ISAIAS ALVES FERREIRA

ANÁLISE DO IMPACTO DA APLICAÇÃO DE PADRÕES DE PROJETO NA MANUTENIBILIDADE DE UM SISTEMA ORIENTADO A OBJETOS

LAVRAS - MG
2012

ISAIAS ALVES FERREIRA

ANÁLISE DO IMPACTO DA APLICAÇÃO DE PADRÕES DE PROJETO NA MANUTENIBILIDADE DE UM SISTEMA ORIENTADO A OBJETOS

Monografia de graduação apresentada ao colegiado do curso de Sistemas de Informação para obtenção do título de Bacharel em Sistemas de Informação.

Orientador:
Prof. Dr. Antônio Maria Pereira de Resende

LAVRAS - MG
2012

ISAIAS ALVES FERREIRA

ANÁLISE DO IMPACTO DA APLICAÇÃO DE PADRÕES DE PROJETO NA MANUTENIBILIDADE DE UM SISTEMA ORIENTADO A OBJETOS

Monografia de graduação apresentada ao colegiado do Curso de Sistemas de Informação para obtenção do título de Bacharel em Sistemas de Informação.

APROVADA em 16 de outubro de 2012.

ANDRÉ PIMENTA FREIRE

IGOR RIBEIRO LIMA

ANTÔNIO MARIA PEREIRA DE RESENDE (orientador)

LAVRAS - MG
2012

Ao meu amigo mais chegado que um irmão Elias Caetano de Souza (in MEMORIAN).
Que mesmo sem saber, foi a chave que abriu a porta para a realização deste grande sonho.
A você dedico este trabalho.

Agradecimentos

Agradeço em especial aos meus pais Jesum Delgado Ferreira e Elizabeth Alves Ferreira, que não mediram esforços, contribuindo em absolutamente tudo durante toda a minha vida para que esse sonho se tornasse realidade. A eles que se doaram por mim.

Agradeço à minha esposa Daniela que esteve ao meu lado em todos os momentos durante toda essa minha jornada, que chorou e se alegrou comigo, que esteve acordada quando eu estive acordado, que me serviu quando precisei e que sempre acreditou em mim.

Aos meus irmãos Wesley, Gláucia e Elias (in Memorian), que longe, perto ou mesmo sem saber, estiveram sempre comigo.

Ao meu orientador, Prof. Antônio Maria pelo apoio, disposição e ajuda durante a execução desse trabalho. Pelos ensinamentos que certamente levarei por toda a vida.

Agradeço ao Laboratório de Estudos e Projetos em Manejo Florestal (LEMAF) e ao seu diretor de TI Samuel Campos por acreditar no meu trabalho e pela imensa oportunidade de aprendizado que me concedeu. Ao meu colega e amigo Fernando Simeone pela paciência, dedicação e grande ajuda e a todos os demais colegas pela contribuição em meu aprendizado.

“Qualquer tonto é capaz de escrever um código que um computador consiga entender. Os bons programadores escrevem códigos que outros humanos consigam entender.”

MARTIN FOWLER

RESUMO

A manutenção é parte indispensável no ciclo de vida de um software, seja ela adaptativa, corretiva, preditiva ou perfectiva. Por esse motivo, é importante que se saiba o grau de dificuldade de se alterar um sistema, o que permitiria traçar estratégias para minimizá-lo, aumentando a qualidade e diminuindo custos. Esse grau de dificuldade é denominado no desenvolvimento de software como "Manutenibilidade de Software". Este trabalho realiza um estudo de caso para analisar e mensurar o impacto sofrido pela manutenibilidade de um sistema orientado a objetos por meio da aplicação de padrões de projeto ao sistema. Para isso, utilizou-se um sistema desenvolvido em JAVA J2EE, onde seu código fonte foi alterado aplicando-se determinados padrões de projeto obtendo-se então um código refatorado. Após a refatoração do sistema legado, foram aplicadas métricas de software voltadas para manutenibilidade como as métricas de Halstead, a Complexidade Ciclomática de McCabe e a MI (Maintainability Index) ao sistema legado e ao novo sistema, a fim de comparar os resultados das métricas em uma análise detalhada relacionando as melhorias obtidas aos padrões aplicados. A análise dos resultados permitiu que a melhoria da qualidade do sistema fosse quantificada e verificada, onde os resultados de todas as métricas aplicadas ao sistema refatorado foram satisfatórios em relação aos resultados das mesmas métricas aplicadas ao sistema legado. Pretendeu-se então evidenciar, a partir de um estudo de caso, que padrões de projeto de software podem contribuir efetivamente para a melhoria da manutenibilidade e consequentemente da qualidade de um software.

Palavras Chave: Manutenibilidade, Padrões de Projeto, Boas Práticas de Programação, Métricas de Software.

ABSTRACT

The maintenance of software systems is an indispensable part in its life cycle, be it adaptive, corrective or perfective. For this reason, it is important to be aware of the difficulty level to modify a system. This can help outline strategies to minimize the difficulty to maintenance, increasing quality and decreasing costs. This difficulty level is called in the software industry as "Software Maintainability". This work performs a case study to analyze and measure the impact on the maintainability of an object-oriented system through the application of design patterns on the system. For this, we used a system developed in JAVA J2EE, where the source code was modified by applying certain design patterns yielding a refactored code. After refactoring the legacy system, we applied software metrics focused on maintainability like Halstead metrics, McCabe cyclomatic complexity and MI (Maintainability Index) to the legacy system and to the new system, with the purpose of comparing the results of the metrics on a detailed analysis relating the improvements obtained to the patterns applied. The analysis of the results of the metrics applied showed evidence of improvement of the quality of the system, which was measured and checked. The results of all metrics applied to the refactored system were satisfactory in relation to the results of the same metrics applied the legacy system. It was intended to show from a case study that software design patterns can contribute effectively to improving maintainability and therefore the quality of software.

Keywords: Maintainability, Design Patterns, Good Programming Practices, Software Metrics.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

SUMÁRIO

1. INTRODUÇÃO

1.1 Contextualização e Motivação

Manutenibilidade de software é um atributo de qualidade de software que determina o grau de facilidade com que o mesmo pode ser corrigido ou alterado diante de uma necessidade (IEEE, 1993). Coleman (1994) citado por Brusamolin (2004), afirma que a manutenção de um software pode consumir tanto ou mais recursos que o desenvolvimento do mesmo, e por esse motivo, é extremamente importante que o fator manutenibilidade seja levado em consideração durante todas as etapas de desenvolvimento.

A manutenção de software pode ser classificada em diferentes tipos. São eles a corretiva, a adaptativa, a preditiva e a perfectiva. Segundo Pigoski (1996) dentre essas classificações, em geral, a maior parte dos esforços de manutenção efetuados são para a manutenção perfectiva, que também é conhecida como manutenção evolutiva. Esse tipo de manutenção objetiva adicionar novas funcionalidades ao sistema. Em outras palavras, a manutenção perfectiva ocorre quando o proprietário, por algum motivo, solicita que um novo requisito ou uma nova funcionalidade faça parte do software.

Com o passar do tempo, cresce a exigência de que os softwares em geral sejam cada vez mais capazes de executar uma quantidade maior de novas funcionalidades. Entretanto, na maioria dos casos, criar um novo sistema para cada nova funcionalidade se mostra como um problema, pois manter vários sistemas ao mesmo tempo pode exigir mais esforços e, consequentemente, mais recursos financeiros. Este problema pode ser corrigido ou minimizado se a manutenibilidade de um sistema for propícia à

adição de novas funcionalidades, bem como à execução de todos os tipos de manutenção. Nesse sentido, a aplicação de determinados padrões e boas práticas de programação podem melhorar significativamente a manutenibilidade de um software. De acordo com Lauder e Kent (1998), esses padrões são resultados de boas práticas que ajudam a minimizar problemas recorrentes no desenvolvimento de sistemas.

Além de poderem contribuir para o aumento da manutenibilidade geral de um sistema, alguns padrões e boas práticas impactam diretamente aos blocos de código escrito quanto à sua estrutura e disposição. Nesses casos, a legibilidade é facilitada quando o código propriamente dito é escrito de uma forma fluente, ou seja, quando a própria escrita se torna intuitiva. Assim, o código não necessita de tantos comentários, ficando mais limpo e podendo ser entendido com mais facilidade.

Em casos de manutenção, a aplicação de padrões e boas práticas de software podem implicar na necessidade de que partes do sistema passem por processos de refatoração, ou seja, que o código de certa forma, seja reescrito para que passe a contemplar os padrões aplicados. De acordo com Fowler (1999) o termo refatoração pode ser definido como "*o processo de alterar um software orientado a objetos de maneira que o seu comportamento externo não seja alterado, mas sua estrutura interna seja melhorada*". A necessidade de refatoração pode ser identificada por meio de métricas de software aplicadas ao sistema revelando resultados insatisfatórios, ou até mesmo, quando novas funcionalidades precisam ser adicionadas ao sistema e a arquitetura do sistema dificulta esta adição não sendo favorável a ações de manutenção.

Sabe-se, entre os profissionais de Tecnologia da Informação, que os padrões e as boas práticas de programação, de fato melhoram e ajudam no desenvolvimento dos sistemas. Entretanto, em geral, a descrição das

aplicações dos padrões não apresentam de forma clara quais atributos de qualidade de software os padrões impactam diretamente.

Nota-se, atualmente, a falta de estudos que demonstrem o real impacto do uso de padrões de projeto de software e de boas práticas de programação na manutenibilidade dos sistemas.

1.2 Objetivo Geral

O objetivo deste trabalho de conclusão de curso é *mensurar quantitativamente o impacto da aplicação de padrões de projeto e boas praticas de programação na refatoração de um sistema legado*.

1.3 Objetivos Específicos

Para a consecução do objetivo geral, os seguintes objetivos específicos foram estabelecidos:

- Fazer um levantamento de padrões de projeto de software, boas práticas de programação e métricas voltadas para manutenibilidade;
- Selecionar um sistema legado;
- Analisar o estado inicial do sistema legado;
- Selecionar padrões de projeto e boas práticas de programação a serem aplicados;
- Refatorar o sistema legado, aplicando os padrões selecionados;
- Aplicar as métricas no sistema legado e no novo sistema;

- Comparar as medidas obtidas;

- Apontar o quanto a medida de uma determinada métrica pode ser melhorada com a aplicação de um determinado padrão;

1.4 Estrutura do documento

Este trabalho de conclusão de curso traz inicialmente no capítulo 1 denominado "*Introdução*", uma contextualização dos assuntos abordados e a motivação pela escolha do tema em questão. Em seguida descrevem-se os principais objetivos da realização dessa pesquisa.

O Capítulo 2, denominado "*Referencial Teórico*" traz um embasamento bibliográfico dos conhecimentos necessários para se desenvolver a pesquisa proposta. Os assuntos abordados por este capítulo iniciam-se com uma introdução da atual evolução de software, seguida por manutenção, manutenibilidade e os fatores que impactam a manutenibilidade de software. Em seguida tem-se a descrição de algumas métricas de software voltadas para manutenibilidade. Por fim, ainda no Capítulo 2 abordam-se alguns padrões de projeto e boas práticas de programação utilizadas na pesquisa.

A "*Metodologia*", descrita no Capítulo 3, classifica esta pesquisa e traz os procedimentos metodológicos adotados durante sua execução.

O Capítulo 4, denominado "*Estudo de Caso*", relata a aplicação prática da pesquisa e como ela se desenvolveu.

O Capítulo 5 denominado "Coleta de Dados e Análise dos Resultados" descreve os resultados obtidos após o estudo de caso e faz a análise desses resultados. Por fim o capítulo 6 traz as "*Conclusões*".

2. REFERENCIAL TEÓRICO

Neste capítulo, apresentam-se os principais conceitos necessários para a compreensão do presente trabalho de conclusão de curso. Inicialmente este capítulo traz uma introdução sobre evolução de software. Em seguida apresentam-se os conceitos de manutenção e manutenibilidade de software. Logo após, são apresentados alguns fatores que impactam diretamente à manutenibilidade de um sistema. São eles a arquitetura, tecnologia, documentação e compreensibilidade. Em seguida são apresentadas algumas métricas de software voltadas para manutenibilidade. Dentre elas estão às medidas de Halstead, a Complexidade Ciclomática de McCabe e a Maintainability Index (MI). Após toda a apresentação conceitual de manutenção, manutenibilidade e métricas de software, detalham-se alguns padrões de projeto e boas práticas de software.

2.1 Evolução de Software

A constante execução de manutenção perfectiva é uma realidade nos sistemas atualmente. Segundo Sommerville (2007), após a implantação dos sistemas, as mudanças são inevitáveis para que eles permaneçam úteis. Cada vez mais, os sistemas precisam evoluir para atender a novas funcionalidades. Isso ocorre à medida que as organizações adicionam novas regras em seus negócios e alteram as regras de negócio já existentes. Se uma organização precisasse de um novo sistema para atender às suas evoluções e às suas mudanças, além de não ser viável financeiramente, ficaria mais complexo administrar tantos sistemas distintos. Eles poderiam ser escritos em linguagens diferentes, não se comunicariam diretamente e exigiriam muitos esforços para manterem-se consistentes. Diante disso, as manutenções perfectivas se apresentam como a melhor opção para que um sistema de informação se mantenha atualizado.

Ainda segundo Sommerville (2007), as organizações atuais estão cada vez mais dependentes de sistemas de software para o bom andamento de seus negócios, os quais na maioria das vezes custam caro. Por esse motivo, os sistemas passaram a ser importantes ativos de negócios e, sendo assim, os investimentos nesses ativos devem ser constantes para que o valor desses ativos seja mantido.

Erlikh (2000) citado por Sommerville (2007) sugere que 90% dos custos de software estão na evolução, mas alerta que existe uma grande variação nesse percentual quando os custos de evolução são distinguidos dos de manutenção. Estas palavras de Erlikh apontam que a maioria das mudanças sofridas pelos sistemas de software é em decorrência dos novos requisitos gerados pelas mudanças nas regras de negócio ou por novas necessidades dos usuários.

Diante dessas informações, pode-se enxergar a Engenharia de Software como um processo em espiral com requisitos, especificação e projeto, implementação e teste durante todo o ciclo de vida do sistema onde versões (ou releases) são criadas a cada volta no espiral. Isso pode ser observado na Figura 1.

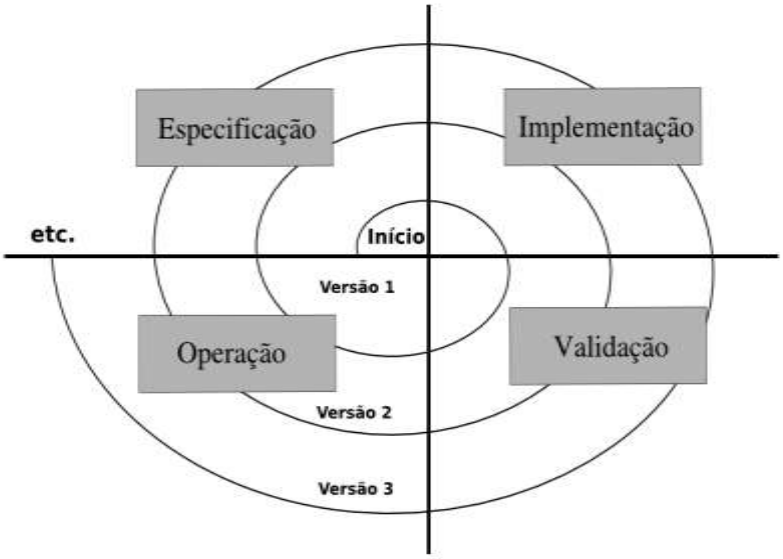

Figura 1 - Ciclo de vida de um Software (adaptado de Sommerville, 2007)

Sommerville (2007) relata que a maioria dos trabalhos na área da evolução de software foram conduzidos por Lehman e Belady. Eles começaram com as pesquisas nas décadas de 1970 e 1980 e o trabalho continuou ainda pelas décadas de 1990 e 2000. Baseado nos seus estudos e pesquisas, eles propuseram 5 leis ou hipóteses relacionadas às mudanças nos sistemas de software, são as Leis de Lehman. Para propor as leis, eles examinaram o crescimento e evolução de um número considerável de sistemas de software de grande porte. Lehman e Belady defendem que essas leis são amplamente aplicáveis e não sofrem variações.

A primeira lei de Lehman afirma que a manutenção de um sistema de software é um processo constante e inevitável. À medida que o ambiente muda, novas necessidades surgem, novos requisitos aparecem, e o sistema deve ser modificado para atendê-las. Quando o sistema modificado é colocado em operação, mais mudanças ocorrem e o ciclo se inicia

novamente dando continuidade ao processo de evolução (Sommerville, 2007).

A segunda lei de Lehman estabelece que quando um sistema é alterado, inevitavelmente sua estrutura é degradada. Ou seja, a complexidade do sistema tende a aumentar durante os processos de mudança. Lehman ainda afirma que uma forma de minimizar esse aumento da complexidade é investir em manutenções preventivas.

A terceira lei de Lehman afirma que os sistemas de grande porte possuem dinâmica própria definida ainda nos primeiros estágios do processo de desenvolvimento. Em outras palavras, essa lei expressa que um sistema pode ter um número de possíveis mudanças limitado e que essa dinâmica própria determina as tendências de manutenção. Lehman e Belady indicam que essa lei é consequência tanto de fatores estruturais do sistema quanto de fatores organizacionais que influenciam diretamente às mudanças no sistema. Sommerville (2007) afirma que essa é a mais interessante e polêmica lei de Lehman, pois ela diz respeito a um lado particular entre os sistemas, a cultura organizacional do seu proprietário e os fatores estruturais e arquiteturais utilizados durante todo o processo de desenvolvimento.

A quarta lei de Lehman diz que a maioria dos sistemas de grande porte acontece no estado saturado. Isso quer dizer que mudanças em recursos têm efeitos imperceptíveis na evolução dos sistemas no longo prazo, ou seja, a taxa de desenvolvimento é constante independentemente dos recursos dedicados.

A quinta lei de Lehman está relacionada aos incrementos de mudanças em cada nova versão do sistema. A cada adição de uma nova funcionalidade ao sistema, inevitavelmente são inseridos novos defeitos. Esta lei ainda sugere que não se devem planejar um grande incremento de

funcionalidades sem levar em consideração uma posterior necessidade de correção dos defeitos.

Lei	Descrição
Mudança Contínua	Um programa usado em um ambiente real deve necessariamente mudar ou irá se tornar progressivamente menos útil.
Complexidade Crescente	À medida que um sistema muda, sua estrutura tende a se tornar mais complexa. Recursos extras devem ser dedicados para preservar e simplificar a estrutura.
Evolução de sistemas de grande porte	A evolução de um sistema é um processo auto-regulável. Atributos de sistemas como tamanho, tempo entre versões e número de erros reportados é quase invariável em cada versão de sistema.
Estabilidade Organizacional	Durante o ciclo de vida de um sistema, sua taxa de desenvolvimento é quase constante e independente de recursos dedicados ao desenvolvimento do sistema.
Conservação de familiaridade	Durante o ciclo de vida de um sistema, mudanças incrementais em cada versão são quase constantes.
Crescimento contínuo	A funcionalidade oferecida pelos sistemas deve aumentar continuamente para atender a satisfação do usuário.
Qualidade em declínio	A qualidade dos sistemas entrará em declínio a menos que eles sejam adaptados a mudanças em seus ambientes operacionais.
Sistema de feedback	Os processos de evolução incorporam sistemas de feedback com vários agentes e loops e eles devem ser tratados como sistemas de feedback para conseguir aprimoramentos significativos de produto.

Tabela 1 – Outras Leis de Lehman (adaptado de Sommerville, 2007)

Estas cinco leis foram as propostas iniciais de Lehman. Existem ainda outras leis que foram adicionadas por trabalhos posteriores. A Tabela 1 retirada de Sommerville (2007) descreve brevemente todas as leis de Lehman.

Dentro do contexto desse trabalho de conclusão de curso, e considerando que as manutenções perfectivas são constantes, os sistemas que possuem maiores facilidades de manutenção acabam se saindo mais competitivos e confiáveis, além de mais baratos. Pode-se considerar que atualmente, o grande desafio na produção de sistemas é o de construí-los de forma que a adição de novas funcionalidades seja do menor impacto possível. Para que esse impacto seja minimizado, existem técnicas e práticas a serem aplicadas durante o desenvolvimento ou até mesmo durante uma eventual refatoração. Este trabalho de conclusão de curso apresenta em outras seções uma série de padrões de projeto de software e boas práticas de programação que, quando aplicadas, melhoram a manutenibilidade do sistema.

A evolução dos sistemas em forma de manutenções perfectivas está diretamente ligada às questões acerca da manutenibilidade. Quando não há uma preocupação em se manter os níveis de manutenibilidade de um sistema, as manutenções efetuadas ao longo do tempo podem deixar o código de uma forma tão complexa que torna as manutenções cada vez mais difíceis de serem realizadas, além de aumentar os riscos de alguma manutenção impactar em outras partes do sistema (Criscuolo, 2008).

Na maioria dos casos, os softwares sofrerão mudanças em sua estrutura e principalmente em seu código para que atenda a novos requisitos. Entretanto, a manutenção que o sistema sofrerá no futuro pode ser facilitada durante todo o seu desenvolvimento e é isso que se espera de um sistema legado.

2.2 Manutenção de Software

Pigoski (1996), sintetiza que "*Manutenção de software é a totalidade de atividades necessárias para prover, minimizando o custo,*

suporte a um sistema de software". Para Sommerville (2007), "*A manutenção de software é um processo geral de mudanças de um sistema depois que ele é entregue*". Pressman (2006) destaca a manutenção como algo "*bem mais do que consertar erros*" depois que o sistema é entregue. Ele descreve a manutenção de software separando-a em quatro diferentes atividades que são a manutenção corretiva, a adaptativa, a perfectiva e a preventiva.

Este trabalho de conclusão de curso adota a definição de Pigoski (1996), tendo a manutenção de software como sendo a totalidade das atividades necessárias para prover suporte a um sistema de software, minimizando o custo.

Segundo a norma NBR ISO/IEC 12207 existem três principais tipos de manutenção de software. São eles:

- **Manutenção Corretiva** que visa unicamente à correção de eventuais erros identificados por qualquer natureza;

- **Manutenção Adaptativa** que corresponde às mudanças que o software precisa sofrer quando se decide por mudá-lo de contexto alterando partes do software para que essa adaptação seja efetuada;

- **Manutenção Perfectiva** que corresponde às mudanças efetuadas para atender a novas necessidades do usuário, ou seja, são aquelas mudanças efetuadas quando o software precisa evoluir, quando precisa ser estendido para que possua novas funcionalidades;

Segundo Pigoski (1996), 55% dos esforços de manutenção de software são para a manutenção perfectiva, 25% para a adaptativa e 20% para a corretiva conforme apresentado no gráfico da Figura 2.

Figura 2 - Aplicação dos tipos de manutenção de software (Pigoski, 1996)

O fato de a maior parte dos esforços de manutenção ser efetuada no intuito de se adicionar novas funcionalidades aos sistemas demonstra um forte indício de que os sistemas precisam estar em constante evolução. À medida que o tempo passa, os sistemas podem ficar defasados ou ultrapassados tanto em relação às suas funcionalidades, que podem mudar com o tempo, quanto pela tecnologia utilizada no seu desenvolvimento.

A manutenção ou o grau de dificuldade de se alterar um sistema é mensurado por um atributo de qualidade de software denominado "*manutenibilidade*" (ISO/IEC 9126-1, 2000). O tópico a seguir caracteriza esse importante atributo de qualidade de software.

2.3 Manutenibilidade de Software

A norma ISO/IEC 9126-1 define seis características e suas subcaracterísticas para se mensurar a qualidade de um produto de software, como apresentada na Figura 3.

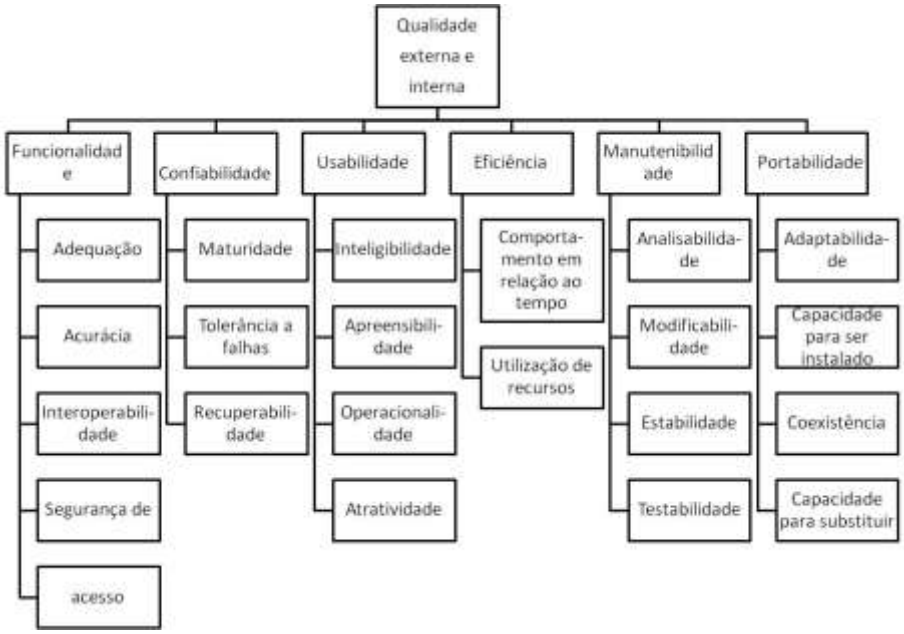

Figura 3 - Atributos chave de qualidade ISO/IEC 9126-1 (2000)

Essas características denominam-se Funcionalidade, Confiabilidade, Usabilidade, Eficiência, Manutenibilidade e Portabilidade, explicadas a seguir:

Funcionalidade é o nível de satisfação das necessidades declaradas que se espera que o software compreenda. Em suma, é a capacidade que o software tem de realizar com eficácia as funcionalidades que lhe foram requeridas.

Confiabilidade é o nível de confiança que se pode ter no software levando em consideração sua maturidade, sua tolerância a falhas e sua capacidade de se recuperar.

Usabilidade é o grau de facilidade de uso que o software possui levando em conta a inteligibilidade, facilidade de aprendizado e operabilidade.

Eficiência diz respeito ao comportamento do software em relação ao tempo e aos recursos existentes onde quanto maior a otimização dos recursos do sistema em relação a estes atributos, maior é sua eficiência.

Manutenibilidade é a facilidade que o software possui de sofrer mudanças e se os impactos sofridos pelas mudanças são suficientemente baixos.

Portabilidade é a facilidade com que o software se permite ser levado de um ambiente para outro.

Dentre estes atributos chave de qualidade descritos, o foco deste trabalho de conclusão de curso se concentra na Manutenibilidade que está sendo detalhada a seguir.

A manutenibilidade é justamente a facilidade encontrada no software para que o mesmo seja modificado (IEEE, 1993). A norma ISO/IEC 9126-1 ainda define cinco subcaracterísticas da manutenibilidade, são elas:

a) **Analisabilidade** que evidencia o esforço necessário para diagnosticar alguma deficiência no sistema seja ela de qualquer natureza;

b) **Modificabilidade** que evidencia o esforço necessário para modificar o software diante de uma eventual necessidade;
c) **Estabilidade** que evidencia o risco de acontecerem efeitos inesperados ocasionados por alguma modificação efetuada;
d) **Testabilidade** que evidencia o esforço necessário para validar o software mesmo após ter sofrido modificações;
e) **Conformidade** que analisa se o software está condizente com padrões que permitirão ao mesmo ser alocado em diferentes ambientes. Em outras palavras, se suas "conexões" estão padronizadas;

Conforme Pigoski (1996), 55% dos esforços de manutenção de software são elaborados no intuito de adicionar novas funcionalidades aos sistemas. Por esse motivo, durante as etapas de implementação do código, deve-se levar em consideração que serão grandes as chances de ocorrer alguma modificação num momento futuro para que alguma nova funcionalidade faça parte do sistema e que isso impacte naquela lógica que está sendo implementada. Seria ideal poder adicionar uma nova funcionalidade sem a necessidade de modificar o que já existe. Esse nível de manutenibilidade é muito difícil de ser alcançado, entretanto é possível deixá-lo satisfatório.

O nível de dificuldade de adicionar uma nova funcionalidade pode ser facilmente notado ao se ter uma ideia da quantidade de código a ser alterada para a adição de uma nova funcionalidade. Obviamente, se ao ser adicionada uma nova funcionalidade, além de implementar o código responsável unicamente pela sua lógica, também for necessário modificar diversas partes do software, a manutenção terá um custo muito maior comparado ao custo de alterar apenas pontos específicos e centralizados no código (Brusamolin, 2004).

De acordo com Martins (2002), alguns fatores que impactam diretamente a manutenibilidade de um sistema são:

- Arquitetura
- Tecnologia
- Documentação
- Compreensibilidade

Algumas influências destes fatores poderão ser observadas nos tópicos a seguir.

2.4 Fatores que impactam a Manutenibilidade

A manutenibilidade pode ser influenciada por diversos fatores, dentre eles, destacam-se arquitetura, tecnologia, documentação e compreensibilidade. Cada um destes fatores pode afetar de maneira positiva ou negativa a manutenibilidade de um sistema de software. A seguir, apresentam-se cada um desses fatores.

2.4.1 Fator Arquitetura

Do ponto de vista da arquitetura, foram elaborados testes de desenvolvimento de um mesmo sistema em diversas arquiteturas diferentes chegando à conclusão de que o esforço para modificar o software foi menor e menos erros foram identificados quando se utilizou unidades de código menores (Martins, 2002).

2.4.2 Fator Tecnologia

No contexto da tecnologia, Henry e Humphrey (1990 apud Brusamolin, 2004) observaram que softwares produzidos em linguagens orientadas a objeto possuem alto índice de manutenibilidade ou podem alcançar esses altos níveis.

2.4.3 Fator Documentação

Quanto à documentação, sua falta pode implicar em um alto gasto de tempo para compreender o produto antes de modificá-lo. Se a manutenção deve ser de menor impacto e custo possíveis, o código deve ser o mais compreensível possível. Uma vez que a arquitetura do software, as tecnologias utilizadas na sua produção e a documentação são consideradas satisfatórias e complementares entre si, pode se dizer que o software tem um bom índice de compreensibilidade.

2.4.4 Fator Compreensibilidade

Ainda destacando o fator Compreensibilidade, pode-se dizer que a legibilidade do código é trivial para alcançar bons níveis desta característica e que estas características estão diretamente ligadas entre si (Posnett, 2011). A compreensibilidade do código talvez seja um dos principais fatores para que a manutenção seja facilitada.

Legibilidade de software é a propriedade que remete a facilidade em que um trecho de código pode ser lido e entendido (Buse, 2008). Como ler e entender o código são tarefas que também precisam ser feitas constantemente durante o desenvolvimento dos sistemas, o fator legibilidade precisa ser considerado durante todo o ciclo de vida do software.

Segundo Buse (2008), existe um consenso entre desenvolvedores de que a legibilidade é uma característica essencial e determinante para a qualidade de um sistema. Um código mais legível é considerado mais manutenível principalmente porque, na maioria dos casos, um código altamente legível diminui o tempo gasto para se entender como a lógica do trecho que está sendo lido está organizada, além de estar mais fácil de ser alterada.

Legibilidade e entendimento de código são fatores que possuem um forte relacionamento. De acordo com Posnett et al. (2011), legibilidade é um aspecto sintático enquanto compreensibilidade é um aspecto semântico. Posnett et al. (2011), ainda ressalta que essencialmente, a legibilidade diz respeito à dificuldade de compreensão que o programador sente antes de alterar o corpo do código, onde quanto maior a legibilidade, menor é a dificuldade de compreensão.

Há certa dificuldade em se medir a legibilidade de um sistema, pois ela vem de percepções subjetivas e essas percepções podem variar bastante. Além disso, é difícil de obter essas percepções subjetivas, pois isso requer intensa análise de aspectos humanos além de grandes aplicações estatísticas. Entretanto, existe uma grande convergência de opiniões de desenvolvedores que consideram determinados aspectos fundamentais para a legibilidade (Buse, 2008). Estes aspectos, que são relativamente simples, podem ser mais facilmente analisados por estarem explícitos no código, como por exemplo:

- Identação
- Quantidade de caracteres utilizados
- Tamanho de métodos
- Quantidade de métodos

Buse (2008) propôs um modelo para mensurar a legibilidade de software de uma forma genérica. Seu modelo apresenta algumas características que foram consideradas como preditoras de legibilidade por terem grande influência sobre facilidade de leitura e compreensão de código. Estas características podem ser observadas na Tabela 2. A tabela também aponta quais dessas características podem ter seu valor máximo utilizado e quais devem ter seu valor médio utilizado.

Média	Máximo	Característica
✓	✓	Comprimento de linha (caracteres)
✓	✓	Identificadores
✓	✓	Comprimento dos identificadores
✓	✓	Identações (precedida por espaço em branco)
✓	✓	Palavras chave
✓	✓	Números
✓		Comentarios
✓		Períodos
✓		Vírgulas
✓		Espaços
✓		Parêntese
✓		Operadores Aritméticos
✓		Operadores de Comparação
✓		Atribuições (=)
✓		Ramificações (if)
✓		Laços (for, while)
✓		Linhas em branco
	✓	Ocorrências de caracteres únicos
	✓	Ocorrências de identificadores únicos

Tabela 2 - Características consideradas pelo modelo de Buse (Buse, 2008)

A partir da proposição destas características, pode-se obter pontos concretos a serem analisados quando se pretende verificar a legibilidade de um determinado trecho de código.

Como dito, vários aspectos influenciam a legibilidade de código. Dentre as características apontadas na Tabela 2 – características estas que estão sendo utilizadas nesse trabalho de conclusão de curso como determinantes para a legibilidade – acredita-se que uma das principais seja a identação do código. O paradigma orientado a objetos permite que uma mesma instrução seja escrita de diversas e diferentes formas. Entretanto, Maeda (2010) descreve como a identação do código e a escrita da forma mais simples possível, melhoram significativamente sua legibilidade. A Figura 4 exemplifica bem, e de uma forma muito simples, a facilidade de ler e entender o que um trecho de código corretamente identado representa. Esse trecho de código é um exemplo de um formato de escrita de dados estruturados denominado RugsOn (Maeda, 2010). Este formato para representação de dados estruturados é semelhante ao JSON e foi criado para representar dados de uma forma altamente legível usando para isso a linguagem Ruby e a sintaxe de Scala. Além de ser utilizado em Ruby e Scala, pode também ser utilizado por diversas linguagens de programação que rodam em *Java Virtual Machine*. O exemplo da Figura 4 descreve um elemento denominado *config* e que possui três elementos filhos, onde *"programming"* é o valor do elemento *time*, *"coffe"* é o valor do elemento *drink* e *"coffe stand"* é o valor do elemento *place*.

```
config {

        "programming"   .time
        "coffe"         .drink
        "coffe stand"   .place
}
```

Figura 4 - Um elemento com três elementos filhos (Maeda, 2010)

O trecho de código apresentado pela Figura 4 deixa claro que *config* é um elemento que possui três atributos e deixa também de forma bem explícita os valores desses três atributos. Essa identação, que ocorre ao descrever os elementos filhos de *config*, é a grande responsável por essa facilidade de leitura e compreensão do código.

Existem diferentes métricas de software que permitem mensurar a manutenibilidade de forma quantitativa. As medidas obtidas por meio de métricas seguem a premissa de que quanto maior o valor da manutenibilidade, maior a facilidade de alterar o código e, consequentemente, menor o esforço a ser empregado.

2.5 Métricas de Software

De uma forma geral, métricas de software podem ser descritas como formas de mensurar algum atributo de software e aferir tempo, esforço e recursos necessários na execução dos processos de software (Pressman, 2006). É muito importante que, durante os processos de desenvolvimento, sejam utilizadas medições para avaliar a qualidade dos produtos de software dentre outros fatores. A engenharia tem em sua natureza uma visão quantitativa sobre os processos que executa. Diferentemente de outras engenharias, a de software não está acondicionada a leis quantitativas básicas e triviais. Entretanto, os processos de software utilizam diversas medições para que sejam executados com qualidade. Membros da comunidade de software afirmam que um software não pode ser mensurável ou que a tentativa de mensurá-lo deveria ser adiada por ainda não se ter conhecimento suficiente para fazê-lo corretamente. Segundo Pressman (2006), isso é um erro. Existem muitas formas para se medir um software, de modo que a avaliação das métricas podem proporcionar resultados significativos no que diz respeito à melhoria dos processos e à qualidade do sistema. Nesse sentido a citação de DeMarco por Pressman (2006), é bem

contundente quando diz que "*a qualidade de um produto é função de quanto ele muda o mundo para melhor*".

Os fatores classificados pela norma ISO/IEC 9126-1 como triviais para identificar a qualidade de um software, podem ser medidos e utilizados de forma indireta para avaliar a qualidade de um sistema.

Analisando os sistemas desenvolvidos no paradigma orientado a objetos, um projetista experiente "*sabe*" como fazer com que um sistema implemente efetivamente os requisitos do cliente. Entretanto, à medida que o projeto cresce pode-se ter uma visão mais objetiva das características do projeto permitindo que se ganhe mais experiência no que diz respeito à qualidade do produto de software. Whitmire, citado por Pressman (2006) descreve nove características distintas que também podem ser mensuráveis num projeto orientado a objetos, são elas: tamanho, complexidade, acoplamento, suficiência, completeza, coesão, primitividade, similaridade e volatividade.

Segundo Sommerville (2001), o processo de medição de software, que pode fazer parte de um processo de controle de qualidade, passa por alguns estágios. São eles:

- **Escolha das medições a serem feitas:** Inicialmente devem ser formuladas as questões que se destinam a responder, ou seja, a escolha da métrica deve ser baseada no que se deseja saber.

- **Seleção dos componentes a serem avaliados:** Pode não ser necessário e nem viável que se utilize todos os componentes de um sistema de software em um processo de medição. Relativamente aos aspectos a serem avaliados, deve-se separar os componentes que

representarão a medição. Esses componentes devem ser os que efetivamente participam diretamente da lógica do caso analisado, ou seja, as componentes centrais que estão em uso constante.

- **Medição de características dos componentes:** Esse estágio significa efetuar as medições e os cálculos utilizando ferramentas específicas para isso, que pode ser uma ferramenta especialmente desenvolvida para o caso daquela medição ou pode estar incorporada nas ferramentas CASE utilizadas.

- **Identificação de medições anômalas:** Já feitas às medições, elas devem ser comparadas entre si e com outras anteriormente registradas. As comparações são feitas no intuito de encontrar valores com diferenças incomuns entre cada métrica. Se identificada tal diferença, há grande possibilidade de se haver problemas com aquele componente.

- **Análise de componentes anômalos:** Apenas a identificação de componentes com medições anômalas não é suficiente. É necessário também examinar esse componente para decidir se de fato, a diferença encontrada representa um problema ou se a influencia de outros fatores neutralizam a diferença incomum encontrada podendo não significar que haja problemas com a qualidade do componente.

Podemos destacar algumas relações existentes entre os atributos internos e externos de um software na Figura 5. Essas relações mostram como determinados fatores internos como a complexidade ciclomática do código, por exemplo, pode influenciar atributos externos como a manutenibilidade.

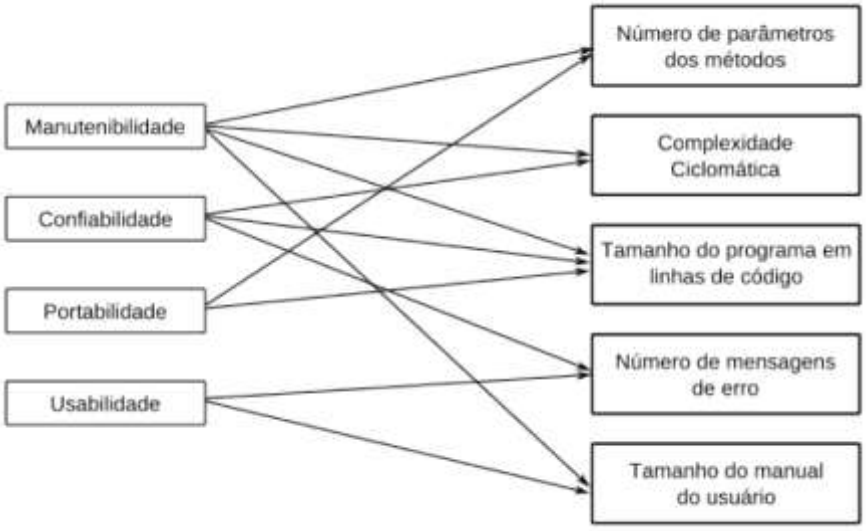

Figura 5 - Relacionamentos entre atributos de software externos e internos (Sommerville, 2001)

Nos tópicos a seguir serão apresentadas métricas voltadas para a manutenibilidade de software onde, pelas quais, é possível obter um dado quantitativo sobre o índice de manutenibilidade de um sistema.

2.5.1 Métricas de Halstead

A teoria de Halstead (Halstead, 1977), foi a primeira a associar leis quantitativas ao desenvolvimento de softwares através de um conjunto de medidas que podem ser retiradas do código após o mesmo ser gerado. Estas medidas são os *operandos*, que são valores operados por alguma instrução, e os *operadores*, que são as instruções que modificam os operandos. Um operando pode ser uma variável do tipo inteiro que é alterada por um operador de soma (+), por exemplo.

- **n1** – O número de operadores distintos que aparece em um programa ;

- **n2** – O número de operandos distintos que aparece em um programa;
- **N1** – O numero total de ocorrências dos operadores;
- **N2** – O número total de ocorrências dos operandos;

Através destas medidas primitivas pode-se obter:

Medida	Fórmula
Extensão do programa	N = N1 + N2
Vocabulário do programa	n = n1 + n2
Volume	V = N*(log_2n)
Dificuldade	D = (n1/2) * (N2/2)
Esforço	E=D*V

Tabela 3 - Medidas de Halstead

Onde:

a) **Extensão do programa**, é a soma da totalidade de operadores e operandos identificados. Ela dá a noção do quão grande e extenso é o código. Essa métrica tem sido defendida como mais eficiente que medir apenas o número de linhas de código (LOC) porque mostra o que realmente interessa no código para ser medido e não a totalidade das linhas que podem ser desproporcionais a essa medida. Quanto maior a extensão, maior a possibilidade de haver erros no código, por isso é interessante que este número seja diminuído.

b) **Vocabulário do Programa** é a métrica obtida pela soma dos operadores e operandos distintos de um programa. Ela traz o tamanho do grupo de símbolos distintos presentes no código,

diferentemente da extensão do programa que traz esse número na totalidade.

c) **Volume do Programa** é dado em função do comprimento e do vocabulário. Essa métrica dá a noção do quão volumoso e denso está o código. Quanto maior for esse número, mais difícil pode ficar a manutenibilidade do software, pois isso indica que pode ser muito mais difícil e complexo o entendimento do código. Além disso, a adição de uma nova funcionalidade, nesse caso, pode afetar um número maior de diferentes locais no código o que também pode tornar a manutenção mais difícil de ser efetuada.

d) **Dificuldade** é a medida que determina o seu grau na implementação do código. Se houve uma grande dificuldade na implementação, pressupõe-se que a dificuldade para a manutenção do código também pode ser grande.

e) **Esforço** que é obtido através do produto da Dificuldade pelo Volume do programa traduz a força necessária para a implementação e para a manutenção do código. Quanto maior essa métrica, mais complexa poderá ser a manutenção do código.

2.5.2 Complexidade Ciclomática de McCabe

Criada por Thomas McCabe em 1976, ela mede o número de caminhos linearmente independentes dentro de um módulo (BRUSAMOLIN, 2004). Ela é largamente utilizada e não depende de uma

linguagem. Montando-se o grafo dos possíveis fluxos dentro do programa podem-se obter os dados para se calcular a complexidade ciclomática. A fórmula do cálculo da complexidade ciclomática é apresentada na Equação 1.

$$CC = E - N + 2p$$

Equação 1 - Fórmula da Complexidade Ciclomática de McCabe (McCabe, 1976)

Onde:

- **E** – número de arestas do grafo
- **N** – número de nodos do grafo
- **p** – número de componentes conectados

A Figura 6 (McCabe, 1976), apresenta um grafo que representa os possíveis caminhos, ou os possíveis fluxos que um algoritmo, por exemplo, pode assumir durante sua execução. O grafo da figura possui 19 arestas, 13 nodos e 8 componentes conectados. Os componentes conectados podem ser medidos contando-se as regiões “fechadas” por arestas do grafo e somando-se mais um.

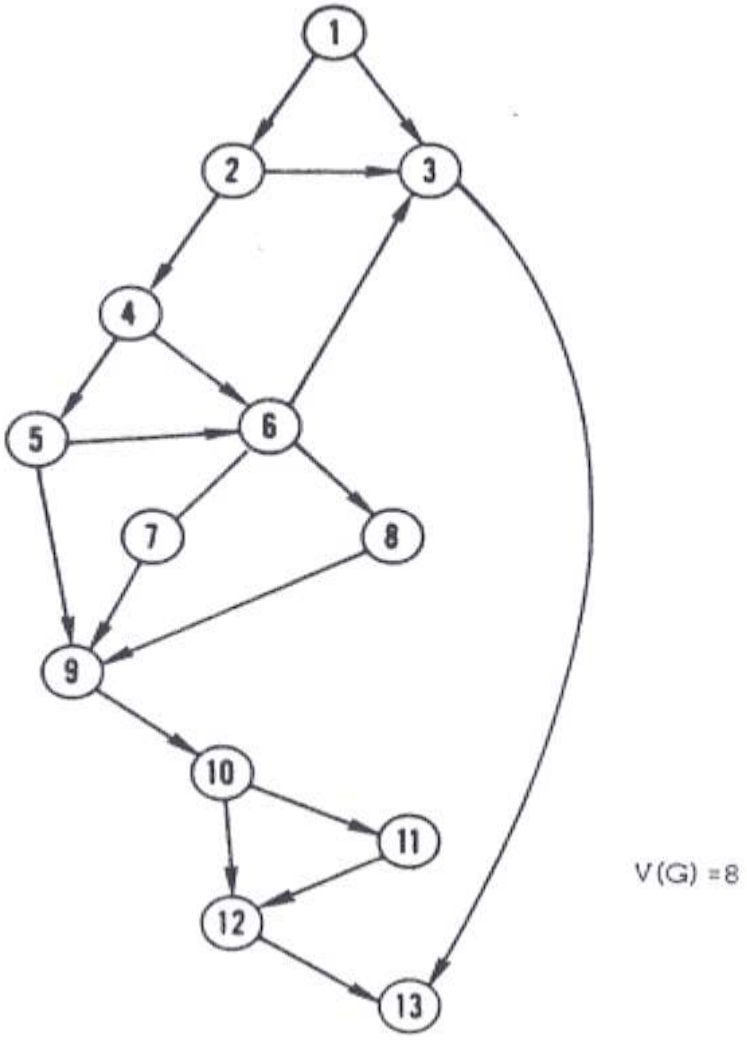

Figura 6 - Exemplo de um grafo representando os possíveis fluxos dentro de um programa (McCabe, 1976)

A complexidade ciclomática do programa pode influenciar diretamente o grau de legibilidade do código. Portanto, uma complexidade ciclomática elevada pode indicar que o código, além de estar mais complexo, pode estar menos manutenível.

2.5.3 Maintainability Index (MI)

O índice de manutenibilidade (Maintainability Index) desenvolvido por Oman (1994) tem como objetivo final diminuir os custos de manutenção podendo calcular o índice de manutenibilidade e apresentá-lo de uma forma conveniente. Coleman (et. al., 1995) explica como o MI foi calibrado validando seu uso no desenvolvimento de software. Essa métrica utiliza as medidas de Halstead e a Complexidade Ciclomática de McCabe onde o índice de manutenibilidade básico de um determinado programa é dado pelo polinômio descrito na Equação 2.

$$171 - (5{,}2 * ln(\text{aveV})) - (0{,}23 * \text{aveV(g')}) - (16{,}2 * ln(\text{aveLOC})) + (50 * sin(\sqrt{2{,}4\text{perCM}}))$$

Equação 2 - Fórmula da Maintainability Index(MI)

Onde:

- **aveV** é a média da medida de Halstead por módulo;
- **aveLOC** é a média de linhas de código por módulo;
- **aveV(g')** é a média estendida da complexidade ciclomática de McCabe por módulo;
- **perCM** é a média do percentual de comentários por linha de código (opcional);

2.5.3.1 Influência das Medidas de Halstead na Métrica MI

O Esforço, que é obtido pelas medidas de Halstead, é uma das medidas que a fórmula do MI utiliza para o cálculo do Índice de Manutenibilidade. A sub-expressão que utiliza a medida de Halstead para subtrair seu impacto do limite de 171 pontos que determina o MI pode ser observada na Equação 3.

$$5{,}2 * ln(aveV)$$

Equação 3 - Primeira sub-expressão da métrica MI

Onde *aveV* é o Esforço obtido pelas medidas de Halstead.

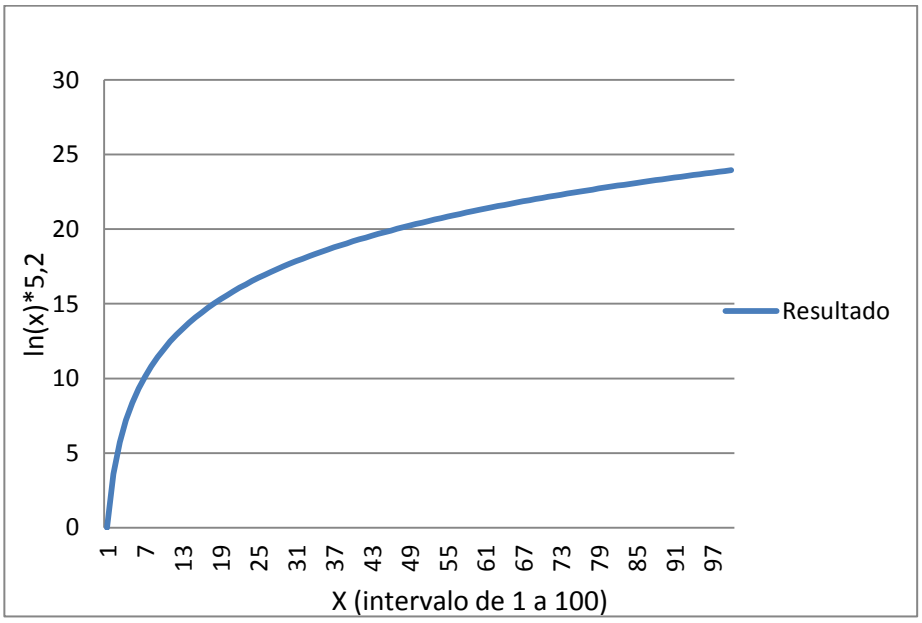

Figura 7 - Curva da primeira sub-expressão da métrica MI

A Figura 7 representa a curva da expressão apresentada na Equação 3. Em termos característicos, esta curva é bem semelhante à curva da função logarítmica. Neste exemplo, o eixo Y contem os valores que estão em função dos resultados da função do logaritmo neperiano aplicada aos valores do intervalo de 1 a 100 multiplicados por 5,2. O eixo X contém os valores desse intervalo.

O exemplo da Figura 7 permite observar que à medida que o valor aplicado à função cresce, a taxa de crescimento da curva diminui. A aplicação de uma função logarítmica a essa sub-expressão permite que haja certo equilíbrio na comparação de diferentes valores retirados de um mesmo sistema através dessa medida.

2.5.3.2 Influência da Complexidade Ciclomática de McCabe na Métrica MI

A Equação 4 apresenta a sub-expressão da métrica MI que utiliza a complexidade ciclomática de McCabe. O resultado dessa sub-expressão indica um valor a ser subtraído do limite de 171 pontos da métrica MI.

0,23 * aveV(g')

Equação 4 - Segunda sub-expressão da métrica MI

Onde *aveV(g')*é o resultado da complexidade ciclomática de McCabe.

Nessa sub-expressão a subtração do limite de 171 pontos da métrica MI é direta. Mais especificamente, subtrai-se 23% do resultado da métrica de McCabe do limite da métrica MI.

2.5.3.3 Influência da medida do número de linhas de código (LOC) na métrica MI

A sub-expressão apresentada na Equação 5, que é a responsável pela subtração relativa à quantidade de linhas de código do limite da métrica MI, também utiliza a função do logaritmo neperiano.

$$16{,}2 * ln(\text{aveLOC})$$

Equação 5 - Terceira sub-expressão da métrica MI

Onde *aveLOC* é o valor da quantidade de linhas de código.

As observações acerca da influência dessa subtração ao limite da métrica MI são as mesmas das apresentadas no tópico 2.5.3.1.

2.6 Padrões de Projeto

Segundo Quan (et al., 2008), os padrões e boas práticas de design no paradigma orientado a objetos, são demonstrações objetivas de como utilizar melhor as possibilidades que o paradigma proporciona. Como citado na seção 2.4.2, programas implementados neste paradigma são altamente manuteníveis, entretanto, o índice de manutenibilidade apenas se mostra satisfatório quando o paradigma é aplicado na sua totalidade. Os padrões e boas práticas de software apresentados a seguir podem ser utilizados para

solucionar diversas e diferentes deficiências que o código ou arquitetura do sistema venha a ter.

2.6.1 Padrão Factory

Este padrão consiste basicamente em criar uma fábrica de objetos que são diferentes, porém possuem a mesma abstração (Freeman et al., 2007). Pode-se entender isso como objetos que implementam uma mesma interface. Essa fábrica de objetos fica encapsulada de modo que todas as vezes que algum objeto precisar criar uma instância daquela abstração, ele não precisa ter a lógica que irá comparar o tipo de instância desejada com os tipos disponíveis. A fábrica se encarregará disso. Ao objeto caberá apenas pedir à fábrica uma instância de um determinado tipo para a abstração ou para a interface por exemplo. Isso ajuda muito a manutenção do código, pois se houver um novo tipo de instância possível – pode-se entender isso como uma nova funcionalidade para o sistema – para aquela abstração (interface), o único local a ser alterado é a fábrica. Se não houvesse a fábrica, todos os locais que instanciam um objeto para a determinada abstração precisariam ser alterados para suportar o novo tipo de instância. A Figura 8 representa bem esse padrão onde uma instância pode pedir à entidade “Creator” que recorre à fábrica (ConcreteCreator) que contém a lógica para escolher a devida instância. Sendo então o objeto devidamente instanciado, é retornado o produto (ConcretProdut).

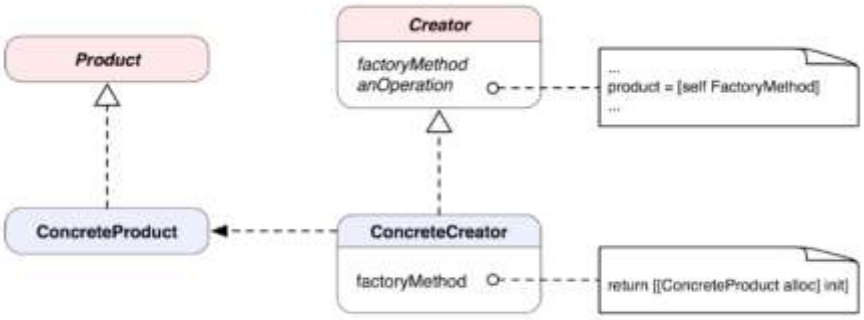

Figura 8 - Diagrama do padrão Factory

2.6.2 Padrão Template

Este padrão ajuda muito a reduzir a duplicidade de partes do código. Utiliza classes e métodos que servem como modelos para realizar determinada tarefa. São classes e métodos abstratos (Freeman et al., 2007). Se um objeto precisa realizar determinada tarefa abstraída numa classe, por exemplo, ele não precisa ter um código próprio que realizará a tarefa podendo recorrer à classe ou método abstrato. A Figura 9 exemplifica isso. No caso de uma manutenção que modificará a regra implementada pela classe abstrata, este será o único local a ser modificado para atender à nova regra. Se esta regra não fosse implementada de modo abstrato, ao se modificar a regra, o impacto seria bem maior, pois consequentemente, várias partes precisariam ser modificadas. Além de a manutenção demandar mais esforço, as possibilidades de erro também aumentariam por haver maior quantidade de código.

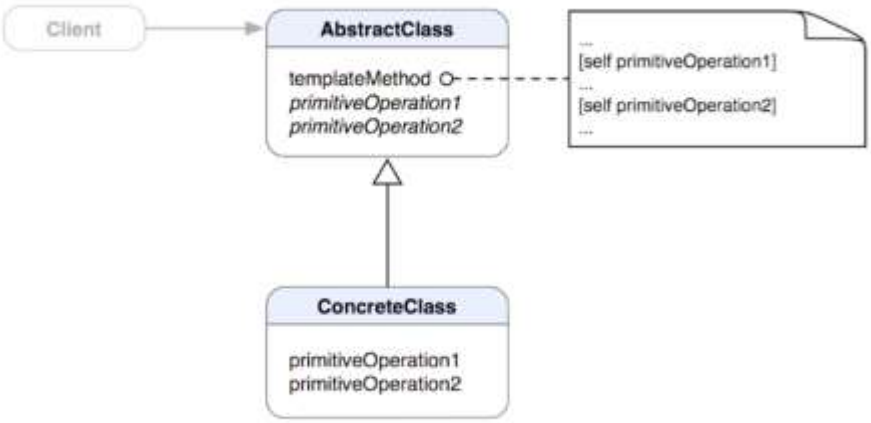

Figura 9 - Diagrama do padrão Template

2.6.3 Padrão Singleton

Este é o padrão que garante a existência de apenas uma instância de um objeto em toda a aplicação (Freeman et al., 2007). Existem objetos que são exclusivamente construídos para comportar a lógica de um caso de uso por exemplo. Nestes casos não é interessante que existam mais de uma instância desse objeto, pois isso poderia causar grandes inconsistências. Além disso, se a aplicação precisasse criar uma instância para executar cada rotina que a classe realiza, além de o sistema ficar sobrecarregado com todas estas instâncias desnecessárias, quando houvesse a necessidade de uma modificação na instanciação dessa classe, o código precisaria passar por modificações em vários locais diferentes. A Figura 10 apresenta esse padrão de forma bem simples, onde a entidade Singleton, sempre retorna a mesma instância de determinada entidade.

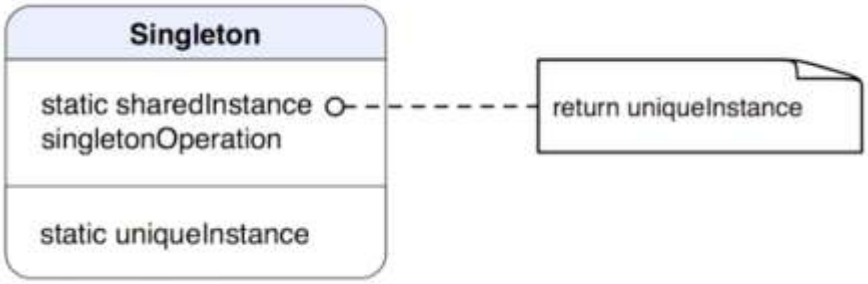

Figura 10 - Diagrama do padrão Singleton

2.6.4 Padrão Command

Este padrão consiste basicamente em encapsular uma chamada a um método (Freeman et al., 2007). Mais detalhadamente, é como se a aplicação fosse como um provedor de serviços a um cliente (entende-se como cliente, outro programa ou sistema que consuma aqueles serviços). O interessante desse paradigma é que não importa quem é o cliente. Por exemplo, para alterar uma determinada entidade haveria um método que seria a interface de comunicação para a realização do serviço que iria alterar a entidade. Este método receberia seus respectivos parâmetros e ao final da execução da sua rotina, todo o serviço estaria feito ou o retorno da chamada seria devolvido. O cliente apenas consumiria determinado serviço e aquele serviço estaria encapsulado numa única solicitação como mostra a Figura 11. Este padrão pode aumentar significativamente o nível de manutenibilidade do sistema. A necessidade de mudar a tecnologia da sua interface de páginas JSP para a tecnologia FLEX, por exemplo, traria a impressão de que todo o sistema precisaria ser refeito. No entanto, se o sistema tiver sido implementado utilizando-se o padrão Command, apenas a implementação da interface precisaria ser refeita. Bastaria apenas à nova implementação da interface consumir os serviços da lógica que permaneceria como está. O cliente só precisará saber a quem solicitar.

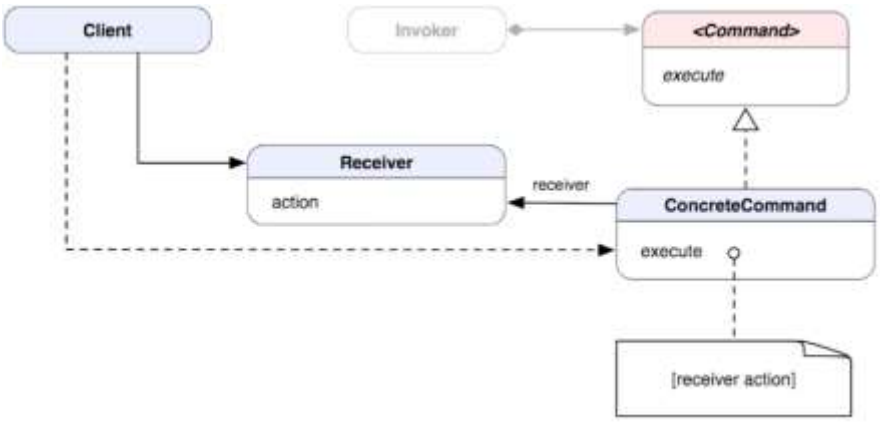

Figura 11 - Diagrama do padrão Command

2.7 Boas Práticas de Programação

Este trabalho de conclusão de curso tratará as boas práticas de programação como sendo práticas ou metodologias a serem adotadas de forma preventiva para evitar que erros considerados comuns possam acontecer durante o desenvolvimento de sistemas. Além disso, as boas práticas de programação, assim como os padrões de projeto de software, têm como objetivos, dentre outros, aumentar a padronização e a legibilidade facilitando a manutenção.

2.7.1 Modelo de Domínio Não Anêmico (Not Anemic Domain Model)

O Modelo de Domínio Anêmico ou *Anemic Domain Model* no inglês foi primeiramente citado por Martin Fowler que identificou que esta é uma prática ruim e que não enfatiza os princípios do paradigma de programação orientada a objetos. Na verdade, este conceito significa um *anti-pattern* ou um anti-padrão e é mais conhecido que o padrão instituído para combatê-lo

denominado “modelo não anêmico”. O *anti-pattern* é um problema específico encontrado que desperta a intenção de se encontrar meios padronizados para combatê-los. Estes meios são denominados os padrões de projeto. A adoção de um modelo não anêmico no desenvolvimento de sistemas pode ser muito favorável ao aumento dos níveis de manutenibilidade (Fowler, 1997).

Talvez essa prática seja a principal dentre as abordadas neste trabalho para o aumento do índice de manutenibilidade de um sistema. O termo “classes anêmicas”, ou “modelo anêmico”, dentre outros com o mesmo significado, classifica classes de sistemas orientados a objeto que não se movimentam ou não se modificam. São classes literalmente atrofiadas. De uma forma objetiva, classes anêmicas são classes que possuem apenas seus atributos e métodos “*getters*” e “*setters*” para eles. Geralmente são classes que representam entidades de um banco de dados.

As classes anêmicas podem ser grandes vilãs da manutenibilidade de um software. As classes anêmicas se assemelham aos registros que existem em linguagens procedurais como o Pascal por exemplo. E por isso, faz com que a orientação a objetos do sistema fique defasada. Uma das principais características de um objeto dentro do paradigma é permitir que ele movimente-se, que ele se altere. Quando há a necessidade de manipular um atributo de um objeto, por exemplo, há uma tendência em fazer sua manipulação dentro da lógica que está se implementando. Isso pode não ser bom. Toda a lógica que diz respeito apenas a um objeto deve estar dentro do próprio objeto. Se a lógica que manipula unicamente atributos do objeto não estiver dentro do objeto, em todas as partes do código que for necessário manipular seus atributos, provavelmente, haverá repetição de código.

Evitar classes anêmicas pode aumentar a manutenibilidade do software em vários aspectos e a legibilidade é um deles (Fowler, 1997). É

muito mais claro entender a chamada de um método com um nome sugestivo do que entender várias linhas de código para executar uma rotina que não está diretamente ligada ou que não é de responsabilidade da lógica que se está implementando como nos exemplos da Figura 12 e Figura 13 que apresentam a mesma lógica em Java dentro e fora do modelo anêmico respectivamente.

```
Objeto objB = new Objeto();

objB.setAtributo1(objA.getAtributo1());
objB.setAtributo2(objA.getAtributo2());

//lógica para validações

//lógica para limpar listas

//outras lógicas

objB.setAtributoN(objA.getAtributoN());
```

Figura 12 - Exemplo de lógica no modelo anêmico

```
Objeto objB = new Objeto();
objB.configuraPorOutroObjeto(objA);
```

Figura 13 - Exemplo de lógica no modelo não anêmico

Ainda destacando a manutenibilidade, no caso de classes não anêmicas, estando à lógica que manipula determinado atributo da classe centralizada em apenas um lugar, se essa lógica sofrer manutenção, o sistema precisará ser alterado em apenas um lugar e este lugar será facilmente encontrado.

2.7.2 Codificação Fluente

A codificação fluente diz respeito a escrever o código de uma forma em que a escrita "flua" de acordo com o entendimento do código e de forma que esse entendimento seja maximizado em todos os seus aspectos, ou seja, escrever de forma a deixar a codificação no mais alto nível possível. Para que a escrita do código seja executada de forma fluente, é extremamente importante que se leve em consideração as características citadas na Tabela 2. Identação, linhas em branco, tamanho das linhas de código, espaços, parênteses, operadores, dentre outros, são características simples, mas que precisam ser colocadas de forma correta durante a escrita do código para que a escrita seja fluente e para aumentar a legibilidade (Buse, 2008).

As linguagens Ruby e Scala, por exemplo, são altamente legíveis e possuem diversas características que proporcionam essa facilidade de leitura e escrita. A linguagem Scala, por exemplo, têm como estilo próprio a combinação dos paradigmas procedural e orientado a objetos em sua escrita objetivando facilitar a leitura e consequentemente o entendimento do código. A linguagem Ruby permite a eliminação de alguns caracteres – parênteses, por exemplo – obrigatórios em outras linguagens, que também deixam a sintaxe de uma forma mais compreensível. O uso de comentários é importante para facilitar a leitura do código, no entanto, a escrita pode ser de uma forma tão legível que o formato da escrita consegue rapidamente induzir o leitor ao objetivo daquele trecho de código sendo desnecessário o uso de tantos comentários. Não apenas em Ruby e Scala, mas em qualquer linguagem, principalmente no paradigma orientado a objetos, pode-se melhorar a legibilidade através da fluência na codificação.

A escrita fluente, assim como a melhoria da legibilidade são fatores que dependem da forma com que o desenvolvedor escreve. Esses são fatores de percepções humanas e subjetivas e por isso, para alcançar um código de

forma altamente legível não basta apenas utilizar linguagens com características favoráveis à legibilidade (Buse, 2008). Entretanto, é extremamente importante que o desenvolvedor pense de uma forma legível para que a escrita flua de uma forma legível.

2.7.3 Lógica por casos de uso

Implementar a lógica por casos de uso permite à lógica estar centralizada. Em muitos sistemas as lógicas relativas às classes são postas separadamente. Isso faz com que o sistema fique saturado de classes que às vezes são criadas para implementarem apenas um método. Se a implementação é feita para um determinado caso de uso, ela fica centralizada. Isso pode aumentar a legibilidade do código, além de evitar repetições desnecessárias. Deixar o código de uma funcionalidade de forma espalhada pode também prejudicar a identificação e a captação de métricas, o que também não é bom. Para a implementação de Repository's (DAO's), por exemplo, é interessante que a implementação seja por classe, pois, nesse caso, a lógica da implementação indica que o objetivo final é manipular uma entidade única. Diferentemente da lógica de um caso de uso, onde os objetivos finais são atender às funcionalidades daquele caso de uso e não de uma entidade ou classe isolada.

2.7.4 Métodos pouco extensos

Na utilização de métodos relativamente grandes para uma determinada lógica, pode não haver junção nem comunicação entre as partes simplesmente porque não haverá partes. Métodos relativamente pequenos pode ser um bom fator para o aumento da legibilidade e, consequentemente, da manutenibilidade do código. Quando se consegue implementar uma pequena ação e der nome a ela, as ações ficam muito mais legíveis e

manuteníveis. Em um método relativamente grande, quando há um problema qualquer e se sabe que o problema está naquele local ou quando aquele local precisa ser modificado por algum motivo, inicialmente todo aquele método ou toda aquela funcionalidade é vista como o problema a ser resolvido. Se o método é relativamente grande, o problema também será. No entanto, quebrar este método em funcionalidades menores pode implicar diretamente, na maioria dos casos, em quebrar o problema em problemas menores e obviamente mais fáceis de serem resolvidos. Além disso, quando houver a necessidade de se adicionar uma nova funcionalidade, muitas das partes que estão quebradas poderão ser reaproveitadas. Isso economiza, além de linhas de código, tempo de raciocínio (Fowler, 1997).

3. Metodologia

3.1 Classificação da Pesquisa

Quanto à natureza, este trabalho pode ser classificado como pesquisa aplicada, pois visa demonstrar a melhoria da manutenibilidade de um sistema orientado a objetos por meio da aplicação de métricas de software em um sistema legado e em outro refatorado por meio da aplicação de padrões de projeto. Este trabalho possui objetivos explicativo e descritivo. Explicativo pois além de citar os padrões de projeto de software aplicados ao sistema legado, analisa como a aplicação dos mesmos impactaram à manutenibilidade do sistema legado. Descritivo por trazer os problemas encontrados no código que consequentemente mantinha baixo os índice de manutenibilidade do sistema e as soluções para esses problemas, que seriam os padrões aplicados. Quanto à sua abordagem, este trabalho pode ser classificado como pesquisa quantitativa e qualitativa. A pesquisa efetuada por este trabalho possui fortes características quantitativas, uma vez que seu principal objetivo é mensurar o impacto da aplicação de padrões de projeto e boas práticas de programação na refatoração de um sistema legado. Entretanto, não deixa de ser uma pesquisa qualitativa, pois busca a melhoria de um importante atributo de qualidade de software denominado manutenibilidade. Quanto aos procedimentos, este trabalho pode ser caracterizado como estudo de caso único, pois a pesquisa se baseia em um caso apenas, onde um mesmo software tem seu índice de manutenibilidade mensurado antes e após passar por um processo de refatoração com a aplicação de determinados padrões de projeto e boas práticas de programação. A coleta de dados deste trabalho se deu pela observação dos valores das métricas de software aplicadas ao sistema antes e após sua refatoração. A Figura 14 apresenta um diagrama com os tipos de pesquisas

científicas onde se pode observar como se desenvolve a classificação deste trabalho.

Figura 14 - Tipos de Pesquisas Científicas (Jung, 2004)

3.2 Procedimentos Metodológicos

Para o alcance dos objetivos propostos, refatorou-se um módulo de um sistema legado aplicando-se alguns padrões e boas práticas de programação e avaliou-se a manutenibilidade dos sistemas legado e refatorado.

Para mensurar a manutenibilidade dos sistemas legado e refatorado, foram utilizadas métricas de software, onde os valores coletados a partir da aplicação das métricas utilizadas poderão então ser comparados e analisados.

Esta metodologia propõe analisar detalhadamente os valores das métricas aplicadas baseando-se na diferença dos valores de uma mesma métrica resultante do sistema legado e do sistema refatorado, além dos

pontos do código afetados por cada métrica. O procedimento utilizado consiste nas etapas descritas a seguir:

1. **Identificação do sistema legado:** nesta etapa o sistema é identificado e estudado.

2. **Análise do estado inicial do sistema legado:** nesta etapa o sistema escolhido é analisado fazendo-se um levantamento de suas características arquiteturais e tecnologias utilizadas em sua implementação. Além disso, nesta etapa são identificadas as deficiências a serem atacadas.

3. **Seleção dos padrões de projeto e boas práticas de programação a serem aplicados:** nesta etapa os padrões a serem aplicados são selecionados com base nas deficiências levantadas na etapa anterior.

4. **Refatoração do sistema legado:** nesta etapa ocorre a refatoração do sistema legado aplicando-se os padrões e boas práticas de programação selecionados na etapa anterior preservando todas as funcionalidades do sistema legado.

5. **Definição e aplicação das métricas de software:** nesta etapa são definidas e aplicadas métricas de software voltadas para manutenibilidade nos sistemas legado e refatorado.

6. **Analise comparativa dos resultados:** nesta etapa os resultados obtidos através da aplicação das métricas são comparados, analisados e apresentados de uma forma conveniente.

4. Estudo de Caso

Como estudo de caso, utilizou-se um módulo de um software legado que tem por objetivo automatizar procedimentos padrão relativos a manejo florestal e análise ambiental da Secretaria de Meio Ambiente do Estado de Minas Gerais. O software foi desenvolvido na linguagem JAVA J2EE dentro do padrão de camadas MVC, utilizando-se as tecnologias Spring Framework e Hibernate/JPA dentre outras.

Mudanças em algumas especificações e a necessidade da adição de novos requisitos ao projeto permitiram que alguns fatores que dificultavam a manutenibilidade do sistema fossem identificados. Diante disso, decidiu-se refatorar um módulo do sistema aplicando-se alguns padrões de software e boas práticas de programação para que sua manutenibilidade fosse melhorada.

A partir do tópico a seguir, tem-se inicialmente a descrição do sistema utilizado no estudo de caso, seguido de como se desenvolveu o estudo.

4.1 Identificação do Sistema Legado

O sistema utilizado para o estudo de caso desse trabalho de conclusão de curso, denominado “Análise Ambiental”, tem como principal objetivo automatizar procedimentos padrão relativos à análise de fatores ambientais do estado de Minas Gerais. A automação desses procedimentos está sendo construída com o fim de auxiliar técnicos ligados à secretaria de meio ambiente do estado de Minas Gerais, a realizar e documentar estudos ambientais necessários para que o governo do estado tome decisões a respeito da utilização dos seus recursos naturais. Esses estudos indicam a

viabilidade da utilização dos recursos naturais do estado, ou da construção e instalação de empreendimentos que possam impactar, de alguma forma, ao meio ambiente. Em suma, esses estudos auxiliam na tomada de decisão da liberação de licenças ambientais assim também como as suas validades.

O "Análise Ambiental" é considerado um sistema de grande porte e possui diversos módulos. É um sistema desenvolvido na linguagem JAVA J2EE dentro do padrão de camadas MVC e utiliza as tecnologias Spring Framework para o gerenciamento das camadas de negócio e Hibernate/JPA como tecnologias de persistência em banco de dados. A interface da aplicação é desenvolvida na tecnologia Flex escrita nas linguagens Action Script e MXML.

Característica	Descrição
Linguagem	Java J2EE
Gerenciamento de Camadas	Spring Framework
Persistência	Hibernate + JPA
Interface Gráfica	Flex (Action Script + MXML)
Gerenciamento de Repositórios	Maven
Banco de Dados	Oracle
Sistema web	Sim
Sistema distribuído	Sim

Tabela 4 - Características do Sistema

Como citado, para o estudo de caso desse trabalho de conclusão de curso foi utilizado um módulo do sistema em questão denominado "Parecer Único". Este módulo resume-se em um formulário a ser preenchido por técnicos analistas da Secretaria de Meio Ambiente do Estado de Minas Gerais para a inserção de resultados e informações de estudos e pela geração de um documento único contendo um parecer técnico a respeito da finalidade daqueles estudos. Na Tabela 4, pode se observar as principais tecnologias e características do sistema.

4.2 Deficiências do Sistema Legado

O projeto do sistema sofreu algumas mudanças nas especificações trazendo a necessidade da adição de novos requisitos ao módulo “Parecer Único” já anteriormente homologado. Diante dessas mudanças foi detectado que alguns fatores dificultariam a manutenção desse módulo para que o mesmo atendesse às novas especificações. Os principais problemas levantados foram:

i. Novas funcionalidades quase sempre implicavam na criação de novas classes;

ii. Repetitividade de partes do código;

iii. Múltiplas instancias de um mesmo objeto sem necessidade;

iv. Dificuldade para identificar a instancia correta a ser utilizada em determinado caso;

v. Métodos saturados e extensos;

vi. Demora em entender o contexto do código;

vii. Demora em encontrar erros;

viii. Incerteza sobre o tamanho do escopo, no código, de uma funcionalidade;

ix. Quantidade considerável de classes com apenas um método;

4.3 Soluções para as Deficiências Levantadas

Diante do levantamento das deficiências que dificultariam a manutenção do módulo “Parecer Único”, utilizado como estudo de caso desse trabalho de conclusão de curso, identificou-se que alguns problemas eram clássicos e recorrentes entre o desenvolvimento de sistemas. Entretanto, a solução encontrada foi a utilização de padrões de projeto de software e de boas práticas de programação que aumentariam o índice de manutenibilidade do sistema e facilitaria a adição de novos requisitos, assim também como eventuais mudanças nas especificações do sistema.

Segundo Quan et al. (2008), os padrões e boas práticas de design no paradigma orientado a objetos, são demonstrações objetivas de como utilizar melhor as possibilidades que o paradigma proporciona. Como dito anteriormente, programas implementados neste paradigma são altamente manuteníveis, entretanto, o índice de manutenibilidade apenas se mostra satisfatório quando o paradigma é aplicado na sua totalidade. Os padrões de projeto e as boas práticas utilizadas durante a refatoração do módulo são apresentadas a seguir.

a. Padrão Factory

b. Padrão Template

c. Padrão Singleton

d. Padrão Command

e. Modelo de Domínio Não Anêmico (Not Anemic Domain Model)

f. Codificação Fluente

g. Lógica por casos de uso

h. Métodos pouco extensos

4.4 Refatoração do Sistema Legado

O sitema legado foi refatorado com base na premissa de que uma refatoração efetuada com sucesso é aquela que preserva as mesmas entradas e saídas, ou seja, se qualquer das versões – inicial e refatorada – receberem uma mesma entrada, a saída deve ser obrigatoriamente a mesma.

A refatoração foi efetuada em cada funcionalidade do módulo por vez. A funcionalidade que fosse passar pela refatoração, tinha seu código minuciosamente analisado e uma nova estratégia de implementação era então traçada com base em determinados padrões de projeto e boas práticas de programação. Quando essa estratégia estava então traçada, o código era reescrito do zero dentro da nova estratégia e com a aplicação dos padrões selecionados.

As boas práticas de programação utilizadas estiveram presentes durante a refatoração de todas as funcionalidades do módulo. Elas foram muito úteis durante a refatoração pelo fato de complementarem alguns padrões de projeto deixando-os mais fáceis de serem implementados.

A reescrita do código durante sua refatoração permitiu que se obtivessem duas diferentes versões do sistema. Essas versões puderam então ser submetidas à aplicação de métricas voltadas para manutenibilidade a fim de se comparar os resultados para a verificação da melhoria da manutenibilidade do sistema pela aplicação dos padrões e boas práticas de

programação. Abaixo, descreve-se como foi a definição das métricas utilizadas para a verificação da melhoria da manutenibilidade do sistema.

4.5 Definição das Métricas

As métricas utilizadas no estudo de caso desse trabalho de conclusão de curso foram métricas voltadas para manutenibilidade. A principal referência utilizada foi a métrica "Maintainability Index" ou Índice de Manutenibilidade, no português, principalmente por ser uma métrica específica para mensurar a manutenibilidade de um programa e também por ter características satisfatórias à comparação de resultados. Essa métrica, desenvolvida por Oman (1994), objetiva calcular o índice de manutenibilidade e apresenta-lo de uma forma conveniente retirando medidas diretamente do código do programa. Como citado, Coleman et al. (1995) explica como o MI foi calibrado e validado para ser usado no desenvolvimento de software.

A MI utiliza as medidas de Halstead e a Complexidade Ciclomática de McCabe em seus cálculos. Medidas essas que são bem aceitas e já estão consolidadas no desenvolvimento de software. Sendo assim, as métricas utilizadas foram as medidas de Halstead, a Complexidade Ciclomática de McCabe e a Maintainability Index.

As ferramentas utilizadas para a retirada das métricas do código foram dois plug-ins para a IDE Eclipse, o Metrics e o CodePro Analytics.

5. Coleta de dados e Análise dos Resultados

Como estudo de caso desenvolveu-se um sistema refatorado a partir da correção de diversas deficiências identificadas no sistema legado. Para verificar se ocorreram melhorias efetivas, aplicaram-se várias métricas de software no sistema legado e refatorado, a fim de comparar os resultados, como mostrado nas tabelas e figuras do tópico abaixo.

5.1 Análise dos resultados Obtidos

Este tópico apresenta os resultados das métricas utilizadas. A Tabela 5 apresenta para cada métrica, sua descrição, sua fórmula e o resultado obtido do sistema legado e do sistema refatorado.

Descrição	Fórmula	Sistema Legado	Sistema Refatorado
Número de operadores distintos	n1	20	17
Número de operandos distintos	n2	215	156
Número total de operadores	N1	1723	440
Número total de operandos	N2	3085	1079
Extensão do programa	N = N1 + N2	4808	1519
Vocabulário do programa	n = n1 + n2	235	173
Volume	V = N*($\boldsymbol{log_2}$n)	37870,29	11293,2
Dificuldade	D=(n1/2)*(N2/2)	143,48	58,79
Esforço	E = D*V	5443946,76	663946,06
Complexidade Ciclomática	CC	35,55	5,42
Número de linhas de código	LOC	1268	503

Tabela 5 - Resultados das métricas abordadas

A Tabela 6 apresenta a fórmula e o resultado da medida do Índice de Manutenibilidade obtido do sistema legado e do refatorado.

Descrição	Fórmula	Sistema Legado	Sistema Refatorado
MI (escala de -∞ a 171)	171-(5,2*ln(V))-(0,23*CC)-(16,2*ln(LOC))	-7,72	20,48

Tabela 6 - Resultado do Índice de Manutenibilidade

A Tabela 7 e a Tabela 8 apresentam os resultados das métricas abordadas de uma forma mais comparativa. Além dos valores dos resultados, essas tabelas também apresentam a característica de cada medida. A coluna "**característica"** que apresenta essas características, as representam por três diferentes ícones, onde:

⇩ = indica que quanto menor o valor da medida, melhor para a manutenibilidade do sistema.

⇔ = indica que é indiferente o valor da medida.

⇧ = indica que quanto maior é o valor da medida, melhor é para a manutenibilidade do sistema.

A coluna "**variação"** apresenta a porcentagem da diferença entre os resultados do sistema legado e do refatorado, ou seja, apresenta o quanto aquela métrica variou. O resultado do Nº de operadores distintos do sistema refatorado, por exemplo, pela coluna "variação" da tabela, caiu em 15 %.

Descrição	Característica	Sistema Legado	Sistema Refatorado	Variação
Nº operadores distintos	⇔	20	17	-15 %
Nº operandos distintos	⇔	215	156	-27,4 %
Nº total de operadores	⇩	1.723	440	-74,4 %
Nº total de operandos	⇩	3.085	1.079	-65 %
Extensão	⇩	4.808	1.519	-68,4 %
Vocabulário	⇩	235	173	-26,3 %
Volume	⇩	37.870,29	11.293,2	-70,1 %
Dificuldade	⇩	143,48	58,79	-59 %
Esforço	⇩	5.443.946,76	663.946,06	-87,8 %
Complexidade Ciclomática	⇩	35,55	5,42	-84,7 %
LOC	⇩	1.268	503	-60,3 %

Tabela 7 - Comparativo dos resultados das métricas abordadas

Descrição	Característica	Sistema Legado	Sistema Refatorado	Variação
MI	⇧	-7,72	20,48	62,3 %

Tabela 8 - Comparativo do resultado da métrica MI

Para verificar a diferença entre os resultados descritos na Tabela 7, foi efetuado um teste de Wilcoxon pareado, comparando a diferença entre os valores obtidos para as métricas nas duas versões do sistema. Para a realização do teste definiu-se as seguintes hipóteses:

H_0 = Não há nenhuma diferença entre os resultados das métricas aplicadas às duas versões do sistema.

H_1 = Há diferença entre os resultados das métricas aplicadas às duas versões do sistema.

O teste apontou uma diferença significativa entre as duas versões (W^+ = -2.934, N = 11, p = 0,003). Diante desse resultado, pode-se rejeitar H_0 e concluir que a aplicação dos padrões de projeto e boas práticas de programação mencionados na seção 4.3 , causou diferença significativa nos resultados das métricas para o sistema legado e o refatorado, como apresentado na Tabela 7.

A Figura 15 apresenta um gráfico de comparação proporcional, o qual permite que se tenha uma visão dimensional da diferença entre os resultados das métricas retiradas do sistema legado e do sistema refatorado. A linha externa do gráfico representa os resultados do sistema legado e a linha interna representa os resultados do sistema refatorado. A linha interna está colocada proporcionalmente aos resultados do sistema legado onde a distância entre os mesmos vértices da linha externa e interna mostra a diferença do resultado de cada métrica.

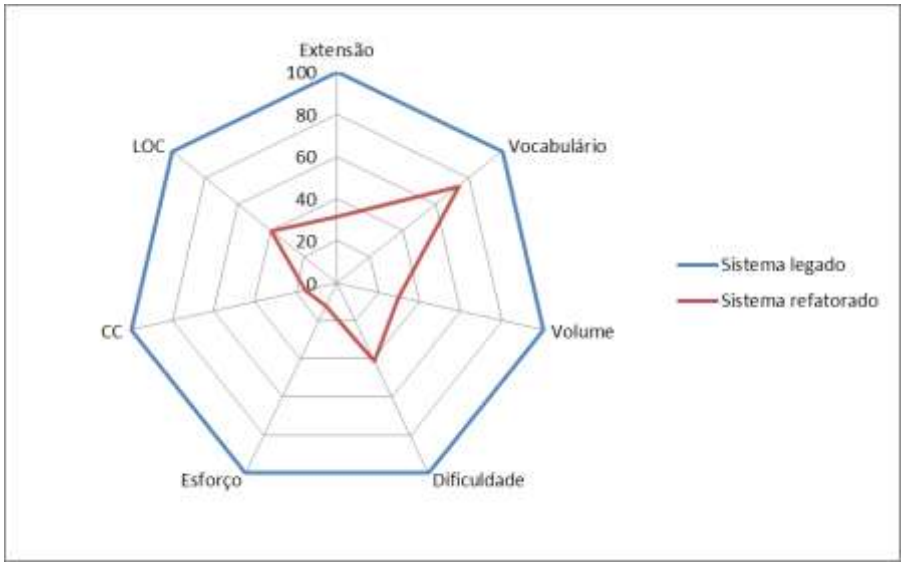

Figura 15 - Gráfico de Comparação Proporcional

Conclui-se que o sistema refatorado melhorou significativamente em relação ao sistema legado pela observação das medidas obtidas com a aplicação das métricas. Nota-se ainda que todas as medidas do sistema refatorado, apresentaram melhor desempenho em relação às medidas do sistema legado. Os tópicos a seguir analisam como os padrões aplicados ao código puderam contribuir para a melhoria, não só do índice de manutenibilidade do código, mas também da qualidade do sistema como um todo.

5.1.1 Análise de melhorias na Eficiência

Um código menos extenso e menos complexo possibilita ao sistema um aproveitamento maior de seus recursos (Freeman et. al., 2007). Como verificado pelos resultados, houve uma redução da Extensão do código de 4808 pontos para 1519 pontos e também uma redução da Complexidade Ciclomática de 35,55 pontos para 5,42 pontos após a refatoração. Quando os caminhos a serem percorridos são menores, o tempo gasto pelo sistema para realizar determinada atividade também passa a ser menor e isso reflete um aumento na eficiência do sistema. Não necessariamente a redução do volume do código garante uma melhor eficiência, pois o código pode ter um volume menor e os caminhos a serem percorridos terem uma distância equivalente. Entretanto, a redução do volume do código pode melhorar a eficiência.

5.1.2 Análise de melhorias na Funcionalidade

Com a melhoria da manutenibilidade, o código ficou mais fácil de ser alterado. Isso significa que, diante da necessidade de adicionar novos requisitos, a implementação destes requisitos tem uma maior probabilidade de ser efetuada com sucesso. A verificação da funcionalidade de um sistema depende de vários fatores e só pode ser confirmada após a validação do

software. Entretanto, a melhoria de fatores como a Complexidade Ciclomática e a melhoria do Índice de Manutenibilidade do sistema contribuem para a sua Funcionalidade ao mesmo passo em que facilitam sua implementação.

5.1.3 Análise de melhorias na Confiabilidade

A diminuição na Dificuldade, após a aplicação dos padrões de 143,48 pontos para 57,79 pontos, indica que uma eventual manutenção será feita dentro de um menor índice de dificuldade e, portanto, permitindo que o sistema esteja mais confiável. A diminuição da Complexidade Ciclomática também indica que erros terão menor chance de ocorrer colaborando para a confiabilidade.

5.1.4 Análise de melhorias na Usabilidade

A Usabilidade de um software é verificada a partir de fatores do sistema que estão em constante contato com o seu usuário como sua interface gráfica. As métricas e os fatores analisados nesse trabalho não permitiram a verificação da usabilidade do sistema legado. Entretanto, a melhoria da manutenibilidade do código pode permitir que eventuais funcionalidades que venham a ser incorporadas ao sistema, sejam efetuadas de uma forma mais inteligente, o que pode ser considerado como uma forma mais facilitada para o usuário executar determinada tarefa.

5.1.5 Análise de melhorias na Manutenibilidade

A métrica Maintainability Index (MI) abordada no tópico 2.5.3, utiliza uma combinação de métricas que mensuram diversos fatores de um sistema como o Volume, o Esforço e a Complexidade Ciclomática.

Percebemos pelos resultados que todas as métricas calculadas tiveram um resultado melhor após a refatoração do código aplicando-se os padrões. Padrões como o Factory, o Template e a utilização de classes não anêmicas, fazem com que o código seja reduzido evitando repetições. São muitas as vantagens de se ter uma menor quantidade de código escrito. Essa quantidade não está relacionada ao tamanho dos nomes dos métodos ou dos atributos. Ou seja, não necessariamente se a codificação de um programa possuir menos caracteres significará que ele possui menos código. Essa quantidade está relacionada à quantidade de comandos e atributos que o código possui.

As práticas utilizadas na refatoração do sistema legado, como evitar métodos extensos, implementar as lógicas por casos de uso e os padrões Singleton e Command, podem ser determinantes na melhoria da legibilidade do código. Como já citado, é melhor ler um comando simples e entender o que acontece naquele momento do que ter que interpretar a lógica daquele comando, o que não é o foco naquele momento. Além de melhorar a legibilidade do código, a redução na Dificuldade de 143,48 pontos para 58,79 pontos, verificada pelo resultado desta métrica após a refatoração aplicando-se os padrões, reflete que numa eventual manutenção, seja ela perfectiva ou não, o código estará mais fácil de ser alterado, de ser entendido, e consequentemente com menos risco de ocorrerem efeitos colaterais. Todos os padrões e práticas utilizados na refatoração do sistema legado contribuíram para que o mesmo fosse reduzido, tornado mais legível, e consequentemente mais manutenível. Uma faceta da prática denominada como encapsulamento, que foi aqui abordada como “classes não anêmicas”, foi a prática que mais refletiu ao aumento da manutenibilidade do código. A prática de executar uma rotina que diz respeito apenas a uma classe, em outra classe qualquer, contribui para que a classe seja fraca, seja anêmica e tenha menos capacidade de se movimentar ou de se alterar. Normalmente quando existem métodos muito extensos, neles estão contidas várias sub-

rotinas que poderiam estar fora daquele método e serem reutilizadas, tornando a manutenção mais facilitada e o código mais legível. Outros fatores que contribuíram para a diminuição do Volume de 37870,29 pontos para 11293,2 pontos, do Vocabulário de 235 pontos para 173 pontos e da Extensão do código de 4808 pontos para 1519 pontos de acordo com os resultados obtidos, foram as reduções dos números de operadores e operandos. O uso de "classes não anêmicas" contribui para a diminuição destes fatores no sentido de que tirar partes da lógica das rotinas principais do sistema na medida do possível, leva para outros locais operadores e operandos que estariam "inchando" a lógica principal dificultando, principalmente, a manutenção perfectiva naquele código. Em suma, a reutilização de grande parte do código e a centralização de determinadas implementações foram os grandes responsáveis pela grande melhoria no índice de manutenibilidade do código.

6. Conclusões

Para a análise do impacto da aplicação de padrões de projeto em um sistema orientado a objetos, utilizou-se um módulo de um sistema legado denominado "Análise Ambiental" desenvolvido na linguagem JAVA J2EE que tem por objetivo automatizar procedimentos padrão relativos à análise de fatores ambientais do estado de Minas Gerais e auxiliar técnicos ligados à secretaria de meio ambiente do estado de Minas Gerais na realização e documentação de estudos ambientais necessários para que o governo do estado tome decisões a respeito da utilização dos seus recursos naturais.

Para a realização deste trabalho de conclusão de curso, inicialmente definiu-se alguns conceitos sobre Evolução de Software, seguidos de definições de Manutenção de Software, onde se adotou a definição de Pigoski (1996), citado por Brusamolin (2004), que diz que "*Manutenção de software é a totalidade de atividades necessárias para prover, minimizando o custo, suporte a um sistema de software*". Apresentado o conceito de Manutenção de Software, pôde-se entender a Manutenibilidade como um atributo de qualidade de software que pode ser utilizado para mensurar o quão manutenível é um sistema de software.

Mudanças nas especificações e a necessidade da adição de novos requisitos ao sistema permitiram a detecção de fatores que dificultariam a sua manutenção. Além disso, pôde-se perceber que muitos desses problemas eram clássicos e recorrentes entre o desenvolvimento de sistemas. Diante disso, decidiu-se aplicar padrões de projeto e de boas práticas de programação ao sistema legado através de processos de refatoração com a finalidade de elevar o índice de manutenibilidade do sistema facilitando também à adição de novos requisitos.

Para a verificação da melhoria da manutenibilidade do sistema, aplicaram-se algumas métricas de software ao sistema legado e ao sistema refatorado a fim de se comparar os resultados. Os resultados de todas as métricas aplicadas ao sistema refatorado foram satisfatórios em relação aos resultados obtidos pelo sistema legado. Com base nos resultados, foi feita uma análise detalhada de vários atributos do software relacionando as melhorias obtidas aos padrões aplicados.

O resultado da análise foi positivo, pois foram verificadas significativas melhorias no código após a aplicação dos padrões. Além disso, a análise permitiu entender como os padrões impactaram os resultados das métricas, ou seja, como cada padrão aplicado permitiu que o valor de uma determinada métrica fosse melhorado.

Este trabalho pôde dar contribuições significativas no desenvolvimento de software, pois, por meio de um estudo de caso, conseguiu demonstrar que padrões de projeto e boas práticas de programação, se devidamente aplicados, aumentam a manutenibilidade de um sistema. Assim sendo, por consequência, aumentam também a qualidade de um sistema que é um atributo muito importante para qualquer sistema de software.

Como trabalhos futuros, pretende-se:

- realizar uma revisão sistemática de literatura para identificar estimativas e métricas que calculem o beneficio da aplicação de padrões de projetos, antes delas serem aplicadas. Isto propiciará ao engenheiro de software escolher qual padrão aplicar e se vale a pena aplicar

- realizar uma revisão sistemática de literatura para identificar valores de referência para as métricas aplicadas

- realizar um estudo em software livre, para estabelecer o valor médio atual das métricas aplicadas neste TCC

- poder então calcular a influência específica de cada padrão de projeto e de cada boa prática utilizados nesse trabalho em um sistema de software

7. Referências Bibliográficas

BRUSAMOLIN, Valerio. Manutenibilidade de Software, Instituto Cientifico de Ensino Superior e Pesquisa - ICESP, Revista Digital Online vol. 2, 2004.

BUSE, R. P. L. WEIMER, W. R. A Metric for Software Readability –ISSTA'08 International Symposium on Software Testing and Analysis, Seattle, Washington – USA, 2008.

COLEMAN, Don. ASH, Dan. – Using Metrics to Evaluate Software System Maintainability. IEEE, 1994.

COLEMAN, Don. LOWTHER, Bruce. OMAN, Paul – The application of Software Maintainability Models in Industrials Software Systems. J. Systems Software, 1995.

CRISCUOLO, Marcelo. Qualidade de Produto de Software: uma abordagem baseada no controle da complexidade. Dissertação de mestrado – Instituto de Ciências Matemáticas e de Computação, ICMC – USP. São Carlos – SP, 2008.

ERLIKH, Len. Leveraging Legacy System Dollars For E-Business. Journal IT Professional vol. 2 p. 17-23, NJ, USA, 2000.

FOWLER, Martin. Analysis Patterns: Reusable Object Models – Addison-Wesley. Indianapolis – USA, 1997.

FOWLER, Martin. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

FREEMAN, Eric. FREEMAN, Elizabeth. Use a Cabeça!: Padrões de Projeto (Design Patterns). Alta Books, Rio de Janeiro, 2007.

HALSTEAD, M. H. Elements of Software Science, Operating and Programming Systems Series. Vol. 7.Elsevier, 1977.

HENRY, Sallie M. HUMPHEY, Matthew. A controlled Experiment to Evaluate Maintainability of Object-Oriented Software. IEEE, 1990.

ISO/IEC 9126-1. Software engineering – Software product quality – Part 1: Quality Model, 2000.

ISO/IEC 12207. Technology Information – Software Life Cycle Process, 2002.

JUNG, Carlos Fernando. Metodologia para pesquisa & desenvolvimento: aplicada a novas tecnologias, produtos e processos. Rio de janeiro: Axcel Books, 2004.

LAUDER, A., KENT, S. Precise Visual Specification of Design Patterns - ECCOP 98, Anais da 12ª Conferência Europeia sobre Programação Orientada a Objetos - Londres, 1998.

LI, Wei. HENRY, Sallie. Maintenance Metrics for the Object Oriented Paradigm. IEEE, 1993.

MAEDA, Kazuaki, Executable Representation for Structured Data Using Ruby and Scala – 10º ISCIT Internacional Symposium on Comunications and Information Technologies, IEEE. Tokio – Japan, 2010.

MARTINS, Vidal. VOLPI, Lisiane M. Influência da Arquitetura na Qualidade do Software. Bate Byte, 2002.

MCCABE, Thomas J. A Complexity Measure. Software Engineering. IEEE 1976.

NBR ISO/IEC 12207 - Tecnologia de informação– Processos de ciclo de vida de software. Rio de Janeiro: ABNT, 1998.

OMAN, Paul. W., HAGEMEISTER, Jack. R. Construction and Testing of Polynomials Predicting Software Maintainability – Journal of Systems and Software, pp. 251 - 266, 1994.

PIGOSKI, Thomas M. Pratical Software Maintenance: Best Practices for Managing Your Software Investment. Wiley Computer Publishing, 1996.

POSNETT, Daryl. HINDLE, Abram. DEVANBU, Premkumar. A Simpler Model of Software Readability – 8º IEEE Intl. Working Conference on Mining Software Repositories (MSR-11), Honolulu, Hawaii – USA, 2011.

PRESSMAN, Roger S. Engenharia de Software, 6ªedicão, McGraw-Hill, Sao Paulo, 2006.

QUAN, Long. ZONGYAN, Qiu. LIU, Zhiming. Formal Use of Design Patterns and Refactoring – T. Margaria and B. Steffen (Eds.): ISoLA 2008, CCIS 17, pp. 323-338, 2008.

SOMMERVILLE, Ian. Software Engineering. Addison Wesley 6ª edição, 2001.

SOMMERVILLE, Ian. Software Engineering. Addison Wesley 8ª edição, 2007.

VALENTIM, Ricardo. A. M., NETO, Plácido. A. S. O Impacto da Utilização de Design Patterns nas Métricas e Estimativas de Projetos de Software – Revista FARN v. 4, p. 63-74. Natal, RN – Brasil, 2004.

WHITMIRE, Scott. A., Object-Oriented Design Measurement, Wiley, 1997.