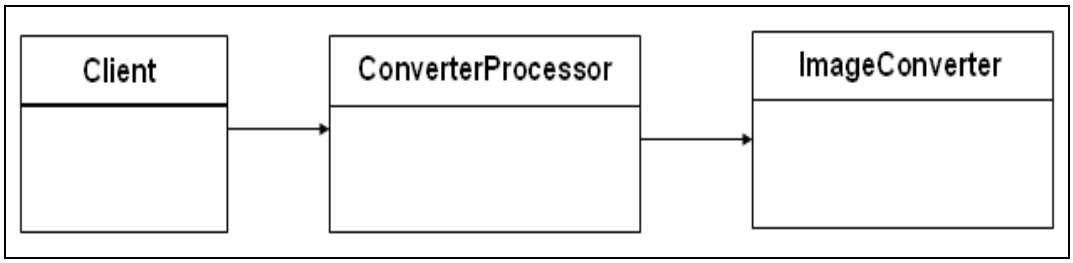

Figura 5 – Exemplo sem a utilização do princípio Open-Closed

Supondo se que a classe *ImageConverter* faça a conversão de imagens do formato *GIF* para o formato *JPEG*, e que posteriormente seja necessário fazer conversões para outros formatos, como *PNG* por exemplo, os códigos de *ConverterProcessor* e de *ImageConverter* terão de ser alterados cada vez que for necessário fazer a conversão de um novo formato, e isso acabará acarretando em diversos problemas, como classes sem coesão, altamente acopladas, e mais suscetíveis a novos erros a cada manutenção feita. Para resolver este tipo de problema, as classes serão modeladas de acordo com a Figura 6, aonde a classe *Client* irá se utilizar da classe *ConverterProcessor*, que desta vez fará uso da classe abstrata *ImageConverterAbstract*, e para cada tipo de formato novo a ser convertido, uma nova classe irá ser criada e irá herdar a classe *ImageConverterAbstract*, e a classe *ConverterProcessor* irá utilizar uma classe chamada *ImageConverterFactory* para a criação das instâncias de cada *ImageConverter*. Com

estas mudanças, o código se torna mais flexível para atender novas mudanças, mantendo a qualidade e sem gerar problemas para o código-fonte existente.

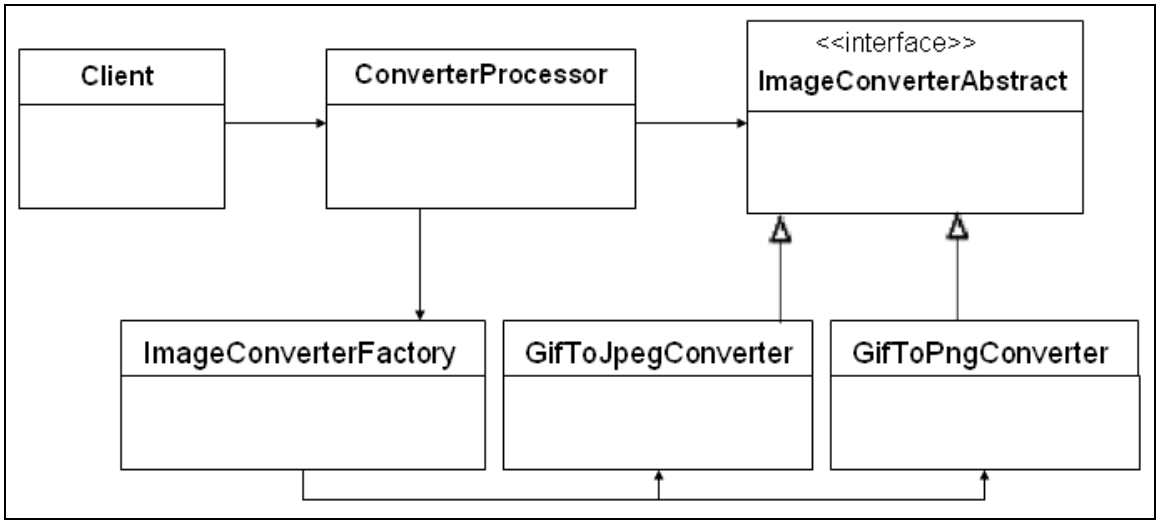

Figura 6 - Exemplo com a utilização do princípio Open-Closed

4.1.2. Princípio da Inversão de Dependência

Outro princípio recomendado pelo Martin (2000) é o da Inversão de Dependências, onde uma classe que se utiliza de outra classe, passará a utilizar-se de uma interface ou uma classe abstrata, ou seja, dependerá de algo abstrato e não de algo concreto. De acordo com Martin (1996):

- "Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações."
- "Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações."

Classes concretas sofrem constantes mudanças, e tendo uma dependência de algo abstrato, novas mudanças podem estender essa interface ou classe abstrata sem afetar as classes que possuem esta dependência. Segue abaixo um exemplo de Inversão de Dependência onde uma classe *Client* utiliza-se de uma interface *Service*:

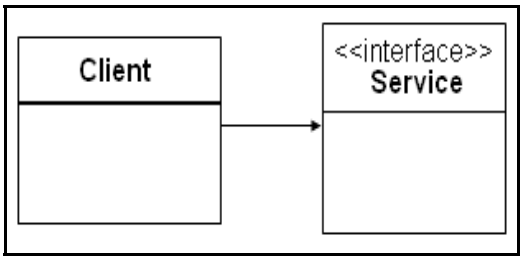

Figura 7 – Exemplo do princípio da Inversão de Dependência

4.2. Martin Fowler

Martin Fowler é uma das referências no mercado em design de software, além de outras especialidades como patterns e refatoração, possui diversos artigos e livros. Por hora, serão abordados dois princípios de design adotados por ele em seus design patterns. Estes princípios são: “Separando a configuração do uso” e “Separando código da interface de usuário”.

4.2.1. Separando a Configuração do Uso

Este é um princípio utilizado em diversos design patterns e o seu objetivo é separar as interfaces de suas respectivas implementações, onde o código que utiliza uma determinada interface não será afetado caso mude a implementação, evitando assim aquelas situações onde caso uma interface seja utilizado em diversos lugares da aplicação e seja modificada a maneira como a instância é criada, ao invés de modificar em diversos lugares, será necessário modificar somente em um único lugar. Segue abaixo um exemplo deste princípio utilizando o design pattern *Service Locator*:

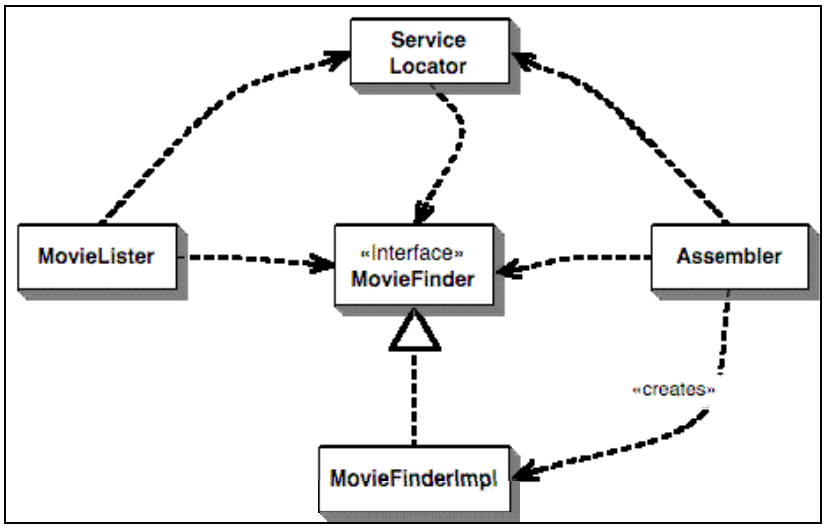

Figura 8 – Exemplo do princípio “Separando a Configuração do Uso”, com o design pattern Service Locator. (FOWLER, 2004.2)

4.2.2. Separando o Código da Interface de Usuário

Este princípio é utilizado em diversos frameworks de desenvolvimento de aplicação e tem como objetivo separar o código de interface de usuário do código referente ao domínio de negócio. Quando se fala do código de interface de usuário, se refere a todo código responsável pelas interações com o usuário, como por exemplo: campos da tela, a captura de informações, a exibição de informações processadas; já quando se fala do código de domínio de negócio, se refere a todo código referente ao processamento das informações capturadas do usuário. Esta separação trás diversos benefícios, pois a camada de apresentação é mais complexa, envolve diversos detalhes e muitas vezes utilizam até uma biblioteca própria para isso, e dessa maneira o desenvolvimento fica mais organizado e fácil de dar manutenção, é possível utilizar outras interfaces de usuário com o mesmo código de negócio, além do código de domínio de negócio se tornar testável.

4.3. Gang of Four

A Gang of Four, é uma das principais referências no mercado em design de software, autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software*, o qual tem sido uma importante contribuição para arquitetos e desenvolvedores de sistemas, onde são propostas diversas soluções de design patterns para resolver problemas presentes no desenvolvimento de sistemas. No próximo capítulo, estes design patterns serão abordados com mais detalhes, por hora serão abordados três princípios de design que a Gang of Four utiliza em seus design patterns, que são: “Programar para uma interface, e não para uma implementação”, “Favorecer composição de objetos sobre herança de classes” e “Considere o que deve ser variável em seu design e encapsule o conceito que varia”.

4.3.1. Programar para uma Interface, e não para uma Implementação

Muito similar aos princípios vistos anteriormente, este é um princípio bem utilizado nos design patterns da Gang of Four, onde uma classe que tem dependência de outra, ela terá

dependência de uma interface e não de uma classe concreta, e de acordo com a Gang of Four (1994), esta abordagem traz dois benefícios:

- "Clientes permanecem sem saber o tipo específico de objeto que eles usam. Enquanto os objetos aderirem à interface que o cliente espera."
- "Clientes permanecem sem saber das classes que implementam estes objetos. Clientes só sabem sobre a classe abstrata definida pela interface."

4.3.2. Favorecer Composição de Objetos sobre Herança de Classes

Este princípio, que também é bem utilizado nos design patterns da Gang of Four, está relacionado com a reutilização de funcionalidades, onde a idéia é fazer estas reutilizações através de composição de objetos, ou seja, através de componentes que irão encapsular estas funcionalidades, ao invés de herdar estas funcionalidades. Tanto a composição de objetos quanto a herança possuem suas vantagens e desvantagens, mas para este tipo de situação, a composição de objetos possui mais vantagens, e de acordo com a Gang of Four (1994):

- A dependência que uma sub-classe tem da sua respectiva super-classe limita a flexibilidade e a reutilização.
- "A composição de objetos é definida dinâmicamente em tempo de execução através da aquisição da referência de outros objetos. [...] [diferente do que é feito com a herança que é definida estaticamente em tempo de compilação, permitindo assim que a substituição de um objeto por outro do mesmo tipo possa ser feita em tempo de execução]."
- "Composição requer que os objetos estejam de acordo com as interfaces, que por sua vez requer interfaces cuidadosamente desenhadas que não te impedem de usar um objeto com muitos outros. Mas existe uma vantagem. Devido ao fato dos objetos serem acessados através de suas respectivas interfaces, nós não quebramos o encapsulamento."
- "Favorecer a composição de objetos sobre a herança de classes ajuda você a manter cada classe encapsulada e focada em uma tarefa."

4.3.3. Considere o que deve ser variável em seu design e encapsule o conceito que varia

Este princípio é bem utilizado nos design patterns da Gang of Four, e o intuito não é simplesmente encapsular uma simples variável, mas sim encapsular o conceito que varia, como por exemplo, o primeiro princípio da Gang of Four citado “Programar para uma interface, e não para uma implementação” é um exemplo de encapsulamento, pois a classe que se utiliza de uma interface ou uma classe abstrata, ela não tem acoplamento com a classe concreta que a implementa, logo esta classe concreta está encapsulada, e garante uma melhor flexibilidade e manutenibilidade, reduzindo as redundâncias no sistema.

5. Design Patterns

A idéia de patterns possui diversas definições, segundo Gang of Four, a primeira definição foi concebida por Christopher Alexandre, que dizia:

> "Cada pattern descreve um problema que ocorre com frequencia em nosso ambiente, e então descreve se um conjunto de soluções para o problema, de tal maneira que você pode usar esta solução um milhão de vezes mais, sem mesmo fazer a mesma coisa duas vezes". (GANG OF FOUR, 1994)

Apesar do Christopher Alexandre se referir a patterns de construções e cidades, o mesmo se procede em design patterns orientado a objeto. Já a Gang of Four (1994), que foram os pioneiros em design patterns orientado a objeto, define design patterns como: "Descrição da comunicação de objetos e classes que são customizados para resolver um problema de design genérico em um contexto particular".

No capítulo anterior foram abordados alguns dos princípios de design importantes que são utilizados nos design patterns, já neste capítulo serão abordados os design patterns. A primeira referência de design patterns foi criada pela Gang of Four (1994). De lá para cá, surgiram diversas referências de design patterns, Fowler (2005) enumera algumas dessas referências:

Referências de Design Patterns	
Referência	**Autor(es)**
Patterns of Enterprise Application Architecture	Fowler
Core JEE Patterns	Deepak Alur, John Crupi and Dan Malks
Enterprise Integration Patterns	Hohpe and Woolf
Microsoft Enterprise Solution Patterns	Trowbridge, Mancini, Quick, Hohpe, Newkirk and Lavigne
Microsoft Data Patterns	Teale, Etx, Kiel and Zeitz
Microsoft Integration Patterns	Trowbridge, Roxburgh, Hohpe, Manolescu and Nadhan
Domain Driven Design	Evans

Analysis Pattern	Fowler
Data Model Patterns	Hay
POSA	Buschmann, Meunier, Rohnert, Sommerlad and Stal

Tabela 1 – Principais referências de Design Patterns. (FOWLER, 2005)

Gang of Four (1994) documentou 23 design patterns, onde cada um deles foca em um problema particular de design orientado a objeto, são divididos em dois tipos de categorias: propósito e escopo. Na categoria de propósito, os design patterns são divididos em 3: criacionais, estruturais e comportamentais, onde os criacionais estão relacionados com o processo de criação dos objetos; os estruturais estão relacionados com a composição das classes e dos objetos; já os comportamentais estão relacionado com a maneira em que as classes e objetos se interagem e distribuem responsabilidades. Na categoria de escopo, determina se o design pattern é aplicado primariamente a uma classe ou a um objeto. Segue abaixo a relação dos design patterns divididos por categoria:

		Propósito		
		Criacional	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsability Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Tabela 2 – Tabela que relaciona os design patterns da Gang of Four com seus propósitos e escopo. (GANG OF FOUR, 1994)

Cada um dos design patterns da Gang of Four (2004) são descritos em um mesmo template e divididos nas seguintes seções: “Nome do Pattern e Classificação”, “Intenção”, “Também Conhecido Como”, “Motivação”, “Aplicabilidade”, “Estrutura”, “Participantes”, “Colaborações”, “Conseqüências”, “Implementação”, “Exemplo de Código”, “Utilizações Conhecidas” e “Patterns Relacionados”.

Os design patterns ajudam a fazer um design mais robusto, com mais reuso e que tenha menos problemas com redesign, ou seja, que esteja preparado para atender novas mudanças. As causas de redesign mais comum e que a Gang of Four (1994) propõe solução através de seus design patterns são:

- “Criando um objeto, especificando uma classe explicitamente.”
- “Dependência de operações específicas.”
- “Dependência de hardware e de plataforma de software.”
- “Dependência de representações de objetos e implementações.”
- “Dependência de algoritmos.”
- “Alto acoplamento.”
- “Estender funcionalidades através de sub-classe.”
- “Inabilidade de alterar classes convenientemente.”

Uma recomendação feita por Christopher Alexandre, de acordo com Bain (2008) é: “considere as forças”, ou seja, mais importante que o pattern são as forças, que segundo Shalloway (2008) são: “as questões no seu domínio de problema que você precisa considerar, no sentido de resolver da melhor maneira o problema que você está enfrentando”. Dado um determinado domínio de problema, deve se analisar as forças envolvidas para se decidir quais os design patterns a serem utilizados para resolver o problema.

5.1. Exemplo de aplicação de Design Patterns

Segue abaixo um exemplo de aplicação de design patterns de Bain (2008):

```
public class SignalProcessor {
      private ByteFilter myFilter;
```

```
      public SignalProcessor() {
            myFilter = ByteFilter.getInstance();
      }

      public byte[] processSignal(byte[] signal) {
            // Do preparatory steps
            myFilter.filter(signal);
            // Do other steps
            return signal;
      }
}


public class ByteFilter {

      private ByteFilter() {
            // do any construction set here
      }

      public static ByteFilter getInstance() {
            return new ByteFilter();
      }

      public byte[] filter(byte[] signal) {
            // do the filtering here
            return signal;
      }
}
```

Listagem 9 – Código-fonte das classes utilizadas no exemplo de aplicação Design Patterns. (BAIN, 2008)

Este exemplo é formado por duas classes: *SignalProcessor* e *ByteFilter*, onde a classe *SignalProcessor* é responsável pelo processamento de *array de bytes*, e o *ByteFilter* pela filtragem desse *array de bytes*. Nesse design, observa se que o construtor do *ByteFilter* está encapsulado pelo método estático *getInstance*, permitindo que o *ByteFilter* se torne uma classe abstrata e o *getInstance* retorne uma sub-classe, ou várias possibilidades de sub-classes sem afetar o objeto cliente que é o *SignalProcessor*.

Ao analisar, verifica se que existirá mais de um *ByteFilter*, sendo assim encontra se uma variação que deve ser tratada da melhor maneira. Ao verificar as intenções dos design patterns, verifica se que existem dois possíveis design patterns para resolver este tipo de problema: *Strategy* e o *Chain of Responsability*.

Se decidir que o método *getInstance* deve pertencer a um objeto separado, no caso uma *factory*, a solução ficaria igual a da Figura 9.

De acordo com a Figura 9, a *factory* é responsável pela criação das classes concretas que implementam o método *filter* da classe *ByteFilter* de diferentes maneiras, que são: *HiPassFilter* e *LoPassFilter*. A classe *SignalProcessor* não sabe qual implementação de *ByteFilter* está utilizando, e outras implementações de *ByteFilter* podem ser adicionadas sem afetar a classe *SignalProcessor*.

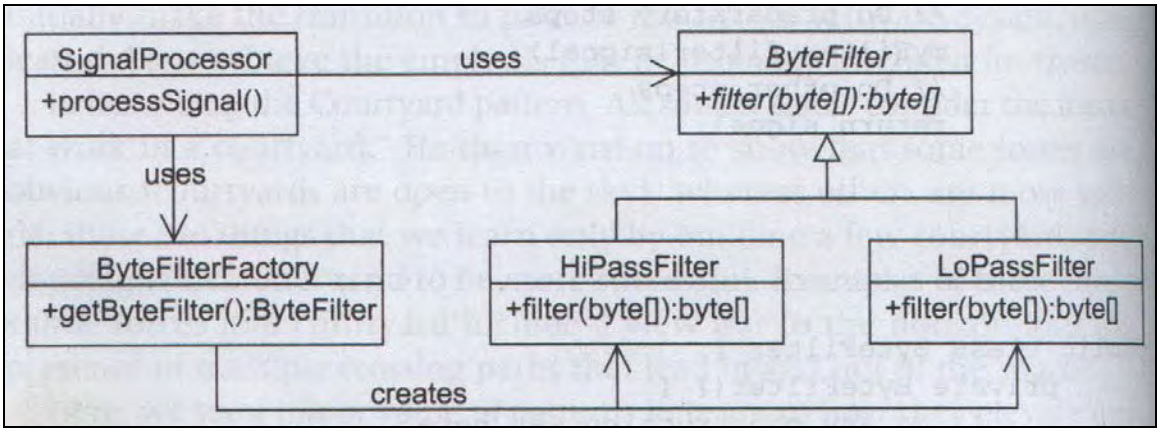

Figura 9 – Solução do exemplo utilizando o design pattern *Strategy*. (BAIN, 2008)

A escolha do pattern *Strategy* foi feita, pois existe uma variação de algoritmos, e de acordo com a Gang of Four (1994), a intenção deste pattern é: "Definir uma família de algoritmos, encapsular cada uma, e torná-las substituíveis. *Strategy* permite que o algoritmo varie independentemente do cliente que o utilize.", baseado nisso, o pattern *Strategy* foi identificado como uma possível solução para este problema.

Porém, a solução apresentada anteriormente não é a única possível, pode se resolver esta solução de outra maneira, onde cada uma das implementações de *ByteFilter* estaria acorrentada a uma lista encadeada, de maneira que cada solicitação de *filter* passaria por toda essa lista até encontrar a implementação de *ByteFilter* correta para tratar esta solicitação.

Esta segunda maneira de implementar mencionada anteriormente, foi feita de acordo com o pattern *Chain of Responsability*, que de acordo com a Gang of Four (1994), a intenção deste pattern é:

> "Evitar o acoplamento entre o remetente e o destinatário da solicitação, de maneira que mais de um objeto tenha chance de tratar a solicitação. Colocar os

objetos destinatários das solicitações em uma lista encadeada e percorrer a solicitação por toda a lista até que um objeto trate-a.”. (GANG OF FOUR, 1994)

A figura 10 ilustra o diagrama, onde a diferença está no auto-relacionamento de *ByteFilter*:

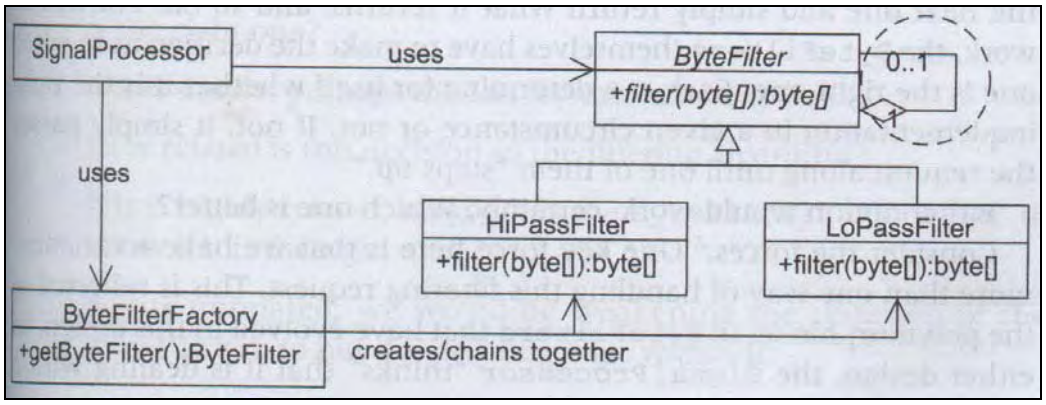

Figura 10 - Solução do exemplo utilizando o design pattern *Chain of Responsability*. (BAIN, 2008)

Em tempo de execução, estas duas soluções serão executadas de maneira diferente, onde no caso do *Strategy*, a *factory* irá decidir qual implementação será criada, e será entregue para o *SignalProcessor* utilizar, sem ao mesmo saber qual implementação está utilizando. No caso do *Chain of Responsability*, a *factory* irá criar a lista encadeada de todas as implementações de *ByteFilter*, e o *SignalProcessor* irá encaminhar a solicitação por esta lista para verificar qual implementação irá tratar a solicitação, onde cada implementação decidirá se irá tratá-la, ou passá-la para a próxima implementação, ou simplesmente retornar. A figura 11 ilustra essa diferença:

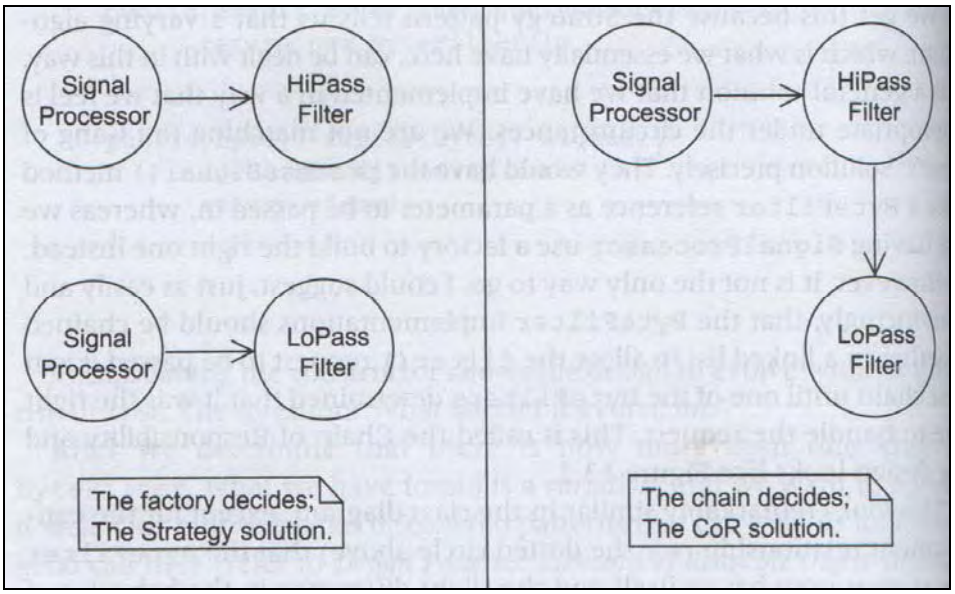

Figura 11 – Comparativo das adoções de design pattern: *Strategy* x *Chain of Responsability*. (BAIN, 2008)

Ao analisar, verifica se que tanto a primeira quanto a segunda opção funcionam, porém fica a questão: Qual das duas opções é a melhor?

Levando em consideração as forças, a principal força que se tem é a variação, onde haverá mais de uma maneira de filtrar o *array de bytes*.

Aonde esta decisão irá afetar? Afetará a *factory* e as implementações de *ByteFilter*. Segue abaixo o comparativo dos impactos de cada design pattern:

Comparativo entre os Design Patterns	
Strategy	**Chain of Resposability**
Factory mais complexa porque precisa fazer mais decisões.	*Factory* simplesmente constrói a lista e retorna.
Se novas implementações de *ByteFilter* surgirem, a *factory* deverá fazer mais decisões.	Se novas implementações de *ByteFilter* surgirem, a *factory* deverá construir uma lista longa.
Somente filtra. Cria uma forte coesão.	Além de filtrar, irá fazer a verificação se esta implementação será utilizada ou não.

Tabela 3 – Tabela comparativa dos impactos na adoção dos design patterns *Strategy* x *Chain of Responsability*. (BAIN, 2008)

Mesmo com estas informações, ao analisar um pouco mais, percebe-se que não se tem informações suficientes para tomar a decisão correta, pois de acordo com Bain (2008):

- "Quão complexa é a lógica necessária para determinar qual *ByteFilter* é o mais correto?
  Caso seja simples, talvez a *factory* possa tratar isso bem."
- "Quanto relacionada é a decisão com o algoritmo do filtro?
  Caso seja altamente relacionada, pode ser que a própria regra de negócio do *ByteFilter* sabe se a determinada implementação é a mais correta. Caso não seja relacionada, o *ByteFilter* ficaria com uma baixa coesão colocando a lógica de decisão junto com o algoritmo."

Baseado nisso, ainda fica a questão: Qual decisão tomar? Se for uma regra mais simples, onde um simples dado sinalizaria qual implementação de filtro deve ser utilizada, seria mais adequado adotar o *Chain of Responsability*; mas caso a regra seja mais complexa, é mais adequado que seja adotado o *Strategy*.

Para finalizar, seguem abaixo duas forças citadas por Bain (2008), que ele recomenda para estar atento, para poder entender o problema com mais clareza e tomar a decisão mais adequada:

- "Como o uso de uma solução contra outra afeta a qualidade global do sistema?"
- "Quão bem uma solução contra outra tratará as possíveis mudanças que poderá acontecer futuramente?"

6. Test-Driven Development

Test-Driven Development (TDD) é um estilo de desenvolvimento de sistemas de software, que está sendo cada vez mais utilizado, principalmente pelos seguidores de metodologias ágeis. Em TDD, o desenvolvedor busca inicialmente entender o negócio e os requisitos do sistema, feito isso o desenvolvedor irá desenvolver os casos de testes para atender os requisitos, após fazer os casos de testes, o desenvolvedor irá desenvolver as classes de negócio para que estas façam os testes funcionarem com sucesso.

Na verdade, TDD é uma verdadeira quebra de paradigma para a maioria dos desenvolvedores. Primeiramente, porque muitos desenvolvedores não testam seus códigos por acreditarem que isso seja função dos profissionais de Quality Assurance (QA), ou por acreditarem que isso será puro desperdício de tempo. Outra quebra de paradigma é que em TDD o procedimento é de primeiro escrever a classe de teste, para depois escrever a classe de negócio, e a maioria dos desenvolvedores estão acostumados a desenvolverem as classes de negócio primeiro, para depois desenvolverem as classes de teste que irão validar as classes de negócio. Ford (2009.2) mostra com mais detalhes o procedimento de um TDD:

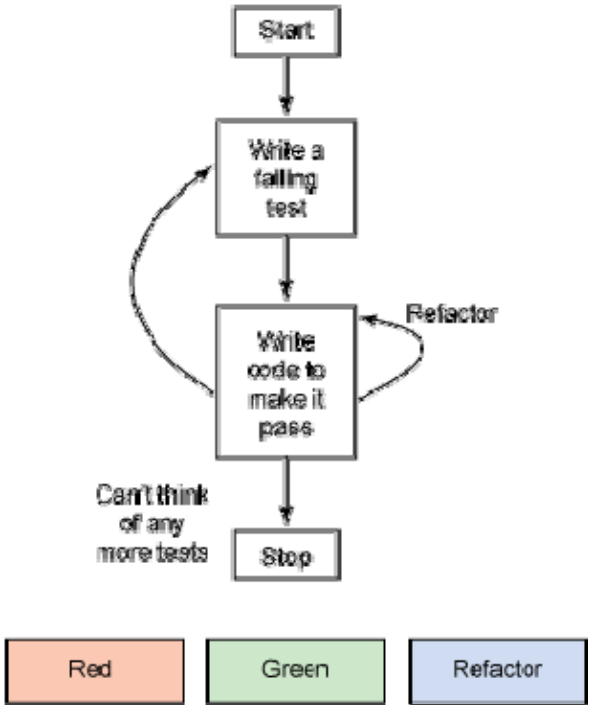

Figura 12 – Workflow de Test-Driven Design. (FORD, 2009.2)

O workflow da Figura 12 é:

1. “Escreva um teste que irá falhar.”
2. “Escreva um código que faça o teste funcionar.”
3. “Repita os passos 1 e 2.”
4. “Ao longo do caminho, refatore agressivamente.”
5. “Quando você não pensar mais em testes, estará concluído.”

Essa abordagem tem uma série de vantagens, dentre elas:

- **Design Emergente**: Após ser desenvolvida a classe de teste que irá falhar, será desenvolvida a classe de negócio para o teste funcionar, e essa classe de negócio precisa ser testável, ou seja, possuir as qualidades mencionadas anteriormente no capítulo 3, como: alta coesão, baixo acoplamento, sem redundância; pois do contrário, as conseqüências serão as seguintes:
    - **Classe com Baixa Coesão**: devido ao fato de não ser uma classe coesa, ou seja, com um único objetivo específico, as possíveis combinações de situações a serem testadas são inúmeras, o que torna difícil a classe de ser testada e consequentemente de garantir a qualidade da mesma.
    - **Classe com Alto Acoplamento**: devido ao fato de ter muitas dependências de outras classes, será necessária a criação e/ou população de dados de diversas instâncias, o que torna o teste difícil de ser executado.
    - **Classes com Redundâncias**: serão criados muitos testes repetitivos.

- **Código sempre Testado**: Isso faz com que aumente a confiabilidade no sistema, pois caso alguma modificação no sistema gere um erro, esse erro poderá ser reportado.
- **Conhecimento do Negócio**: O desenvolvedor para desenvolver os testes precisará saber o que ele está testando, logo precisará entender os requisitos e o negócio envolvido no escopo do sistema.
- **Economia de Tempo em Manutenções**: O desenvolvedor não perderá tanto tempo fazendo debugs e analisando detalhadamente os códigos para encontrar bugs no sistema.

6.1. Ferramentas

Existem atualmente diversas ferramentas que facilitam o trabalho do desenvolvedor que trabalha com TDD, cada uma com sua finalidade específica. Levando em consideração o desenvolvimento de sistemas em Java, podem ser citadas as seguintes ferramentas:

- **JUnit (2010)**: Framework Java amplamente utilizado no mercado pelos desenvolvedores, para a realização de testes unitários, ou seja, testes individuais feitos nas classes e métodos do sistema com o intuito de identificar possíveis erros até então não-detectados durante o desenvolvimento.

- **EasyMock (2010)**: Framework Java utilizado para a criação de mock objects, ou seja, objetos fake que permite testes em códigos que possuem dependência efêmera, ou seja, dependência de um sistema legado, ou de um banco de dados, ou qualquer outro tipo de dependência que tenha um estado imprevisível.

6.2. Exemplo de Desenvolvimento utilizando TDD

Para explicar mais claramente como funciona o processo de TDD, segue um exemplo de uma classe que verifica se a String informada é um palíndromo, ou seja, é igual lida nos dois sentidos, para isso será necessário seguir dois requisitos:

- Gerar a versão invertida de uma *String*.
- Verificar se a *String* é um palíndromo.

Baseado nos requisitos informados anteriormente, será criada uma classe de teste, com um teste unitário para cada requisito:

```
Import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class PalindromoUtilsTest {

	private PalindromoUtils pUtils;

	@Before
	public void setUp() {
		pUtils = new PalindromoUtils();
	}

	@Test
	public void getReversoTest() {
		assertEquals("bbbaa", pUtils.getReverso("aabbb"));
		assertEquals("aabab", pUtils.getReverso("babaa"));
		assertEquals("abcde", pUtils.getReverso("edcba"));
		assertEquals("aabaa", pUtils.getReverso("aabaa"));
	}

	@Test
	public void isPalindromoTest() {
		assertTrue(pUtils.isPalindromo("aaaaa"));
		assertFalse(pUtils.isPalindromo("aaaba"));
		assertTrue(pUtils.isPalindromo("aabaa"));
		assertFalse(pUtils.isPalindromo("abaaa"));
		assertTrue(pUtils.isPalindromo("bbbbb"));
	}
}
```

Listagem 10 – Código-fonte da classe que irá testar os requisitos do exemplo de TDD

Como pode ser visto, foi criada uma classe de teste inicial, com um teste para cada requisito. O método que possui a anotação *@Before* será o primeiro método a ser executado, pois é lá que são feitas as execuções iniciais. A princípio, este código não irá compilar, pois é necessário criar a classe *PalindromoUtils* e os seus respectivos métodos para fazer os testes funcionarem sem erros. Segue abaixo a versão inicial da classe *PalindromoUtils* que permitirá que a classe *PalindromoUtilsTest* compile:

```
public class PalindromoUtils {

	public String getReverso(String string) {
		// TODO Auto-generated method stub
		return null;
	}

	public boolean isPalindromo(String string) {
		// TODO Auto-generated method stub
		return false;
	}
}
```

Listagem 11 – Código-fonte da classe gerada para permitir que a classe de teste do exemplo seja compilada

Após a criação da classe *PalindromoUtils*, a *PalindromoUtilsTest* irá compilar, porém ao executar os testes no *JUnit*, será lançado o erro *java.lang.AssertionError* indicando que os testes não funcionaram. Será necessário implementar os métodos da classe *PalindromoUtils* para que o teste funcione corretamente. Segue abaixo a classe *PalindromoUtils* com seus respectivos métodos implementados:

```
public class PalindromoUtils {

	public String getReverso(String string) {
		StringBuilder stringInvertida = new StringBuilder();
		for (int i = string.length() - 1; i >= 0; i--) {
			stringInvertida.append(string.charAt(i));
		}
		return stringInvertida.toString();
	}

	public boolean isPalindromo(String string) {
		return string.equals(getReverso(string));
	}
}
```

Listagem 12 – Código-fonte da classe *PalindromoUtils* após a implementação dos métodos para atender o caso de teste

Após a implementação da classe *PalindromoUtils*, executa-se novamente os testes da classe *PalindromoUtilsTest* no *JUnit*, e desta vez os testes serão executados com sucesso. Mas ainda existem detalhes que precisam ser pensados, como por exemplo, testar as condições de fronteira, ou seja, os casos que podem gerar bugs ou *Exception*. Ao analisar o código da *PalindromoUtils*, identifica se como condições de fronteira, as situações em que os dois

métodos recebem valor nulo. Segue abaixo o código da classe *PalindromoUtilsTest* com mais dois testes unitários:

```
import org.junit.Before;
import org.junit.Test;

public class PalindromoUtilsTest {

	private PalindromoUtils pUtils;

	@Before
	public void setUp() {
		pUtils = new PalindromoUtils();
	}

	@Test
	public void getReversoTest() {
		assertEquals("bbbaa", pUtils.getReverso("aabbb"));
		assertEquals("aabab", pUtils.getReverso("babaa"));
		assertEquals("abcde", pUtils.getReverso("edcba"));
		assertEquals("aabaa", pUtils.getReverso("aabaa"));
	}

	@Test
	public void isPalindromoTest() {
		assertTrue(pUtils.isPalindromo("aaaaa"));
		assertFalse(pUtils.isPalindromo("aaaba"));
		assertTrue(pUtils.isPalindromo("aabaa"));
		assertFalse(pUtils.isPalindromo("abaaa"));
		assertTrue(pUtils.isPalindromo("bbbbb"));
	}

	@Test
	public void getReversoNullArgumentTest() {
		assertNull(pUtils.getReverso(null));
	}

	@Test
	public void isPalindromoNullArgumentTest() {
		assertTrue(pUtils.isPalindromo(null));
	}
}
```

Listagem 13 – Código-fonte da classe de teste do exemplo após inclusão de mais dois testes referentes a valores nulos

Ao executar novamente os testes da classe *PalindromoUtilsTest*, é lançada a exceção *NullPointerException* em cada um dos testes novos. É necessário incluir uma validação em cada um dos métodos da classe *PalindromoUtils* para verificar se o valor enviado é nulo. Segue abaixo o código da classe *PalindromoUtils* após a inclusão das validações:

```
public class PalindromoUtils {

	public String getReverso(String string) {
		if (string == null) {
			return null;
		} else {
			StringBuilder stringInvertida = new StringBuilder();
			for (int i = string.length() - 1; i >= 0; i--) {
				stringInvertida.append(string.charAt(i));
			}
			return stringInvertida.toString();
		}
	}

	public boolean isPalindromo(String string) {
		return string == null ?
			false		:
			string.equals(getReverso(string));
	}
}
```

Listagem 14 – Código-fonte da classe *PalindromoUtils*, após a implementação das validações de valores nulos

Feita as alterações, os testes da classe *PalindromoUtilsTest* serão executados com sucesso.

6.3 Exemplo de Desenvolvimento utilizando TDD com Objeto Efêmero

Neste próximo exemplo que segue, será feito um teste que envolve um objeto efêmero, ou seja, um objeto que gera resultados imprevisíveis, que neste exemplo é um objeto de acesso ao banco de dados. Inicialmente, temos duas classes e uma interface. A interface *CotacaoDAO* e a classe *CotacaoDAOImpl* são responsáveis pelo acesso ao banco de dados, já a classe *Cambio* é responsável por calcular a conversão de moedas. Segue abaixo o código fonte das classes citadas:

```
public class Cambio {
	private CotacaoDAO cotacaoDAO;

	public Cambio() {
		cotacaoDAO = makeDAO();
	}

	public Double converterDolarParaReais(double valorEmDolar) {
		Double cotacaoDolar = cotacaoDAO.retrieveCotacaoDolar();
		return valorEmDolar * cotacaoDolar;
	}

	protected CotacaoDAO makeDAO() {
		return new CotacaoDAOImpl();
```

```
	}
}

public interface CotacaoDAO {
	public Double retrieveCotacaoDolar();
}

public class CotacaoDAOImpl implements CotacaoDAO {
	@Override
	public Double retrieveCotacaoDolar() {
		Double cotacaoDolar = null;
		// CÓDIGO DE ACESSO AO BANCO DE DADOS
		return cotacaoDolar;
	}
}
```

Listagem 15 – Código-fonte das classes do exemplo de desenvolvimento utilizando TDD com objeto efêmero

Para que a classe *Cambio* possa ser testada, é necessário criar uma classe *fake* que implemente a interface *CotacaoDAO***,** onde esta classe terá uma variável de instância mais um método *setter* para receber o valor previsto que será retornado. Segue abaixo a classe *CotacaoFakeDAOImpl*:

```
public class CotacaoFakeDAOImpl implements CotacaoDAO {

	private Double cotacaoDolar;

	@Override
	public Double retrieveCotacaoDolar() {
		return cotacaoDolar;
	}

	public void setCotacaoDolar(Double cotacaoDolar) {
		this.cotacaoDolar = cotacaoDolar;
	}
}
```

Listagem 16 – Código-fonte da classe *fake CotacaoFakeDAOImpl*

E para finalizar, será criada a classe *CambioTest*, onde nela será criado um objeto *fake* para *CotacaoDAO* e atribuído um valor que é previsto de ser retornado. Segue abaixo a classe *CambioTest*:

```
import org.easymock.*;
import static org.junit.Assert.*;
import org.junit.*;

public class CambioTest extends TestCase {
      private Cambio       cambio;
      private MockControl  mockControl;
      private CotacaoDAO   mockDAO;

      public void setUp() {
            mockControl = EasyMock.controlFor(CotacaoDAO.class);
            mockDAO     = (CotacaoDAO) mockControl.getMock();
            cambio      = new Cambio() {
                  protected CotacaoDAO makeData() {
                        return new CotacaoFakeDAOImpl();
                  }
            };
      }

      public void testConverterDolarParaReais() {
            mockDAO.retrieveCotacaoDolar();
            mockControl.setReturnValue(2.0);
            mockControl.activate();

            assertEquals("Valor de U$200.00 equivale a R$400,00.",
                    cambio.converterDolarParaReais(200.0).doubleValue(),
                    400.0D);
      }
}
```

Listagem 17 – Código-fonte da classe de teste do exemplo de desenvolvimento utilizando TDD com objeto efêmero

7. Refatoração

Refatoração é uma disciplina de grande importância quando se trata de design emergente, pois um design emegente é um design flexível para atender as mudanças que ocorrem tanto durante o projeto, quanto durante o ciclo de vida do sistema de software. De acordo com Fowler Bbor (1999), “Refatoração é o processo de modificação em sistemas de software de tal maneira que não altere o comportamento externo do código e ainda melhore a estrutura interna”.

Quando Fowler Bbor (1999) fala em melhorar a estrutura interna, ele se refere a fazer correções no código-fonte do sistema para que se tenham as qualidades que foram explicadas com detalhes no capítulo 3: encapsulamento, alta coesão, baixo acoplamento, sem redundância, legibilidade e testabilidade. Para Fowler Bbor (1999), os principais motivos de fazer uma refatoração são: melhorar o design do software, tornar o software fácil de entender, ajudar a encontrar bugs e ajudar a programar mais rápido.

Em muitas situações do dia-dia, os profissionais ficam na dúvida se vale a pena refatorar ou se deve prosseguir do jeito que está. Fowler Bbor (1999) diz em que situações se deve e em que situações não se deve refatorar, por exemplo:

Deve se refatorar quando:

- **A regra do três**: quando você verifica que pela terceira vez está fazendo algo redundante.
- **Precisa adicionar uma função**: pois se o código está com problema de qualidade, a tendência é piorar a cada manutenção que é feita.
- **Precisa corrigir um bug**: para deixar o código mais fácil de entender.
- **Se faz uma revisão de código**: no caso de se encontrar possíveis melhorias de qualidade a serem feitas.

Não se deve refatorar quando:

- **Quando o código atual não funciona**: é de extrema importância que um código que será refatorado esteja funcionando, pois caso o código esteja cheio de bugs, é mais viável reescrever.
- **Quando se está perto do prazo de entrega**: nesse caso é mais vantajoso fazer a refatoração depois da entrega da versão do sistema que está para ser entregue, pois as conseqüências de não entregar no prazo são mais impactantes.

Mas fazer refatoração em algumas situações pode ser muito complicado, situações estas que geram muito trabalho e colocam o sistema em risco. Fowler Bbor (1999) enumera algumas situações como:

- **Banco de Dados**: pois o banco de dados é altamente acoplado com a aplicação, além do fato que uma refatoração neste caso irá necessitar de migração de dados, que acaba gerando muito trabalho e risco.
- **Mudança em interfaces**: quando se modifica uma interface, todos os códigos clientes que fazem chamadas para esta interface precisarão ser modificados, e isso se torna complicado a partir do momento em que a interface é pública e não se tem acesso a alguns códigos que acessam esta interface.

No código-fonte de um sistema existem diversas evidências que podem ser identificadas, que indicam uma má-qualidade no código e que naquela determinada parte do sistema seria ideal se fazer uma refatoração. Fowler Bbor (1999) possui inúmeros exemplos, segue abaixo uma tabela relacionando os problemas de qualidade com as evidências do problema:

Problemas de Qualidade X Evidências	
Problema	**Evidências**
Baixo Encapsulamento	• Lista longa de parâmetros em um método. • Diversas mudanças referentes a um único tipo de mudança. • Amontoado de dados. • Obsessão por primitivos onde caberiam melhor objetos.
Baixa Coesão	• Classe grande. • Método longo.
Alto Acoplamento	• Mudanças em diferentes classes referentes a um único tipo

	de mudança. • Método que utiliza características de outras classes. • Cadeia de mensagens.
Redundância	• Código duplicado. • Instruções *switch*. • Hierarquias paralelas.
Sem Legibilidade	• Variáveis temporárias. • Comentários (em excesso).
Sem Testabilidade	• Todas as evidências de baixa coesão, alto acoplamento e redundância.

Tabela 4 – Tabela que relaciona os problemas de qualidade de código com suas respectivas evidências. (FOWLER BBOR,1999)

Durante o processo de refatoração, é importante utilizar as técnicas de TDD que foram mostradas com detalhes no capítulo anterior, onde a cada correção que é feita, executa se em seguida os testes unitários e caso apareçam bugs, efetua se a correção e novamente se realiza o teste unitário até que estes bugs desapareçam.

De acordo com Ford (2009.1), para tornar o código-fonte de um sistema capaz de receber o maior número possível de refatorações:

> "Primeira opção seria se recusar a escrever qualquer novo código antes de adicionar testes para o projeto inteiro. Logo que você propor isso, você será demitido e poderá trabalhar para uma empresa que valoriza o teste unitário. Esta abordagem não será boa. A sua melhor aposta será fazer com que os outros da sua equipe compreendam o valor do teste unitário e comecem aos poucos adicionando testes em torno das partes de código mais críticas." (FORD, 2009.1)

Fowler Bbor (1999) catalogou dezenas de técnicas de refatoração, onde cada uma delas é um conjunto de passos para realizar um determinado tipo de refatoração com sucesso, estas técnicas estão também em Fowler (2010.2), juntamente com outras técnicas novas que foram criadas tanto por ele quanto por outros colaboradores. São inúmeras técnicas de refatoração divididas por categorias, onde cada categoria tem um determinado propósito, seria impossível

explicar cada uma delas. Por hora, será detalhada como exemplo a primeira categoria que Fowler Bbor (1999) explica que é a *Composing Method*, que se refere ao conjunto de técnicas para reestruturar um método longo que está com baixa coesão e baixa legibilidade. Segue abaixo uma tabela, listando as técnicas de refatoração relacionadas com esta categoria, junto com seus respectivos propósitos:

Técnicas de Refatoração da categoria Composing Method	
Técnica	**Propósito**
Extract Method	"Transforme um fragmento do método em um método em que o nome explique o propósito do método."
Inline Method	"Coloque o corpo do método, no corpo do método que faz a chamada e remova-o."
Inline Temp	"Substitua todas as referências de variáveis temporárias pelo valor dessas variáveis."
Replace Temp With Query	"Extraia a expressão em um método. Substitua todas as referências de variáveis temporárias com a expressão. O novo método pode então ser usado em outros métodos."
Introduce Explaining Variable	"Coloque o resultado da expressão, ou partes da expressão, em uma variável temporária com um nome que explique o propósito."
Split Temporary Variable	"Faça uma variável temporária separada para cada instrução."
Remove Assignments to Parameters	"Use variáveis temporárias no lugar."
Replace Method with Method Object	"Transforme o método em um objeto de maneira que todas as variáveis locais se tornem variáveis de instância no objeto. Você poderá decompor o método em outros métodos no mesmo objeto."
Substitute Algorithm	"Substituir o corpo de um método com um novo algoritmo."

Tabela 5 – Tabela que relaciona algumas técnicas de refatoração com suas respectivas definições

Outra importante referência de refatoração, que complementa a referência de Fowler Bbor (1999) é o Kerievsky (2004), onde são ensinadas técnicas que ajudam a combinar as refatorações com os patterns para que se possa fazer uma refatoração orientada à patterns. Kerievsky (2004) catalogou 27 técnicas de refatoração para que se possa fazer uma refatoração orientada a pattern, muitos deles utilizando técnicas do Fowler (1999). Um exemplo de técnica de refatoração de Kerievsky (2004) é a *Encapsulate Classes with Factory*, onde no caso de um cliente que instancia diretamente classes que estão em um package e implementam uma interface comum, deve-se tornar os construtores destas classes não-públicos e criar uma Factory que se encarrega dos detalhes de instanciação destas classes, onde o cliente passará a se utilizar desta Factory para a criação da instância.

7.1. Exemplo de Refatoração

Segue abaixo um exemplo de código Java:

```
import java.util.List;

public class Venda {
     public List<Vendedor> vendedores;

     public void gerarRelatorioVendas() {
          double totalVendasMes = 0.0;
          double totalComissoesMes = 0.0;

          System.out.println("================================");
          System.out.println("=== RELATÓRIO GERAL DO MÊS  ===");
          System.out.println("================================");

          for(Vendedor vendedor : vendedores) {
             double totalVendas = vendedor.getTotalVendas();
             double comissao    = 0.0;
             if(totalVendas < 40000) {
                  comissao = totalVendas * 0.04;
             } else if(totalVendas < 60000) {
                  comissao = totalVendas * 0.05;
             } else if(totalVendas < 80000) {
                  comissao = totalVendas * 0.06;
             } else if(totalVendas < 100000) {
                  comissao = totalVendas * 0.06;
             }
             System.out.println("=== Nome " + vendedor.getNome() +
                            ", Total Vendas: " + totalVendas +
                            ", Comissão: " + comissao);
             totalVendasMes+=totalVendas;
             totalComissoesMes+=comissao;
          }
```

```
            System.out.println("============================");
            System.out.println("=== TOTAL DE VENDAS    = " +
                               totalVendasMes);
            System.out.println("=== TOTAL DE COMISSÕES = " +
                               totalComissoesMes);
            System.out.println("============================");
      }
}

public class Vendedor {
      private String nome;
      private double totalVendas;

      public String getNome() {
            return nome;
      }
      public void setNome(String nome) {
            this.nome = nome;
      }

      public double getTotalVendas() {
            return totalVendas;
      }
      public void setTotalVendas(double totalVendas) {
            this.totalVendas = totalVendas;
      }
}
```

Listagem 18 – Exemplo de código-fonte que será refatorado

Como pode ser observado, o código exibido da classe *Venda* apresenta diversos problemas de qualidade, onde há um método longo, com baixa coesão, baixa legibilidade e consequentemente uma baixa testabilidade. Será necessário refatorar o código-fonte da classe *Venda*.

Para dar início ao processo de refatoração, será aplicada a técnica *Extract Method* para extrair as linhas de código referentes à geração do cabeçalho relatório, geração do rodapé do relatório e cálculo de comissão; e em seguida colocá-las em métodos privado. Será criado o método *gerarCabecalhoRelatorio* e lá serão colocadas as três linhas referentes à geração do cabeçalho do relatório, e no lugar onde estão estas linhas será colocada a chamada para o método *gerarCabecalhoRelatorio*. O mesmo procedimento será feito com as quatro linhas referentes à geração do rodapé do relatório, no método *gerarRodapeRelatorio*, com a diferença que estas linhas de comando utilizam duas variáveis locais, que serão passadas através de parâmetros do método. O trecho de código que calcula a comissão do vendedor, será extraído e colocado no método *getComissao*, que receberá como parâmetro a variável *totalVendas*, referente ao total de vendas do vendedor, e terá como retorno a variável *comissao*, que é o resultado do

cálculo da comissão do vendedor. Com relação à variável de instância *vendedores* que não está encapsulada, será aplicada a técnica *Self Encapsulate Field*, onde a variável será transformada em privada e serão criados métodos de acessos para esta. Segue abaixo o código da classe *Venda* após a primeira etapa de refatoração:

```
import java.util.List;

public class Venda {
     private List<Vendedor> vendedores;

    public void gerarRelatorioVendas() {
         double totalVendasMes = 0.0;
         double totalComissoesMes = 0.0;

         gerarCabecalhoRelatorio();

         for(Vendedor vendedor : getVendedores()) {
              double totalVendas = vendedor.getTotalVendas();
              double comissao    = 0.0;
              comissao = getComissaoVendedor(totalVendas);
              System.out.println("=== Nome " + vendedor.getNome() +
                                 ", Total Vendas: " + totalVendas +
                                 ", Comissão: " + comissao);
              totalVendasMes+=totalVendas;
              totalComissoesMes+=comissao;
         }

         gerarRodapeRelatorio(totalVendasMes, totalComissoesMes);
    }

    private double getComissaoVendedor(double totalVendas) {
        double comissao = 0.0;

        if(totalVendas < 40000) {
            comissao = totalVendas * 0.04;
        } else if(totalVendas < 60000) {
            comissao = totalVendas * 0.05;
        } else if(totalVendas < 80000) {
            comissao = totalVendas * 0.06;
        } else if(totalVendas < 100000) {
            comissao = totalVendas * 0.06;
        }
        return comissao;
    }

    private void gerarCabecalhoRelatorio() {
        System.out.println("==============================");
        System.out.println("=== RELATÓRIO GERAL DO MÊS ===");
        System.out.println("==============================");
    }

   private void gerarRodapeRelatorio(double totalVendasMes,
                                     double totalComissoesMes) {
       System.out.println("==============================");
       System.out.println("=== TOTAL DE VENDAS    = " + totalVendasMes);
       System.out.println("=== TOTAL DE COMISSÕES = " +
                          totalComissoesMes);
       System.out.println("==============================");
```

```
    }

    public void setVendedores(List<Vendedor> vendedores) {
         this.vendedores = vendedores;
    }

    public List<Vendedor> getVendedores() {
         return vendedores;
    }
}
```

Listagem 19 – Código-fonte do exemplo após a primeira etapa de refatoração

Como pode ser visto, houve uma melhora na qualidade do código, mas ainda há algumas refatorações a serem feitas. Dentro do método *getComissaoVendedor* será aplicada a técnica *Consolidate Duplicate Conditional Fragments*, onde o código referente ao cálculo da comissão, que está sendo repetido dentro da condicional, será retirado e colocado depois da condicional, e dentro da condicional será recuperada a porcentagem da comissão em uma nova variável chamada *porcentagemComissao*. Feito isso, ainda dentro do método *getComissaoVendedor*, será aplicada a técnica *Extract Method* para retirar a código que recupera a porcentagem da comissão, e colocá-lo em um novo método chamado *getPorcentagemComissao*, que receberá como parâmetro a variável *totalVendas* e retornará a variável *porcentagemComissao*. Dentro do método *gerarRelatorioVendas*, será aplicada a técnica *Replace Temp with Query*, em cima das variáveis *totalVendas* e *comissao*, onde estas variáveis temporárias serão substituídas pela própria expressão que é atribuída a elas. Segue abaixo o código da classe *Venda* após a segunda etapa de refatoração:

```
import java.util.List;

public class Venda {
    private List<Vendedor> vendedores;

    public void gerarRelatorioVendas() {
         double totalVendasMes    = 0.0;
         double totalComissoesMes = 0.0;

         gerarCabecalhoRelatorio();

         for(Vendedor vendedor : getVendedores()) {
              System.out.println("=== Nome " + vendedor.getNome() +
                                 ", Total Vendas: "               +
                                 vendedor.getTotalVendas()        +
                                 ", Comissão: "                   +
                            getComissaoVendedor(vendedor.getTotalVendas()));

              totalVendasMes    += vendedor.getTotalVendas();
              totalComissoesMes += getComissaoVendedor(vendedor
                                        .getTotalVendas());
```

```
        }

        gerarRodapeRelatorio(totalVendasMes, totalComissoesMes);
    }

    private double getComissaoVendedor(double totalVendas) {
        double comissao            = 0.0;
        double porcentagemComissao = 0.0;

        porcentagemComissao = getPorcentagemComissao(totalVendas);

        comissao = totalVendas * porcentagemComissao;
        return comissao;
     }

     private void gerarCabecalhoRelatorio() {
        System.out.println("==============================");
        System.out.println("=== RELATÓRIO GERAL DO MÊS ===");
        System.out.println("==============================");
     }

     private void gerarRodapeRelatorio(double totalVendasMes,
                                       double totalComissoesMes) {
        System.out.println("==============================");
        System.out.println("=== TOTAL DE VENDAS    = " + totalVendasMes);
        System.out.println("=== TOTAL DE COMISSÕES = " +
                           totalComissoesMes);
        System.out.println("==============================");
     }

     private double getPorcentagemComissao(double totalVendas) {
        double porcentagemComissao = 0.0;
        if(totalVendas < 40000) {
            porcentagemComissao = 0.04;
        } else if(totalVendas < 60000) {
            porcentagemComissao = 0.05;
        } else if(totalVendas < 80000) {
            porcentagemComissao = 0.06;
        } else if(totalVendas < 100000) {
            porcentagemComissao = 0.06;
        }
        return porcentagemComissao;
     }

     public void setVendedores(List<Vendedor> vendedores) {
        this.vendedores = vendedores;
     }

     public List<Vendedor> getVendedores() {
          return vendedores;
     }
}
```

Listagem 20 - Código-fonte do exemplo após a segunda etapa de refatoração

Após essa segunda etapa de refatoração, o código teve outra melhora notável, agora será feita as últimas refatorações. Nos métodos *getComissaoVendedor* e *getPorcentagemComissao*, será aplicada a técnica *Introduce Parameter Object*, onde o parâmetro *totalVendas* será substituído

pelo objeto *Vendedor*, que possui uma variável referente este parâmetro, e consequentemente dentro destes dois métodos, será substituída a chamada do parâmetro *totalVendas* pela variável que está dentro do objeto *Vendedor*, através de seu método de acesso: *vendedor.getTotalVendas()*. Será aplicada a técnica *Extract Method* no código que gera a linha de relatório para gerar o método *gerarLinhaRelatorio*, e como argumento será passada a variável vendedor. No método *getComissaoVendedor*, será aplicada a técnica *Replace Temp With Query*, onde as variáveis *comissao* e *porcentagemComissao* serão substituídas pelas expressões atribuídas a elas. E por fim, alguns ajustes como identação e espaçamento entre as linhas de código para deixar o código mais legível. Segue abaixo a versão final da classe *Venda* refatorada:

```
import java.util.List;

public class Venda {
     private List<Vendedor> vendedores;

     public void gerarRelatorioVendas() {
          double totalVendasMes    = 0.0;
          double totalComissoesMes = 0.0;

          gerarCabecalhoRelatorio();

          for(Vendedor vendedor : getVendedores()) {
              gerarLinhaRelatorio(vendedor);
              totalVendasMes    += vendedor.getTotalVendas();
              totalComissoesMes += getComissaoVendedor(vendedor);
          }

          gerarRodapeRelatorio(totalVendasMes, totalComissoesMes);
     }

     private void gerarCabecalhoRelatorio() {
         System.out.println("================================");
         System.out.println("=== RELATÓRIO GERAL DO MÊS ===");
         System.out.println("================================");
     }

     private void gerarLinhaRelatorio(Vendedor vendedor) {
         System.out.println("=== Nome " + vendedor.getNome() +
                     ", Total Vendas: " + vendedor.getTotalVendas() +
                     ", Comissão: "     + getComissaoVendedor(vendedor));
     }

     private void gerarRodapeRelatorio(double totalVendasMes,
                                       double totalComissoesMes) {
         System.out.println("============================");
         System.out.println("=== TOTAL DE VENDAS        = " +
                            totalVendasMes);
         System.out.println("=== TOTAL DE COMISSÕES = " +
                            totalComissoesMes);
         System.out.println("============================");
     }
```

```
    private double getComissaoVendedor(Vendedor vendedor) {
        return vendedor.getTotalVendas() *
               getPorcentagemComissao(vendedor);
    }

    private double getPorcentagemComissao(Vendedor vendedor) {
        double porcentagemComissao = 0.0;

        if(vendedor.getTotalVendas() < 40000) {
            porcentagemComissao = 0.04;
        } else if(vendedor.getTotalVendas() < 60000) {
            porcentagemComissao = 0.05;
        } else if(vendedor.getTotalVendas() < 80000) {
            porcentagemComissao = 0.06;
        } else if(vendedor.getTotalVendas() < 100000) {
            porcentagemComissao = 0.06;
        }

        return porcentagemComissao;
    }

    public void setVendedores(List<Vendedor> vendedores) {
         this.vendedores = vendedores;
    }

    public List<Vendedor> getVendedores() {
         return vendedores;
    }
}
```

Listagem 21 - Código-fonte do exemplo após a terceira etapa de refatoração

Como pode ser visto, após as etapas de refatorações houve uma notável melhora no código da classe *Vendas*, eliminando os problemas de qualidade mencionados anteriormente e permitindo que esta possa receber futuras manutenções sem grandes impactos no sistema.

Conclusão

As dificuldades ocorridas em muitos dos projetos de sistema de software nesses últimos anos, apesar de parecer algo insolúvel, é algo que tem solução, soluções estas onde algumas são até antigas, e que seriam capazes de resolver boa parte dessas dificuldades. As mudanças de diversas naturezas que ocorrem durante um projeto tem sido um dos principais motivos que levaram muitos projetos ao fracasso. Encarar as mudanças ao invés de evitá-las, passou a fazer parte do dia-dia de muitos projetos, principalmente nos projetos onde se adotam metodologias ágeis, pois o mercado em que se vive hoje em dia é muito dinâmico, fazendo com que seja quase impossível não haver mudanças no decorrer de um projeto, e é por essas e outras que ter uma arquitetura evolutiva é algo muito importante, pois como ocorrem essas mudanças no decorrer do projeto, nunca se sabe o que poderá acontecer, e uma mudança de requisito não-funcional poderá ser fatal.

O problema maior que se observa em boa parte das empresas, é que ainda não houve essa conscientização, e muitos gestores e líderes técnicos ainda insistem em trabalhar de maneira tradicional, seguindo um escopo de trabalho fechado, uma arquitetura engessada, com a expectativa de concluir no menor prazo possível, deixando de lado diversas preocupações relacionadas ao design e à qualidade, e só passam a ter preocupação quando surge algum problema muito sério, e muitas vezes essa preocupação acaba surgindo tarde demais.

Ao analisar as medidas adotadas para se elaborar um design emergente, percebe se que uma parte destas medidas são antigas, foram criadas com o intuito de resolver muitos problemas que ainda acontecem nos projetos, e que se houvesse uma preocupação maior com qualidade o índice de projetos fracassados seria bem menor.

Entrando mais em detalhes, as qualidades de código como redundância e legibilidade já são preocupações existentes desde as linguagens procedurais, onde se escreviam longos trechos de código. O próprio paradigma de orientação a objetos surgiu com o intuito de resolver muitos problemas. A primeira referência de design patterns, foi escrita pelo GoF em 1994, onde foram propostas soluções para problemas comuns no dia-dia de desenvolvimento de sistemas de software. Os princípios mencionados são tão antigos quanto os design patterns, pois estes são a base para um design pattern. Refatoração, ou seja, melhorar a qualidade do código-fonte sem afetar o comportamento externo do sistema, sempre existiu, e em 1999,

Fowler escreveu um livro mostrando algumas técnicas de refatoração que ele desenvolveu no decorrer da sua carreira, e que tem gerado resultados.

Algo que é um pouco mais recente no mercado é a disciplina de Test-Driven Development, criada por Kent Beck, em 2003, onde Kent escreveu um livro explicando a técnica passo-a-passo, e a mesma vem sendo cada vez mais adotada devido a sua eficácia.

Enfim, é difícil de se saber por quanto tempo esses problemas irão persistir, pois verifica se que muitas empresas ainda resistem em persistirem nos paradigmas tradicionais, mas espera-se que cada vez mais surjam novas ferramentas e frameworks que facilitem cada vez mais o trabalho de se criar um design emergente.

Referências Bibliográficas

(AMBLER, 2002): AMBLER, Scott W. **Agile Enterprise Architecture: Beyond Enterprise Data Modeling**. 2002. Disponível em: <http://www.agiledata.org/essays/enterpriseArchitecture.html>. Acesso em: 24 nov. 2010.

(AMBLER, 2006): AMBLER, Scott W. **The TAGRI (They Aren't Gonna Read It) Principle of Software Development**. 2006. Disponível em: <http://www.agilemodeling.com/essays/tagri.htm>. Acesso em: 24 nov. 2010.

(BAIN, 2008): BAIN, Scott L. **Emergent Design: The Evolutionary Nature of Professional Software Development**. Reading, MA: Addison-Wesley Professional, 2008.

(BUG MILENIO, 2010): **Year 2000 problem**. 2010. Disponível em: <http://en.wikipedia.org/wiki/Year_2000_problem>. Acesso em: 24 nov. 2010.

(CARNEGIE, 2010): **Software Engineering Institute | Carnegie Mellon: Software Architecture Definitions**. 2010. Disponível em: <http://www.sei.cmu.edu/architecture/start/bibliographicdefs.cfm>. Acesso em: 24 nov. 2010.

(CHECKSTYLE, 2010): **Checkstyle**. 2010. Disponível em: <http://checkstyle.sourceforge.net/>. Acesso em: 24 nov. 2010.

(CODE CONVENTIONS, 1999): **Oracle: Code Conventions for the Java TM Programming Language**. 1999. Disponível em: <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>. Acesso em: 24 nov. 2010.

(EASYMOCK, 2010): **Easymock**. 2010. Disponível em: <http://easymock.org/>. Acesso em: 24 nov. 2010.

(FORD, 2009.1): FORD, Neal. **Evolutionary architecture and emergent design: Refactoring toward design**. 2009. Disponível em:

<http://www.ibm.com/developerworks/java/library/j-eaed5/index.html>. Acesso em: 24 nov. 2010.

(FORD, 2009.2): FORD, Neal. **Evolutionary architecture and emergent design: Test-driven design, Part 1**. 2009. Disponível em: <http://www.ibm.com/developerworks/java/library/j-eaed2/index.html>. Acesso em: 24 nov. 2010.

(FORD, 2009.3): FORD, Neal. **Evolutionary architecture and emergent design: Test-driven design, Part 2**. 2009. Disponível em: <http://www.ibm.com/developerworks/java/library/j-eaed3/index.html>. Acesso em: 24 nov. 2010.

(FORD, 2010): FORD, Neal. **Evolutionary architecture and emergent design: Evolutionary architecture**. 2010. Disponível em: <http://www.ibm.com/developerworks/java/library/j-eaed10/index.html#distinguishing>. Acesso em: 24 nov. 2010.

(FOWLER, 2001): FOWLER, Martin. **Separating User Interface Code**. 2001. Disponível em: <http://martinfowler.com/ieeeSoftware/separation.pdf>. Acesso em: 24 nov. 2010.

(FOWLER, 2002): FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Reading, MA: Addison-Wesley Professional, 2002.

(FOWLER, 2003): FOWLER, Martin. **Who Needs an Architect?.** 2003. Disponível em: <http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf>. Acesso em: 24 nov. 2010.

(FOWLER, 2004.1): FOWLER, Martin. **Is Design Dead?.** 2004. Disponível em: <http://martinfowler.com/articles/designDead.html>. Acesso em: 24 nov. 2010.

(FOWLER, 2004.2): FOWLER, Martin. **Inversion of Control Containers and the Dependency Injection pattern**. 2004. Disponível em: <http://martinfowler.com/articles/injection.html>. Acesso em: 24 nov. 2010.

(FOWLER, 2005): FOWLER, Martin. **Patterns in Enterprise Software**. 2005. Disponível em: <http://martinfowler.com/articles/enterprisePatterns.html>. Acesso em: 24 nov. 2010.

(FOWLER, 2010.1): **Martin Fowler**. 2010. Disponível em: <http://martinfowler.com/>. Acesso em: 24 nov. 2010.

(FOWLER, 2010.2): **Refactoring in Alphabetical Order**. 2010. Disponível em: <http://refactoring.com/catalog/index.html>. Acesso em: 24 nov. 2010.

(FOWLER BBOR, 1999): FOWLER, Martin; BECK, Kent; BRANT, John; OPDYKE, William; ROBERTS, Don. **Refactoring: Improving the Design of Existing Code**. Reading, MA: Addison-Wesley Professional, 1999.

(GANG OF FOUR, 1994): GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John M. **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA: Addison-Wesley Professional, 1994.

(JUNIT, 2010): **JUnit.org Resources for Test Driven Development**. 2010. Disponível em: <http://www.junit.org/>. Acesso em: 24 nov. 2010.

(KARLESKY WBF, 2007): KARLESKY, Michael; WILLIAMS, Greg; BEREZA, William; FLETCHER, Matt. **Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns**. 2007. Disponível em: <http://www.methodsandtools.com/archive/archive.php?id=59>. Acesso em: 24 nov. 2010.

(KERIEVSKY, 2004): KERIEVSKY, Joshua. **Refactoring to Patterns**. Reading, MA: Addison-Wesley Professional, 2004.

(MARTIN, 1996.1): MARTIN, Robert C. **The Open-Closed Principle**. 1996. Disponível em: <http://www.objectmentor.com/resources/articles/ocp.pdf>. Acesso em: 24 nov. 2010.

(MARTIN, 1996.1): MARTIN, Robert C. **The Dependency Inversion Principle**. 1996. Disponível em: <http://www.objectmentor.com/resources/articles/dip.pdf>. Acesso em: 24 nov. 2010.

(MARTIN, 2000): MARTIN, Robert C. **Design Principles and Design Patterns**. 2000. Disponível em: <http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>. Acesso em: 24 nov. 2010.

(SIERRA; BATES, 2005): SIERRA, Katherine; BATES, Bert. **SCJP Sun Certified Programmer for Java 5 Study Guide**. San Francisco, CA: McGraw-Hill Osborne Media, 2005.

(SHALLOWAY, 2008): SHALLOWAY, Alan. **Can Patterns Be Harmful?**. 2008. Disponível em: <http://www.netobjectives.com/files/resources/CanPatternsBeHarmful.pdf>. Acesso em: 24 nov. 2010.

(WATERS, 2007): WATERS, Kelly. **Agile Development: Enough's enough!**. 2007. Disponível em: <http://kw-agiledevelopment.blogspot.com/2007/04/agile-principle-8-enoughs-enough.html>. Acesso em: 24 nov. 2010.