Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Felipe Hackmann Dobri Leite

Como evitar ou amenizar os riscos de manutenção de software na inclusão e alteração de requisitos

Americana, SP
2016

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Felipe Hackmann Dobri Leite
felipe.hackmann@gmail.com

Como evitar ou amenizar os riscos de manutenção de software na inclusão e alteração de requisitos

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Fatec-Americana, sob orientação do Prof. MSC. Wagner Siqueira Cavalcanti.

Área de concentração: Engenharia de Software

Americana, SP
2016

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

L552c LEITE, Felipe Hackmann Dobri
Como evitar ou amenizar os riscos de manutenção de software na inclusão e alteração de requisitos./ Felipe Hackmann Dobri Leite. – Americana: 2016.
45f.

Monografia (Curso de Tecnologia em Análise e Desenvolvimento de Sistemas). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Ms.Wagner Siqueira Cavalcanti

1.Engenharia de software I. CAVALCANTI, Wagner Siqueira II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.3.05

Felipe Hackmann Dobri Leite

Como evitar ou amenizar os riscos de manutenção de software na inclusão e alteração de requisitos

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo CEETEPS/Faculdade de TECNOLOGIA – FATEC/Americana.
Área de concentração: Engenharia de Software

Americana, 07 de Dezembro de 2016.
Banca Examinadora:

Wagner Siqueira Cavalcante (Presidente)
Mestre
FATEC - Americana

Eduardo Antonio Vicentini (Membro)
Mestre
FATEC - Americana

Antonio Alfredo Lacerda (Membro)
Especialista
FATEC - Americana

DEDICATÓRIA

Dedico este trabalho a Deus, aos meus pais, pelo apoio e paciência, a minha família e aos amigos que aqui fiz.

AGRADECIMENTOS

Agradeço primeiramente a Deus por me dar o conhecimento e me proporcionar tantas graças vividas e alcançadas durante minha vida, por ter me dado a cura da epilepsia.

Agradeço a minha família, em especial os meus pais, Fernando e Edna, que sempre me apoiaram em minhas decisões e me ajudam a conquistar cada batalha que a vida proporciona. Também às minhas irmãs Fernanda Hackmann e Fabiola Hackmann que sempre me ajudaram muito. A minha namorada Bruna Gaseta, que sempre esteve ao meu lado mesmo nas dificuldades encontradas. Agradeço também aos nossos cachorros Suri e Baruk, que sempre trazem a alegria para todos em minha casa.

Agradeço aos meus amigos que conquistei no decorrer deste curso na Fatec – Americana, principalmente ao Bruno Costa, Caio Salvador e Luis Santarosa, que sempre me apoiaram no percorrer destes anos de aprendizagem. Estes levarei eternamente em minhas lembranças e em minha vida.

Agradeço aos amigos e colegas de trabalho na empresa *MicroWork Softwares* principalmente ao André Chiosini e Mayara Alves que me ajudaram no desenvolvimento deste trabalho. Além deles, outros amigos que me ajudaram a crescer profissionalmente na área e também com os assuntos e matérias que estava sendo ensinado na faculdade.

Agradeço também à empresa DataHex, que me permitiram utilizar seu sistema como estudo para o desenvolvimento deste, e prestarem determinadas informações quando foi necessário.

“No que diz respeito ao empenho, ao compromisso, ao esforço, à dedicação, não existe meio termo. Ou você faz bem feito ou não faz.”

Ayrton Senna

RESUMO

Este trabalho monográfico apresenta o que é uma manutenção de *software* e os riscos em que um software e a organização estão submetidos nesta tarefa, seja para a inclusão de novos requisitos ao sistema ou para a alteração de algum já existente. Mostrando um procedimento que poderia ser seguido na execução de uma manutenção no sistema, evitando-se que a empresa afunde em alguns riscos dispostos desta etapa da engenharia de *software*. Para a realização deste, foi utilizado um método ágil, descrevendo sua equipe e os papéis de cada um dentro deste ambiente, adequando-se para evitar os riscos expostos neste trabalho, de acordo como a metodologia adotada é seguida e determinada. Seguindo esses processos, chegou-se à conclusão de que o *software* teve uma boa manutenção e de resultado qualitativo, deixando-o de uma manutenibilidade mais fácil e qualidade avançada de acordo com sua evolução. O objetivo deste trabalho foi alcançado em partes, pois, um dos riscos abordados, seguindo o que foi apresentado, dificilmente seria evitado, este cabe a um dos grandes desafios da manutenção de *software* de se evitar totalmente todos os riscos, estudado ainda por grandes autores da área.

Palavras Chave: Manutenção de *Software*; *Feature Driven Development*; Riscos

ABSTRACT

This monograph shows what is a software maintenance procedure and the risks that the software and the organization are submitted throughout this action, either to include new requirement on the system or even to change existing ones. This study will present a procedure that could be used along the maintenance execution on the system in order to avoid the risks of this software engineering stage, hence provide to the company a safe productive environment. To perform it, an agile method was used, describing the team and the roles for each member into this environment, adjusting to avoid the risks mentioned on this work, according to how chosen methodology is followed and fixed. Following these processes, was possible to conclude that software had a good maintenance by quantitative results, allowing an easier maintainability and higher quality according to its evolution. The aim of this work was achieve partially, since was not possible to avoid one of the studied risks. Its regards one of the major software maintenance challenges, how to avoid all of the risks is still being studied by the greats authors of this subject.

Keywords: Software Maintenance; Feature Driven Development; Risks

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

CASE	*Computer Aided Software Engineering*
CLF	Construir a Lista de *Features*
CMMI	*Capability Maturity Model Integration*
CPF	Construir por *Feature*
DMA	Desenvolver um Modelo Abrangente
DPF	Detalhar por *Feature*
FDD	*Feature Driven Development*
IEEE	*Institute of Electrical and Eletronics Engineers*
IEC	*International Electrotechnical Commission*
ISO	*International Organization Stardardization*
PPF	Planejar por *Feature*
SQuaRE	*Software product Quality Requirements and Evaluation*
UML	*Unified Modeling Language*

SUMÁRIO

1. INTRODUÇÃO

A manutenção de *software* é a responsável por grandes esforços gastos nas organizações de desenvolvimento de *softwares*. Pode-se dizer que futuramente essas empresas estarão disponíveis somente para a realização das manutenções dos *softwares* antigos, pois, os recursos existentes, estão todos sendo gastos para a manutenção, não havendo mais recursos para o desenvolvimento de novos *softwares,* diz Pressman (2011, p. 663) sobre a manutenção de *software*. Para Sommerville (2007, p. 326), as mudanças feitas nas manutenções de *software* podem ser para correção de erros de codificação, correção ou melhorias de componentes de projetos ou então para a acomodação de novos requisitos ao sistema. Nosek e Palvia e Lientz e Swanson (1980, *apud* SOMMERVILLE 2007, p. 326), dizem que 65% das manutenções realizadas, são voltados para a implementação de novos requisitos.

Bellin (1993, p. XV) retrata que muitas pessoas pensam que a manutenção de *software* é uma opção desnecessária, e acham que seu atual sistema nunca precisará de uma manutenção. Porém, um *software* que nunca passou por uma manutenção pode estar tanto gerando dados incorretos, quanto estar deficitário.

Muitos problemas existentes nos sistemas não são encontrados na etapa de elaboração dos testes, para isso que se faz necessária a realização de uma manutenção de *software*, onde para cada tipo de problema encontrado após a entrega do sistema ao usuário, será necessária a realização de um tipo de manutenção, que serão apresentadas nos capítulos neste trabalho.

Com o passar do tempo, podem ocorrer vários fatores que determinarão a necessidade de inclusão de novos requisitos ao sistema, a mudança de uma legislação ou uma necessidade do usuário são alguns exemplos para a execução de uma manutenção de *software.* No decorrer das manutenções de *software* existem vários riscos dispostos, onde estes serão apresentados no capítulo sobre a manutenção de *software*, estes serão também a **problematização** do trabalho a serem resolvidas. Para isso, é preciso saber qualificar estes riscos e também,

analisar como esquivar-se para que não haja defeitos no *software* na entrega deste após a manutenção.

A **justificativa** para a escolha deste tema se deu devido um *software* estar sempre em constantes manutenções, para que não fique obsoleto ou defasado, e sempre enfrentar riscos aos atuais requisitos existentes no sistema, onde muitas empresas têm dificuldades em se organizarem para uma constante evolução do *software* e da própria empresa.

Os capítulos que se seguem nesta monografia abordarão sobre estes riscos, e como evitá-los no processo da manutenção, tendo como **objetivo geral** apresentar os processos da engenharia de *software* em que uma empresa passa para a execução de uma manutenção de *software* com boa qualidade, e tendo como **objetivo específico**, especificar e apresentar as medidas e procedimentos a serem tomadas pelas empresas de desenvolvimento de sistemas, para evitar que o atual *software* dar-se-á com algum dos riscos que serão expostos.

Para o desenvolvimento deste, o método de pesquisa adotado como **metodologia** será do tipo aplicada, através dos procedimentos da pesquisa bibliográfica, que é produzida através de artigos e conteúdos já existentes, principalmente por publicações; pesquisa explicativa, que busca identificar causas e motivos que interferem ou provocam as ocorrências; e por fim, um estudo de caso, com o propósito de levantar e elaborar uma hipótese de possíveis casos que poderiam ocorrer sobre o que foi abordado. Ambos foram escolhidos visando alcançar os objetivos deste trabalho.

O trabalho será estruturado pelos seguintes capítulos após este:

- No segundo capítulo será abordado sobre a engenharia de *software,* onde fará uma breve descrição do que é um sistema, seus processos e engenharia;

- O terceiro capítulo abordará sobre os requisitos, onde é apresentado o que são estes dentro de um sistema;

- A Manutenção de *software* será abordada no quarto capítulo*,* onde será apresentada a definição de manutenção, evolução e qualidade de *software*, e alguns riscos dispostos em uma manutenção;

- O quinto capítulo será apresentado um estudo de caso, onde será simulada uma manutenção em um determinado *software*;

- Por fim, o sexto capítulo será exposto com as considerações finais, em que conterá a estrutura apresentada no trabalho dentro do estudo de caso e seus resultados obtidos. Também com trabalhos futuros sugeridos ao tema.

2. A ENGENHARIA DE *SOFTWARE*

O objetivo deste trabalho é realizar uma manutenção de boa qualidade, evitando-se ou, então, quando não possível, amenizando-se as chances de que a empresa e o *software* sofram algum dos riscos da manutenção. Para um melhor entendimento do leitor, neste capítulo será abordada a conceituação de *software,* a engenharia de *software,* os processos de *software* e algumas metodologias que são adotadas para o desenvolvimento do *software*.

2.1. Conceitos de *software*

Para Pressman (2011, p. 32), *software* pode ser descrito como:

> (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso dos programas.

Sommerville (2007, p.4), acrescenta para este conceito: "*Software* não é apenas o programa, mas também todos os dados de documentação e configuração associados, necessários para que o programa opere corretamente".

O IEEE[1], uma importante instituição para a engenharia de *software*, (IEEE, 1993 apud Pressmann, 2011, p. 39) desenvolveu uma definição para a engenharia de *software* afirmando que esta é:

> (1) A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de *software*; isto é, a aplicação de engenharia de *software*. (2) O estudo de abordagens definido em (1).

[1] *Cf.* http://www.ieee.org/about/index.html

2.2. Processos de *software*

A definição de processos de *software* é apresentada por Sommerville como "um conjunto de atividades que leva à produção de um produto de *software"* (SOMMERVILLE, 2007, p.44). Ademais, o IEEE, define o processo de *software* como "uma sequência de passos executados com um determinado objetivo" (IEEE 2003, *apud* PAULA FILHO, 2009, p.89). Já para o CMMI[2], modelo de referência para as empresas obterem maturidade no desenvolvimento de *softwares*, é apresentado como "um conjunto de ações e atividades inter-relacionadas realizadas para obter um conjunto especificado de produtos, resultados ou serviços" (CMMI 2006, *Apud* PAULA FILHO, 2009, p.89).

Implementando estes, Pressman (2011, p. 40) define os processos de *software* como

> [...] um conjunto de atividades, ações e tarefas realizadas na criação de algum produto de trabalho (*work product*). Uma atividade esforça-se para atingir um objetivo amplo e é utilizada independentemente do campo de aplicação [...] uma ação envolve um conjunto de tarefas que resultam num artefato de *software* [...] uma tarefa se concentra em um objetivo pequeno, porém, bem definido e produz um resultado tangível.

Complementando sua definição, o mesmo autor diz que "um processo não é uma prescrição rígida de como desenvolver o *software*, e sim, uma abordagem adaptável que possibilite as pessoas (a equipe de *software*) realizar o trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas" (PRESSMANN, 2011, p. 40). Os processos de *software* dependem do julgamento humano, e devido este julgamento, as ferramentas CASE da engenharia de *software,* dão suporte para algumas atividades dos processos de *software* (Sommerville, 2007, p.42).

Com isso, são adotadas as metodologias que são imprescindíveis segundo Silva (2007) no desenvolvimento de *software*, dizendo que um roteiro para o trabalho focado em planejamentos bem elaborados, faz com que a pressa exagerada na

[2] *Cf.*http://cmmiinstitute.com/about-cmmi-institute

entrega do sistema, não resulte em novos problemas ainda maiores dos que existiam a serem resolvidos.

Mesmo havendo uma grande variabilidade de processos de *softwares*, existem atividades em comuns para todas, que são apresentadas por Sommerville (2007, p. 43) no Quadro 1:

Quadro 1 – Atividades de processos de *software*

Atividade	Descrição
Especificação de *software*	A funcionalidade do *software* e as restrições sobre sua operação devem ser definidas
Projeto e implementação de *software*	O *software* que atenda à especificação deve ser produzido
Validação de *software*	O *software* deve ser validado para garantir que ele faça o que o cliente deseja
Evolução de *software*	O *software* deve evoluir para atender às necessidades mutáveis do cliente

Fonte: Adaptado pelo autor de Sommerville (2007, p.43)

A escolha da metodologia a ser utilizada é de livre arbítrio para a organização de desenvolvimento de *software*, desde que a metodologia escolhida "seja suficiente e garanta que o *software* em todo seu ciclo de vida seja passível de gerência e controle e que possua a qualidade esperada" (Magela, p. 26, 2006a).

Para um melhor entendimento, será apresentado o ciclo de vida clássico do sistema, também conhecido como modelo tradicional ou modelo cascata, que tem a sua fundamentação tradicional; e o modelo FDD, que tem a sua fundamentação nas metodologias ágeis.

Os modelos tradicionais sugerem uma abordagem sequencial e sistemática para o desenvolvimento do *software*, são indicados quando as definições dos requisitos foram bem elaboradas. Já os métodos ágeis foram criados para um

desenvolvimento mais rápido do *software*, que surgiu através do Manifesto Ágil[3], em que valorizam mais os indivíduos e as interações do que os processos e ferramentas no desenvolvimento do *software*. Lobo (2008, p.42) acrescenta que "a metodologia ágil permite maior adaptação a mudanças no projeto, isso porque as iterações têm um período curto e a documentação é dinâmica, permitindo assim uma maior integração e rapidez na comunicação entre as equipes".

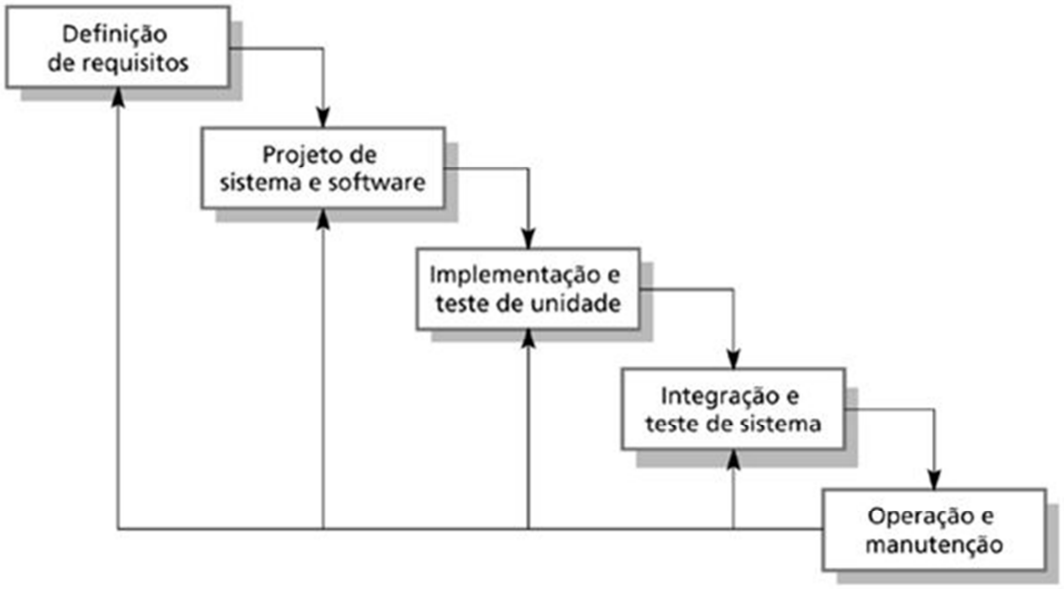
2.2.1. Modelo Cascata

O termo ciclo de vida do *software* é dado por Lobo (2008, p. 20) com o propósito de que o *software* esteja sempre em utilização, sendo assim, o *software* não tem o ciclo de vida finalizado após a implementação, pois o ciclo de vida retorna muitas vezes ao início devido alguma modificação.

Bellin (1993, p.104) diz que "Todas as metodologias estruturadas possuem um ciclo de vida idealizado para todo o processo de desenvolvimento e manutenção de sistemas", sendo que as demais metodologias possuem seus ciclos de vida variantes do modelo cascata. O ciclo do modelo cascata é apresentado por Sommerville (2007, p. 44) na Figura 1, que devido o encadeamento de uma fase com a outra dá-se o nome de cascata. Nota-se que o processo neste tipo de modelo é **Linear**, pois não faz iteração com as fases já passadas em seu decorrer.

[3] *Cf.* http://agilemanifesto.org/iso/ptbr

Figura 1 – Ciclo de vida do modelo Cascata



Fonte: Sommerville (2007)

Onde, para o mesmo autor, cada fase tem como objetivo:

- **Definição de requisitos:** Definir os serviços, restrições e objetivos do sistema por meio de consulta ao usuário.

- **Projeto de sistema e *software*:** Dividir os requisitos entre *hardware* ou *software*, envolvendo a identificação, descrição e suas relações, estabelecendo uma arquitetura ao sistema.

- **Implementação e teste de unidade:** Realizar o projeto de *software* como unidades ou conjunto de programas.

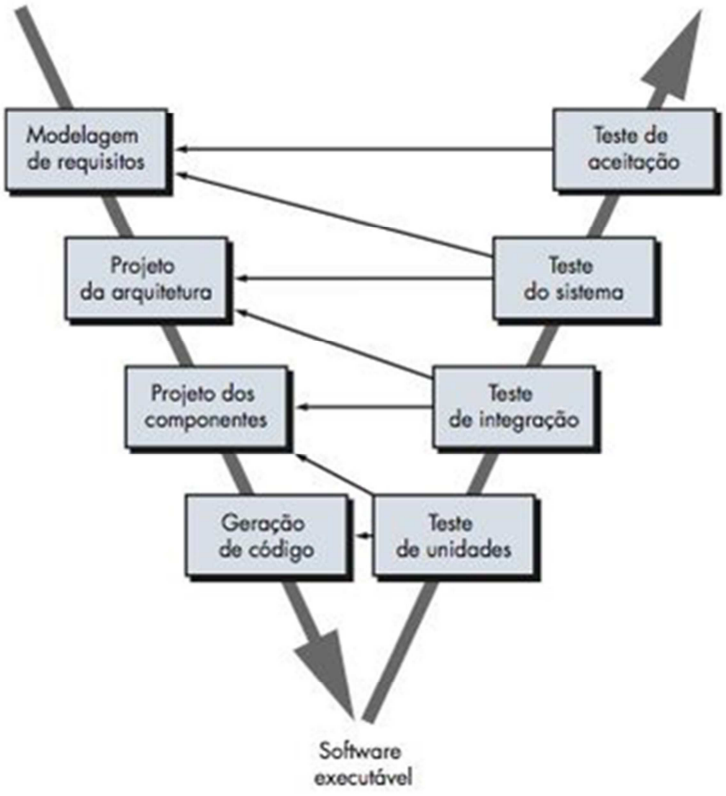
- **Integração e teste do sistema:** Juntar os programas desenvolvidos e testados para verificar se os requisitos ao sistema foram atendidos. Depois de testados, o *software* é liberado ao cliente.

- **Operação e manutenção:** Corrigir os erros que não foram identificados nas fases anteriores, seja para aprimoramento das unidades de sistema, ou para ampliar os serviços do sistema para novos requisitos.

O início de cada fase somente se dá quando a fase anterior já consiste de documentações aprovadas, ou seja, enquanto uma fase não é completada, as demais não podem prosseguir, o que pode resultar em retrabalhos quando se está na fase de implementação (SOMMERVILLE, p. 43-44).

Uma adaptação tomada ao modelo cascata foi inseri-lo em um modelo “V”, conforme Figura 2, onde faz uma relação entre as ações de garantia da qualidade e as comunicações na segunda etapa quanto a modelagem e atividades nos processos iniciais da primeira etapa, sendo assim um processo **Iterativo**, que se comparado ao ciclo de vida clássico da metodologia cascata, torna-se mais rápido, porém defasado, devido a necessidade de produção de documentação textual excessiva.

Figura 2 – Modelo Cascata em V



Fonte: Pressman (2011, p.60)

2.2.2. FDD *(Feature Driven Development)*

O FDD é uma metodologia que já existia antes do Manifesto Ágil. Pressman (2011, p.98) apresenta o FDD como abordagens ágeis:

> [...] o FDD adota uma filosofia que (1) enfatiza a colaboração entre pessoas da equipe FDD; (2) gerencia problemas e complexidade de projetos utilizando a decomposição baseada em funcionalidades, seguida pela integração dos incrementos de *software*, e (3) comunicação de detalhes técnicos usando meios verbais, gráficos e de texto. O FDD enfatiza as atividades de garantia da qualidade de *software* por meio do encorajamento de uma estratégia incremental, o uso de inspeções do código e do projeto, a aplicação de auditorias para garantia da qualidade de *software*, a coleta de métricas e o uso de padrões (para análise, projeto e construção).

O seu desenvolvimento é feito através das *features* (funcionalidades), que são os **requisitos funcionais** do sistema, que serão explicados no capítulo 3 (Requisitos) deste trabalho.

As *features* são elaboradas para as etapas de atividades e processos de uma determinada área que podem ser automatizadas pelo sistema (RETAMAL, 2014, p.68). Coad (1999, Apud Pressman 2011, p. 98) diz que a funcionalidade é “uma função valorizada pelo cliente passível de ser implementada em duas semanas ou menos". No FDD existe um modelo para a denominação de uma *feature*, sendo:

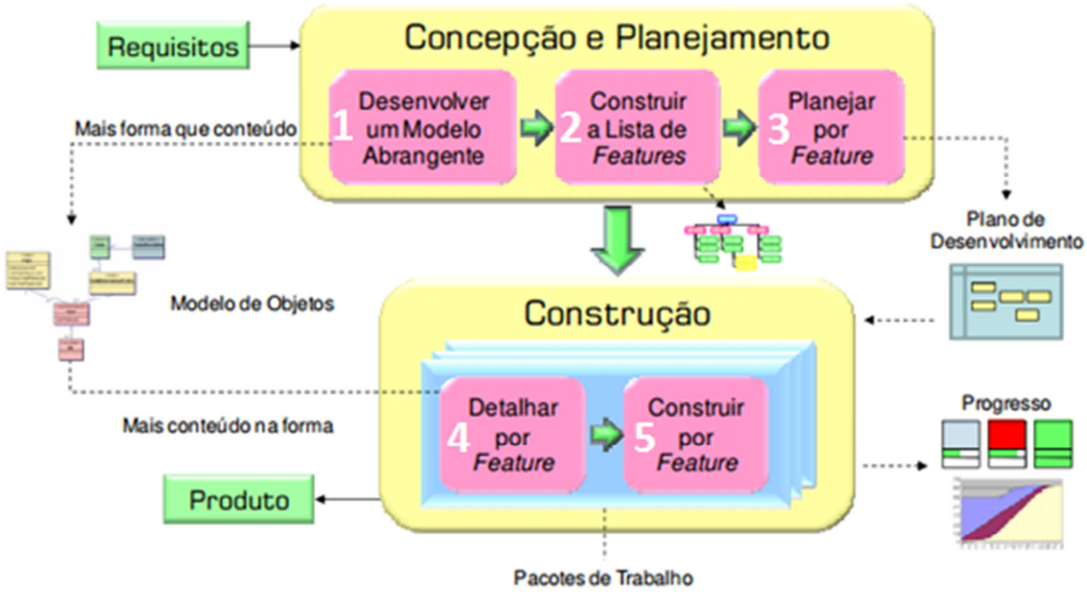
<Ação><Resultado><Objeto>

A **<Ação>** é o ato a ser tomado para um determinado **<Resultado>** do **<Objeto>**. Seguindo este modelo será possível uma padronização para a identificação das *features* que serão criadas. Para saber se uma *feature* criada está com o detalhamento necessário, é preciso que ela atenda aos seguintes critérios: [1] o desenvolvedor conseguir fornecer uma estimativa razoável para a implementação e [2] o testador escrever um teste que a *feature* esteja funcionando (RETAMAL, 2014, p.69).

O FDD é composto pelos cinco processos que são apresentados na Figura 3, onde os requisitos do sistema são as matérias primas. O ciclo de vida do FDD pode ser separadas em duas “fases”: [1] Linear (Concepção e Planejamento): criam-se os planos para as entregas incrementais, servindo como guia para a fase seguinte e [2]

Iterativa (Construção): criam-se os incrementos do produto já para o uso do cliente (RETAMAL, 2014, p.74).

Figura 3 – Estrutura do FDD



Fonte: Adaptado pelo autor de HEPTAGON (2015)

Para Retamal (2014, p.78) o primeiro processo, também chamado de DMA (Desenvolver um Modelo Abrangente), é uma atividade inicial que envolve todo o projeto, ocasião em que são compartilhados conhecimentos, levantamento dos requisitos, análise e síntese. Fazendo-se estudos sobre o escopo do sistema, são analisados os domínios de negócio a serem modelados e cria-se o **Modelo de Objetos**, modelo que será iterativamente atualizado em sua forma e conteúdo no quarto processo. A classificação dos objetos é de acordo com suas cores, sendo quatro cores distintas, onde o [1] Rosa representa algo que precisa ser registrado para eventos ocorridos em determinados momentos ou um intervalo de tempo; [2] Verde representa uma pessoa caracterizada como agente, um local onde a ação se desenvolve, ou objeto que participa da ação ou sofre alteração; [3] Amarelo representa o que irá ser feito por uma pessoa, lugar ou coisa para determinado evento de negócio (Rosa); [4] Azul define as caracterizações dos lugares, pessoas ou coisas.

O mesmo autor, diz que o segundo processo, também chamado de CLF (Construir a Lista de *Features*) identifica as *features* que serão importantes e que

satisfaçam os requisitos. Faz-se uma decomposição funcional do domínio de negócio, para assim, criar atividades de negócio para dentro das atividades de negócio, serem criados os passos de negócio, formando-se uma lista de *features*, geralmente categorizadas conforme seus níveis.

O terceiro processo, também chamado de PPF (Planejamento por *Feature*), é apresentado por Retamal (2014, p. 82) onde determina a ordem das *features* que foram criadas e serão implementadas conforme dependências entre uma *feature* e outra, carga de trabalho da equipe, e suas complexidades para a implementação, que para um bom planejamento ágil, cria-se o Plano de Desenvolvimento que deverá ter uma atualização ao longo das iterações.

O quarto processo, chamado também de DPF (Detalhar por *Feature*), para Retamal (2014, p.84), é uma atividade realizada para cada *feature*, tendo como objetivo, o desenvolvimento do *design* necessário, seguindo o modelo de objetos que fora criado no segundo processo, resultando em um pacote para assim poder ser implementado.

No quinto e último processo do ciclo de vida do FDD, também chamado de CPF (Construir por *Feature*), “é uma atividade realizada em cada funcionalidade, para produzir algo com valor para o cliente” (RETAMAL, 2014, p.86). Fazendo-se a implementação de itens necessários para que a classe suporte o projeto. Assim que passado pelos testes e pela inspeção, o código é incrementado como compilação atual (*build*).

Na segunda fase do FDD (DPF e CPF), também é criado o Quadro de Progresso, que faz o monitoramento do progresso individual de cada funcionalidade a ser desenvolvida, segundo Retamal (2014, p.87).

3. REQUISITOS

Medeiros (2013) faz uma relação de requisitos para a manutenção de *software*:

> [...] as falhas em requisitos estão entre as principais razões para a necessidade de uma manutenção de *software*. Entre as principais razões destacam-se os requisitos mal organizados, requisitos mal expressos, requisitos desnecessários para os clientes e a dificuldade para lidar com requisitos frequentemente mutáveis.

A definição de requisitos é descrevida por SOMMERVILLE (2007, p.79) sendo "[...] descrições dos serviços fornecidos pelo sistema e as suas restrições operacionais. Esses requisitos refletem as necessidades dos clientes de um sistema que ajuda a resolver algum problema". Para as instituições do CMMI e do IEEE se dão as seguintes definições (CMMI 2006, IEEE 2003 apud PAULA FILHO, 2009, p. 165):

> (1) Condição ou potencialidade de que um usuário necessita para resolver um problema ou atingir um objetivo. (2) Condição ou potencialidade que um sistema, componente ou produto deve possuir para que seja aceito (isto é, satisfaça a um contrato, padrão, especificação ou outro documento formalmente imposto). (3) Expressão documentada dessa característica.

Sommerville (2007, p.80) destaca que para uma boa definição dos requisitos de um sistema, é preciso fazer níveis de especificações, separando estes em requisitos funcionais e requisitos não funcionais.

3.1. Requisitos Funcionais

Os requisitos funcionais do sistema são dados como "O QUE" deve ser feito no *software* (Magela, 2006b, p.9). Acrescentado, Sommerville (2007, p. 80) descreve os requisitos funcionais como

> [...] declarações de serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como o sistema deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais podem também estabelecer explicitamente o que o sistema não deve fazer.

Já para o IEEE, os requisitos funcionais "representam os comportamentos que um programa ou sistema deve apresentar diante de certas ações de seus usuários" (IEEE, 2003 apud PAULA FILHO, 2009, p. 7).

3.2. Requisitos Não Funcionais

Sommerville (2007, p.80) aborda os requisitos não funcionais como:

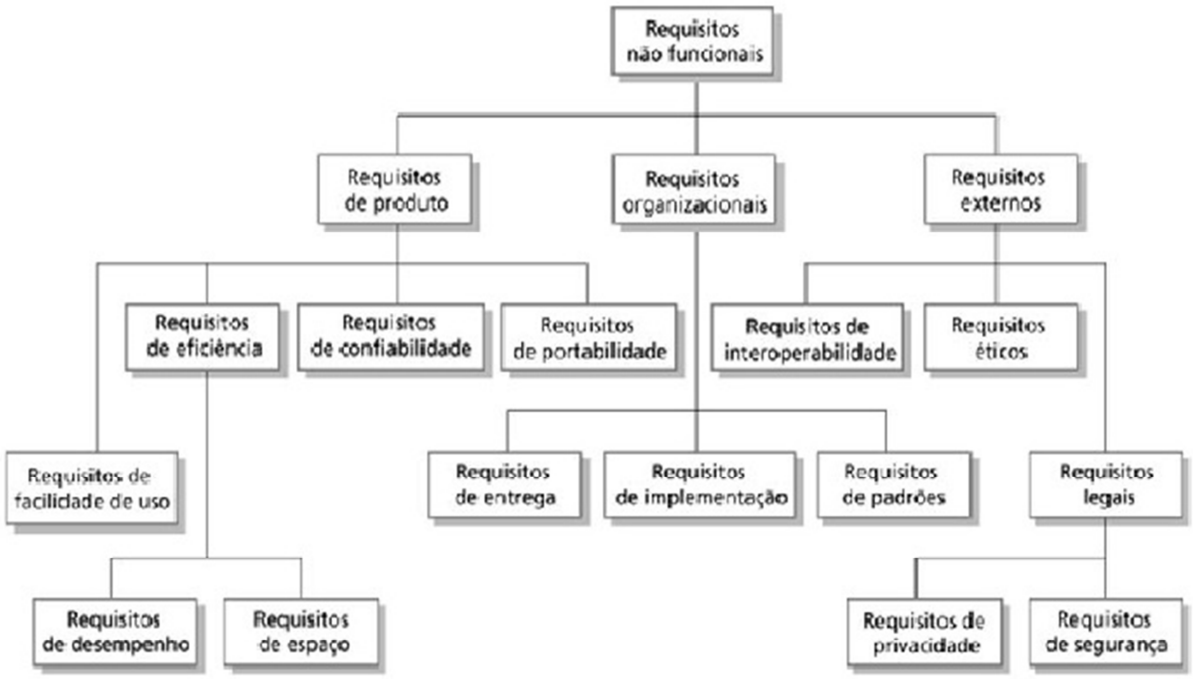
> Restrições sobre os serviços ou as funções oferecidos pelo sistema. Eles incluem restrições de *timing*[4], restrições sobre o processo de desenvolvimento e padrões. [...] em geral, eles não se aplicam às características ou serviços individuais de sistema.

Além disso, o IEEE (2003, *apud* PAULA FILHO, 2009, p. 7), diz que os requisitos não funcionais "quantificam determinados aspectos do comportamento".

Para os requisitos não funcionais, também são incluídos outros tipos de requisitos, como por exemplo, os Requisitos de Ambiente, que segundo Magela (2006b, p.4) trata-se do "estabelecimento de restrições ou necessidades externas ao *software* que está sendo produzido". Na Figura 4 são apresentados os requisitos não funcionais e suas derivadas.

[4] Tempo para execução.

Figura 4 – Tipos de Requisitos Não Funcionais



Fonte: Sommerville (2007, p.82)

4. MANUTENÇÃO DE *SOFTWARE*

A manutenção de *software* é caracterizada por Spinola (2015) sendo uma "modificação do *software* já entregue ao cliente, ou seja, a manutenção é qualquer alteração no *software* após sua entrada em produção", podendo assim descrever a manutenção como "um pequeno ciclo de vida do sistema" (BELLIN, 1993, p.106). Logo, a manutenção tem por função a correção e remoção dos defeitos no *software* que está em utilização, para que o sistema esteja sempre em evolução e não se torne obsoleto (PAULA FILHO, 2009, p. 662). Sendo que essas modificações, para Sommerville (2007, p. 326) podem ser "mudanças simples para corrigir erros de codificação, podem ser mudanças mais extensas para corrigir erros de projetos ou melhorias significativas para corrigir erros de especificação ou para acomodar novos requisitos".

Para Paula Filho (2009, p. 662-663), a manutenção de *software* é dividida em diferentes categorias, apresentadas no Quadro 2, onde cada uma tem uma função distinta entre elas. Importante verificar que, para a manutenção corretiva não há alteração de requisitos desde que o problema no sistema não seja a má interpretação de um requisito. Para a manutenção preventiva, não há alteração de requisitos, já para as manutenções adaptativa e perfectiva, ocorrem alterações nos requisitos existentes no *software*.

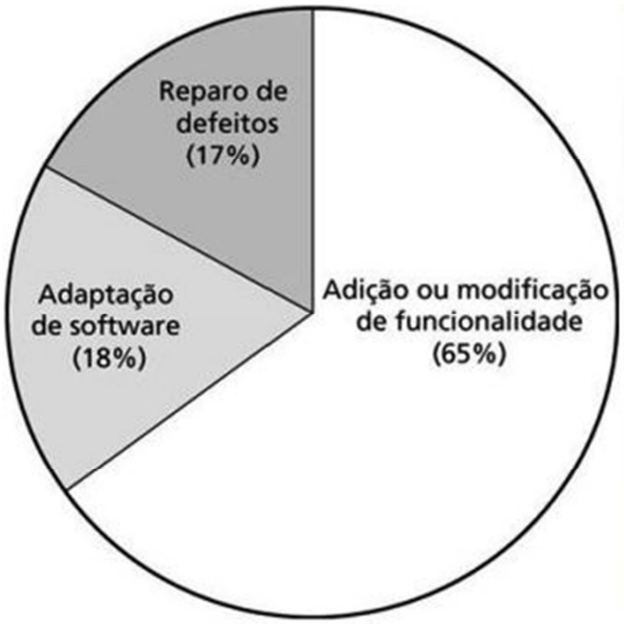
Quadro 2 – Tipos de Manutenção de *Software*

Tipo	Função
Corretiva	Correção de erros (*bugs*) após o fim do projeto de desenvolvimento
Preventiva	Localizar pontos no código ou desenho do *software* que podem ser melhorados
Adaptativa	Modificações nas interfaces de usuário, pequenas expansões funcionais e alterações em requisitos não funcionais
Perfectiva	Introduzir melhorias solicitadas pelos usuários

Fonte: Adaptado pelo autor de Paula Filho (2009, p.662-663)

Sommerville (2007, p. 326) destaca que as distinções desses tipos de manutenções não são claras na prática, pois quando adaptado o sistema ao novo ambiente pode ser feito adição de funcionalidades, que por sua vez, cada usuário tem uma maneira diferente de utilizar o sistema de alguma forma imprevisível, expondo os defeitos do *software* a qualquer momento de sua utilização. Por isso é preciso realizar mudanças no sistema para acomodação do modo de trabalho do usuário ao sistema. Na Figura 5 é apresentado um gráfico que relaciona a distribuição dos esforços que são gastos nas manutenções, que segundo Sommerville (2007, p. 326) a inserção e modificação de novos requisitos são os principais esforços para a realização das manutenções.

Figura 5 – Distribuição de esforços de manutenção



Fonte: Sommerville (2007, p.327)

Importante ressaltar a equipe de suporte técnico do sistema para as manutenções, pois, na maioria dos casos, são eles que recebem as alterações e falhas encontradas pelos usuários no sistema que precisam ser verificados.

4.1. Evolução de *Software*

Uma boa manutenção de *software*, consequentemente gera uma boa evolução de *software*. Sommerville (2007, p. 324) diz que a evolução de *software* é o estudo de mudanças no sistema, que tendo uma transição contínua de

desenvolvimento visando a evolução, é caracterizado como manutenção de *software*.

A dinâmica da evolução de sistemas foi definida por Lehman (1985, *Apud* Sommerville, 2007, p.324) através de suas leis que são relacionadas às mudanças no sistema, apresentada na Figura 6, sendo que as cinco primeiras leis foram propostas por Lehman, e as demais foram incrementadas através de outros trabalhos.

Figura 6 – Leis de Lehman

Lei	Descrição
1ª Mudança contínua	Um programa usado em um ambiente real deve mudar necessariamente ou tornar-se progressivamente menos útil.
2ª Complexidade crescente	À medida que um programa muda, sua estrutura tende a se tornar mais complexa. Recursos extras devem ser dedicados para preservar e simplificar a estrutura.
3ª Evolução de programa de grande porte	A evolução de programa 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.
4ª Estabilidade organizacional	Durante o ciclo de vida de um programa, sua taxa de desenvolvimento é quase constante e independente de recursos dedicados ao desenvolvimento do sistema.
5ª Conservação de familiaridade	Durante o ciclo de vida de um sistema, mudanças incrementais em cada versão são quase constantes.
6ª Crescimento contínuo	A funcionalidade oferecida pelos sistemas deve aumentar continuamente para manter a satisfação do usuário.
7ª Qualidade em declínio	A qualidade dos sistemas entrará em declínio a menos que eles sejam adaptados a mudanças em seus ambientes operacionais.
8ª Sistema de feedback	Os processos de evolução incorporam sistemas de feedback com vários agentes e loops e você deve tratá-los como sistemas de feedback para conseguir aprimoramentos significativos de produto.

Fonte: Sommerville (2007, p.325)

Sommerville (2007, p.324-325) faz as seguintes definições para cada lei:

A primeira lei diz que todo sistema inevitavelmente passará pelo processo da manutenção.

A segunda lei diz que conforme o sistema passa pelo processo de manutenção, sua estrutura vai sendo degradada. Tendo assim que ser investido em manutenções preventivas para evitar-se isso.

A terceira lei diz que para sistemas de grande porte, são limitadas determinadas mudanças no sistema a serem feitas na manutenção, consequente de fatores estruturais do sistema, podendo acarretar em defeitos no sistema devido a alguma modificabilidade.

A quarta lei diz que mudanças de pessoal ou de recursos tem efeitos imperceptíveis na evolução de *software* no longo prazo.

A quinta lei diz que novos defeitos irão surgir conforme adicionadas novas funcionalidades ao sistema, sugerindo não adicionar um grande incremento de funcionalidade se não considerar a necessidade de correções futuramente.

A sexta e a sétima lei estão relacionadas entre si, dizendo que os usuários ficarão cada vez mais infelizes em utilizar o sistema se este não for adicionado novas funcionalidades.

A oitava lei reflete a evolução para sistemas de *feedback*[5].

Devido a evolução envolver primeiramente a compreensão do sistema a ser mudado para depois ser implementada esta mudança, Sommerville (2007, p.331) diz que a **Reengenharia de Sistemas** pode ser utilizada para “simplificar os problemas de mudanças” no *software* e “para aprimorar sua estrutura e facilidade de compreensão” nas mudanças necessárias. Pode-se dizer que a reengenharia ajuda a investigar o atual *software* para a reconstrução de um produto já em uso (PRESSMAN, 2011, p. 668). Segundo Sommerville (2007, p.332) o processo da reengenharia de *software* pode ser dividido em cinco partes:

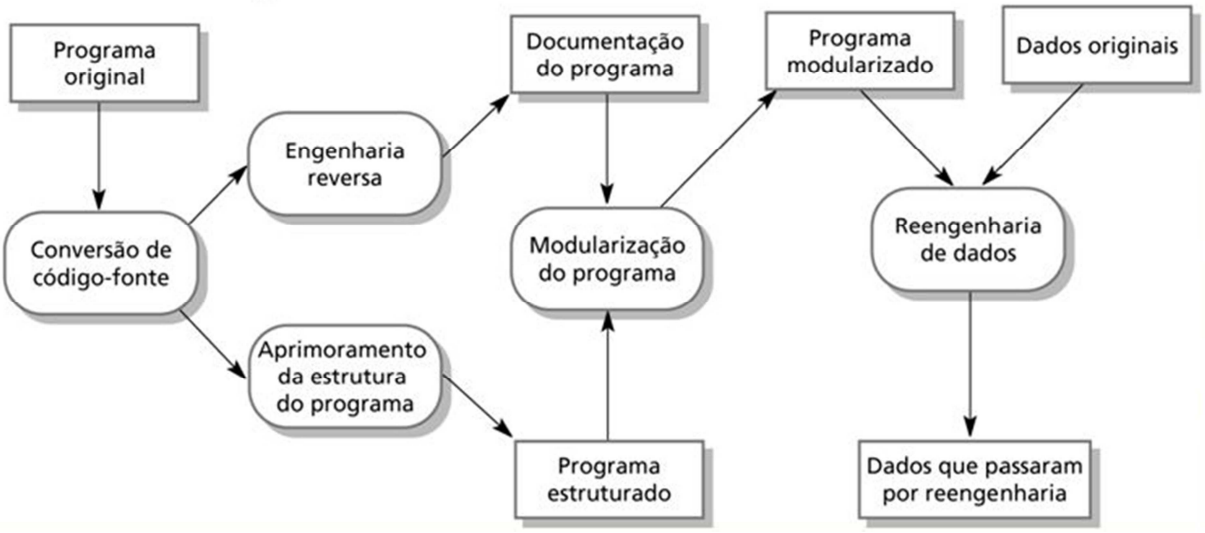
- **Conversão de código-fonte:** alteração de uma linguagem de programação antiga para uma mais moderna.

[5] Dados sobre o desempenho do sistema.

- **Engenharia Reversa:** análise e extração das informações do programa.
- **Aprimoramento da estrutura do programa:** análise e modificação da estrutura de controle do programa, visando facilitar sua compreensão e leitura.
- **Modularização de Programas:** agrupamento das partes relacionadas do programa e remoção de redundâncias se apropriado.
- **Reengenharia de Dados:** alteração dos dados processados para retratar as mudanças do sistema.

Estes são apresentados na Figura 7 que segundo Sommerville (2007, p. 332) “pode não requerer necessariamente todos os passos da figura”, devido alguns passos não serem necessários conforme o que está sendo analisado.

Figura 7 – Processos de Reengenharia



Fonte: Sommerville (2007, p.332)

4.1.1. Testes de *Software*

Uma fase do ciclo de vida do sistema que também visa a evolução do *software*, é a etapa de testes. PRESMANN (2011, p. 402) apresenta os testes como

> [...] um elemento de um tópico mais amplo, muitas vezes conhecido como verificação e validação (v&v). Verificação refere-se ao conjunto de tarefas que garantem que o *software* implementa corretamente uma função específica. Validação refere-se a um conjunto de tarefas

> que asseguram que o *software* foi criado e pode ser rastreado segundo os requisitos do cliente.

Implementando a definição de Pressman, Neto (2010) apresenta os testes de *software* como a execução do produto após o desenvolvimento, para verificar se foram atingidas suas especificações e funcionou corretamente dentro do ambiente projetado. Tendo como objetivo delatar as falhas deste produto, podendo assim ser feita a correção antes da entrega final.

Existem vários tipos de testes que podem ser executados para estas verificações. Um tipo de teste importante para a manutenção de *software* é o **Teste de Regressão** que para Neto (2010) “é uma estratégia importante para redução de “efeitos colaterais”. Consiste em se aplicar, a cada nova versão do *software* ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema”. Este também pode ser feito para “assegurar-se que as mudanças feitas no código não afetaram nenhuma funcionalidade existente” (ECLIPSE).

4.2. Riscos da Manutenção de *Software*

PAULA FILHO (2009, p. 572) diz que os riscos “são possíveis futuros eventos que podem trazer consequências indesejáveis”. Agregando este, Sommerville (2007, p. 69) diz que o risco “é algo que seria preferível não ocorrer [...] e podem ameaçar o projeto, o *software* que está sendo desenvolvido ou a organização”. Com isso, é feita a categorização dos riscos disponíveis no desenvolvimento de sistemas, apresentado no Quadro 3 por Sommerville (2007, p. 69).

Quadro 3 – Categorização dos Riscos de *Software*

Tipo	Ameaça
Riscos de Projeto	Ameaça ao planejamento do projeto (custos, equipe)
Riscos de Produto	Ameaça para a qualidade ou o desempenho do *software*
Riscos de Negócio	Ameaçam a organização (concorrência)

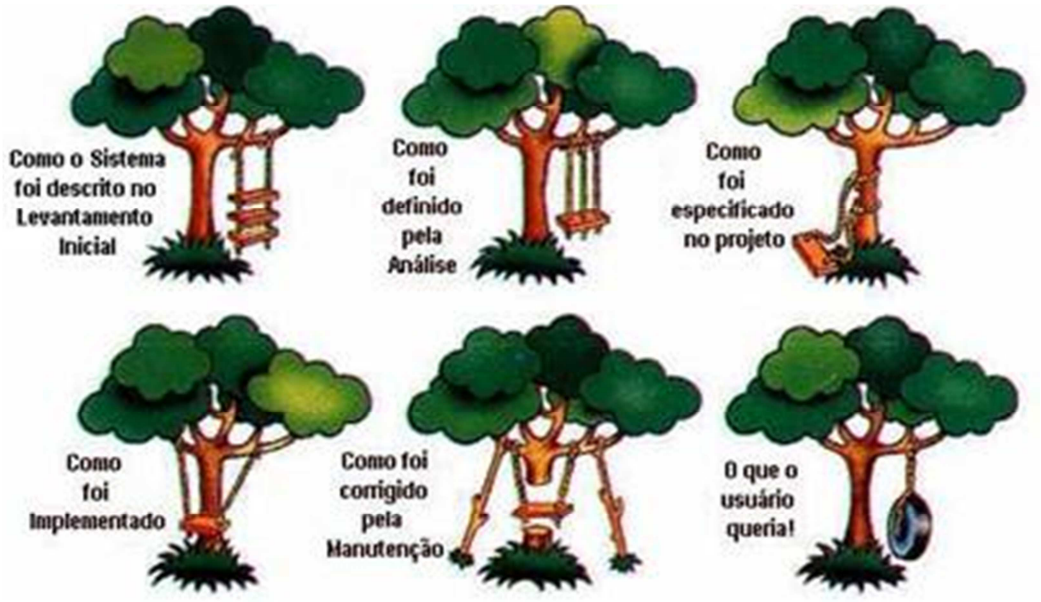
Fonte: Adaptado pelo autor de PAULA FILHO (2009, p. 572)

Os riscos existentes na manutenção de *software* são praticamente os mesmos riscos da etapa do desenvolvimento do *software*. Nos subcapítulos na sequência serão apresentados alguns deles, categorizados como risco de projeto e produto, riscos que serão os principais abordados no estudo de caso deste trabalho. Importante destacar o que diz Pivetta (2002 *Apud* Nogueira 2009, p. 45) que "embora a eliminação total e completa dos riscos seja um ideal da gestão de riscos, esta possibilidade é questionada pelos autores e especialistas da área".

4.2.1. Requisitos Mal Elaborados

"As principais causas de falhas em projetos são relativas aos requisitos. Essas falhas se devem às dificuldades em entender o que o usuário quer, descrições incompletas e mudanças não controladas nos requisitos" (KOSCIANSKI e SOARES, 2007, p. 174). Os mesmos autores (Koscianski e Soares, 2007, p.173), retratam que este fato ocorre devido aos requisitos passarem por outros departamentos antes de chegar à equipe de desenvolvimento, passando pelo analista no levantamento dos requisitos, em seguida para um gerente de projeto para modificações, retornar ao analista para assim, finalmente, ser entregue aos desenvolvedores. Considerando a etapa da manutenção, o requisito pode ser interpretado de uma outra maneira também, resultando no que é apresentado na figura 8 por Neto (2010).

Figura 8 – Má especificação de Requisitos



Fonte: Neto (2010)

4.2.2. Documentação

Nos métodos ágeis apresentados anteriormente, por serem desenvolvidos através de funcionalidades, não é comum fazer muita documentação para o desenvolvimento como diz Borges (2013, 00min36s), pois, acreditam que a comunicação verbal seja melhor que uma documentação para o desenvolvimento, economizando assim, tempo e custos que seriam utilizados para o desenvolvimento de documentações.

Mesmo assim, faz-se necessário documentações dentro dos métodos ágeis para a compreensão e registro para uma manutenção, como apresenta Bellin (1993, p. 80) para a documentação, dizendo que “a documentação de programa consiste em material escrito que permite que um programador seja capaz de efetuar manutenção em seu sistema de computador”. Portanto, dentro do método ágil costuma-se fazer a documentação somente se este tiver valor ao incremento que fora desenvolvido, e que seja menor que o custo de fazer e manter essa documentação (BORGES, 2013, 06min50s), o que pode acarretar em perda de tempo e esforços para localizar um erro necessário em uma manutenção, devido à falta de alguma documentação (ENGHOLM JUNIOR, 2010, p.31).

4.2.3. Equipe Mantenedora

Diversas vezes, a equipe para realização das manutenções de *software* não são as mesmas da etapa do desenvolvimento. O que pode resultar da nova equipe responsável pela manutenção não entender o código ou a base de decisões do projeto no sistema, além de geralmente as pessoas nesta equipe de manutenção serem inexperientes e sem familiaridade com a aplicação (SOMMERVILLE, 2007, p.32).

Assim como Bellin (1993, p.110), Sommerville (2007, p.328) diz que o processo da manutenção de *software* também é visto como uma "atividade de segunda classe" para muitas empresas de desenvolvimento de *software*, desvalorizando estas equipes. Além, de o mesmo processo, não ter muita interação com o pessoal que desenvolveu o código na etapa do desenvolvimento, ou então muitas vezes a equipe que desenvolveu nem se encontra mais na mesma instituição, resultando em uma manutenção de código alienígena (PRESMANN, 2011, p. 663).

4.2.4. Custos

O custo pode ser considerado como um dos riscos mais significativos da manutenção de *software*. Sommerville (2007, p.327) diz que os custos da manutenção vão de acordo com a manutenibilidade[6] do *software*. O mesmo autor, diz também que a inserção de novas funcionalidades depois da entrega do sistema é mais gravoso. Tendo assim que investir nos riscos anteriormente apresentados de documentação, e de estabilidade da equipe mantenedora, para não optar em terceirização deste processo da manutenção, que também pode ser uma opção de custo mais elevado.

4.3. Qualidade de *Software*

A qualidade de *software* é definida por Bessin (2004 *Apud* Pressman, 2011, p. 360) para um termo mais geral como "uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o

[6] "Facilidade de modificação de um produto de *software".* (KOSCIANSKI e SOARES, 2007, p.212)

produzem e para aqueles que o utilizam". Modificando este, Pressman (2011, p. 360) enfatiza essa definição em três pontos:

- **Qualidade Efetiva:** Infraestrutura utilizada para a produção de um *software*, práticas e maturidade utilizadas na engenharia de *software* durante sua construção.
- **Produto Útil:** Conteúdo, funções e recursos que o usuário deseja do produto, além de conceder confiabilidade[7], dispensa de erros e que atenda aos requisitos esperados.
- **Valores para o Fabricante e o Usuário:** Menos recursos gastos para realização de manutenções para correções de erros ao fabricante, e maior disponibilidade[8] do produto ao usuário.

4.3.1. SQuaRE: ISO/IEC 25000

O SQuaRE (*Software product Quality Requirements and Evaluation)* traduzido como Requisitos de Qualidade e Avaliação de Produtos de *Software*, também conhecido como ISO/IEC 25000[9], surgiu através da junção das normas [1] ISO/IEC 9126 que apresenta os instrumentos para verificação e validação da qualidade no *software*; e a [2] ISO/IEC 14598, junções de aspectos gerenciais, metodologias e indicação a documentações relevantes. Ambas se tratando de qualidade do produto de *software* (KOSCIANSKI e SOARES, 2007, p. 204-205). Norma que apresenta padrões e medições para a qualidade no *software* tendo como atributos seis pontos, sendo: [1] funcionalidade; [2] manutenibilidade; [3] usabilidade; [4] confiabilidade; [5] eficiência e [6] portabilidade (KOSCIANSKI e SOARES, 2007, p. 204).

5. ESTUDO DE CASO

Conforme o que foi exposto no capítulo 1 deste trabalho, será feito um estudo de caso aplicando os conhecimentos adquiridos e que foram expostos nos capítulos 2, 3 e 4 através de uma revisão bibliográfica. O estudo de caso visa evitar os possíveis riscos da manutenção de *software* e quando não possível, amenizá-los.

---

[7] "O quanto se pode esperar que um programa realize a função pretendida com a precisão exigida". (McCall, 1977 *Apud* Pressman, 201, p. 361)
[8] "[...] tempo durante o qual um produto pode ser utilizado". (KOSCIANSKI e SOARES, 2006, p. 244)
[9] *Cf.* <https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-2:v1:en>

Mostrando como a organização que desenvolveu o *software* Compre Bem *Corporate* poderia se comportar em uma manutenção para adição de novos requisitos à este sistema e aos requisitos que poderiam ser alterados. Todas as informações referentes ao sistema neste estudo, não teve acesso ao código em si. Estas foram coletadas entrando em contato diretamente com o fornecedor do *software.*

A escolha do *software* Compre Bem *Corporate* para utilização no estudo de caso se deu devido o mesmo possuir liberação de licença de distribuição livre e também de ter uma fácil compreensão de suas funcionalidades. Onde "*Software* Livre é aquele que vem com a permissão para qualquer um usar, copiar e distribuir, tanto a versão original do *software* como versões modificadas, de forma gratuita ou cobrando uma taxa" (FERREIRA, 2009, p.41). Dando assim, liberdade de utilização e estudo do sistema desenvolvido para utilização neste trabalho. *Software* que tem como função a gestão de compras de variados tipos de produtos de estoque.

5.1. Compre Bem *Corporate*

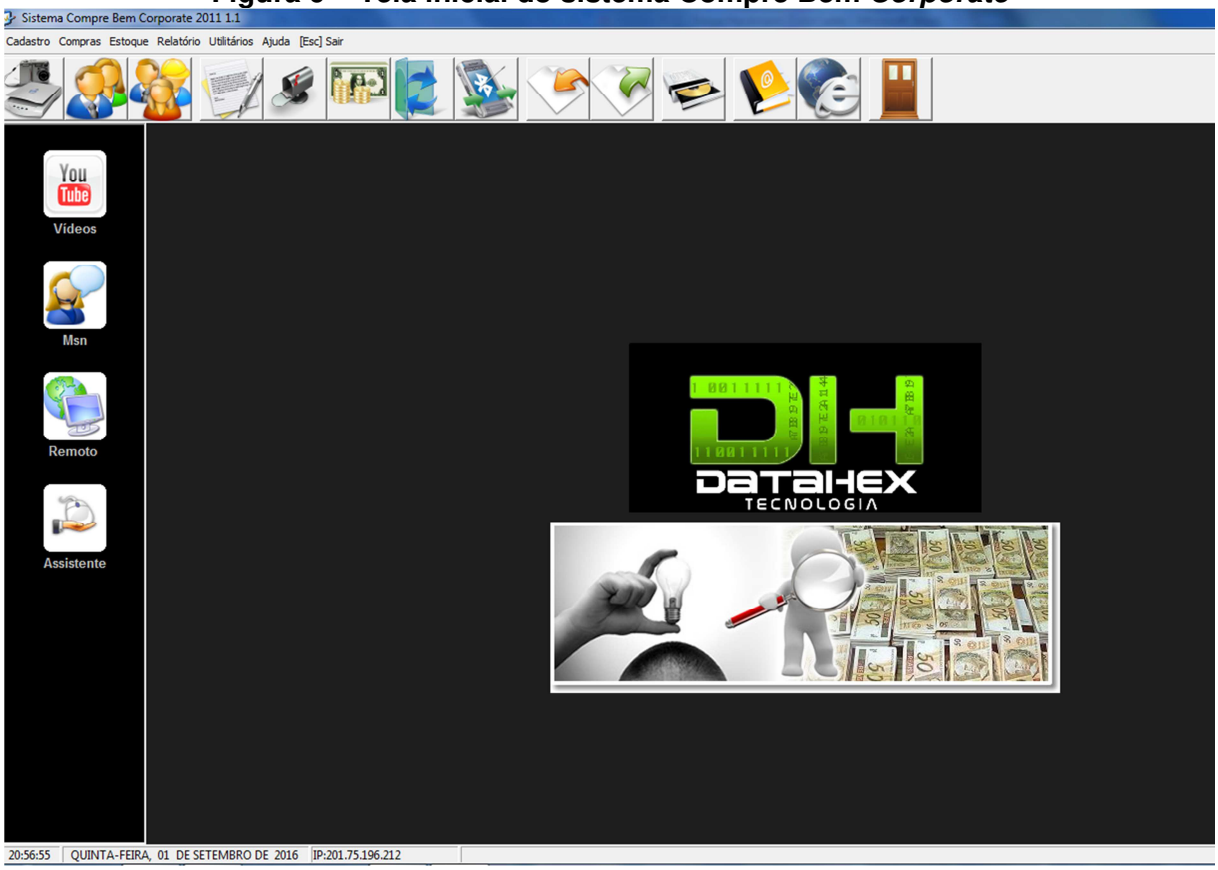
O Compre Bem *Corporate* é um sistema para uso empresarial que faz análise de preços dos produtos de diversos fornecedores, objetivando desenvolver uma lista de compra dos produtos ao menor preço possível dentre os fornecedores, tendo assim, uma maior economia ao usuário. Em outras palavras, pode-se dizer que o objetivo do sistema é gerir o processo de compras das empresas, desde a criação de uma lista de compra, até a geração de um pedido de compra filtrando os produtos com os melhores preços (mais baratos)[10].

O sistema foi desenvolvido na linguagem *Delphi 7* juntamente com o banco de dados *FireBird*. O uso do sistema é localmente no computador em que foi instalado, sendo assim, todas informações inseridas no mesmo são salvas no computador em que se está utilizando, porém, o sistema também utiliza alguns recursos via *web* para envio de e-mails, por exemplo, do modo em que a própria linguagem que foi desenvolvida e lhe é disponibilizada.

[10] Disponível em: http://www.datahex.com.br/produtos

As principais funcionalidades do sistema seriam as seguintes: [1] Cadastro de produtos; [2] Cadastro de fornecedores; [3] Lista de compras; [4] Cotação de fornecedores; [5] Pedidos de compras; [6] Envio de e-mails. Na tela inicial do sistema apresentado na Figura 9, é exibida a rotina para se chegar a estas funcionalidades, além de contar com botões de atalhos para se chegar a estes e outras funcionalidades secundárias.

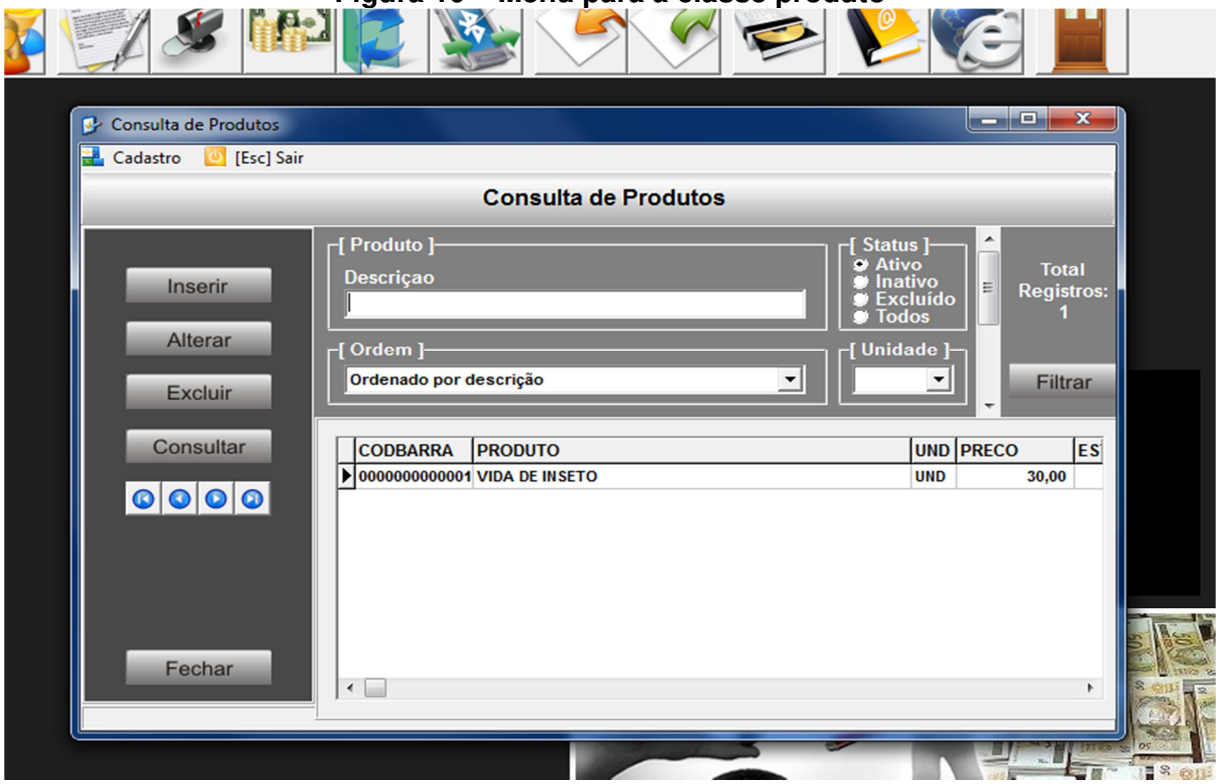
Figura 9 – Tela inicial do sistema Compre Bem *Corporate*



Fonte: Elaborado pelo autor.

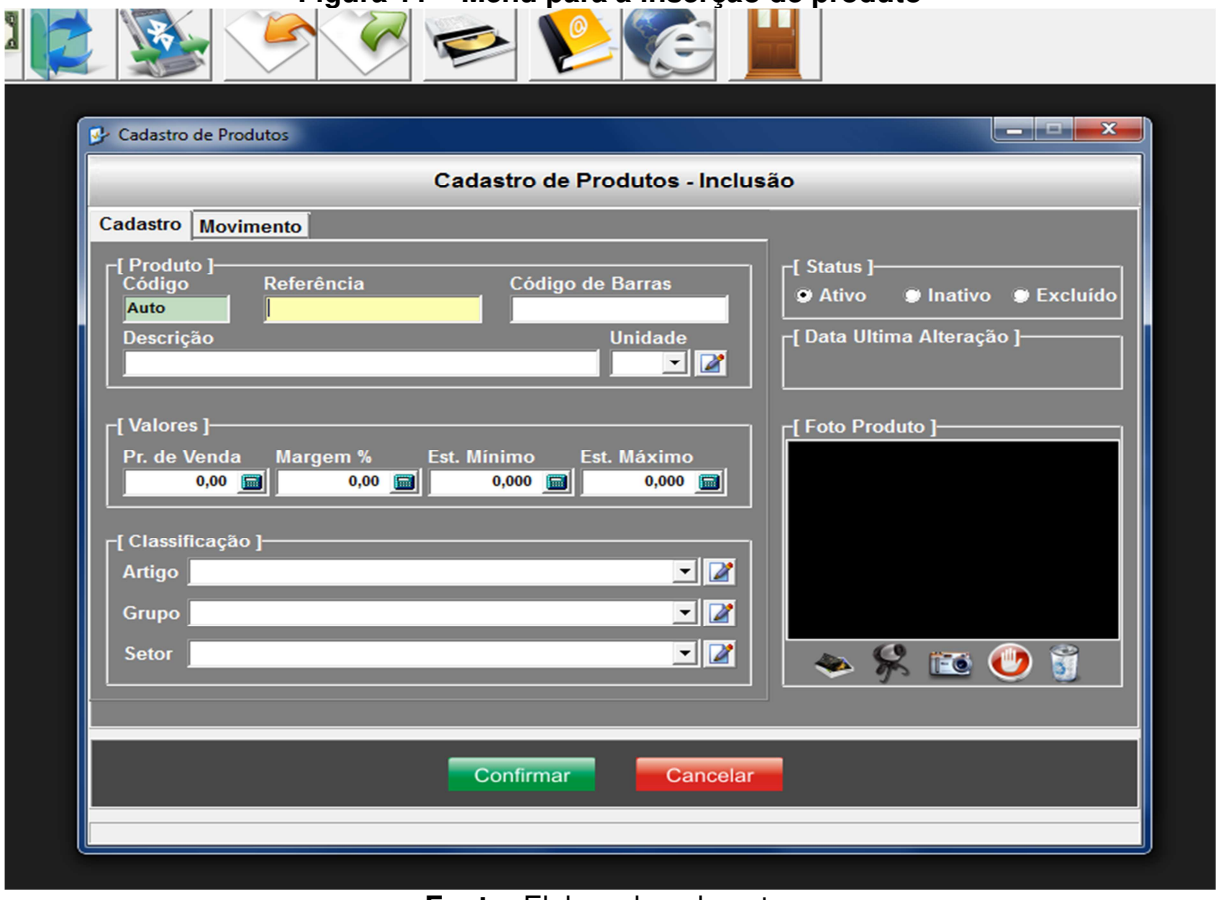
Cada funcionalidade principal do sistema tem em seus menus outras funcionalidades para as classes do sistema que se está utilizando (área de negócio), como por exemplo na Figura 10 no menu da classe Produto, contém quatro funcionalidades principais: [1] Inserir, [2] Alterar, [3] Excluir e [4] Consultar. Já a Figura 11, apresenta a tela de inserção de um novo produto ao sistema.

Figura 10 – Menu para a classe produto



Fonte: Elaborado pelo autor

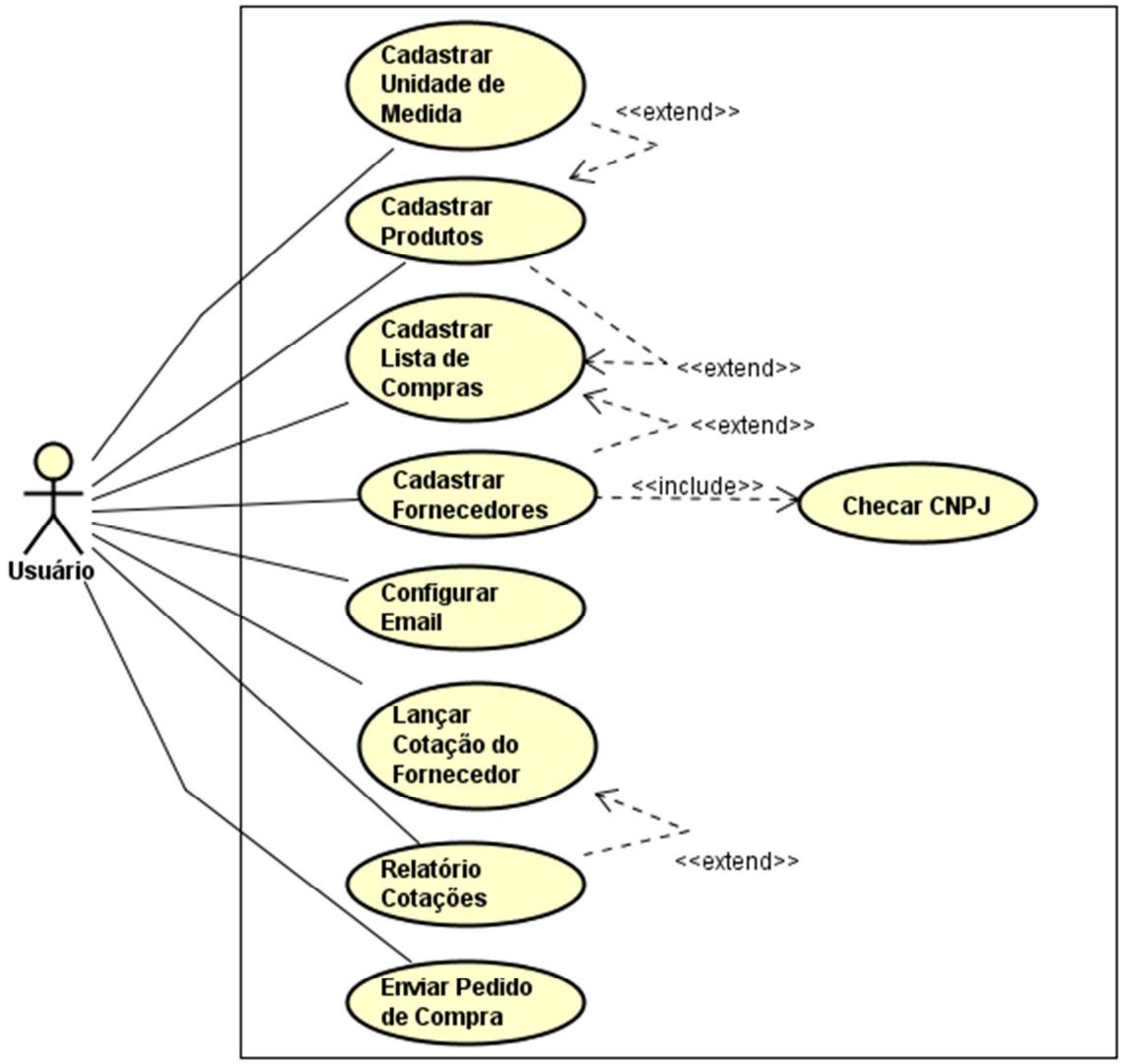
Figura 11 – Menu para a inserção de produto



Fonte: Elaborado pelo autor

A seguir serão apresentados alguns diagramas da ferramenta de UML que são utilizados para documentar, visualizar ou especificar o atual sistema Compre Bem *Corporate*, começando com o diagrama mais simples desta ferramenta, o diagrama de casos de uso apresentado na Figura 12. Diagrama que mostra as principais funcionalidades do sistema, ou, o que o sistema faz segundo o usuário (Ribeiro, 2012).

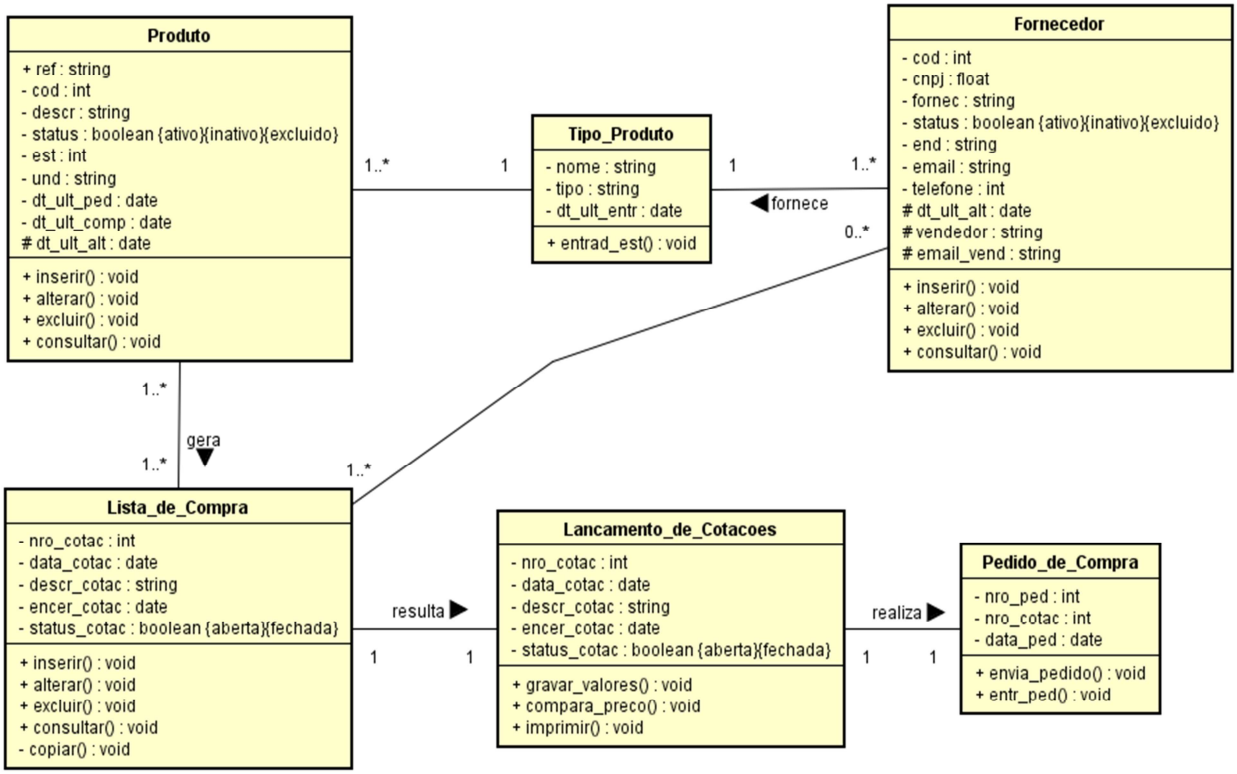
Figura 12 – Diagrama de Casos de Uso do Compre Bem *Corporate*



Fonte: Elaborado pelo autor.

O segundo diagrama da ferramenta UML a ser apresentado do *software* Compre Bem *Corporate* na Figura 13, será o diagrama de classe, apresentado de uma forma reduzida do sistema. Diagrama que já tem uma abordagem mais técnica, que demonstra determinados atributos e métodos que uma classe possui no sistema, apresentando também como as classes interagem entre si.

Figura 13 – Diagrama de Classe Reduzida do Compre Bem *Corporate*

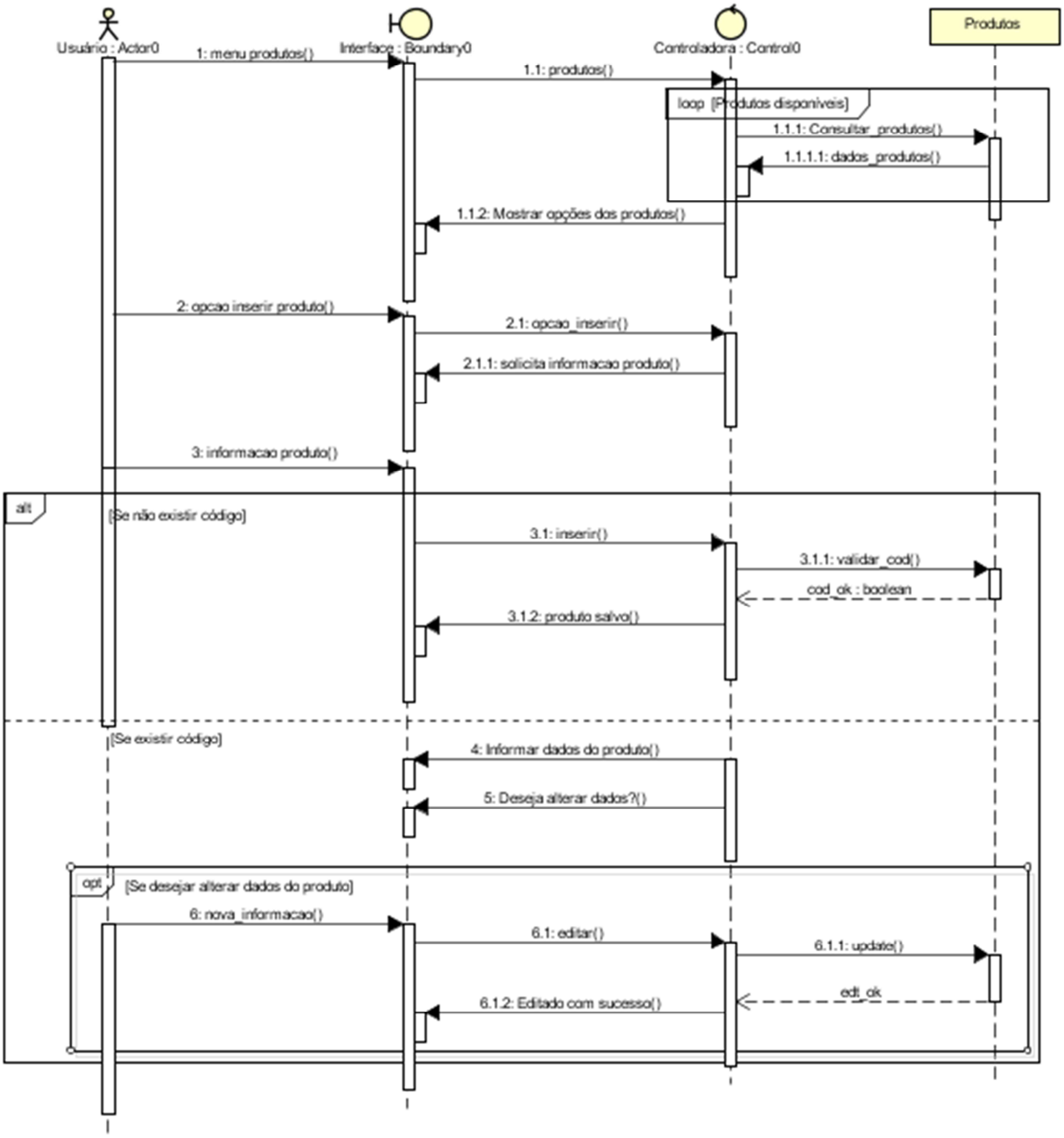


Fonte: Elaborado pelo autor.

O terceiro diagrama apresentado é referente ao diagrama de sequência para a classe Produto do *software* Compre Bem *Corporate*, apresentado na Figura 14. Diagrama que tem por finalidade, mostrar o passo a passo de como o sistema irá se comportar no percorrer das rotinas realizadas no sistema, apresentando as mensagens que emitem e recebem. Em outras palavras, o diagrama de sequência é apresentado por César (2009) da seguinte maneira:

> O diagrama de sequência preocupa-se com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em determinado processo, ou seja, quais condições devem ser satisfeitas e quais métodos devem ser disparados entre os objetos envolvidos e em que ordem durante um processo. Dessa forma, determinar a ordem em que os eventos ocorrem, as mensagens que são enviadas, os métodos que são chamados e como os objetos interagem entre si dentro de um determinado processo é o principal objetivo deste diagrama.

Figura 14 – Diagrama de Sequência do Compre Bem *Corporate* para inserção de produtos



Fonte: Elaborado pelo autor.

5.2. Manutenção no sistema Compre Bem *Corporate*

Como já foi apresentado no capítulo 2 deste trabalho, para os processos de *software*, utiliza-se uma metodologia. Será suposto neste estudo de caso, utilizando o método ágil do FDD, demonstrado também no item 2.2.2 deste trabalho. A escolha desta metodologia se deu devido o mesmo possuir modelos e planejamento nas

medidas sobre os riscos apresentados; favorecer a aproximação da equipe; utilizar a prática de proprietários de classes, tendo assim, uma familiaridade e particularidade dos desenvolvedores com determinadas partes do código, garantindo uma manutenção de qualidade e rápida; além também de possuir uma inspeção de código do que fora desenvolvido, para que assim o *software* possa ter uma boa evolução conforme apresentado anteriormente. Lembrando que o FDD não resolverá todos os problemas, apenas auxiliará para que as etapas da engenharia sejam mais eficientes. Para um melhor entendimento de um possível caso de manutenção no sistema Compre Bem *Corporate*, será apresentado uma suposta equipe da empresa que irão participar deste processo no sub-item 5.2.1.

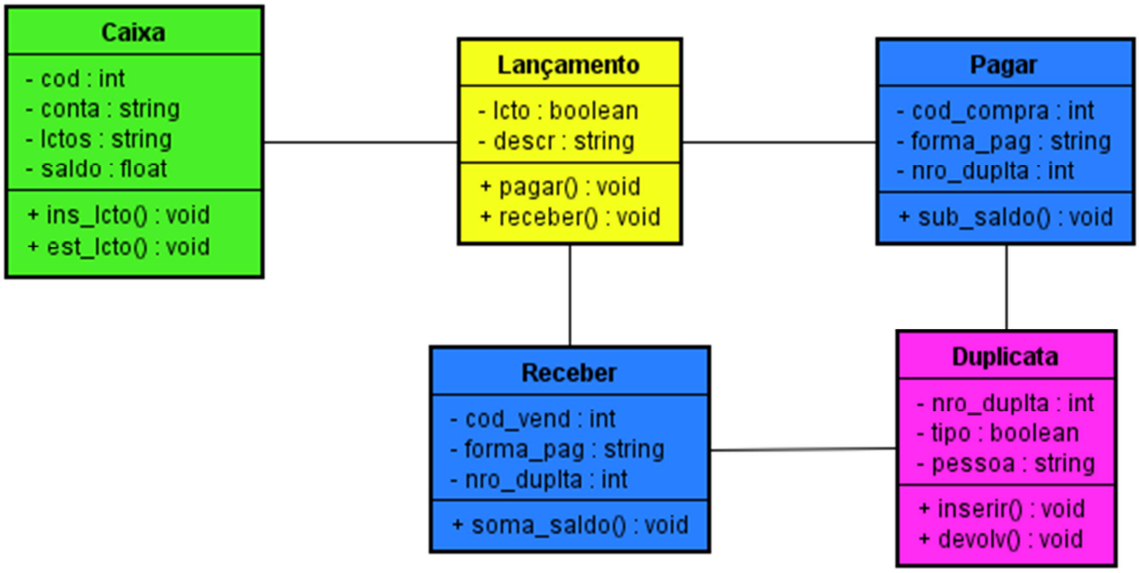
A manutenção a ser feita no sistema, veio através de um atendimento do suporte do sistema ao cliente, onde o mesmo fez duas solicitações de mudanças no sistema: [1] Incluir um módulo que faça o gerenciamento financeiro sobre as compras (caixas e pagamentos); e [2] Configurar para que um produto possa ser fornecido somente por um fornecedor. Para controle dos incidentes de atendimentos do suporte, foi levado em conta que a empresa dispõe de uma ferramenta CASE para registros e acompanhamento do percorrer das tarefas solicitadas e registradas por toda a empresa. Sendo assim, após registrar o que foi solicitado pelo usuário, o analista do suporte encaminha este registro para um departamento de análise (gerando um item de trabalho), onde verificam se o atual sistema já não atende às funcionalidades que foi registrado nas solicitações do usuário.

Se o que foi solicitado procede, o departamento de análise encaminha este item para o gerente de projetos, que irá verificar se será possível a mudança solicitada sem que ocorra uma implicação nas atuais funcionalidades do sistema, fazendo-se também uma análise dos requisitos, a fim de se evitar o primeiro risco apresentado no capítulo 4.2.1, verificando se este é possível causar um problema maior e pôr em risco as demais funcionalidades, desenvolvendo então o modelo abrangente, como de exemplo a Figura 15. Iniciando assim a fase 1 do ciclo do FDD na manutenção.

Para a primeira fase do FDD é levado em conta que os requisitos já passaram por uma engenharia e gerência para seu desenvolvimento. O Especialista de

Negócio também faz a organização dos processos das atividades, e, após feito o processo das atividades, estes são estudados com o Arquiteto Líder e Programadores, juntamente com as documentações existentes do sistema.

Figura 15 – Exemplo de Modelo de Objetos



Fonte: Elaborado pelo autor.

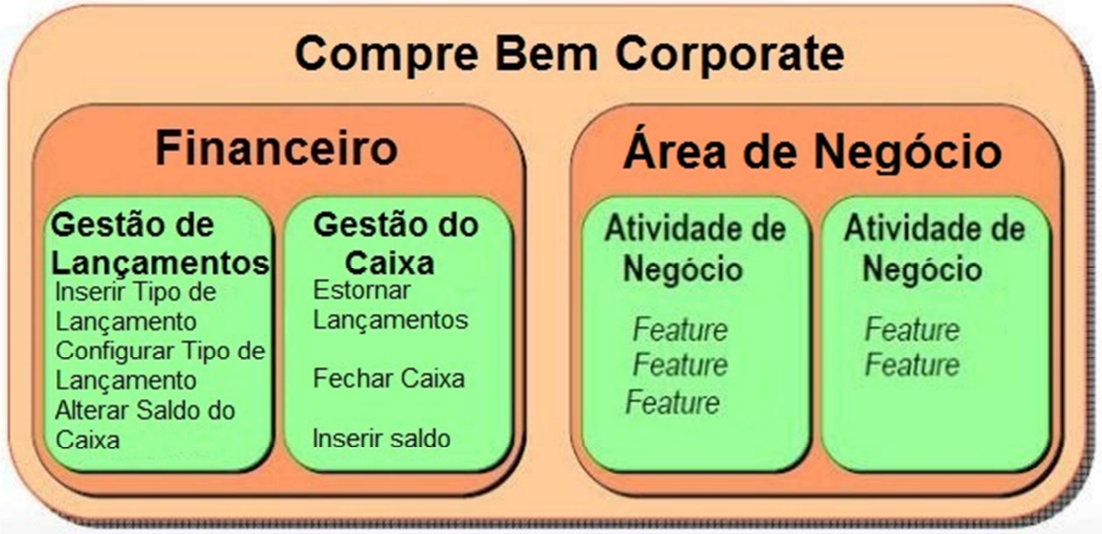
Em seguida, é dada a sequência com a segunda fase do FDD com os programadores líderes construindo a lista de funcionalidades, tendo como objetivo o que o sistema precisa ter para satisfazer as necessidades solicitadas pelo usuário. A funcionalidade de inclusão e alteração dos requisitos ao sistema neste estudo poderiam ser apresentados da seguinte forma:

<Controlar><Lançamentos><de um Caixa>

<Fornecer><Produtos><por um só Fornecedor>

Também podem ser desenvolvidos da seguinte maneira apresentada na Figura 16, também chamado de Mapa Mental (ao lado esquerdo, um caso para esta manutenção, ao lado direito, como se é baseado para a criação da lista), onde fica mais explícito as funcionalidades para seu entendimento.

Figura 16 – Exemplo de Construção da Lista de Funcionalidades (Mapa Mental)



Fonte: Adaptado pelo autor de Costa (2011).

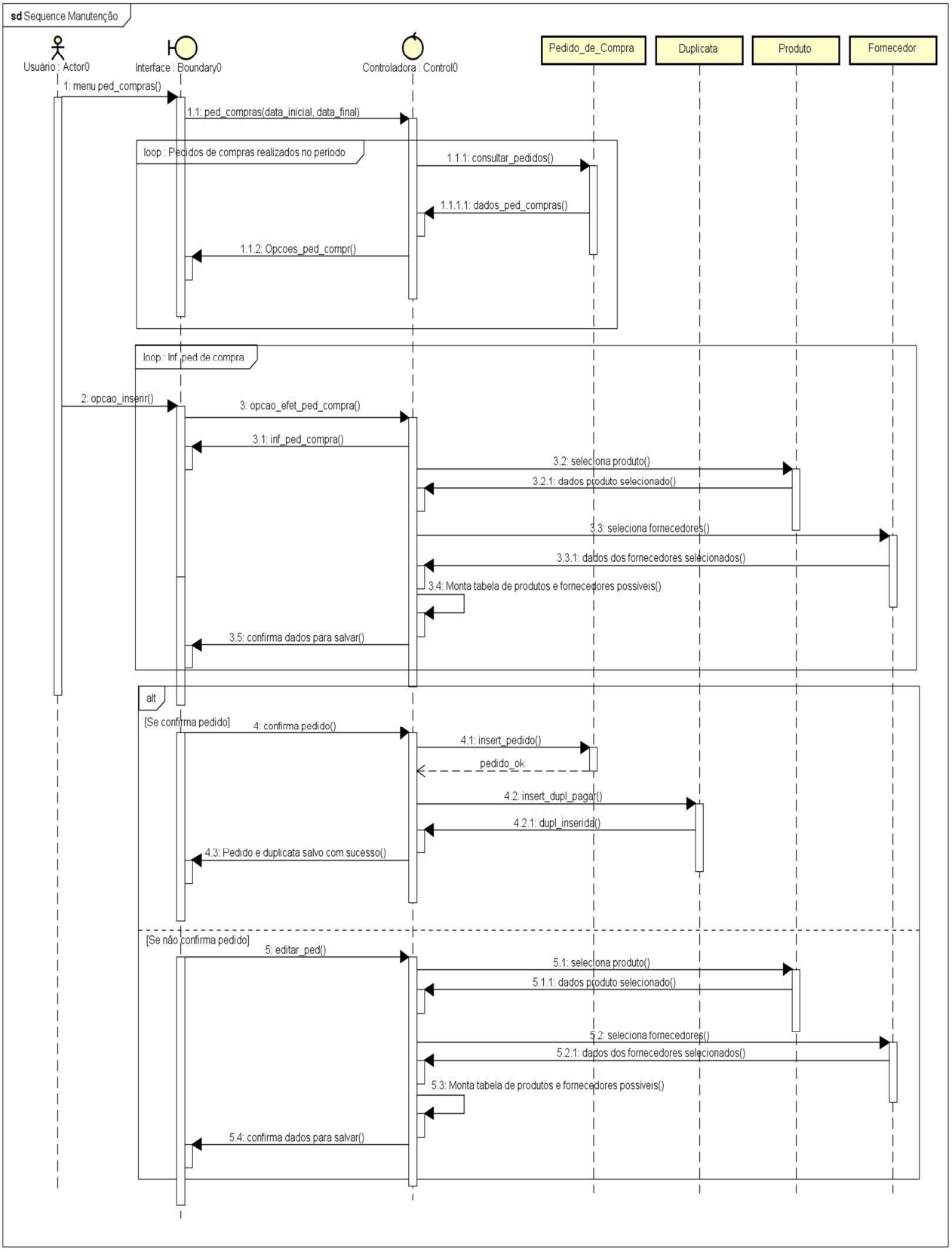
Após a construção da lista de funcionalidades na segunda fase, é feito o planejamento das funcionalidades (terceira fase). Onde o gerente de projeto, gerente de desenvolvimento e programadores líderes, definem uma ordenação das funcionalidades a serem desenvolvidas. Baseando-se na dependência das funcionalidades entre si, na complexidade para o desenvolvimento e na carga de trabalho da equipe.

Feito a atribuição dos conjuntos de funcionalidades para os programadores líderes, é atribuída as classes para os desenvolvedores. Fazendo-se as interações entre as equipes no percorrer do desenvolvimento de acordo com a necessidade observada pelos gerentes de projeto e desenvolvimento, juntamente com os programadores líderes.

Iniciando a quarta fase do FDD, é executada uma atividade com o objetivo de criar um *design* para a alteração para que, assim, os desenvolvedores façam o que foi planejado para o determinado pacote de alteração do sistema. Geralmente nesta etapa, é desenvolvido o diagrama de sequência referente a funcionalidade a ser implementada, apresentado na Figura 17 para o caso desta manutenção exposta a ser feita, refinando-se este pelo programador líder. Em seguida, é escrito um prefácio das classes e métodos das funcionalidades pelos desenvolvedores e também testes unitários. Finalizando esta fase, é realizado uma revisão no desenho

das funcionalidades, sendo proveitoso para localizar defeitos encontrados antecipadamente do desenvolvimento por completo.

Figura 17 – Diagrama de Sequência para manutenção executada



Fonte: Elaborado pelo autor.

Por fim, na quinta fase do FDD para esta manutenção seria produzidas as atividades em cada funcionalidade com valor para o cliente, de acordo com o que foi detalhado na fase anterior, em que os proprietários de classes planejaram os elementos necessários para as classes suportarem as funcionalidades com o desenho planejado. Em seguida, o código desenvolvido passa pelos testes unitários e para uma maior segurança sobre as existentes funcionalidades, é feito o teste de regressão. Após passar pelas inspeções, o atual código é acarretado para a atual compilação do sistema (*build*). Finalizando a manutenção feita, a fim de se esquivar dos riscos de documentação, é feito o registro das alterações feitas em comentários no código fonte e salvo os diagramas em um local para *backup*[11] e de segurança para a empresa. Por fim, é feito uma análise para liberação das novas funcionalidades, para verificar se ambas funcionalidades podem ser liberadas no mesmo pacote (versão) do sistema liberando este novo pacote aos usuários.

5.2.1. A Equipe

Para o desenvolvimento e manutenção de um *software*, “as pessoas, claro, são o fator primordial, a motivação e o propósito do sistema. São as pessoas que fazem tudo acontecer, seguindo processos apropriados (documentados ou não, padronizados ou não)” (RETAMAL, 2014, p. 71). Com isso, para este estudo, foi levado em conta o valor que a empresa dá aos seus funcionários, principalmente aos mantenedores, prevalecendo-os desde a etapa em que o sistema foi desenvolvido, seja por promoções ou por planos de carreiras para o incentivo, a fim de se esquivar do risco da equipe mantenedora.

Sendo assim, a FDD organiza a sua equipe com a seguinte estrutura e função segundo Retamal (2014, p. 72-74):

- **Gerente de Projeto:** Responsável pela administração do projeto, coordenar as ações da equipe e reportar o progresso para a alta gerência, cliente e demais partes interessadas.

[11] Cópia de segurança

- **Gerente de Desenvolvimento:** Responsável por liderar e organizar as ações da equipe de desenvolvimento de acordo com o que foi orientado pelo gerente de projeto.

- **Especialista de Negócio:** Responsável por comunicar e explicar à equipe de projeto as necessidades do *software* para o usuário, de acordo com o que foi solicitado pelo cliente no levantamento. Guiando-os para o que deve ser feito.

- **Arquiteto Líder:** Experientes em análise e modelagem de sistemas, lideram uma equipe para seguir um modelo de desenvolvimento de acordo com as funcionalidades determinadas, atuando também como "um facilitador na absorção nas regras de negócio" (AXMAGNO, 2007, p. 4, visto que o documento encontra-se em *pdf*).

- **Programadores Líderes:** Desenvolvedores mais reconhecidos e experientes, reconhecidos como líderes pela equipe. Responsáveis por coordenar o desenvolvimento, montar as equipes de funcionalidades e participar de definições técnicas. Além também de serem proprietários de classes.

- **Proprietários de Classes:** Desenvolvedores de uma equipe ao qual lhes foram atribuídas as classes do modelo. São compostos por pequenas equipes de funcionalidades onde se é feito o desenvolvimento. Dentre os proprietários de classes, também lhes podem ser atribuídos os demais papéis que são apresentados na sequência (assistente de projeto, testadores, gerente de versão, gerente de configuração, implantadores e escritores técnicos).

- **Assistente de Projeto:** Acompanha o percorrer do projeto, para que assim, o gerente de projeto possa se preocupar em outras questões de estratégia e táticas para novas funcionalidades. Podem também fazer eventos de integração e motivação da equipe, organizar documentações e cuidar da intercomunicação das equipes.

- **Testadores:** Fazem os testes no código que foi desenvolvido, dando ênfase ao teste de regressão que foi apresentado anteriormente.

- **Gerente de Versão:** Definem as funcionalidades desenvolvidas que serão liberadas em determinada versão do sistema para o ambiente de produção com o cliente.

- **Gerente de Configuração:** Garantem a integridade e segurança do repositório de artefatos. Cuidam também da montagem geral do produto e de sua compilação.

- **Implantadores:** Fazem a instalação da nova versão do sistema com as funcionalidades desenvolvidas no ambiente de produção do cliente, podendo também fazer um treinamento aos mesmos sobre as novas mudanças.

- **Escritores técnicos:** Descrevem manuais e documentações sobre o sistema e suas mudanças. Também podem fazer o material para treinamento dos clientes e suporte.

6. CONSIDERAÇÕES FINAIS

Este trabalho monográfico fez uma apresentação sobre a manutenção de um *software* e seus riscos, processo este que é tão presente na rotina das empresas de desenvolvimento de *softwares* e aplicativos, porém, pouco apresentado e valorizado pelas mesmas. Os riscos para a execução da manutenção, são variados de acordo com cada novo requisito ou alteração a ser feita, sendo alguns dos principais apresentados no estudo de caso que foi exposto neste trabalho. Para a elaboração da manutenção e compreensão para o desenvolvimento dos diagramas abordados no estudo de caso deste trabalho, foi levado em conta o conhecimento dos processos para a engenharia reversa e reengenharia, processo este que também seria preciso para a empresa caso não tivessem uma boa documentação de seu sistema.

Foi exposto que para um *software* estar ativo no mercado e não se tornar obsoleto, precisa passar por constante evolução, tendo assim que ser feita a manutenção, que para a execução desta com boa qualidade, também foi relatada a adoção de uma metodologia para controle do ciclo de vida do *software*, que seguindo o que foi exposto em cada etapa da manutenção no estudo de caso apresentado, as chances do *software* estudado obter os quesitos da SQuaRE ISO/IEC 25000 citado no trabalho, são de grande valia e chances, obtendo assim também uma maturidade nos processos para o CMMI.

Quanto a problematização deste trabalho (riscos da manutenção de *software* para a execução desta rotina), foram apresentados alguns riscos existentes na elaboração do estudo de caso. Esta problematização juntamente ao objetivo exposto foram alcançados parcialmente, devido ao fato de que seguindo-se o que foi feito no estudo de caso, a empresa conseguiria uma manutenção de boa qualidade, obtendo um *software* com boa manutenibilidade, confiabilidade, usabilidade e demais quesitos que definem uma qualidade de *software* como exposto, atingindo assim um de seus objetivos. Porém, um dos riscos explicado que seria o custo de manutenção, dificilmente poderia ser evitado, devido à empresa ter gasto seus recursos para documentação e estabilidade de sua equipe, outros riscos desta etapa da engenharia a fim de se obter qualidade.

Por fim, pode-se levar este objetivo não alcançado (como reduzir os custos de manutenção) para futuros temas de trabalhos e pesquisas. Pode-se também explorar o tema de testes de *softwares* e automatização de testes dentro da rotina de manutenção, para aumento de qualidade ou então para obter mais tempo livre para execução de novas manutenções, ou mesmo para investimento a novos sistemas para a empresa.

REFERÊNCIAS

AXMAGNO. Um guia de rápido aprendizado para a feature-driven development. *In.*: **Axmagno.com**, 2007. Disponível em: <http://homes.dcc.ufba.br/~mauricio052/Engenharia%20de%20Software%20I/FDD/FDD%20Em%20Uma%20Casca%20De%20Banana.pdf>. Acesso em: 06 out. 2016, às 22h49min.

BELLIN, David. **Manutenção de *software***: guia para administração de pequenos sistemas. Traduzido por José Renato Adorni Martins, São Paulo: Makron Books, 1993.

BORGES, Eduardo. Métodos ágeis e documentação de software (vídeo), 4 nov. 2013. *In.*: **Youtube.com**. Disponível em: <https://www.youtube.com/watch?v=3Smbhnmue7Y>. Acesso em: 03 Maio 2016, às 17h04min.

CÉSAR, Paulo. **Utilizando UML:** Diagrama de sequência. Artigo eletrônico SQL Magazine, número 64, ano 2009. Devmedia. Disponível em: < http://www.devmedia.com.br/artigo-sql-magazine-64-utilizando-uml/12665> Acesso em 21 Out, 2016.

CMMI. ***About* CMMI *Institute***. Disponível em: <http://cmmiinstitute.com/about-cmmi-institute>. Acesso em 03 Mar. 2016.

COSTA, Vinicius. Feature Driven Development, Laboratório de FDD. Blog do WordPress, 2011. Disponível em: <https://fddprocess.wordpress.com/laboratorio-de-fdd/>. Acesso em 26 Out. 2016.

DATAHEX. **Produtos**. Disponível em: <http://www.datahex.com.br/produtos>. Acesso em 01 Set. 2016.

ECLIPSE. **Diretriz:** Tipos de Teste de Desenvolvedor. Disponível em: <http://epf.eclipse.org/wikis/openuppt/openup_basic/guidances/guidelines/types_of_developer_tests,_eRutgC5QEduVhuZHT5jKZQ.html>. Acesso em: 07 Maio 2016.

ENGHOLM JUNIOR, Hélio. **Engenharia de *software* na prática**. São Paulo: Novatec, 2010.

FERREIRA, Marcelo. Entendendo o *Software* Livre. *In.*: de Melo, Tiago(Org.). **A Revolução do *Software* Livre**. Manaus: Comunidade Sol, 2009. p. 31-58 (Capítulo 1°).

HEPTAGON: Tecnologia da informação. **FDD: Estrutura**. Disponível em: <http://www.heptagon.com.br/fdd-estrutura>. Acesso em 25 Mar. 2016.

IEEE: INSTITUTE OF ELECTRICAL AND ELETRONICS ENGINEERS. ***About* IEEE.** Disponível em: <http://www.ieee.org/about/index.html>. Acesso em 03 Mar. 2016.

ISO. **ISO/IEC 25000:2014.** Disponível em: <https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-2:v1:en>. Acesso em 09 Maio 2016.

KOSCIANSKI, André. SOARES, Michel dos S. **Qualidade de *software***: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de *software*. 2ªed. São Paulo: Novatec, 2007.

LOBO, Edson J. R. **Curso de engenharia de *software***: Métodos e processos para garantir a qualidade no desenvolvimento de *softwares*. São Paulo: Digerati Books, 2008.

MAGELA, Rogério. **Engenharia de *software* aplicada**: Fundamentos. Rio de Janeiro: Alta Books, 2006a.

_____. **Engenharia de *software* aplicada**: Princípios. Rio de Janeiro: Alta Books, 2006b.

MANIFESTO ÁGIL PARA DESENVOLVIMENTO DE SOFTWARE. Disponível em: <http://agilemanifesto.org/iso/ptbr>. Acesso em: 22 Mar. 2016.

MEDEIROS, Higor. **Introdução a requisitos de *software***. Revista Eletrônica Engenharia de *Software*, número 58, ano 2013. Devmedia. Disponível em: <http://www.devmedia.com.br/introducao-a-requisitos-de-software/29580> Acesso em 04 Abr, 2016.

NETO, Arilo Claudio Dias. **Qualidade de *software***. Revista eletrônica Engenharia de *Software*, número 43, ano 2010. Devmedia. Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035> Acesso em 29 Abr, 2016.

NOGUEIRA, Marcelo. **Engenharia de *software***: Um Framework para a Gestão de Riscos em Projetos de *Software*. Rio de Janeiro: Ciência Moderna Ltda. 2009.

PAULA FILHO, W.P. **Engenharia de *software***: fundamentos, métodos e padrões. 3ªed. Rio de Janeiro: LTC,2009.

PRESSMAN, Roger S. **Engenharia de *software:*** uma abordagem profissional. Traduzido por Ariovaldo Griosi. 7ªed. Porto Alegre: AMGH, 2011.

RETAMAL, Adail Muniz. FDD - *Feature-Driven Development*. In.: PRIKLADNICKI, Rafael; WILLI Renato; MILANI, Fabiano (Org.). **Métodos Ágeis para Desenvolvimento de *Software***. Porto Alegre: Bookman, 2014. p. 66-101 (Capítulo 6°).

RIBEIRO, Leandro. **O que é UML e diagramas de casos de uso:** Introdução prática à UML. Artigo eletrônico Engenharia de *Software*, número 23408, ano 2012. Devmedia. Disponível em: <http://www.devmedia.com.br/o-que-e-uml-e-diagramas-de-caso-de-uso-introducao-pratica-a-uml/23408> Acesso em 10 Set, 2016.

SILVA, Nelson Peres. **Análise e estruturas de sistemas de informação.** São Paulo: Erica, 2007.

SOMMERVILLE, Ian. **Engenharia de *software***. Traduzido por Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edílson de Andrade Barbosa. 8ªed. São Paulo: Pearson Addison-Wesley, 2007.

SPINOLA, Rodrigo. **Manutenção de Software**: Definições e Dificuldades. Revista eletrônica SQL Magazine, número 86, ano 2015. Devmedia. Disponível em: <http://www.devmedia.com.br/manutencao-de-software-definicoes-e-dificuldades-artigo-revista-sql-magazine-86/20402#ixzz45lNHKHvC> Acesso em 13 Abr, 2016.